Bernd Finkbeiner
Laura Kovács (Eds.)

ARCoSS

LNCS 14571

Tools and Algorithms for the Construction and Analysis of Systems

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

2 Part II

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

Springer

OPEN ACCESS

Lecture Notes in Computer Science 14571

Founding Editors

Editorial Board Members

Advanced Research in Computing and Software Science

Subline of Lecture Notes in Computer Science

Subline Series Editors

Subline Advisory Board

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

Bernd Finkbeiner · Laura Kovács
Editors

Tools and Algorithms for the Construction and Analysis of Systems

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

Springer

Editors

Bernd Finkbeiner
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Laura Kovács
TU Wien
Vienna, Austria

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-57248-7 ISBN 978-3-031-57249-4 (eBook)
https://doi.org/10.1007/978-3-031-57249-4

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

Paper in this product is recyclable.

ETAPS Foreword

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

ETAPS 2024 is the 27th instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference established in 1998, and consists of four conferences: ESOP, FASE, FoSSaCS, and TACAS. Each conference has its own Program Committee (PC) and its own Steering Committee (SC). The conferences cover various aspects of software systems, ranging from theoretical computer science to foundations of programming languages, analysis tools, and formal approaches to software engineering. Organising these conferences in a coherent, highly synchronized conference programme enables researchers to participate in an exciting event, having the possibility to meet many colleagues working in different directions in the field, and to easily attend talks of different conferences. On the weekend before the main conference, numerous satellite workshops took place that attracted many researchers from all over the globe.

ETAPS 2024 received 352 submissions in total, 117 of which were accepted, yielding an overall acceptance rate of 33%. I thank all the authors for their interest in ETAPS, all the reviewers for their reviewing efforts, the PC members for their contributions, and in particular the PC (co-)chairs for their hard work in running this entire intensive process. Last but not least, my congratulations to all authors of the accepted papers!

ETAPS 2024 featured the unifying invited speakers Sandrine Blazy (University of Rennes, France) and Lars Birkedal (Aarhus University, Denmark), and the invited speakers Ruzica Piskac (Yale University, USA) for TACAS and Jérôme Leroux (Laboratoire Bordelais de Recherche en Informatique, France) for FoSSaCS. Invited tutorials were provided by Tamar Sharon (Radboud University, the Netherlands) on computer ethics and David Monniaux (Verimag, France) on abstract interpretation.

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

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

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

The ETAPS Steering Committee consists of an Executive Board, and representatives of the individual ETAPS conferences, as well as representatives of EATCS, EAPLS, and EASST. The Executive Board consists of Marieke Huisman (Twente, chair), Andrzej Wąsowski (Copenhagen), Thomas Noll (Aachen), Jan Kofroň (Prague), Barbara König (Duisburg), Arnd Hartmanns (Twente), Caterina Urban (Inria), Jan Křetínský (Munich), Elizabeth Polgreen (Edinburgh), and Lenore Zuck (Chicago).

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

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

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

April 2024

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

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

TACAS is a forum for researchers, developers and users interested in rigorous tools and algorithms for the construction and analysis of systems. The conference aims to bridge the gaps between different communities with this common interest and to support them in their quest to improve the utility, reliability, flexibility, and efficiency of tools and algorithms for building systems. TACAS 2024 interleaves and integrates various disciplines, including formal verification of software and hardware systems, static analysis, probabilistic programming, program synthesis, concurrency, testing, simulations, verification of machine learning/autonomous systems, Cyber-Physical Systems, SAT/SMT solving, automated and interactive theorem proving, and proof checking.

There were four submission categories for TACAS 2024:

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

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

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

Similarly to previous years, it was possible to submit an artifact alongside a paper. Artifact submission was mandatory for regular tool and tool demo papers, and voluntary for regular research and case study papers at TACAS 2024. An artifact might consist of a tool, models, proofs, or other data required for validation of the results of the paper. The Artifact Evaluation Committee (AEC) was tasked with reviewing the artifacts, based on their documentation, ease of use, and, most importantly, whether the results presented in the corresponding paper could be accurately reproduced. Most of the evaluation was carried out using a standardized virtual machine to ensure consistency of the results, except for those artifacts that had special hardware or software requirements. Artifact evaluation at TACAS 2024 consisted of two rounds. The first round implemented the mandatory artifact evaluation of regular tool and tool demonstration papers; this round was carried out in parallel with the work of the PC. The judgment of the AEC was communicated to the PC and weighed in their discussion. The second round of artifact evaluation carried out the voluntary artifact evaluation of regular research and case study papers, and took place after paper acceptance notifications were sent out; authors of accepted regular research and case study papers were able to update and revise their respective artifacts before artifact evaluation started. In both rounds, the AEC provided 3 reviews per artifact and anonymously communicated with the authors to resolve apparent technical issues. In total, 104 artifacts were submitted and the AEC evaluated a total of 62 artifacts regarding their availability, functionality, and/or reusability. Papers with an artifact that were successfully evaluated include one or more badges on the first page, certifying the respective properties.

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

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

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

April 2024

Bernd Finkbeiner
Laura Kovács
PC Chairs

Hadar Frenkel
Michael Rawson
AEC Chairs

Dirk Beyer
SV-Comp Chair

Organization

Program Committee Chairs

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

Program Committee

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

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

Artifact Evaluation Committee Chairs

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

Artifact Evaluation Committee

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

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

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

SV-COMP Program Committee and Jury

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

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

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

TACAS Steering Committee

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

Additional Reviewers

Parosh Aziz Abdulla
Guy Amir
Andrei Arusoaie
Shaun Azzopardi
Thom Badings
Milan Banković
Chinmayi Prabhu Baramashetru
Sebastien Bardin
Ludovico Battista
Anna Becchi
Lena Becker
Sidi Mohamed Beillahi
Yoav Ben Shimon

Csaba Biró
León Bohn
Alberto Bombardelli
Wael-Amine Boutglay
Eline Bovy
Matías Brizzio
Gianpiero Cabodi
Francesca Cairoli
Marco Campion
Marco Carbone
Martin Ceresa
Kevin Cheang
Md Solimul Chowdhury

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

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

Contents – Part II

Model Checking

JPF: From 2003 to 2023 . . . 3
Cyrille Artho, Pavel Parízek, Daohan Qu, Varadraj Galgali, and Pu (Luke) Yi

Hitching a Ride to a Lasso: Massively Parallel On-The-Fly LTL Model Checking . . . 23
Muhammad Osama and Anton Wijs

Towards Safe Autonomous Driving: Model Checking a Behavior Planner during Development . . . 44
Lukas König, Christian Heinzemann, Alberto Griggio, Michaela Klauck, Alessandro Cimatti, Franziska Henze, Stefano Tonetta, Stefan Küperkoch, Dennis Fassbender, and Michael Hanselmann

Enhancing GenMC's Usability and Performance . . . 66
Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis

Automata and Learning

Scalable Tree-based Register Automata Learning . . . 87
Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist

Small Test Suites for Active Automata Learning . . . 109
Loes Kruger, Sebastian Junges, and Jurriaan Rot

MATA: A Fast and Simple Finite Automata Library . . . 130
David Chocholatý, Tomáš Fiedor, Vojtěch Havlena, Lukáš Holík, Martin Hruška, Ondřej Lengál, and Juraj Síč

Software Verification

Accelerated Bounded Model Checking Using Interpolation Based Summaries . . . 155
Mayank Solanki, Prantik Chatterjee, Akash Lal, and Subhajit Roy

Weakest Precondition Inference for Non-Deterministic Linear Array Programs 175
Sumanth Prabhu S, Deepak D'Souza, Supratik Chakraborty, R Venkatesh, and Grigory Fedyukovich

Automated Software Verification of Hyperliveness 196
Raven Beutner

A Comprehensive Specification and Verification of the L4 Microkernel API 217
Leping Zhang, Yongwang Zhao, and Jianxin Li

Probabilistic Systems

Accurately Computing Expected Visiting Times and Stationary Distributions in Markov Chains 237
Hannah Mertens, Joost-Pieter Katoen, Tim Quatmann, and Tobias Winkler

CTMCs with Imprecisely Timed Observations 258
Thom Badings, Matthias Volk, Sebastian Junges, Marielle Stoelinga, and Nils Jansen

Pareto Curves for Compositionally Model Checking String Diagrams of MDPs 279
Kazuki Watanabe, Marck van der Vegt, Ichiro Hasuo, Jurriaan Rot, and Sebastian Junges

Learning Explainable and Better Performing Representations of POMDP Strategies 299
Alexander Bork, Debraj Chakraborty, Kush Grover, Jan Křetínský, and Stefanie Mohr

Simulations

Dissipative quadratizations of polynomial ODE systems. . . . 323
Yubo Cai and Gleb Pogudin

Forward and Backward Constrained Bisimulations for Quantum Circuits 343
A. Jiménez-Pastor, K. G. Larsen, M. Tribastone, and M. Tschaikowski

A Parallel and Distributed Quantum SAT Solver Based on Entanglement and Teleportation 363
Shang-Wei Lin, Tzu-Fan Wang, Yean-Ru Chen, Zhe Hou, David Sanán, and Yon Shin Teo

Author Index 383

Model Checking

JPF: From 2003 to 2023⋆

Cyrille Artho[1](✉), Pavel Parízek[2], Daohan Qu[3], Varadraj Galgali[4], and Pu (Luke) Yi[5]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
artho@kth.se
[2] Charles University, Prague, Czech Republic
parizek@d3s.mff.cuni.cz
[3] Nanjing University, Nanjing, China
daohanqu@gmail.com
[4] Belgaum, India
varad23711@gmail.com
[5] Stanford University, Stanford, USA
lukeyi@stanford.edu

Abstract. We give an account of JPF's current architecture as it has evolved over the last 20 years. Key changes include a modular, extensible design, and Java 11 support.
Java 11 brought with it fundamental changes in the language and its runtime, in particular, a new modular library system, different compilation of string expressions to bootstrap methods, and changes in many internal interfaces that allow access to the loaded code and the virtual machine state. These changes required numerous adaptations in JPF to ensure a successful compilation and correct behavior under Java 11.

Keywords: JPF · Java · Software model checking · Program analysis.

1 Introduction

JPF is a framework for Java bytecode analysis [1,2] that can be used to verify and search for bugs in programs written in Java-like languages. At the core of the system is an explicit-state model checker [3], which can be extended to allow many other analyses, such as symbolic execution [4].

Earlier papers covered the original architecture of JPF as a virtual machine for Java bytecode [1] or gave an abridged account of the current architecture [2]. This paper gives a detailed description of the current architecture, which is much more modular and extensible than 20 years ago and supports native methods through a well-designed interface.

As part of the developments of the last two decades, Java 11 was the first long-term release that brought major changes to Java and gave up on full backward compatibility.[6] Key changes at the bytecode level include a new modular library system, a different compilation of string expressions to bootstrap methods, and the removal of or changes in many internal interfaces [5].

[⋆] Supported by Google Summer of Code.

[6] While most of these changes were introduced with Java 9 as a development release, we will group any changes between Java 8 and Java 11 under the latter.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 3–22, 2024.
https://doi.org/10.1007/978-3-031-57249-4_1

For its program analysis, JPF has to support the full functionality of Java bytecode and integrate closely with the underlying Java Virtual Machine (JVM). Thus, it is impacted by internal changes of the Java platform that do not affect most other applications. This was evidenced by JPF first not even compiling under Java 11. After one year, we had adapted the code base so it compiled, but about 75 % of all regression tests failed initially. Four years of additional work addressed the major changes that were needed to support Java 11. During this time, we added over 140 new regression tests (and removed a few obsolete ones) and implemented new functionality with about 10,000 lines of additional code.

This paper is the first detailed publication presenting JPF's architecture and capabilities as they have evolved over the last 20 years since the early version of JPF, which was designed differently [1]. It also gives an overview of the challenges involved in adapting a bytecode-level program analysis tool to major architectural and implementation-level changes of the underlying platform. The remainder of this paper is organized as follows: Section 2 covers the background and related work, while Section 3 describes JPF's architecture. Section 4 describes the key changes from Java 8 to Java 11 and the adaptations in JPF to support them. Section 5 covers other major enhancements from the last five years, and Section 6 shows the evolution of JPF over that time. Finally, Section 7 summarizes our work and concludes.

JPF is freely available on GitHub, in the repository `https://github.com/javapathfinder/jpf-core/`. Extensive user and developer documentation, including an installation and how-to-run guide, is provided in the form of wiki pages at `https://github.com/javapathfinder/jpf-core/wiki`.

2 Background and Related Work

JPF is an extensible framework for Java bytecode analysis [1,2] and built as an explicit-state model checker [3].

In order to explore all possible and relevant outcomes of a program execution, JPF explores all possible outcomes of non-deterministic choices. Such choices can be induced by a non-deterministic thread schedule or unspecified variables/inputs. JPF has the ability to backtrack an execution to a previous point to analyze alternative outcomes. Therefore, it implements a fully-fledged JVM by itself but uses the underlying JVM (the "host JVM"; also see Section 3) to access the underlying platform's functionality. This access happens by delegating native methods at the JPF level to the host JVM.

By default, JPF reports a failure if an uncaught exception occurs or an assertion is violated. Another type of failure can occur due to a deadlock (defined as no remaining runnable thread being able to make progress, either due to waiting on a lock that is being held by another thread or waiting for a notification that never occurs). The set of built-in properties that JPF is able to check includes also the absence of data races. JPF reports a data race when multiple threads access the same memory location without synchronization and at least one of the accesses is write. To implement their own properties, users can add listeners to support, e. g., temporal-logic properties [6].

2.1 History of JPF

JPF started in 1999; it and its community evolved significantly in the time since, due to JPF being reimplemented and rewritten and eventually published under the Apache 2 License. We outline the key milestones here:

1999: First version of JPF, developed at the NASA Ames Research Center in the form of a translation from Java to Promela [7]. This first version was limited because regular model checkers like SPIN [3], which analyzes Promela models, cannot handle the dynamic creation of objects and threads (unless an upper bound is known at compile time).

2000: Reimplementation as a concrete virtual machine for Java bytecode that can backtrack execution [1]. JPF has used this approach since then.

2003: Introduction of extension interfaces and the architecture that modern JPF has until today.

2005: JPF was released as open-source software on Sourceforge, being the first NASA software project to be released in this way.

2008: First participation in Google Summer of Code, which supports students working on open-source software with stipends.

2009: JPF moved to its own server, hosting extension projects and the documentation (wiki).

2017: Moved to GitHub. This allowed JPF to implement continuous integration [8] and accept outside contributions more easily.

2.2 Related Work

JPF is an explicit-state model checker for Java bytecode at its core. It inspired similar works such as JNuke [9] and Moonwalker [10], which implement model checking for Java bytecode or .NET code without native methods, respectively.

Other tools that analyze programs by using their own execution engine include Valgrind [11], which looks for incorrect memory usage (corruption, leaks) in binary programs using dynamic analysis, and KLEE [12], which implements a symbolic execution engine on top of the LLVM [13] infrastructure.

Other software model-checking approaches are closer to the first version of JPF [7] and analyze code after transforming it into a representation that can express that entire state space at compile time. Examples include SLAM [14], which converts C code to a Boolean program for model checking, and CBMC [15], which uses a SAT solver on propositions derived from C code. This approach is more popular for analyzing C code because the inability to handle dynamic memory allocation and thread creation is less relevant for C programs where memory and thread usage are often bounded, especially for safety-critical systems [16].

Also, there exist several dynamic analysis frameworks that can be used to detect runtime errors by monitoring the execution of a program within a particular virtual machine. Notable examples include RoadRunner [17] and DiSL [18] for Java programs running on JVM, and SharpDetect [19] for C#/.NET programs. All these frameworks work on the same principle, recording specific events and the program runtime state on a dynamic execution trace, and forwarding this information to a custom analysis plugin that detects the actual errors. They are useful especially for multi-threaded programs and discovering possible concurrency errors (e. g., deadlocks and race conditions).

Table 1 gives an overview summary of the aforementioned tools by looking at their overall approach (state space exploration vs. runtime monitoring) and supported platforms. In this table, “state space exploration” can refer to model checking or symbolic execution. The table shows that related tools are too different (in terms of the principal approach or platform) for a straightforward quantitative comparison. In particular, we are not aware of any tool for Java bytecode with similar types of capabilities that JPF now has.

Table 1: Comparison of JPF with related tools by approach and target platform

		Platform			
		Java bytecode	x86 code	CIL bytecode	C source code
Approach	State space exploration	JNuke (Java 5) JPF (Java 11)	KLEE	Moonwalker	SLAM CBMC
	Runtime monitoring	RoadRunner DiSL	Valgrind	SharpDetect	

3 JPF’s Architecture

JPF’s architecture separates bytecode execution from the functionality of library classes, access to the underlying host JVM, and various ways of adapting and extending the built-in functionality.

3.1 Functionality

At its core, JPF implements a virtual machine for bytecode instructions. The entire state of a program (with the state of all its threads and the shared heap) is managed by that virtual machine. Unlike a regular virtual machine, JPF is capable of keeping copies of past program states and comparing them to other states. Past states can be restored from these copies, which allows JPF to implement a state space search of a program.

The state space search algorithm is configurable (depth-first search being the default setting) and by default explores all outcomes of all non-deterministic choices. This includes thread scheduling in case multiple threads are enabled at a given state and the outcomes of all possible values of a non-deterministic data choice. Such choices are either implemented through the `Verify` application programming interface (API) or extensions such as SPF [4] and typically model unspecified user inputs.

Sequences of instructions that do not exhibit any choices are executed inside a *transition* in JPF. A transition ends whenever a choice is hit during execution. Therefore, JPF computes the state space of a program *on the fly*, as the extent of a transition depends on the instructions therein and their side effects. Partial-order reductions optimize the state space and avoid unnecessary thread scheduling choices. JPF is close to being a sound verification tool in the sense that there are very few cases of property violations that it misses, but a few implementation choices in the state hashing and partial-order reductions are not sound in all cases, which makes JPF a bug-finding rather than a verification tool in the strict sense [2], unless the default behavior is changed so a fully exhaustive search is used, at the cost of being significantly slower in certain cases. Several extensions aiming to make JPF a sound verification tool have been already proposed, including the support for sound dynamic partial-order reduction [20] and coverage of all behaviors permitted by the Java memory model [21,22].

3.2 Design

JPF itself is written in Java and runs on the so-called *host JVM*, which provides internal functionality such as loading classes or interacting with the system via native methods [1,2]. The system under test is executed by JPF's virtual machine, which keeps track of any effects (such as changes to memory) of an executed instruction (see Figure 1).

The main functionality of JPF is implemented in `jpf-core`, while optional extensions can extend that functionality.

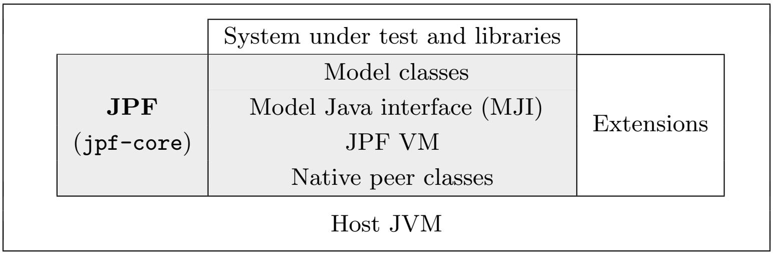


Fig. 1: Architecture of JPF

The functionality of JPF (`jpf-core`) itself is divided into modules (see Table 2). Module `main` implements the core analysis capability, while other modules (explained below) implement models of library classes (`classes`), a bridge to the underlying host JVM (`peers`), or auxiliary functionality.

Table 2: Main JPF modules and their purpose

Module	Purpose
`annotations`	Run-time annotations in programs analyzed by JPF
`classes`	Model library classes
`examples`	Small example programs
`main`	Core functionality (VM, state search, etc.)
`peers`	Native peers for accessing JPF from model classes
`tests`	Unit tests

Main. JPF has a very extensible design that allows developers to customize almost any functionality. The base implementation of the VM in `main` and its instruction set are generic, and while Java bytecode is the default concrete implementation, other instruction sets can be supported.

The extensibility of JPF is achieved by all key functionality being customizable through interfaces. We present the key interfaces below:

– A `SearchListener` can track events arising from the state space search (e. g., when program analysis starts or ends, when a new state is created, or when an existing already visited state is backtracked to).
– A `VMListener` gets notifications from program execution (e. g., when a method call begins or a certain type of instruction is executed).
– A `ChoiceGenerator` can override the way how non-deterministic events are explored or implement new types of choices.
– Instruction-related interfaces allow changing how sets of instructions or individual instructions are analyzed.
– A `Scheduler` has the capability of changing how the state space is analyzed, e. g., to analyze the state space of multiple processes [23]
– A `PublisherExtension` creates reports on program analysis results.

Classes. Any Java program has to access functions of the Java library; this starts by using the common super class `Object` as the first application-specific class is loaded. Many Java library classes have functionality that is not suitable for JPF's analysis, as they include functionality that incurs globally visible side effects (such as writing to a file) and use native methods. Native methods are not available as Java bytecode but instead implemented by a system-specific run-time library, usually written in C or C++. As JPF only interprets the application-level bytecode, it is not able to track the effects of native methods.

Therefore, classes using native methods have to be replaced by *model classes* (in `classes`), which represent a Java implementation of code that makes invisible side effects (through native method calls) in the regular library implementation visible to JPF at the model class level. In this way, model classes solve the problem of not being able to track the outcome of native code execution. Other approaches have been attempted, such as using process-level virtualization to track the state of an entire operation-system-level process. This approach is less efficient because it can only analyze the state of a process at a “black-box” level, preventing state compaction or partial-order reduction [24].

A model classes can completely replace the functionality of a library class, if it can implement this entirely in Java, making all (side-) effects visible to JPF. However, access to native methods requires the peers module (see below).

Annotations. This module implements annotations that are visible to JPF at run time. The most important annotations are `@MJI`, which marks a method as a bridge to a native peer, and attributes that affect the state space exploration by ignoring fields during program analysis (`@FilterField`) or marking them as not shared by multiple threads (`@NonShared`).

Peers. Model classes by themselves are limited to functionality that can be implemented directly through bytecode. Much functionality, such as printing to the console, requires access to the underlying run-time environment. *Native peers* bridge this gap between bytecode and native code (see Figure 2). At the model class level, native methods are declared normally (see Fig. 2b, line 12).

To delegate a native method call, JPF uses a so-called Model Java Interface (MJI) layer to specify the *native peer*, a JPF-level class implementing native methods. MJI methods handle parameters from bytecode at the JPF level and pass them in the appropriate form to an actual native function on the host JVM. A native peer usually accesses the host JVM and maps the state of the host JVM object to the JPF-level object and also manages potential side effects of the host JVM method (see Figure 1).

MJI classes and methods follow a name-mangling scheme to encode the package name as part of the class name, and the return type, method name, and parameter list of the underlying native method as part of the method name (see Fig. 2b, line 15). This way, JPF can identify the right method at run time.

Figure 2 shows how the different layers of JPF interact when executing code that prints “Hello, World!”. The bytecode first loads a reference to the PrintStream instance and the string “Hello, World!”, in order to call `println`.[7] This method (available entirely in bytecode through the Java library) constructs the correct final string with the newline character at the end and internally calls `OutputStreamWriter.write`. That method uses `OutputStreamWriter.encode` to convert the string to a byte array. Because the encoding of a string uses a native method, `encode` is declared as such in the model class. The JPF native peer counterpart is an MJI method, which internally accesses the native character-to-byte conversion functionality of the host JVM.

Generally, a native peer can delegate its functionality to the underlying native method for calls that have no side effects on the Java-level object [25], or it can implement its own logic and manage side effects in a complex way, such as when

[7] To save space, we elide Java package names (java/lang for System and java/util for PrintStream), instruction and constant pool offsets, and the “L” denoting a fully qualified class name.

```
getstatic       // Field System.out:PrintStream;
ldc             // String Hello, World!
invokevirtual   // Method PrintStream.println:(String;)V
```

(a) Bytecode to be executed

```
   /** Java library: java.io.PrintStream (provided by the JVM) */
   public void PrintStream.println(String s) {
     ...
     OutputStreamWriter.write(...);
 5 }

     /** JPF model class: java.io.OutputStreamWriter (in ”classes”) */
     public void write(String s, int off, int len) throws IOException {
       ... = encode(...);
10   }
     // native method declaration in the model class
     private native int encode (String s, int off, int len, byte[] buf);

     /** JPF native peer: JPF_java_io_OutputStreamWriter (in ”peers”) */
15   @MJI public int encode__Ljava_lang_String_2II_3B__I (MJIEnv env,
                         int objref, int sref, int off, int len, int bref) {
       ... // access to the host JVM
     }
```

(b) Model class interfacing with the native peer via MJI

Fig. 2: Interaction between code of the Java library (line 2), a model class with a native method (line 12), and its native peer (line 15).

synchronizing program states between an application that is analyzed by JPF and external applications that are connected through the network [26].

Tests. Tests contain over 1000 unit tests that check the internal functionality of JPF, ensuring that key Java language or library features work correctly. Some tests verify the verdict of a full program analysis on a small example.

Examples. A couple of small examples are also provided, along with their configuration files, to exemplify the usage of JPF.

3.3 JPF extensions

JPF *extensions* are modular implementations of enhancements to JPF. They are separate projects that are independent of the main part of JPF and implement additional functionality, e. g., by overriding how unspecified values of variables are interpreted or how an application interacts with its environment.

Notable extensions include symbolic execution [4], automated support of stateless native methods [25], and automated support of certain types of networked applications using either a centralization or a caching approach [23,26].

3.4 Program execution using JPF

When JPF is used, it typically is run with a configuration file that specifies the application under test. JPF then proceeds as follows:

1. The configuration file is parsed and extensions are loaded as specified.
2. The application main class and the necessary library classes are loaded. Program execution begins at `main`.[8] Execution covers the bytecode of the program under test, Java libraries (without native methods), and model classes.
3. Any instruction that creates a new thread, affects the state of another thread, or produces a non-deterministic choice for other reasons is handled by a `ChoiceGenerator`, which causes the current transition to end. New transitions are scheduled and added to the state space search.
4. Any time a model class declares a Model Java Interface method, execution is handled by the corresponding native peer method inside JPF. These methods are able to access (possibly native) methods of the underlying host JVM.
5. The analysis stops when JPF has explored the entire state space of a program, runs out of memory, or finds a property violation to report.[9]

3.5 Challenges in modeling Java library classes

The main challenges in writing a model class (and if needed, its native peer) are the following:

- A model class has to reflect the functionality of the original Java class faithfully; differences may result in an overapproximation of the behavior under JPF, or an unsound underapproximation.
- While a model class can hide side effects of native methods, these side effects are often an essential part of the program behavior, such as for networked applications [23,26]. When native methods interact with the environment, major changes in JPF are necessary to handle the side-effects of both the host JVM and its environment (the underlying operating system) [26,24].
- Because a native peer interacts with the host JVM, it often has additional state information compared to the model class. Care has to be taken that this additional state (which is used by native methods) remains consistent with the state of the model class (which is visible and used by non-native methods).
- It is not possible to create a model for only selected methods, so a model class has to support the entire public API that the program under test requires. Furthermore, it is often not possible to replace a single class in isolation, as multiple classes in the same package or even related packages (such as `java.io` and `java.net`) often interact and have to be replaced as an ensemble.
- The Java base classes (in module `java.base`) in Java 11 alone contain 190 native methods, so the effort of supporting all of them is prohibitive. JPF therefore focuses on the most commonly used native methods.

[8] More classes are loaded at run time as needed [27].

[9] One can specify that JPF continue the state space search after a property violation.

4 Adaptations in JPF for Java 11

Several changes in compiled Java code (bytecode) from Java 8 to Java 11 heavily affect runtime environments, including JPF, and even the build system. Here, we describe these challenges and our solutions to them.

4.1 Module system

Java 11 introduces a module system that provides an additional unit of encapsulation on top of Java packages [28]. Modules have to declare their dependencies explicitly and can also declare the services they provide. In Java 11, built-in modules are bundled in a new archive format (JMOD). Thus, JPF no longer reads the classes directly but delegates reading class files from these archives to the host JVM. To support the module layer, we extended JPF's class loader with new functionality to support the module API. In particular, the Proxy API (which is used for reflection) has to support module information in Java 11.

4.2 Bootstrap methods

Java 8 introduces support for dynamic languages and lambda expressions, which are functions that are not bound to an identifier. These functions cannot be fully resolved at compile time. Internally, they are compiled into so-called *bootstrap methods* that instantiate a valid anonymous function at class load time in order to accommodate concrete uses.

This change allows for optimized string handling: In Java 8 and prior, string output is handled by creating a *StringBuilder* instance and appending strings to its buffer. This requires expensive creations of intermediate objects and subsequent conversions of these objects to *String* instances for tasks as simple as adding a number to an output string. In Java 11, specialized bootstrap methods handle string output with non-string parameters much more efficiently.

Figure 3 illustrates this with a simple example. As can be seen, a simple string expression (Fig. 3a) compiles into complex bytecode (Fig. 3b) under Java 8. The resulting code produced by the Java 11 compiler is much more compact (Fig. 3c); however, most of the functionality is delegated to the bootstrap method *makeConcatWithConstants*, which takes an integer parameter and returns a string. While we cannot show the details here, one can see that parameter `i` is part of the bootstrap method, but the string constant "Number" is not a runtime parameter. The bootstrap method is expanded into a callable anonymous function (a *call site* [29]) at runtime by the class loader, which adds the string constant, before it can be called by the bytecode instruction `invokedynamic`.

When a class is loaded, the target of an *invokevirtual* instruction (a call to a dynamically generated method) has to resolve to a valid call site [29]. The call site is generated from the bootstrap method. The bootstrap method, e. g., `makeConcatWithConstants` in Fig. 3c, instantiates a complete call site by creating an anonymous function using a concrete value ("Number" in Fig. 3d) for the string constant.

```
public static void print(int i) {
  System.out.println("Number␣" + i);
}
```

(a) Source code

```
getstatic       // Field System.out:PrintStream;
new             // class StringBuilder
dup
invokespecial   // Method StringBuilder."<init>":()V
ldc             // String Number
invokevirtual   // Method StringBuilder.append:(String;)StringBuilder;
iload_0
invokevirtual   // Method StringBuilder.append:(I)StringBuilder;
invokevirtual   // Method StringBuilder.toString:()String;
invokevirtual   // Method PrintStream.println:(String;)V
return
```

(b) Compilation with Java 8

```
getstatic       // Field System.out:PrintStream;
iload_0
invokedynamic   // makeConcatWithConstants:(I)String;
invokevirtual   // Method PrintStream.println:(String;)V
return
```

(c) Compilation with Java 11

```
BootstrapMethods:
  0: #19 REF_invokeStatic StringConcatFactory.makeConcatWithConstants:
  (MethodHandles$Lookup;String;MethodType;String;[Object;)CallSite;
    Method arguments:
      #20 Number \u0001
```

(d) Bootstrap method structure

Fig. 3: String handling and output under Java 8 and Java 11

Lambda expressions are also handled internally via bootstrap methods. Some lambda expressions are serializable, in particular in `java.util.Comparator`, which is often used for sorting.

4.3 Bootstrap methods in JPF

JPF for Java 8 had limited support for bootstrap methods, handling a few common cases. Due to the more widespread use of lambda expressions in Java 11 (especially for string concatenation and output), shortcomings in the earlier implementations had to be addressed. The internal implementation of OpenJDK for bootstrap method resolution generates the bytecode of the call site at load time. This is complex and involves internal APIs and native calls that JPF does not support. JPF instead models the behavior of the bootstrap method and implements its own support of `invokedynamic` for string concatenation.

When used to concatenate strings, instruction `invokedynamic` calls a function that takes arguments to be concatenated and returns the resulting `String` instance. JPF can easily implement this at VM level if all the arguments are `String` instances or primitive types, since their string representation could be easily constructed from their meta-data stored at VM runtime. However, for other reference type arguments, their `toString()` methods have to be called. This method call no longer happens in the form of a method call in the bytecode but is done automatically by the JVM. JPF mimics this behavior to support Java 11 string expressions by analyzing all arguments on the operand stack to convert them to a string if needed. Our approach therefore avoids the complex dynamic bytecode generation of OpenJDK 11.

To support serializable lambda expressions, OpenJDK uses a special bootstrap method called `altMetafactory`. As JPF cannot use that mechanism, we need another approach to handle serializable lambda expressions: JPF creates an object that implements the target interface of a lambda expression, which happens to make serialization easier — we only need to add `Serializable` to this object's implemented interface list, and the object serialization mechanism can then serialize the lambda expression automatically.

4.4 Reflection

One of the primary goals of adding modules in Java 11 was strong encapsulation, which attempts to limit reflection and promote the modular approach to improve security. As a consequence of this, the reflection API no longer permits access to private fields without issuing a warning (as of Java 11); in later releases, such access is denied by default [30] or removed entirely [31]. Under Java 11, various tests raised warnings due to illegal reflective access taking place. The problem was caused by the executing code using reflection, trying to access non-public fields/methods residing in different modules.

To fix the problem, it needs to be ensured that the accessing code present in the module has access rights to the module it is trying to access the field/method from. Specifying the `--add-opens` option to `jvmArgs` within `build.gradle` for the necessary packages fixes the problem, ensuring that JPF will not break due to illegal-reflective access errors when using newer versions of Java.

4.5 Internal APIs

Many implementations of the Java API (`java.*` packages) use internal APIs, such as `com.sun.*` and `jdk.internal.*`, in their implementation. Such internal APIs are often highly dependent on the native methods and the underlying JVM.

After Java 8, many such internal packages have been replaced with new packages under `jdk.internal`. Replacement for existing functionality is usually available under a redesigned API, such as `java.lang.StackWalker`, which provides more flexible and efficient stack traversal functionalities, like lazy traversal, frame filtering, and criteria for stopping.

The most straightforward strategy to minimize dependency on internal APIs is to use a model class, which replaces the problematic class entirely. However, providing an accurate model class is a challenge of its own. For example, JPF for Java 8 uses a model class for `DateFormat`, providing a simplified implementation of key date methods. The changes in Java 11 also affected date handling and caused four regression tests to fail. The increased complexity of the Java 11 implementation made it difficult to maintain a model class for such functionality.

To enforce strong encapsulation of JDK internals, any JPF constructs using internal data have been rewritten by leveraging model classes and Model Java Interface (MJI) components to intercept method invocations and delegate them to dedicated classes. Notable updates include the `StackFrameInfo` model class that provides support for the *StackWalker* API and the `ServiceLoader` model class that ensures support for the *DateFormat* API. Additionally, the `MethodHandles` mechanism is used to support *CountDownLatch*, *ExecutorService*, and *Semaphore*, which are utility classes to support concurrent programming. We also added support for the `PlatformClassLoader` class and improved `SecureClassLoader` to help with service provider loading.

Many of these changes (such as the `ServiceLoader` model class and the `MethodHandles` mechanism) lift the layer at which a model class is used to a higher level by supporting general delegation mechanisms in the Java library better. This reduces the number of model classes needed and makes it easier to fully support internal functionality, especially when it has few dependencies on native methods.

4.6 Build system

Since 2018, JPF has been using *Gradle* as the build automation tool of choice. For a user, Gradle has the advantage of automatically downloading any dependencies, as long as the available JVM used to run Gradle supports them.

This results in multiple dependencies: The version of Java used has to be able to both run Gradle and compile and execute the software being built and tested. Therefore, targeting a different Java version often requires updating Gradle as well. This ensures the ability to support newer versions of Java and benefit from other improvements in Gradle.

However, these Gradle updates often include major changes that deprecate an old build mechanism or even change Gradle's domain-specific language that is used to declare build tasks. Therefore, major Gradle updates may require a redesign of some build tasks. Deprecations of certain features also usually have an effect with the next major version.

The JPF project has made two major updates in using Gradle: from version 4.7 to version 5.4.1 in 2019, and again to version 8.2.1 in 2023. We eliminated any use of deprecated features to allow at least one more major version upgrade without changes in the future. By redefining tasks using the task configuration avoidance API, which deters creation of unnecessary tasks, the build process became more efficient. We also now use plugins to support publishing to a local Maven repository and measure statement coverage (using JaCoCo).

4.7 Other adaptations

Various other internal changes or defects that were discovered during development for Java 11 support required corresponding adaptations in JPF:

- Support for new internal string representation (JEP 254 [32]), which allows strings in memory to be encoded as either LATIN1 or UTF16. The content of a string is now stored as an array of bytes rather than an array of characters.
- Removal of unnecessary explicit manual boxing of primitive values in objects; adaptations to changes in the unboxing API of primitive classes (`valueOf`).
- Removal of string buffer-related model classes (StringBuilder, StringBuffer) that are no longer needed, because their functionality can be safely delegated to the library of the host JVM.
- Support for the stream API.
- Better support for loading classes from JAR files and URLs.
- Better support for correct type handling.
- Various other fixes (file I/O, internals of threads and concurrency packages).
- Updates in the documentation and renaming of the git branches to make Java 11 the default Java version for JPF.

5 Other Enhancements

Other enhancements that have been included with the Java 11 support include test pollution detection and bit-flip simulation, which are both compact enough to be included in `jpf-core` rather than as an extension of their own.

5.1 Test pollution detection

Flaky tests are software tests that non-deterministically pass or fail. They undermine developers' trust in the test infrastructure, and waste many man-hours to investigate non-existing bugs. Order-dependent flaky tests are a prominent type of flaky tests [33], where polluter tests, the kind of software tests that modify the program state shared among other tests, cause other tests to fail. It is therefore worthwhile to proactively detect test pollution and prevent order-dependent flakiness. A technique dubbed PolDet was developed to detect polluter tests [34].

Because JPF provides infrastructure support for detecting other potential issues in software tests, such as race conditions and deadlock, PolDet is re-implemented in JPF (PolDet@JPF) [35] to combine the capability of PolDet and JPF. PolDet@JPF captures and compares the states before and after the test execution with JPF's state serialization mechanism. Its implementation is only about 200 lines of code on top of JPF but can detect 26 polluter tests existing in 13 Java projects. This demonstrates JPF's versatility for rapid prototyping of various software tools in research.

5.2 Systematic Bit-Flip Fault Injection and Exploration

Computer hardware is susceptible to errors. Hardware defects or radiation can induce errors to the hardware, which can result in a memory bit being flipped. With the increasing complexity of computer hardware according to Moore's law [36], bit flips become more likely [37], making it important to improve the resiliency

```
public static void foo(@BitFlip int n) {
  System.out.println(n);
}
```

Fig. 4: An annotation triggering a bit-flip analysis for the method parameter

of software against hardware errors. However, these hardware errors are nondeterministic and hard to reproduce. To evaluate software's resiliency against bit flips, fault injection [38] can simulate the outcome of such events at the software level.

To support such fault injection, we use JPF to systematically inject and explore bit-flip faults in the user specified variables in Java programs. Specifically, the users can specify a list of variables and for each variable v_i specify k_i, the number of bits to flip in it. Considering that any k_i bits of v_i are possible to be flipped in real hardware faults, we let JPF execute the program in all possible ways to systematically evaluate programs' resiliency to bit-flip faults. For a simple example, consider the code in Figure 4. We want to know what happens if some error causes a bit to be flipped in the argument to method `foo`. Since `n` is of type `int`, our implementation explores all the 32 cases in which bit is flipped, so the expected output of calling `foo(0)` is `1 2 4...-2147483648`.

Our implementation provides a `BitFlipListener`, a JPF listener that monitors the list of user-specified variables and performs bit-wise fault injection before the relevant instructions execute. Specifically, we support injecting bit-flip faults to three kinds of variables, (1) static and instance fields, (2) method arguments, and (3) local variables, whose type can be any primitive data types. For the fields and local variables, the bit flips are injected when they are written by the programs. For the method arguments, the bit flips are injected when the method is invoked and the arguments are assigned. `BitFlipListener` registers a Choice Generator to inject all possible bit flips to the corresponding operand in the operand stack before the store/write instruction or the invoke instruction, depending on the variable type.

A user can specify the variables to flip in three ways: (1) calling `getBitFlip` API in the application code, (2) adding `@BitFlip` annotation to the variables, and (3) specifying in the command line arguments without changing the application code. For example, in Figure 4, we can (1) add `n=getBitFlip(n,1)` at the beginning of the method `foo`, (2) annotate `n` with `@BitFlip(1)` (where `(1)` can be omitted because $k = 1$ by default), or (3) specify bit-flip fault injection in method `foo`, parameter `n`, and $k = 1$ in the command line arguments.

Our implementation is based on JPF's `Verify.getInt`, which generates all possible integer values in a given range. The `BitFlipListener` parses the annotations and the command line arguments and adds the specified variables to a watch list.

A key challenge in the implementation is that JPF cannot register several choice generators at the same point of the application code. We resolve this issue by registering only one choice generator even when the number of bits to flip, k, in a variable is $k > 1$. Specifically, we register only one `IntIntervalGenerator` that produces an integer m in range $[0, \binom{n}{k})$ where n is the number of bits of the variable, and then decode the integer using binomial coefficients to get the set of k in n bits to flip. The decoding process is as follows: Because $\binom{n-1}{k}$ out of $\binom{n}{k}$ combinations do not select the n^{th} bit, if $m > \binom{n-1}{k}$, we select the n^{th} bit, let $m' = m - \binom{n-1}{k}$ and $k' = k - 1$; otherwise, we do not select the n^{th} bit and let $m' = m, k' = k$. If we then let $n' = n - 1$, with the same process on n', m', k', we can decode out the set of k' in the remaining n' bits, and then recursively get all the k bits to flip.

We implemented a JPF regression test class that checks our injection engine in various scenarios and documents the basic usage. It verifies all bits are flipped exactly once in a variable using a global counter at JPF level. Besides, we used our tool to check the resiliency of Cyclic Redundancy Check (CRC) and International Standard Book Number (ISBN) algorithms against bit-flip faults. We confirmed that both algorithms can detect all one-bit flips, but cannot detect all two-bit flips, as expected. Our implementation has been included in JPF.[10]

6 Project Evolution and Evaluation

The evaluation of the validity of JPF is based on automated unit tests, which execute key features of JPF. JPF-level unit tests internally run small applications using JPF's analysis engine, thus effectively implementing system tests [39]. Our experimental evaluation is therefore based on these unit tests, which grew from 864 tests, for the code base supporting Java 8, to 1002 tests at the time of writing.

Figure 5 shows how the code base grew in the last five years, to accommodate for Java 11 functionality.[11] We sometimes observe sudden code size increases and drops in failing tests, which is usually because larger amounts of work had been merged into the Java 11 development branch at that time.

The first large decrease in failing tests was thanks to preliminary support for Java 11 string handling; the final large code growth was from incorporating patches and tests from the mainline development (for Java 8) into Java 11.

The graph shows that while we also had contributions at other times of the year, virtual summer internships supported by Google Summer of Code were a major factor in the contributions, as it was possible to carry out development that was not directly tied to a short-term research goal.

7 Conclusions

This paper has provided a detailed account of JPF's current architecture. JPF's modular design separates bytecode execution from code that models key library

[10] `https://github.com/javapathfinder/jpf-core/pull/295`

[11] JPF used to be handled as a Mercurial repository, and older versions of Mercurial were too slow to handle a large project history, so the project history was purged in 2018 by importing the entire code base as an initial commit in a new git repository.

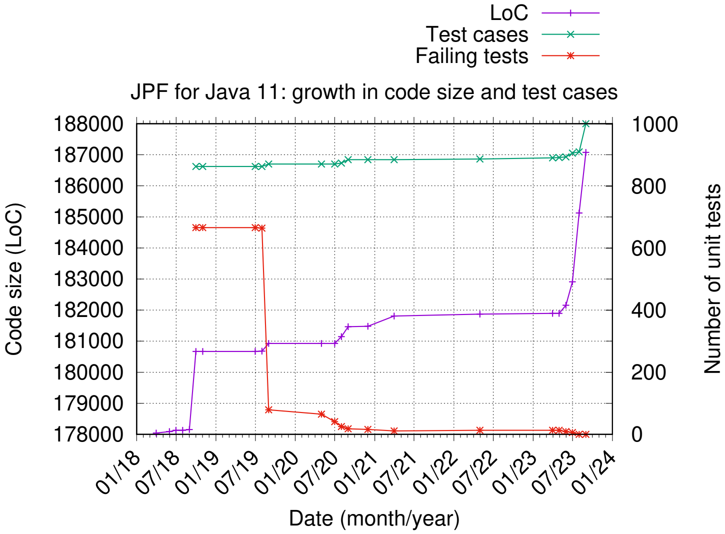


Fig. 5: Evolution of Java 11 development from 2018–2023.

functions and code that interfaces with the underlying run-time environment. Thanks to a very extensible design, JPF's functionality can be modified from concrete to symbolic execution or from a single application to multiple processes, which can even be networked.

Most of the development effort of the last five years was focused on supporting Java 11. Its major changes required corresponding adaptations in JPF: The new modular library system required extensions in the class loader; a different compilation of string expressions to bootstrap methods required support for them; and internal API changes brought both simplifications in the code base (because some model classes could be dropped) as well as complications, because new types of interfaces to internal VM data structures had to be supported.

Future work includes ensuring seamless Java 11 support for JPF's extensions and support for the next stable Java release (Java 17).

Acknowledgments

We would like to thank Alexander Kohan, Dan Smith, Gayan Weerakutti, Jeanderson Barros Candido, John Toman, Malte Mues, Nastaran Shafiei, Quoc-Sang Phan, Vaibhav Sharma, Wen Zhang, Willem Visser, Amgad Rady, and Yuvaraj Anbarasan for their contributions to supporting Java 11.

We would also like to thank the Google Summer of Code program for their support and Soha Hussein and other JPF organization administrators.

This work was partially supported also by the Czech Science Foundation project 23-06506S.

References

1. Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
2. Cyrille Artho and Willem Visser. Java Pathfinder at SV-COMP 2019 (competition contribution). In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 224–228, Cham, 2019. Springer International Publishing.
3. G. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2004.
4. Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 179–180, 2010.
5. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Language Specification, Java SE 11 Edition.* Oracle, 2018.
6. Matt Walker, Parssa Khazra, Anto Nanah Ji, Hongru Wang, and Franck van Breugel. jpf-logic: a framework for checking temporal logic properties of Java code. *ACM SIGSOFT Software Engineering Notes*, 48(1):32–36, 2023.
7. Klaus Havelund. Java PathFinder, a translator from Java to Promela. In *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops Trento, Italy, July 5, 1999 Toulouse, France, September 21 and 24, 1999 Proceedings 5*, pages 152–152. Springer, 1999.
8. Martin Fowler and Matthew Foemmel. Continuous integration. `http://www.martinfowler.com/articles/continuousIntegration.html`, 2006.
9. Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller. JNuke: Efficient dynamic analysis for Java. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 462–465, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
10. Niels HM Aan de Brugh, Viet Yen Nguyen, and Theo C Ruys. Moonwalker: Verification of .NET programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*, pages 170–173. Springer, 2009.
11. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
12. Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
13. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
14. Thomas Ball and Sriram K Rajamani. The SLAM toolkit. In *Proceedings of CAV 2001 (13th Conference on Computer Aided Verification)*, volume 2102, pages 260–264, 2000.
15. Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, pages 389–391. Springer, 2014.

16. Les Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 46(7):465–472, 2004.
17. Cormac Flanagan and Stephen N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In Sorin Lerner and Atanas Rountev, editors, *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010*, pages 1–8. ACM, 2010.
18. Lukas Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Petr Tuma, and Zhengwei Qi. Java bytecode instrumentation made easy: The DiSL framework for dynamic program analysis. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, volume 7705 of *Lecture Notes in Computer Science*, pages 256–263. Springer, 2012.
19. Andrej Čizmárik and Pavel Parízek. SharpDetect: Dynamic analysis framework for C#/.NET programs. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2020.
20. Kyle Storey, Eric Mercer, and Pavel Parizek. A sound dynamic partial order reduction engine for Java Pathfinder. *ACM SIGSOFT Software Engineering Notes*, 44(4):15–15, 2021.
21. Jeremy Manson, William Pugh, and Sarita V Adve. The Java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, 2005.
22. Huafeng Jin, Tuba Yavuz-Kahveci, and Beverly A Sanders. Java memory model-aware model checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings 18*, pages 220–236. Springer, 2012.
23. Nastaran Shafiei and Peter Mehlitz. Extending JPF to verify distributed systems. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.
24. Cyrille Artho, Kuniyasu Suzaki, Masami Hagiya, Watcharin Leungwattanakit, Richard Potter, Eric Platon, Yoshinori Tanabe, Franz Weitl, and Mitsuharu Yamamoto. Using checkpointing and virtualization for fault injection. *IJNC*, 5(2):347–372, 2015.
25. Nastaran Shafiei and Franck van Breugel. Automatic handling of native methods in Java PathFinder. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, pages 97–100, 2014.
26. Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, Mitsuharu Yamamoto, and Koichi Takahashi. Modular software model checking for distributed systems. *IEEE Transactions on Software Engineering*, 40(5):483–501, 2013.
27. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification.* Addison-Wesley, 2013.
28. Alan Bateman, Alex Buckley, Jonathon Gibbons, and Mark Reinhold. JEP 261: The module system. `https://openjdk.org/jeps/261`, 2014.
29. Oracle. Class CallSite. `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/CallSite.html`, 2018.
30. Alex Buckley and Mark Reinhold. JEP 396: Strongly encapsulate JDK internals by default. `https://openjdk.org/jeps/396`, 2020.
31. Alex Buckley and Mark Reinhold. JEP 403: Strongly encapsulate JDK internals. `https://openjdk.org/jeps/403`, 2021.

32. Brent Christian and Xueming Shen. JEP 254: Compact strings. `https://openjdk.org/jeps/254`, 2014.
33. Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 312–322, 2019.
34. Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 223–233, New York, NY, USA, 2015. Association for Computing Machinery.
35. Pu Yi, Anjiang Wei, Wing Lam, Tao Xie, and Darko Marinov. Finding polluter tests using Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 46(3):37–41, 2021.
36. Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.
37. Laszlo B Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149, 2002.
38. M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.
39. The JPF Team. Writing JPF tests. `https://github.com/javapathfinder/jpf-core/wiki/Writing-JPF-tests`, 2023.

Hitching a Ride to a Lasso: Massively Parallel On-The-Fly LTL Model Checking

Muhammad Osama[1,2]⋆ and Anton Wijs[2](✉)

[1] Leiden Institute of Advanced Computer Science (LIACS), Leiden University,
Leiden, The Netherlands
`m.o.mahmoud@liacs.leidenuniv.nl`
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
`a.j.wijs@tue.nl`

Abstract. The need for massively parallel algorithms, suitable to exploit the computational power of hardware such as graphics processing units, is ever increasing. In this paper, we propose a new algorithm for the on-the-fly verification of Linear-Time Temporal Logic (LTL) formulae [45] that is aimed at running on such devices. We prove its correctness and termination guarantee, and experimentally compare a GPU implementation with state-of-the-art LTL model checkers. Our new GPU LTL-checking algorithm is up to 150× faster on proving the correctness of a system than LTSmin running on a 32-core high-end CPU, and is more economic in using the available memory.

Keywords: Temporal logic, LTL model checking, automata-based verification, finite-state machines, GPU.

1 Introduction

With hardware developments increasingly focussing on parallel computing capabilities, the need for massively parallel algorithms, in which thousands of threads contribute to a computation, continues to grow [34]. In the last decade, we successfully deployed Graphics Processing Units (GPUs) to accelerate various computations relevant for computer-aided verification [36–40, 42, 43, 50–52, 54, 56, 57, 60, 61]. Being computationally heavy, this applies in particular to model checking [1]. Verifying whether a system model satisfies a given *Linear-Time Temporal Logic* [45] (LTL) formula is usually done with algorithms employing Depth-First Search (DFS) [8, 15, 22, 27, 33], as it involves the detection of cycles in the state space of the model, and DFS is very suitable for this. Unfortunately, DFS is not suitable as a basis for massive parallelism; for single and multi-core platforms, using in the order of tens of threads to run the same number of DFSs in parallel still works, but maintaining *thousands* of call stacks is not practical.

Algorithms based on Breadth-First Search (BFS) are more promising for massive parallelism [3, 11, 13, 25]. However, so far, these have only been applied

⋆ This work is supported by NWO grant OCENW.M.21.061 for the GAP project and by an Amazon Research Award (Fall 2021).

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 23–43, 2024.
https://doi.org/10.1007/978-3-031-57249-4_2

after the state space has been explored [2, 28], while a strong point of some LTL algorithms is their ability to detect counter-examples *on-the-fly*, i.e., while the state space, implicitly described by the system model, is being explored. This allows for early termination once a counter-example has been detected. The recent emergence of GPU-based state space exploration engines [16, 49, 51, 52, 54, 57, 59] has provided an important step for massively parallel on-the-fly LTL model checking, but high-performant BFS-based on-the-fly cycle detection is highly non-trivial. So far, only [53] has addressed this, but it relies on the piggyback algorithm [23], which only guarantees finding cycles up to a predefined length.

In this paper, we propose the Hitchhiking algorithm for massively parallel on-the-fly LTL model checking. It is based on the Map algorithm [11]. We present the algorithm, prove its correctness and that it is guaranteed to terminate, and discuss our experimental results, obtained by comparing the performance of an implementation of this algorithm for GPUs with state-of-the-art CPU-based algorithms.

Section 2 contains the necessary background. In Section 3, Map is discussed, followed by Hitchhiking. The implementation of Hitchhiking together with the experimental results are presented in Section 4. Related work is discussed in Section 5, and finally, conclusions are drawn in Section 6.

2 Preliminaries

The semantics of a (concurrent) system is represented by a *Kripke structure.*

Definition 1 (Kripke structure). *A* Kripke structure *over a set of atomic propositions AP is a 4-tuple $K = (\mathcal{S}, \mathcal{S}_0, \rightarrow, \lambda)$, where*

- *$\mathcal{S}$ is a finite set of states;*
- *$\mathcal{S}_0 \subseteq \mathcal{S}$ is the set of initial states;*
- *$\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation, such that $\rightarrow$ is* left-total, *i.e., $\forall s \in \mathcal{S}. \exists s' \in \mathcal{S}.(s, s') \in \rightarrow$;*
- *$\lambda : \mathcal{S} \rightarrow 2^{AP}$ is an interpretation (or labelling) function that maps each state to its set of valid atomic propositions.*

With $s \rightarrow s'$, we denote that $(s, s') \in \rightarrow$.

Definition 2 (Path). *Given a Kripke structure $K = (\mathcal{S}, \mathcal{S}_0, \rightarrow, \lambda)$, a* path *in K is an infinite sequence of states $\pi_K = s_0 s_1 s_2 \ldots$, such that $s_0 \in \mathcal{S}_0$ and $\forall i \geq 0. s_i \rightarrow s_{i+1}$.*

Sometimes, we interpret a path $\pi_K = s_0 s_1 \ldots$ as a set $\{s_0, s_1, \ldots\}$, for instance to reason about membership, e.g., $s \in \pi_K$.

Functional properties of the paths of a Kripke structure can be formalised by means of a *temporal logic* such as *Linear-Time Temporal Logic* (LTL) [45]. We refrain from defining the syntax and semantics of LTL. What is relevant for this paper is that verifying whether a given Kripke structure K satisfies a given

LTL formula φ can be done via the *automata-based* method [47] that uses a *Non-deterministic Büchi Automaton* [12] (NBA) $B_{\neg\varphi}$ derived from the negation of φ. To explain this, we first introduce the relevant notions of an NBA and NBA path.

Definition 3 (Non-deterministic Büchi Automaton). *A* Non-deterministic Büchi Automaton *(NBA) is a 5-tuple* $B = (\mathcal{Q}, \Sigma, \hookrightarrow, \mathcal{Q}_0, \mathcal{Q}_F)$*, where*

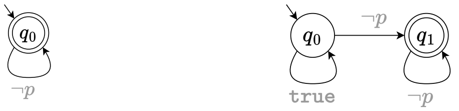
- $\mathcal{Q}$ *is a finite set of states;*
- Σ *is an alphabet;*
- $\hookrightarrow \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ *is a transition relation;*
- $\mathcal{Q}_0 \subseteq \mathcal{Q}$ *is the set of initial states;*
- $\mathcal{Q}_F \subseteq \mathcal{Q}$ *is the set of* accepting *states.*

Definition 4 (NBA path). *Given an NBA* $B = (\mathcal{Q}, \Sigma, \hookrightarrow, \mathcal{Q}_0, \mathcal{Q}_F)$*, a* path *in B is an infinite sequence of states* $\pi_B = q_0 q_1 q_2 \ldots$*, such that* $q_0 \in \mathcal{Q}_0$*,* $\forall i \geq 0. q_i \hookrightarrow q_{i+1}$*,* $\exists q_i \in \pi_B . q_i \in \mathcal{Q}_F$*, and* $\forall q_i \in \pi_B \cap \mathcal{Q}_F . \exists j > i . q_j \in \mathcal{Q}_F$*.*

In Def. 2, the latter two constraints refer to a path being *accepting*: an infinite number of states in the path are accepting states.

The labels of NBA transitions are usually written as propositional logic formulae. For a Kripke structure $K = (\mathcal{S}, \mathcal{S}_0, \rightarrow, \lambda)$ and a state $s \in \mathcal{S}$, we denote with $K, s \models \phi$ that s satisfies the propositional logic formula ϕ. The semantics of $K, s \models \phi$ is defined as follows, with ϕ_1, ϕ_2 propositional logic formulae:

$$
\begin{array}{llll}
K, s \models \mathsf{true} & & K, s \models \neg\phi & \Leftrightarrow K, s \not\models \phi \\
K, s \models p & \Leftrightarrow p \in \lambda(s) & K, s \models \phi_1 \vee \phi_2 & \Leftrightarrow K, s \models \phi_1 \text{ or } K, s \models \phi_2
\end{array}
$$



(a) Not eventually p. (b) Not globally eventually p.

Fig. 1: Example NBAs.

For every LTL formula φ, it is possible to construct an NBA $B_{\neg\varphi}$ that accepts exactly all paths of K that do not satisfy φ. Fig. 1 presents some example NBAs, with the accepting state having a double border, and the initial states having an incoming transition without a source state. Fig. 1a shows an NBA with a single path in which $\neg p$ holds globally, i.e., p is never eventually satisfied, and Fig. 1b contains an NBA for the negation of "globally eventually p": its paths eventually lead to $\neg p$ globally holding.

With $B_{\neg\varphi}$, LTL model checking can be performed by solving the *emptiness problem*: a Kripke structure K satisfies LTL formula φ iff the *product* of K and $B_{\neg\varphi}$ is empty, i.e. has no (accepting) paths. This product is defined as follows:

Definition 5 (Product of Kripke structure and NBA). *Given a Kripke structure $K = (\mathcal{S}, \mathcal{S}_0, \rightarrow, \lambda)$ and an NBA $B = (\mathcal{Q}, \Sigma, \hookrightarrow, \mathcal{Q}_0, \mathcal{Q}_F)$. The* product *of K and B is an NBA $K \otimes B = (\mathcal{Q}_\otimes, \Sigma_\otimes, \hookrightarrow_\otimes, \mathcal{Q}_{0,\otimes}, \mathcal{Q}_{F,\otimes})$, with*

- *$\mathcal{Q}_\otimes = \{\langle s, q\rangle \mid s \in \mathcal{S} \land q \in \mathcal{Q}\}$;*
- *$\Sigma_\otimes = \Sigma$;*
- *$\hookrightarrow_\otimes$ is the smallest relation satisfying the rule*

$$\frac{s \rightarrow s' \quad q \overset{\phi}{\hookrightarrow} q' \quad K, s' \models \phi}{\langle s, q\rangle \overset{\phi}{\hookrightarrow}_\otimes \langle s', q'\rangle}$$

- *$\mathcal{Q}_{0,\otimes} = \{\langle s, q\rangle \in \mathcal{Q}_\otimes \mid s \in \mathcal{S}_0 \land \exists q_0 \in \mathcal{Q}_0 . q_0 \overset{\phi}{\hookrightarrow} q \land K, s \models \phi\}$;*
- *$\mathcal{Q}_{F,\otimes} = \{\langle s, q\rangle \in \mathcal{Q}_\otimes \mid q \in \mathcal{Q}_F\}$.*

With $\mathsf{succ}_\otimes(\langle s, q\rangle)$, we refer to the set of *successors* of $\langle s, q\rangle \in \mathcal{Q}_\otimes$ according to the transition relation, i.e., $\mathsf{succ}_\otimes(\langle s, q\rangle) = \{\langle s', q'\rangle \mid (\langle s, q\rangle, \langle s', q'\rangle) \in \hookrightarrow_\otimes\}$.

When it is not needed to reason about s and q of a state $\langle s, q\rangle \in \mathcal{Q}_\otimes$ individually, we refer to states in $\mathcal{Q}_\otimes$ with $\bar{q}$, $\bar{q}'$, etc. The transitive closure of $\hookrightarrow_\otimes$ is denoted by $\hookrightarrow_\otimes^+$, i.e., $\bar{q} \hookrightarrow_\otimes^+ \bar{q}'$ denotes that there exists a finite sequence of states $\bar{q}_0 \bar{q}_1 \ldots \bar{q}_n$ with $n > 0$, $\bar{q}_0 = \bar{q}$, $\bar{q}_n = \bar{q}'$, and for all $0 \leq i < n$, $\bar{q}_i \hookrightarrow_\otimes \bar{q}_{i+1}$.

On the one hand, paths in $K \otimes B_{\neg\varphi}$ are by definition infinite, but $K \otimes B_{\neg\varphi}$ has a finite number of states. Therefore, a path in $K \otimes B_{\neg\varphi}$ actually consists of a finite sequence of states from an initial state $\bar{q}_0 \in \mathcal{Q}_{0,\otimes}$ to a state $\bar{q} \in \mathcal{Q}_\otimes$ and an *accepting cycle* including $\bar{q}$, i.e., a finite sequence of states $\sigma = q_0 q_1 \ldots q_n$, with $\bar{q} \in \sigma$, $q_0 = q_n$, and for at least one i, with $0 \leq i \leq n$, it holds that $q_i \in \mathcal{Q}_F$. This means that a path in $K \otimes B_{\neg\varphi}$ traverses a *lasso* in the graph structure.

3 The Hitchhiking Algorithm

3.1 Map: the Basis of Hitchhiking

The algorithm we propose is based on the Map algorithm, introduced in [11] and adapted for use on many-core systems such as GPUs in [2]. The main idea behind this algorithm is as follows: we use a strictly total order $>$ for the states in $\mathcal{Q}_{F,\otimes}$, and define the *maximal accepting predecessor* of a state $\bar{q} \in \mathcal{Q}_\otimes$ as $\mathtt{map}(\bar{q}) = \mathtt{max}(\{\bar{q}' \in \mathcal{Q}_{F,\otimes} \mid \bar{q}' \hookrightarrow_\otimes^+ \bar{q}\})$, with $\mathtt{max}$ returning the maximal state according to the order defined by $>$. Now, if there exists a state $\bar{q} \in \mathcal{Q}_{F,\otimes}$ with $\mathtt{map}(\bar{q}) = \bar{q}$, then it is in an accepting cycle. The reverse, however, is not true: if $\mathtt{map}(\bar{q}) \neq \bar{q}$, it does not mean that $\bar{q}$ is not in an accepting cycle. To overcome this, Map computes a function $\mathtt{p}$ that defines for every state in $\mathcal{Q}_\otimes$ its maximal accepting predecessor w.r.t. an evolving subset of $\mathcal{Q}_{F,\otimes}$.

Alg. 1 presents the many-core Map algorithm of [2]. As there were no state space exploration engines running on GPUs at the time, the Map algorithm was applied after the state space had been generated, i.e., post-exploration. In Alg. 1, the grey area highlights the many-core part, for instance to be executed on a GPU, with the loops indicating the parallelism, while the *explore* function (l.1, i.e., line 1) identifies all the states reachable from $\mathcal{Q}_{0,\otimes}$ via $\hookrightarrow_\otimes$. Initially,

Algorithm 1: The parallel post-exploration MAP algorithm [2].

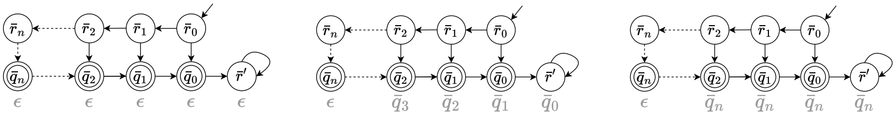
```
1  function explore($\mathcal{Q}_{0,\otimes}$, $\mathsf{succ}_\otimes()$):
2      $\mathcal{O} \leftarrow \mathcal{Q}_{0,\otimes}$; $\mathcal{Q}_\otimes \leftarrow \mathcal{O}$
3      forall $\bar{q} \in \mathcal{O}$ do
4          $\mathcal{O} \leftarrow \mathcal{O} \setminus \{\bar{q}\}$
5          forall $\bar{q}' \in \mathsf{succ}_\otimes(\bar{q})$ do
6              if $\bar{q}' \notin \mathcal{Q}_\otimes$ then  $\mathcal{Q}_\otimes \leftarrow \mathcal{Q}_\otimes \cup \{\bar{q}'\}$; $\mathcal{O} \leftarrow \mathcal{O} \cup \{\bar{q}'\}$
7      MAP($\mathcal{Q}_\otimes$, $\hookrightarrow_\otimes$, $\mathcal{Q}_{0,\otimes}$)
8  function MAP($\mathcal{Q}_\otimes$, $\hookrightarrow_\otimes$, $\mathcal{Q}_{0,\otimes}$):
9      $\mathcal{A} \leftarrow \mathcal{Q}_{F,\otimes} = \{\langle s, q\rangle \in \mathcal{Q}_\otimes \mid q \in \mathcal{Q}_F\}$
10     while $\mathcal{A} \neq \emptyset$ do
11         $\mathtt{p} \leftarrow$ update-$\mathtt{p}$($\mathcal{Q}_{0,\otimes}$, $\hookrightarrow_\otimes$, $\mathcal{A}$)
12         forall $\bar{q} \in \mathcal{Q}_\otimes$ do in parallel
13             if $\bar{q} = \mathtt{p}(\bar{q})$ then  report counter-example found
14             else  $\mathcal{A} \leftarrow \mathcal{A} \setminus \{\mathtt{p}(\bar{q})\}$
15     report no counter-example found
16 function update-p($\mathcal{Q}_\otimes$, $\hookrightarrow_\otimes$, $\mathcal{A}$):
17     forall $\bar{q} \in \mathcal{Q}_\otimes$ do in parallel  $\mathtt{p}(\bar{q}) \leftarrow \epsilon$;
18     repeat in parallel
19         $\mathtt{p}' \leftarrow \mathtt{p}$
20         forall $(\bar{q}, \bar{q}') \in \hookrightarrow_\otimes$ do in parallel
21             $\mathtt{p}(\bar{q}') \leftarrow \mathtt{max}(\mathtt{p}(\bar{q}), \mathtt{p}(\bar{q}'))$; if $\bar{q} \in \mathcal{A}$ then  $\mathtt{p}(\bar{q}') \leftarrow \mathtt{max}(\mathtt{p}(\bar{q}'), \bar{q})$
22     until $\mathtt{p} = \mathtt{p}'$
23     return $\mathtt{p}$
```

the initial states are inserted in an *open set* $\mathcal{O}$ and added to $\mathcal{Q}_\otimes$ (l.2). Then, each state in $\mathcal{O}$ is *explored* (l.3), meaning that it is removed from $\mathcal{O}$ (l.4) and its successors are *visited*. This happens in the loop of l.5. Each successor not yet seen before is added to $\mathcal{Q}_\otimes$ and $\mathcal{O}$ (l.6). Once all reachable states have been processed, the MAP function is called (l.7).

In MAP, initially, all accepting states are added to a set of *active* states $\mathcal{A}$ (l.9). While $\mathcal{A}$ is not empty (l.10), the function *update*-$\mathtt{p}$ is called to update the $\mathtt{p}$-function (l.11). Once $\mathtt{p}$ has been updated, for every $\bar{q} \in \mathcal{Q}_\otimes$, we have that $\mathtt{p}(\bar{q}) = \mathtt{max}(\{\bar{q}' \in \mathcal{A} \mid \bar{q}' \hookrightarrow_\otimes^+ \bar{q}\})$. If any state is now its own maximal accepting predecessor w.r.t. $\mathcal{A}$, an accepting cycle, and therefore a counter-example, has been found (l.13). All other accepting states referred to by $\mathtt{p}$ are removed from $\mathcal{A}$ (l.14), as these cannot be in an accepting cycle. This is proven in [11]. A similar property is proven for our HITCHHIKING algorithm in Section 3.2. If, at some point, $\mathcal{A}$ is empty, a counter-example cannot be present (l.15).

In the *update*-$\mathtt{p}$ function, first, $\mathtt{p}$ is reset, i.e., for all $\bar{q} \in \mathcal{Q}_\otimes$, $\mathtt{p}(\bar{q})$ is set to the special value ϵ, which is smaller than all the states in $\mathcal{Q}_{F,\otimes}$. Next, in the loop of l.18, $\mathtt{p}$ is updated until a fix-point has been reached, which is detected by comparing in each iteration the updated $\mathtt{p}$ to its definition $\mathtt{p}'$ at the start of the iteration. The update is performed by processing all transitions in parallel. For each transition $\bar{q} \hookrightarrow_\otimes \bar{q}'$, $\mathtt{p}(\bar{q}')$ is updated to be the maximum of $\mathtt{p}(\bar{q})$, the current value of $\mathtt{p}(\bar{q}')$, and, if $\bar{q} \in \mathcal{A}$, $\bar{q}$ (l.21). Note that the updating of $\mathtt{p}(\bar{q}')$ needs to be done atomically, as it can be updated simultaneously by multiple threads processing different transitions to $\bar{q}'$.

While adapting Alg. 1 to an on-the-fly algorithm is certainly possible, two observations led to the development of a new algorithm based on MAP. Firstly, in each iteration of the loop of l.20, *all* transitions are inspected, but actually, in


$\bar{r}_n$
$\bar{r}_2$
$\bar{r}_1$
$\bar{r}_0$
$\bar{q}_n$
$\bar{q}_2$
$\bar{q}_1$
$\bar{q}_0$
$\bar{r}'$
ϵ ϵ ϵ ϵ ϵ
ϵ $\bar{q}_3$ $\bar{q}_2$ $\bar{q}_1$ $\bar{q}_0$
ϵ $\bar{q}_n$ $\bar{q}_n$ $\bar{q}_n$ $\bar{q}_n$

(a) At start of *update*-p. (b) After one iteration. (c) End of *update*-p.

Fig. 2: Applying update-p once on an example NBA.

the first iteration, only transitions $\bar{q} \hookrightarrow_{\otimes} \bar{q}'$ for which $\bar{q} \in \mathcal{A}$ lead to an update of p, and in subsequent iterations, these updates are propagated along the paths in $K \otimes B_{\neg\varphi}$. The use of an open set would avoid unnecessary processing of transitions, but initially, at the start of *update*-p, only adding the initial states to such a set would likely restrict parallel computation too much; since the states in $\mathcal{A}$ lead to an update of p, and typically $|\mathcal{A}| > |\mathcal{Q}_{0,\otimes}|$, adding all states in $\mathcal{A}$ to the open set directly leads to more parallelism and faster updating of p.

Secondly, and most importantly, Map is very conservative in keeping states in $\mathcal{A}$ from one iteration of the loop of l.10 to the next. Fig. 2 illustrates this. The accepting states are ordered as follows: $\bar{q}_i > \bar{q}_j$ iff $i > j$. Fig. 2a shows the situation at the beginning of the first *update*-p call, with the grey labels representing the p-values (the $\bar{r}_i$ states keep the p-value ϵ throughout execution of the algorithm). After one iteration of the loop of l.18, the situation is as shown in Fig. 2b: the states $\bar{q}_{n-1}$ to $\bar{q}_0$ and $\bar{r}'$ all point to their left neighbour with their p-value. In subsequent iterations, these p-values are propagated to the right, with in the end, all the previously mentioned states pointing to $\bar{q}_n$ (Fig. 2c). In Alg. 1, this results in only removing $\bar{q}_n$ from $\mathcal{A}$ at l.14. However, referring with the $\bar{q}_i$*-search* to the propagation of $\bar{q}_i$ through $K \otimes B_{\neg\varphi}$, note that the $\bar{q}_0$-search was actually never interrupted: $\bar{q}_0$ was propagated to $\bar{r}'$, and after considering the self-loop of $\bar{r}'$ this search ended. Actually, none of the $\bar{q}_i$-searches were interrupted, and restarting them in the next execution of *update*-p will obviously not lead to finding an accepting cycle.

We observe that Map fails to distinguish situations in which a $\bar{q}_i$-search is interrupted from situations in which it is not, and that being able to distinguish them will likely positively impact the practical runtime.

3.2 Hitchhiking

In this section, we present our Hitchhiking algorithm, which addresses the observations made in the previous section. Compared to Map, Hitchhiking is not only on-the-fly, but also keeps track of which accepting state propagations are interrupted, by means of a set of states $\mathcal{F}$: an accepting state is added to $\mathcal{F}$ when its propagation is interrupted. Once a fix-point has been reached, $\mathcal{F}$ provides the accepting states that have to be reopened. All the accepting states that require reopening are added to $\mathcal{O}$, so they can be processed in parallel.

We refer with $\mathcal{C} \subseteq \mathcal{Q}_{\otimes}$ to the set consisting of all states in $\mathcal{Q}_{\otimes}$ that are part of at least one accepting cycle, and with Θ to the set of all accepting cycles in

Algorithm 2: The parallel on-the-fly HITCHHIKING algorithm.

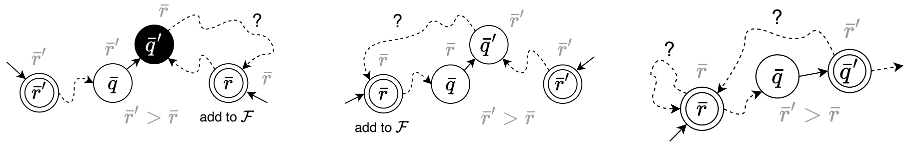
```
 1  function void HITCHHIKING($(\mathcal{Q}_{0,\otimes}, \mathsf{succ}_{\otimes}())$):
 2      $\mathcal{O} \leftarrow \emptyset$                // The open set, containing states to be explored
 3      $\mathcal{F} \leftarrow \emptyset$                // Set to keep track of cycle detection restarts
 4      $\mathcal{A} \leftarrow \emptyset$                // Set of accepting states for which cycle detection is ongoing
 5      forall $\bar{q} \in \mathcal{Q}_{0,\otimes}$ do
 6          $\mathcal{O} \leftarrow \mathcal{O} \cup \{\bar{q}\}$
 7          if $\bar{q} \in \mathcal{Q}_{F,\otimes}$ then $\mathcal{A} \leftarrow \mathcal{A} \cup \{\bar{q}\}$; $\mathtt{p}(\bar{q}) \leftarrow \bar{q}$      // $\bar{q}$ is cycle active
 8          else $\mathtt{p}(\bar{q}) \leftarrow \epsilon$
 9      while $\mathcal{O} \neq \emptyset$ do                // Beginning of a round
10          forall $\bar{q} \in \mathcal{O}$ do in parallel
11              $\mathcal{O} \leftarrow \mathcal{O} \setminus \{\bar{q}\}$; $\alpha \leftarrow \mathtt{p}(\bar{q})$
12              forall $\bar{q}' \in \mathsf{succ}_{\otimes}(\bar{q})$ do in parallel
13                  if $\alpha = \bar{q}'$ then return counter-example found
14                  if $\mathtt{p}(\bar{q}') = \bot$ and $\bar{q}' \in \mathcal{Q}_{F,\otimes}$ then
15                      $\mathcal{A} \leftarrow \mathcal{A} \cup \{\bar{q}'\}$
16                      if $\bar{q}' > \alpha$ then $\alpha \leftarrow \bar{q}'$      // Interrupted by $\bar{q}' \in \mathcal{A}$ (Fig. 3c)
17                      else $\mathcal{F} \leftarrow \mathcal{F} \cup \{\bar{q}'\}$      // Fig. 3a with $\bar{q}' = \bar{r}$
18                  [$\beta \leftarrow \mathtt{p}(\bar{q}')$; $\mathtt{p}(\bar{q}') \leftarrow \mathtt{max}(\alpha, \beta)$]      // Atomic update of $\mathtt{p}(\bar{q}')$
19                  if $\alpha > \beta$ and $\beta > \epsilon$ and $\bar{q}' \in \mathcal{O}$ then $\mathcal{F} \leftarrow \mathcal{F} \cup \{\beta\}$      // Fig.3a
20                  if $\beta > \alpha$ and $\alpha > \epsilon$ and $\bar{q}' \notin \mathcal{A}$ then $\mathcal{F} \leftarrow \mathcal{F} \cup \{\alpha\}$      // Fig.3b
21                  if $\alpha > \beta$ then $\mathcal{O} \leftarrow \mathcal{O} \cup \{\bar{q}'\}$
22          if $\mathcal{F} \neq \emptyset$ then                // Post-processing of a round
23              forall $\bar{q} \in \mathcal{Q}_{\otimes}$ do in parallel
24                  if $\bar{q} \in \mathcal{A}$ then
25                      if $\bar{q} \in \mathcal{F}$ and $\mathtt{p}(\bar{q}) \neq \bar{q}$ then $\mathcal{O} \leftarrow \mathcal{O} \cup \{\bar{q}\}$
26                      else $\mathcal{A} \leftarrow \mathcal{A} \setminus \{\bar{q}\}$
27                      if $\bar{q} \in \mathcal{F}$ then $\mathcal{F} \leftarrow \mathcal{F} \setminus \{\bar{q}\}$
28                  if $\bar{q} \in \mathcal{A}$ then $\mathtt{p}(\bar{q}) \leftarrow \bar{q}$ else $\mathtt{p}(\bar{q}) \leftarrow \epsilon$
29      return no counter-example found
```

$K \otimes B_{\neg\varphi}$. HITCHHIKING never actually constructs $\mathcal{C}$ and Θ, but we reason about their contents to prove the algorithm's correctness.

A function $\mathtt{p} : \mathcal{Q}_{\otimes} \to \mathcal{Q}_{F,\otimes} \cup \{\bot, \epsilon\}$ maps each state in $\mathcal{Q}_{\otimes}$ to either an accepting state in $\mathcal{Q}_{F,\otimes}$ or one of the special values $\bot$ and ϵ, with $\bot$ representing never having reached that state before, i.e., initially, for all $\bar{q} \in \mathcal{Q}_{\otimes}$, $\mathtt{p}(\bar{q}) = \bot$, and ϵ representing that the state has been encountered, but has not been assigned a state reference. A strict total order $>$ on $\mathcal{Q}_{F,\otimes} \cup \{\bot, \epsilon\}$ defines that $\epsilon > \bot$ and for all $\bar{q} \in \mathcal{Q}_{F,\otimes}$, we have $\bar{q} > \epsilon$. How the states in $\mathcal{Q}_{F,\otimes}$ are ordered is not important, only that there exists a fixed order.

Alg. 2 presents HITCHHIKING in pseudo-code. First, we explain the algorithm, ignoring parallel execution. Later, we reason about correctness of the parallelisation. In the loop of l.5, all initial states are added to $\mathcal{O}$, and any accepting initial states are added to a set of active states $\mathcal{A}$. For a state $\bar{q} \in \mathcal{Q}_{0,\otimes}$, $\mathtt{p}(\bar{q})$ is set to $\bar{q}$ if $\bar{q} \in \mathcal{Q}_{F,\otimes}$, otherwise to ϵ. Note that this is different from how $\mathtt{p}$ is constructed in MAP, but this logically follows from the fact that HITCHHIKING detects the interruption of cycle searches. We address this later.

Similar to the loop of l.10 in Alg. 1, HITCHHIKING works in *rounds*. A new round is started iff $\mathcal{O} \neq \emptyset$ (l.9). During a round, in each iteration of the loop of l.10, a state $\bar{q}$ is taken from $\mathcal{O}$ to be explored. We refer to $\mathtt{p}(\bar{q})$ with α (l.11). In the loop of l.12, the successors are inspected. If α is encountered as successor, an accepting cycle has been found (l.13). If successor $\bar{q}'$ is visited for the first time, i.e., $\mathtt{p}(\bar{q}') = \bot$, and $\bar{q}' \in \mathcal{Q}_{F,\otimes}$, it is added to $\mathcal{A}$ (l.15). If $\bar{q}' > \alpha$, then



(a) Interrupted $\bar{r}$-search. (b) Interrupted $\bar{r}$-search. (c) Interrupted by $\bar{q}' \in \mathcal{A}$.

Fig. 3: Various cycle search interruption situations.

the $\bar{q}'$-search should be started, and the α-search interrupted in this direction, otherwise the $\bar{q}'$-search is interrupted, which is recorded by adding $\bar{q}'$ to a set of *interrupted* states $\mathcal{F}$ (l.16–17). Note that in the first case, α is not added to $\mathcal{F}$. Why that is not needed is illustrated by Fig. 3c: If in an $\bar{r}$-search, another accepting state $\bar{q}'$ is encountered with $\mathtt{p}(\bar{q}') > \bar{r}$, then any cycle containing both $\bar{r}$ and $\mathtt{p}(\bar{q}')$ will be detected by either the $\mathtt{p}(\bar{q}')$-search or yet another search with a higher priority. Any other possible cycles that contain $\bar{r}$ but not $\mathtt{p}(\bar{q}')$ can still be found by the $\bar{r}$-search.

Next, $\mathtt{p}(\bar{q}')$ is updated, and we refer to its value before the update with β. The state reference is updated to the maximum of α and β (l.18). The possible search interruption situations are handled next, see Figs. 3a and 3b.

If for $\bar{q}'$, we have $\mathtt{p}(\bar{q}) > \mathtt{p}(\bar{q}') > \epsilon$ and $\bar{q}' \in \mathcal{O}$, then the $\mathtt{p}(\bar{q}')$-search is interrupted by the $\mathtt{p}(\bar{q})$-search, and $\mathtt{p}(\bar{q}')$ must be added to $\mathcal{F}$ (l.19). In Fig. 3a, this is illustrated, with $\bar{q}'$ being black indicating that it is in $\mathcal{O}$, and $\mathtt{p}(\bar{q}) = \bar{r}'$, $\mathtt{p}(\bar{q}') = \bar{r}$. Note that if $\bar{q}'$ was not in $\mathcal{O}$, the $\bar{r}$-search would not be affected by involving $\bar{q}'$ in the $\bar{r}'$-search, as the $\bar{r}$-search had already moved on.

If for $\bar{q}'$, we have $\mathtt{p}(\bar{q}') > \mathtt{p}(\bar{q}) > \epsilon$ and $\bar{q}' \notin \mathcal{A}$, then the $\mathtt{p}(\bar{q})$-search is interrupted by the $\bar{q}'$-search, and $\mathtt{p}(\bar{q})$ must be added to $\mathcal{F}$ (l.20). In Fig. 3b, this is illustrated. If there is no cycle containing both $\bar{r}$ and $\bar{r}'$, the $\bar{r}'$-search will not detect any cycles containing $\bar{r}$, hence $\bar{r}$ must be added to $\mathcal{F}$. At l.21, $\bar{q}'$ is added to $\mathcal{O}$ if it needs to be (re-)explored, i.e., if $\mathtt{p}(\bar{q}')$ was updated to a larger value.

At l.22, the post-processing of the round starts. If there are states in $\mathcal{F}$, a next round is needed. In the loop of l.23, all the states are inspected. Any state $\bar{q} \in \mathcal{A}$ in $\mathcal{F}$ with $\mathtt{p}(\bar{q}) \neq \bar{q}$ is added to $\mathcal{O}$. In particular, a state $\bar{q} \in \mathcal{F}$ with $\mathtt{p}(\bar{q}) = \bar{q}$ does *not* need to be reopened. This is addressed by the proof of Lemma 6. All the states in $\mathcal{A}$ that are not added to $\mathcal{O}$ are removed from $\mathcal{A}$ (l.26). Next, all states are removed from $\mathcal{F}$ (l.27). Finally, $\mathtt{p}$ is updated: if $\bar{q}$ is in $\mathcal{A}$, it starts referencing itself, and otherwise $\mathtt{p}(\bar{q})$ is reset to ϵ (l.28).

The fact that search interruptions are detected when inspecting successors, and that it is relevant whether a successor $\bar{q}'$ is in $\mathcal{A}$ or not (l.20), makes it logical to check whether $\bar{q}' \in \mathcal{Q}_{F,\otimes}$ when first visiting $\bar{q}'$ (l.14). Since at that stage, $\mathtt{p}(\bar{q}')$ must be updated, it is more elegant to set it to $\bar{q}'$ when the $\bar{q}'$-search needs to be started, as opposed to using ϵ, as is done in MAP. When $\bar{q}'$ is explored, the former option leads to the successors of $\bar{q}'$ naturally propagating $\bar{q}'$ as $\mathtt{p}$-value.

With the following Lemmas and Theorems, we prove that HITCHHIKING is correct and terminates.

Lemma 1. *Every time the loop of l.10 is entered, and* Hitchhiking *does not return at l.13, eventually, l.22 is reached with $\mathcal{O} = \emptyset$.*

Proof. When the loop of l.10 is entered, $\mathcal{O} \neq \emptyset$. In every iteration of this loop, a state $\bar{q} \in \mathcal{O}$ is selected and removed from $\mathcal{O}$ at l.11. Each successor $\bar{q}' \in \mathtt{succ}_{\otimes}(\bar{q})$ is processed in the loop of l.12. Since Hitchhiking does not return at l.13, $\mathtt{p}(\bar{q}')$ may or may not be updated at l.18, depending on whether $\mathtt{p}(\bar{q}')$ is smaller or not than α, which was set to $\mathtt{p}(\bar{q})$ at l.11 and possibly updated to $\bar{q}'$ at l.16 if $\bar{q}' \in \mathcal{Q}_{F,\otimes}$ and this is the first time $\bar{q}'$ is visited. If $\mathtt{p}(\bar{q}')$ is updated, $\bar{q}'$ is added to $\mathcal{O}$ at l.21. Since $\mathcal{Q}_{F,\otimes}$ is strictly totally ordered by $>$ and finite, $\mathtt{p}(\bar{q}')$ can only be updated a finite number of times, and hence $\bar{q}'$ is only added a finite number of times to $\mathcal{O}$. Therefore, as there are a finite number of states, eventually $\mathcal{O} = \emptyset$, and the loop of l.10 is exited, and l.22 is reached. □

Lemma 2. *When l.22 is reached for the first time, $\forall \bar{q} \in \mathcal{Q}_{\otimes}.\mathtt{p}(\bar{q}) \neq \bot$ holds until* Hitchhiking *terminates.*

Proof. Initially, $\forall \bar{q} \in \mathcal{Q}_{\otimes}.\mathtt{p}(\bar{q}) = \bot$. For all $\bar{q} \in \mathcal{Q}_{0,\otimes}$, $\mathtt{p}(\bar{q})$ is set to either $\bar{q}$ at l.7 or to ϵ at l.8. For any state $\bar{q}' \in \mathcal{Q}_{\otimes}$ reached inside the loop of l.10, $\mathtt{p}(\bar{q}')$ is updated at l.18, and since $\bot$ is the bottom element, if $\mathtt{p}(\bar{q}') = \bot$ before l.18, this always results in $\mathtt{p}(\bar{q}') > \bot$ after execution of l.18. Since all states in $\mathcal{Q}_{\otimes}$ are reachable from $\mathcal{Q}_{0,\otimes}$, once l.22 is reached, $\forall \bar{q} \in \mathcal{Q}_{\otimes}.\mathtt{p}(\bar{q}) \neq \bot$. □

In the following, we refer with $\bar{q}_{max}$ to the maximum element of $\mathcal{A}$ ordered by $>$, i.e., $\bar{q}_{max} \in \mathcal{A} \land \forall \bar{q} \in \mathcal{A}.\bar{q}_{max} = \bar{q} \lor \bar{q}_{max} > \bar{q}$.

Lemma 3. $\mathcal{A} \neq \emptyset \implies \forall \bar{q} \in \mathcal{Q}_{\otimes}.\bar{q}_{max} \geq \mathtt{p}(\bar{q})$ *is an invariant of* Hitchhiking.

Proof. After initialisation, at the start of the first round at l.9, $\mathcal{A} = \mathcal{Q}_{0,\otimes} \cap \mathcal{Q}_{F,\otimes}$, and $\forall \bar{q} \in \mathcal{A}.\mathtt{p}(\bar{q}) = \bar{q}$, and $\forall \bar{q} \notin \mathcal{A}.\mathtt{p}(\bar{q}) = \epsilon \lor \mathtt{p}(\bar{q}) = \bot$. Hence, if $\mathcal{A} \neq \emptyset$, by definition of $\bar{q}_{max}$ and the fact that ϵ and $\bot$ are smaller than all $\bar{q} \in \mathcal{Q}_{F,\otimes}$, we have $\forall \bar{q} \in \mathcal{Q}_{\otimes}.\bar{q}_{max} \geq \mathtt{p}(\bar{q})$. If this holds at the beginning of a round (l.9), then during execution of the round, this remains valid, and holds subsequently at l.18 and l.22, as no state is removed from $\mathcal{A}$, and only states in $\mathcal{A}$ can be assigned to the $\mathtt{p}(\bar{q})$ of some state $\bar{q} \in \mathcal{Q}_{\otimes}$, and $\bar{q}_{max}$ is by definition the maximum element of $\mathcal{A}$. If in the loop of l.23, $\bar{q}_{max} = \bar{q}'$ and $\bar{q}_{max}$ is removed from $\mathcal{A}$ at l.26, then for each $\bar{q} \in \mathcal{Q}_{\otimes}$ with $\mathtt{p}(\bar{q}) = \bar{q}'$, $\mathtt{p}(\bar{q})$ is either set to $\bar{q}$ if $\bar{q} \in \mathcal{A}$, or to ϵ otherwise at l.28. In fact, for all $\bar{q}$, $\mathtt{p}(\bar{q})$ will be updated to either $\bar{q}$ or ϵ. Hence, at the end of the loop of l.23, if $\mathcal{A} \neq \emptyset$, we again have $\forall \bar{q} \in \mathcal{Q}_{\otimes}.\bar{q}_{max} \geq \mathtt{p}(\bar{q})$, for possibly a new $\bar{q}_{max}$. Since $\mathcal{A} \neq \emptyset \implies \forall \bar{q} \in \mathcal{Q}_{\otimes}.\bar{q}_{max} \geq \mathtt{p}(\bar{q})$ holds after processing the end of a round for which $\mathcal{A} \neq \emptyset \implies \forall \bar{q} \in \mathcal{Q}_{\otimes}.\bar{q}_{max} \geq \mathtt{p}(\bar{q})$ holds, and since it holds at the beginning of the first round, it always holds. □

Lemma 4. *At l.9: If $\bar{q}_{max} \in \mathcal{C}$ and $\forall \bar{q} \in \mathcal{Q}_{\otimes}.\mathtt{p}(\bar{q}) \neq \bot$, then* Hitchhiking *returns at l.13 before l.22 is reached.*

Proof. First, we must have $\bar{q}_{max} \in \mathcal{O}$: if $\bar{q}_{max}$ was added to $\mathcal{A}$ at l.7, then it was added to $\mathcal{O}$ at l.6. Otherwise, $\bar{q}_{max}$ must have been in $\mathcal{A}$ or added to $\mathcal{A}$ in the previous round, at the end of which, since it was not removed from $\mathcal{A}$ at l.26, it must have been added to $\mathcal{O}$ at l.25. Because $\bar{q}_{max} \in \mathcal{O}$, each successor $\bar{q}' \in \mathsf{succ}_\otimes(\bar{q}_{max})$ is inspected in the loop of l.12. If $\bar{q}' = \bar{q}_{max}$, Hitchhiking returns at l.13, and the lemma holds. If $\bar{q}' \neq \bar{q}_{max}$, then since $\mathsf{p}(\bar{q}') \neq \bot$, l.15–17 are not executed. At l.18, $\mathsf{p}(\bar{q}')$ is set to $\bar{q}_{max}$, since $\bar{q}_{max} \geq \mathsf{p}(\bar{q}')$ by Lemma 3. Hence, l.20 is not executed, i.e., the $\bar{q}_{max}$-search is not blocked. At l.21, $\bar{q}'$ is added to $\mathcal{O}$. By the same reasoning as above, when $\bar{q}'$ is taken from $\mathcal{O}$ at l.11, for all its successors $\bar{q}''$, $\mathsf{p}(\bar{q}'')$ will be set to $\bar{q}_{max}$, unless for some successor, $\bar{q}'' = \bar{q}_{max}$, in which case Hitchhiking returns at l.13. Since $\bar{q}_{max} \in \mathcal{C}$, and the $\bar{q}_{max}$-search is never blocked, this eventually happens.

Finally, any other search for a $\bar{q} \in \mathcal{A}$ with $\bar{q} \neq \bar{q}_{max}$ cannot interrupt the $\bar{q}_{max}$-search either: if the $\bar{q}$-search reaches a state $\bar{r} \in \mathcal{O}$ with $\mathsf{p}(\bar{r}) = \bar{q}_{max}$, then the update at l.18 fails, as $\bar{q}_{max} \geq \mathsf{p}(\bar{r})$ by Lemma 3, and l.19 is not executed. □

Lemma 5. *At l.22: $\bar{q}_{max} \notin \mathcal{C} \wedge \bar{q}_{max} \notin \mathcal{F}$.*

Proof. This follows from the same reasoning as in the proof of Lemma 4. The $\bar{q}_{max}$-search is not interrupted before l.22 is reached, i.e., l.19–20 are never executed to add $\bar{q}_{max}$ to $\mathcal{F}$. When l.22 is reached, Hitchhiking did not return at l.13 in the current round, hence by Lemma 4 and the fact that by Lemma 2, $\forall \bar{q} \in \mathcal{Q}_{F,\otimes}.\mathsf{p}(\bar{q}) \neq \bot$, we must have that $\bar{q}_{max} \notin \mathcal{C}$. □

Lemma 6. *At l.22: $\forall \sigma \in \Theta . \exists \bar{q} \in \mathcal{A} \cap \sigma . \bar{q} \in \mathcal{F} \wedge p(\bar{q}) \neq \bar{q}$.*

Proof. By induction on the number of rounds performed when l.22 is reached.

- 1: When a state $\bar{q} \in \mathcal{Q}_{F,\otimes}$ is reached for the first time, it is added to $\mathcal{A}$ at either l.7 or l.15 (in the latter case, $\mathsf{p}(\bar{q}) = \bot$), and during the round, states are not removed from $\mathcal{A}$. By Lemma 2, at l.22, $\forall \bar{q} \in \mathcal{Q}_\otimes.\mathsf{p}(\bar{q}) \neq \bot$, meaning that $\mathcal{A} = \mathcal{Q}_{F,\otimes}$. Consider an accepting cycle $\sigma \in \Theta$ at l.22. Since $\mathcal{A} = \mathcal{Q}_{F,\otimes}$, we must have $\mathcal{A} \cap \sigma \neq \emptyset$. First we prove $\exists \bar{q} \in \mathcal{A} \cap \sigma . \bar{q} \in \mathcal{F}$. Consider a state $\bar{r} \in \mathcal{A} \cap \sigma$. Since Hitchhiking did not return at l.13 in the first round, the $\bar{r}$-search over σ must have been interrupted by one of the situations in Fig. 3. Either a state $\bar{q}' \in \mathcal{O}$ with $\mathsf{p}(\bar{q}') = \bar{r}$ was visited by a $\bar{r}'$-search with $\bar{r}' > \bar{r}$ (case a), which led to $\bar{r}$ being added to $\mathcal{F}$ at l.19, or the $\bar{r}$-search visited a state $\bar{q}'$ with $\mathsf{p}(\bar{q}') > \bar{r}$. In the latter case, either $\bar{q}' \notin \mathcal{A}$, leading to $\bar{r}$ being added to $\mathcal{F}$ at l.20 (case b), or $\bar{q}' \in \mathcal{A}$ (case c), but then, either $\bar{q}' \notin \sigma$, and the $\bar{r}$-search continued along σ, or $\bar{q}' \in \sigma$ and the $\mathsf{p}(\bar{q}')$-search continued along σ, which apparently was in turn interrupted by one of the blocking situations, since σ was not detected. Since σ is finite, the latter case can only be applicable a finite number of times, so eventually, a state $\bar{r}'' \in \mathcal{A} \cap \sigma$ must have been added to $\mathcal{F}$ (case b).

  Next, we prove that for a state $\bar{q} \in \mathcal{A} \cap \mathcal{F} \cap \sigma$, $\mathsf{p}(\bar{q}) \neq \bar{q}$ holds. Consider a state $\bar{r} \in \mathcal{A} \cap \mathcal{F} \cap \sigma$. Some $\bar{r}'$-search caused the $\bar{r}$-search to be interrupted along σ, with $\bar{r}$ added to $\mathcal{F}$. This $\bar{r}'$-search must have traversed σ (cases a and b). Since both $\bar{r} \in \sigma$ and $\bar{r}' \in \sigma$, $\bar{r}$ is reachable from $\bar{r}'$, i.e., there is some path π from $\bar{r}'$ to $\bar{r}$. Two cases can be distinguished: 1) the $\bar{r}'$-search continued until $\bar{r}$ was reached from some state $\bar{q} \in \sigma$, at which point, since $\bar{r}' > \bar{r}$, $\mathsf{p}(\bar{r})$ was updated to $\bar{r}'$ at l.18, or 2) the $\bar{r}'$-search was interrupted along π due to an interruption situation involving a state $\bar{q}' \in \sigma$ with $\mathtt{p}(\bar{q}')$ having some value $\bar{r}'' > \bar{r}'$. In the latter case, however, in turn, the same two cases are applicable for the $\bar{r}''$-search. Since π is finite, if case 2 continued to be applicable, eventually case 1 had to be applicable: some state $\bar{q}''$ with $\bar{r} \in \mathsf{succ}_{\otimes}(\bar{q}'')$ must have been reached, with $\mathtt{p}(\bar{q}'') > \mathtt{p}(\bar{r})$, after which exploration of $\bar{q}''$'s successors (l.12) led to $\mathtt{p}(\bar{r})$ being updated to $\mathtt{p}(\bar{q}'')$ at l.18.

- $\underline{n{+}1}$: By the induction hypothesis, after n rounds, Lemma 6 holds. After n rounds, each state $\bar{q} \in \mathcal{A} \cap \mathcal{F}$ with $\mathtt{p}(\bar{q}) \neq \bar{q}$ is added to $\mathcal{O}$ at l.25 and not removed from $\mathcal{A}$ at l.26. Subsequently, $\bar{q}$ is removed from $\mathcal{F}$ at l.27 and $\mathtt{p}(\bar{q})$ is set to $\bar{q}$ at l.28. In the next round, therefore, for every $\sigma \in \Theta$, at least one search commences of a state $\bar{q} \in \mathcal{A} \cap \sigma$. If no cycle is detected, by the same reasoning as for the base case, it follows that Lemma 6 again holds when l.22 is reached after round $n+1$. □

Theorem 1. Hitchhiking *returns at l.13 iff* $\mathcal{C} \neq \emptyset$.

Proof. $\Rightarrow$: If Hitchhiking returns at l.13, a successor $\bar{q}'$ of a state $\bar{q}$ was encountered with $\mathtt{p}(\bar{q}) = \bar{q}'$. The fact that $\mathtt{p}(\bar{q}) = \bar{q}'$ means that there exists some path π from $\bar{q}'$ to $\bar{q}$. At some point, when $\bar{q}'$ was visited, $\mathtt{p}(\bar{q}')$ was set to $\bar{q}'$, either at l.7 or at l.18 (because of l.16). In both cases, note that $\bar{q}'$ was added to $\mathcal{A}$ at either l.7 or l.15, respectively, and that this means that $\bar{q}' \in \mathcal{Q}_{F,\otimes}$. Subsequently, the successor $\bar{q}_0$ of $\bar{q}'$ along π was considered at l.12, with $\mathtt{p}(\bar{q}_0)$ set to $\bar{q}'$ at l.18. In turn, the successor $\bar{q}_1$ of $\bar{q}_0$ along π was visited and $\mathtt{p}(\bar{q}_1)$ updated to $\bar{q}'$, etc., until $\bar{q}$ was reached. From the existence of π, the fact that $\bar{q}' \in \mathsf{succ}_{\otimes}(\bar{q})$, and $\bar{q}' \in \mathcal{Q}_{F,\otimes}$, there is an accepting cycle, i.e., $\mathcal{C} \neq \emptyset$.

$\Leftarrow$: If $\mathcal{C} \neq \emptyset$, then at the start of the first round, if a $\bar{q} \in \mathcal{Q}_{F,\otimes}$ is reached the first time, when $\mathtt{p}(\bar{q}) = \bot$, then either at l.7 or l.15, $\bar{q}$ is added to $\mathcal{A}$. Next, either not all searches through $\mathcal{C}$ are interrupted, and eventually, Hitchhiking returns at l.13, or all searches through $\mathcal{C}$ are interrupted, and by Lemma 1, Hitchhiking reaches l.22. From the second round onwards, we can distinguish two cases at the start of executing a round (l.9):

1. $\bar{q}_{max} \in \mathcal{C}$. By Lemma 2 and Lemma 4, Hitchhiking returns at l.13.
2. $\bar{q}_{max} \notin \mathcal{C}$. Then, Hitchhiking may or may not return at l.13. If it does not, eventually, by Lemma 1, l.22 is reached. At l.22, by Lemma 5, $\bar{q}_{max} \notin \mathcal{F}$, and therefore, $\bar{q}_{max}$ is removed from $\mathcal{A}$ at l.26. Whenever a state $\bar{q} \in \mathcal{Q}_{F,\otimes}$ is visited for the first time, it is added to $\mathcal{A}$ at either l.7 or l.15, and can only be removed from $\mathcal{A}$ at l.26. However, by Lemma 6, for each cycle $\sigma \in \Theta$, there exists a state $\bar{q} \in \mathcal{A} \cap \sigma$ with $\bar{q} \in \mathcal{F}$ and $\mathtt{p}(\bar{q}) \neq \bar{q}$. This $\bar{q}$ is added to $\mathcal{O}$ at l.25, and therefore not removed from $\mathcal{A}$ at l.26. Hence, at l.28, $\mathcal{C} \cap \mathcal{A} \neq \emptyset$, and some state $\bar{q} \in \mathcal{A}$ now acts as $\bar{q}_{max}$. At l.28, each $\mathtt{p}(\bar{q}')$, with $\bar{q}' \in \mathcal{Q}_{\otimes}$, is either set to a value in $\mathcal{A}$ or ϵ.

   In the next round, the same two cases can be distinguished for the new $\bar{q}_{max}$. Eventually, case 1 must be applicable, since $\mathcal{A}$ is finite, and as states in $\mathcal{A} \setminus \mathcal{C}$ keep being removed from $\mathcal{A}$, eventually, $\mathcal{A} \subseteq \mathcal{C}$, meaning that $\bar{q}_{max} \in \mathcal{C}$. □

Theorem 2. Hitchhiking *returns at l.29 iff* $\mathcal{C} = \emptyset$.

Proof. $\Rightarrow$: Follows from Theorem 1. Since HITCHHIKING returns at l.13 iff $\mathcal{C} \neq \emptyset$, at l.29, we must have $\mathcal{C} = \emptyset$.

$\Leftarrow$: Since $\mathcal{C} \neq \emptyset$, by Theorem 1, HITCHHIKING does not return at l.13. By the same reasoning as for the proof of Theorem 1, case $\Leftarrow$, at the end of each round, states in $\mathcal{A} \setminus \mathcal{C}$ are removed from $\mathcal{A}$, and since $\mathcal{A}$ is finite and $\mathcal{A} \cap \mathcal{C} = \emptyset$, eventually, $\mathcal{A} = \emptyset$ and no state is added to $\mathcal{O}$ at l.25 since that line is not reached. By Lemma 1, at that point, $\mathcal{O} = \emptyset$, and therefore subsequently, l.9 is reached with $\mathcal{O} = \emptyset$. This leads to HITCHHIKING returning at l.29. □

Thread-safety. In the parallel version of HITCHHIKING, the parallel for-loops are performed with each element in the involved set, i.e., $\mathcal{O}$, $\mathsf{succ}_{\otimes}(\bar{q})$, or $\mathcal{Q}_{\otimes}$, being processed by a separate thread. Furthermore, we assume that every parallel for-loop is followed by a global thread-barrier (in GPU code, this means that every parallel for-loop is implemented as a separate GPU kernel).

A parallel, shared-memory implementation can use a thread-safe shared hash table to maintain $\mathcal{O}$ and p, while using bookkeeping bits for each element to maintain $\mathcal{A}$ and $\mathcal{F}$. These bits can be manipulated in a thread-safe way using atomic bit operations. This makes the loops of l.10 and l.12 largely thread-safe. To make the updating of $\mathsf{p}(\bar{q})$ thread-safe, an atomic maximum operation can be used, which atomically updates the original value iff the new value is larger, and returns the original value. This may result in some values of $\mathsf{p}(\bar{q})$ never being considered when processing $\bar{q}$, i.e., after having updated $\mathsf{p}(\bar{q})$, a thread may reach l.21 after another thread has again updated $\mathsf{p}(\bar{q})$, but this is not a concern, as $\mathsf{p}(\bar{q})$ monotonically increases, and for correctness, it is only important that in each round, the largest possible value for $\mathsf{p}(\bar{q})$ is processed at some point. In the loop of l.23, threads inspect and manipulate individual states and their bookkeeping bits without following references to other states, which is thread-safe.

Complexity. Consider Fig. 2. If HITCHHIKING explores this NBA strictly in a BFS order, i.e., $\mathcal{O}$ evolves from $\{\bar{r}_0\}$ to $\{\bar{r}_1, \bar{q}_0\}$ to $\{\bar{r}_2, \bar{q}_1, \bar{r}'\}$, etc., then every state $\bar{q}_i$ is explored $n+1-i$ times with an increasing p-value. This is the worst-case in a single round: n accepting states leading to $n \cdot \frac{n+1}{2}$ explorations. With BFS, every time a $\bar{q}_i$-search reaches a state $\bar{q}_j$ with $i > j$, $\bar{q}_j \notin \mathcal{O}$, meaning that at l.19 of Alg. 2, $\bar{q}_j$ is not added to $\mathcal{F}$, and after one round, HITCHHIKING terminates, whereas MAP needs n rounds to terminate. Worst-case, however, HITCHHIKING has the same complexity as MAP, i.e., $O(n^2 \cdot \frac{n+1}{2})$, if the states can be explored in an arbitrary, not strictly BFS order. Still, the practical runtime of HITCHHIKING is susceptible to the exploration order, in contrast to MAP, and can therefore be much better (see Section 4.2).

4 Implementation and Experiments

4.1 Parallel Implementation for GPUs

We implemented HITCHHIKING in CUDA C++ as part of GPUEXPLORE 3.0 [51, 52]. We describe here the general workflow of GPUEXPLORE (Fig. 4). For more

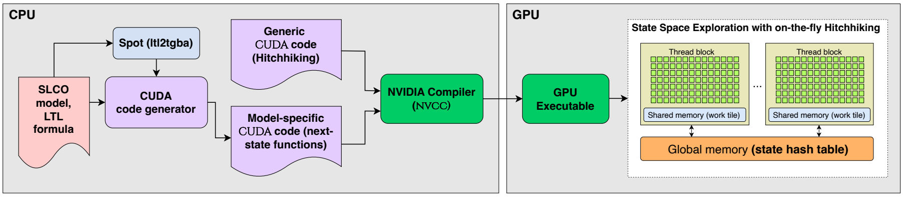

CPU
SLCO model, LTL formula
Spot (ltl2tgba)
CUDA code generator
Generic CUDA code (Hitchhiking)
Model-specific CUDA code (next-state functions)
NVIDIA Compiler (NVCC)
GPU
GPU Executable
State Space Exploration with on-the-fly Hitchhiking
Thread block
Shared memory (work tile)
...
Thread block
Shared memory (work tile)
Global memory (state hash table)

Fig. 4: The workflow of GPUExplore with Hitchhiking.

details, see [51, 52]. Given an input model implicitly describing a Kripke structure K in the Slco language [9, 46], which allows the specification of concurrent systems by means of state machines with shared variables, and an LTL formula φ, a code generator, implemented in Python using textX [17] and Jinja2,[3] produces *model-specific* CUDA C++ code to explore $K \otimes B_{\neg\varphi}$. The Büchi automaton is constructed with the `ltl2tgba` tool of the Spot library [18]. The generated code entails next-state computation functions, i.e., functions that given a state $\bar{q} \in \mathcal{Q}_{\otimes}$, produce the successor states $\mathsf{succ}_{\otimes}(\bar{q})$. One next-state computation function is generated for each state machine in the model, which allows the parallel construction of $\mathsf{succ}_{\otimes}(\bar{q})$, with each function executed by a different thread. In addition, a separate function is generated to update the state of $B_{\neg\varphi}$. Together, the threads explore all the states reachable from $\mathcal{Q}_{0,\otimes}$.

GPUExplore's *generic* code implements the control flow and state storage. In GPU global memory, a large hash table $\mathcal{H}$ is maintained to store the visited states with their current p-value. States are stored as binary trees [9, 51], which enables states to share sub-trees, and *root compression* [14, 30, 51] is applied, which allows compressed storage of tree roots. As tree roots in practice vastly outnumber non-roots, this can compress the overall stored data 2–6 times [51].

On an NVIDIA CUDA GPU, threads run in *blocks*. In GPUExplore, 512 threads form a block, and by default, around 3,000 blocks are launched. The overall workflow is as follows. First, the initial states $\mathcal{Q}_{0,\otimes}$ are identified and stored in $\mathcal{H}$. Next, an exploration *kernel*, i.e., GPU function, is launched, in which the blocks scan $\mathcal{H}$ for unexplored states, each block focussing on a specific region that it was assigned to. This kernel essentially implements the loop of l.12 of Alg. 2. Encountered unexplored states are added to the *work tile* of the block, which resides in fast but small *shared memory*. The threads in the block can use this memory together. Once the fixed-size work tile is full, or there are no more unexplored states in the assigned region, the block assigns threads to state / state machine combinations to generate the successors in parallel. These successors are first stored in shared memory, after which they are added to $\mathcal{H}$ if they are not already present. Global memory has a high latency, and using shared memory for successor storage reduces the number of accesses to $\mathcal{H}$.

In an exploration kernel launch, the blocks fill their tiles and generate and store successors a pre-determined number of iterations. After this, a CPU thread determines via a progress flag whether another kernel launch is necessary.

[3] https://palletsprojects.com/p/jinja.

To avoid frequent scanning of $\mathcal{H}$, GPUExplore employs *work claiming*: as a block stores a state $\bar{q}$ in $\mathcal{H}$ that requires exploration, it can immediately claim $\bar{q}$ for exploration in the next iteration. If it does this, $\bar{q}$ is added to the work tile, and marked as explored in $\mathcal{H}$. Work claiming has a significant performance impact [51]. Due to non-synchronised parallel exploration by many blocks, and work claiming, GPUExplore does not strictly adhere to a BFS order.

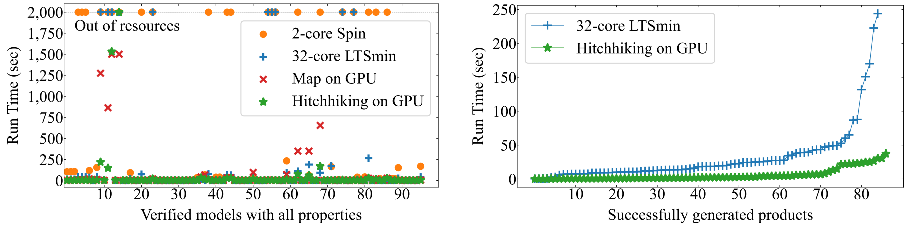
Hitchhiking has been integrated into the exploration kernel, with the $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{F}$ sets implemented with flags associated to the entries in $\mathcal{H}$.

4.2 Experimental Evaluation

We conducted our GPU experiments on a machine running Linux Mint 20 equipped with a Titan RTX GPU from 2018, with 4,608 cores at 1.35 GHz and 24 GB global memory. The generated code is compiled with CUDA 12.2 targeting compute capability 7.5. The GPU experiments include a comparison between an implementation of Hitchhiking with a version of the algorithm in which $\mathcal{F}$ is not used and after every round, all states are added to $\mathcal{O}$. The latter boils down to implementing Map into GPUExplore, as proposed in [2], apart from the fact that our implementation is integrated into the exploration phase and can detect cycles on-the-fly. The DiVinE-CUDA tool [28] contained a GPU implementation of Map, but used the no longer maintained DiVinE v2.0 [4] tool, and is itself no longer maintained or supported by current GPUs. In addition, we compare with state-of-the-art multi-core CPU tools: Spin 6.5.1 [24] and LTSmin 3.0.2 [29,32] running 2-core Nested DFS (NDFS) [26] and multi-core (Combined) NDFS (CNDFS) [22] for LTL checking, respectively, using 32 GB memory. Spin's NDFS can only use up to two cores.

All tools were configured to use state compression, to use the available memory as efficiently as possible. Spin's bit-state hashing and hash compaction were not used, as they make exploration imprecise. That is also the reason to not compare to Spin's Piggyback algorithm [23,25] (see Section 5), as in precise mode, it requires too much memory. Also, all tools support *Partial-Order Reduction* [10,31,35], but we did not use it, as it results in the tools deviating w.r.t. the number of states they explore, making the comparison not exclusively about the LTL checking algorithms. The CPU experiments were performed on an Amazon Web Services machine, running Ubuntu 22.04 with a 32-core 2.65 GHz AMD EPYC 7R13, from 2021.

For benchmarks, we used models from the Beem benchmark suite [44], translated to Slco and Promela (for Spin). All models with state spaces of at least 100,000 states, and without communication channels (support for which in Slco is still experimental) were selected. A few models were scaled up to have larger state spaces. Those are marked in Table 1 with '+'. The state spaces of the benchmarks range in size between 150,000 and 1.2 billion states. The minimum runtime was recorded after having verified a model w.r.t. a single property five times. Timeout was set to 2,000 seconds per run. Furthermore, to be in line with the theory, any deadlocks in a model were removed by adding self-loops to those states. Models that were altered in this way are highlighted yellow in Table 1.



(a) Verification of all properties

(b) Product state space generation of all properties

Fig. 5: LTL model checking performance of various tools

Regarding LTL properties, we initially used three formulae per model, to verify safety and liveness properties, as provided by BEEM. However, we observed for almost all formulae that counter-examples were found very quickly. To obtain better insight into the algorithms' performance, we changed some properties to make them *satisfied*, such that it takes much more time to prove the models correct. For a description of all properties and models, see [41].

Fig. 5a gives a general insight into the performance achievable by all tools running on 32 models × 3 LTL formulae, i.e., 96 instances. Timed out cases are illustrated at the top. For the majority of cases, HITCHHIKING spent between 0 and 250 seconds, while the other tools either ran out of memory or took significantly more time.

Fig. 5b compares the performance of state space exploration without cycle detection using LTSMIN and GPUEXPLORE, i.e., the products are explored but cycle detection is not performed. The runtimes were accumulated, sorting the instances by their individual runtime. The gentle curve of GPUEXPLORE demonstrates that the thousands of GPU cores are being used effectively: as the state space grows, the runtime increases only mildly. On the other hand, LTSMIN has a steeper curve. The individual cores of the state-of-the-art CPU it uses are much more advanced than the GPU cores, but there are relatively few of them.

Table 1 compares in-depth the runtimes of HITCHHIKING with those of other algorithms for the 96 instances. As mentioned earlier, *not-satisfied* formulae (highlighted in red) were trivially violated within one second for both LTSMIN and HITCHHIKING. Among those cases, LTSMIN resulted in incorrect results for the `telephony` models w.r.t. the formula φ_3. We suspect the culprit is a data overflow during the index evaluation of array accesses in the `telephony` models. Some transitions in those models involve complex expressions with array indices to perform indirect memory accesses. With regard to *satisfied* formulae (highlighted in green), HITCHHIKING drastically outperformed its multi-core competitors. For example, HITCHHIKING took only 1 second to verify φ_3 for the six dining philosophers model (`phils.6`, with φ_3 stating that globally, a particular fork is either picked up or lying on the table) compared to 375 seconds by LTSMIN running on 32 cores.

Finally, as indicated by the flipped triangles (▼), GPUEXPLORE with MAP experienced a massive slowdown in performance compared to HITCHHIKING. This confirms our expectations previously discussed in Section 3.

Table 1: Runtime (sec.) of LTL checking on GPU vs. contemporary multi-core tools. 🟨: Model was altered to remove deadlocks. m.: Out of memory. t.: Time out. 🟩: φ is *satisfied*. 🟥: φ is *not satisfied*. ✗: Incorrect result. Times in bold indicate that Hitchhiking was faster than Spin and LTSmin. Significantly worse Map times (at least 0.1 second slower) are marked ▼.

Model	Spin (2-core NDFS)			LTSmin (1-core CNDFS)			LTSmin (32-core CNDFS)			GPUexplore Hitchhiking			GPUexplore Map		
	φ_1	φ_2	φ_3	φ_1	φ_2	φ_3	φ_1	φ_2	φ_3	φ_1	φ_2	φ_3	φ_1	φ_2	φ_3
adding.20+	103	104	106	110.7	110.8	112.8	14.65	14.47	14.2	**0.88**	**0.88**	**0.899**	0.879	0.91	0.898
adding.50+	m.	m.	m.	677.5	673.3	667.2	42.24	43.55	44.34	**3.66**	**3.67**	**3.679**	3.663	3.684	3.679
anderson.6	118	0.17	155	162.4	2.62	195.2	37.43	0.02	41.95	**4.57**	0.168	**3.238**	▼ 19.704	0.168	▼ 12.85
anderson.7	m.	3.9	m.	m.	50.63	m.	m.	9.07	m.	**219**	**1.963**	**148.39**	▼ 1275	▼ 2.381	▼ 865.2
anderson.8	m.	3.84	m.	m.	29.11	m.	m.	0.08	m.	**1531**	0.945	m.	▼ t.	▼ 2.062	m.
at.5	0.06	0.06	92.5	0.001	0.001	120.3	0.001	0.001	22.53	0.144	0.001	**0.6**	0.145	0.001	0.602
at.6	0.06	0.06	m.	0.001	0.001	911.1	0.001	0.01	73.74	0.148	0.001	**2.706**	0.148	0.001	2.708
at.7	0.06	0.06	m.	0.001	0.001	m.	0.001	0.001	m.	0.165	0.001	**19.63**	0.167	0.001	19.67
bakery.5	0.05	0.05	0.04	0.001	0.001	0.001	0.01	0.001	0.001	0.042	0.042	0.041	0.042	0.045	0.044
bakery.6	0.05	0.05	0.05	0.001	0.001	0.001	0.01	0.001	0.001	0.042	0.043	0.047	0.043	0.044	0.068
bakery.7	0.04	0.05	0.05	0.001	0.001	0.001	0.01	0.001	0.001	0.041	0.042	0.042	0.043	0.042	0.042
elevator2.3	3.21	5.87	34.9	2.5	0.01	34.24	0.46	0.04	11.28	1.63	1.965	**0.608**	1.678	▼ 2.946	0.633
elevator2.4+	32.5	66.9	m.	33.94	0.001	570.1	18.21	0.05	76.64	32.39	11.18	**9.083**	▼ 43.66	▼ 67.57	▼ 9.432
frogs.4	0.07	39.5	40.6	0.001	43.16	42.23	0.001	8.65	8.42	0.052	**0.69**	**0.698**	0.062	0.698	0.699
frogs.5	0.05	m.	m.	0.001	1141	1105	0.01	66.27	63.75	0.046	**26.1**	**26.4**	0.046	▼ 33.6	▼ 34.49
lamport.6	0.05	0.05	0.04	0.001	0.001	0.001	0.01	0.01	0.001	0.043	0.043	0.042	0.046	0.043	0.042
lamport.7	0.05	0.05	36.4	0.01	0.01	289.7	0.01	0.001	33.2	0.192	0.204	**17.02**	0.193	0.204	▼ 95.82
lamport.8	0.05	0.05	0.05	0.001	0.001	0.001	0.01	0.001	0.001	0.043	0.042	0.042	0.043	0.043	0.042
loyd.3	m.	m.	m.	m.	m.	m.	m.	m.	m.	**2.59**	**2.6**	**2.627**	2.59	2.6	2.631
mcs.5	0.12	0.12	233	0.001	1.02	m.	0.01	0.001	89.8	0.219	0.229	**9.915**	0.22	0.229	▼ 72.56
peterson.5	0.12	0.12	m.	0.001	0.001	1449	0.02	0.02	109.9	3.425	2.094	**75.78**	3.425	2.094	▼ 348.6
peterson.6	0.07	0.07	60.6	0.001	0.001	m.	0.01	0.01	189.4	0.54	0.543	**58.85**	0.565	0.54	▼ 348
peterson.7	0.18	0.17	m.	0.04	0.001	1232	0.06	0.04	92.64	1.202	0.96	169.4	1.198	0.961	▼ 654.2
phils.6	1.1	1.36	164	0.001	0.001	190.6	0.001	0.001	173.1	0.041	0.031	**1.156**	0.042	0.042	1.155
phils.7	5.99	10.3	m.	0.001	0.02	m.	0.001	0.001	m.	0.001	0.031	**5.156**	0.001	0.034	▼ 5.228
phils.8	3.09	2.58	m.	0.001	0.01	m.	0.01	0.001	m.	0.001	0.026	**2.443**	0.041	0.028	2.484
telephony.4	39.7	0.05	42.4	59.81	0.001	✗	12.51	0.001	✗	**0.44**	0.051	**0.265**	0.442	0.052	0.265
telephony.5	m.	0.05	m.	m.	0.001	✗	264.5	0.01	✗	**28.5**	0.066	**14.6**	28.45	0.067	▼ 14.91
telephony.6	23.3	0.05	m.	0.04	0.001	✗	0.05	0.01	✗	0.314	0.057	**26.56**	0.324	0.058	▼ 26.63
szymanski.5	25.1	24.3	152	10.62	0.01	268	0.01	0.01	25.93	0.287	0.555	**2.363**	▼ 1.122	0.555	2.388
leader_filters.5	0.05	0.05	5.34	0.001	0.001	7.46	0.01	0.01	3.71	0.058	0.07	**0.145**	0.054	0.07	0.151
leader_filters.6	0.05	0.05	169	0.001	0.001	583	0.01	0.01	40.93	0.051	0.045	**5.294**	0.051	0.045	5.336

5 Related Work

Prominent DFS-based LTL model checking algorithms are mentioned in Section 1. In this section, we discuss BFS-based algorithms, and massively parallel state space exploration. To begin with the latter, initially, GPUs were used to accelerate parts of state space exploration, such as duplicate detection [20] and next-state computation [19]. However, the frequent copying of data between GPU and CPU formed the main performance bottleneck. In [21], some state spaces for planning problems were explored on a GPU, assuming a perfect hash function. GPUexplore 1.0 [54] and 2.0 [57] completely perform GPU state space exploration, for models expressed as networks of Labelled Transition Systems. Later, support was added for safety property checking [55] and LTL model checking for cycles up to a given size [53]. Spin was extended with GPU support for reachability analysis [7], which evolved into an incomplete bug-hunting technique [16]. Another tool for reachability analysis was presented in [59], and in [48], a model checker for pushdown automata was presented. Whereas the latter tool supports LTL checking, it is restricted to small models with transitions that can be encoded in 64 bits. Finally, constructing shortest counter-examples with a GPU after LTL checking with the CPU was investigated in [58]. We do not focus on finding shortest counter-examples. This is planned for future work.

BFS-based algorithms were initially developed for distributed model checking. Back-Level Edges [6] searches for edges that close a cycle, but Map is shown to be better in [11]. One-Way-Catch-Them-Young (Owcty) can find counter-examples post-exploration [13]. A heuristic version of Map is added to Owcty, to make it on-the-fly [3], but only to the extent that it *may* find counter-examples early if they exist; it may also overlook them. This heuristic Map is very similar to Piggyback [25], developed in Spin, which was extended to always find a counter-example, as long as the involved cycle is not longer than a given bound [23]. Finally, GPU accelerated post-exploration versions of Map and Owcty were implemented in [2, 5]. Different from these GPU algorithms, ours is truly on-the-fly, i.e., it is guaranteed to find a counter-example, if one exists, before the entire state space has been explored.

6 Conclusions and Future Work

We presented Hitchhiking, a new algorithm for massively parallel LTL model checking, and proved it correct. With a GPU implementation in the GPUexplore 3.0 model checker, we experimentally demonstrated its efficiency, in particular when a model satisfies an LTL formula. Speed-ups up to 150× were measured, compared to a 32-core version of the LTSmin tool running multi-core Nested DFS. This is the first time that GPU on-the-fly LTL model checking has been applied on state spaces in the order of billions of states.

For the future, we plan to work on counter-example construction [58].

Data Availability Statement. The datasets generated and analysed during the current study are available in the Zenodo repository [41].

References

1. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
2. 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
3. Barnat, J., Brim, L., Ročkai, P.: A Time-Optimal On-the-fly Parallel Algorithm for Model Checking of Weak LTL Properties. In: ICFEM. LNCS, vol. 5885, pp. 407–425. Springer (2009). https://doi.org/10.1007/978-3-642-10373-5_21
4. Barnat, J., Brim, L., Ročkai, P.: DiVinE 2.0: High-Performance Model Checking. In: HIBI. pp. 31–32. IEEE (2009). https://doi.org/10.1109/HiBi.2009.10
5. Barnat, J., Brim, L., Ceska, M., Lamr, T.: CUDA Accelerated LTL Model Checking. In: ICPADS. pp. 34–41. IEEE Computer Society (2009). https://doi.org/10.1109/ICPADS.2009.50
6. Barnat, J., Brim, L., Chaloupka, J.: Parallel Breadth-First Search LTL Model-Checking. In: ASE. pp. 106–115. IEEE Computer Society (2003). https://doi.org/10.1109/ASE.2003.1240299
7. 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
8. Bloemen, V., van de Pol, J.: Multi-Core SCC-based LTL Model Checking. In: HVC. LNCS, vol. 10028, pp. 18–33. Springer (2016). https://doi.org/10.1007/978-3-319-49052-6_2
9. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A Database Approach to Distributed State Space Generation. Electron. Notes Theor. Comput. Sci. **198**(1), 17–32 (2008). https://doi.org/10.1016/j.entcs.2007.10.018
10. Bošnački, D., Holzmann, G.: Improving Spin's Partial-Order Reduction for Breadth-First Search. In: SPIN. LNCS, vol. 3639, pp. 91–105. Springer (2005). https://doi.org/10.1007/11537328_10
11. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In: FMCAD. LNCS, vol. 3312, pp. 352–366. Springer (2004). https://doi.org/10.1007/978-3-540-30494-4_25
12. Büchi, J.: On a decision method in restricted second order arithmetic. In: CLMPST. pp. 425–435. Stanford University Press (1962)
13. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection. In: SPIN. LNCS, vol. 2648, pp. 49–73. Springer (2003). https://doi.org/10.1007/3-540-44829-2_4
14. 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
15. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: CAV. LNCS, vol. 531, pp. 233–242. Springer (1990). https://doi.org/10.1007/BFB0023737
16. 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
17. Dejanović, I., Vaderna, R., Milosavljević, G., Vuković, Ž.: TextX: A Python tool for Domain-Specific Language implementation. Knowledge-Based Systems **115**, 1–4 (2017). https://doi.org/10.1016/j.knosys.2016.10.023
18. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Gbaguidi Aisse, A., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: CAV. LNCS, vol. 13372, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9

19. 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
20. Edelkamp, S., Sulewski, D.: External memory breadth-first search with delayed duplicate detection on the GPU. In: MoChArt. LNCS, vol. 6572, pp. 12–31. Springer (2010). https://doi.org/10.1007/978-3-642-20674-0_2
21. Edelkamp, S., Sulewski, D., Yücel, C.: Perfect Hashing for State Space Exploration on the GPU. In: ICAPS. pp. 57–64. AAAI (2010). https://doi.org/10.1609/icaps.v20i1.13414
22. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved Multi-core Nested Depth-First Search. In: ATVA. LNCS, vol. 7561, pp. 269–283. Springer (2012). https://doi.org/10.1007/978-3-642-33386-6_22
23. Filippidis, I., Holzmann, G.: An Improvement of the Piggyback Algorithm for Parallel Model Checking. In: SPIN. pp. 48–57. ACM (2014). https://doi.org/10.1145/2632362.2632375
24. Holzmann, G.: The Model Checker Spin. IEEE Trans. Software Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
25. Holzmann, G.: Parallelizing the SPIN Model Checker. In: SPIN. LNCS, vol. 7385, pp. 155–171. Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_12
26. Holzmann, G., Bošnački, D.: The Design of a Multicore Extension of the SPIN Model Checker. IEEE Trans. on Software Engineering **33**(10), 659–674 (2007). https://doi.org/10.1109/TSE.2007.70724
27. Holzmann, G., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: SPIN. pp. 23–32. American Mathematical Society (1996). https://doi.org/10.1090/DIMACS/032/03
28. J.Barnat, Brim, L., Češka, M.: DiVinE-CUDA - A Tool for GPU Accelerated LTL Model Checking. In: PDMC. EPTCS, vol. 14, pp. 107–111. Open Publishing Association (2009). https://doi.org/10.4204/EPTCS.14.8
29. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_61
30. Laarman, A.: Optimal Compression of Combinatorial State Spaces. Innov. Syst. Softw. Eng. **15**, 235–251 (2019). https://doi.org/10.1007/s11334-019-00341-7
31. Laarman, A., Wijs, A.: Partial-Order Reduction for Multi-core LTL Model Checking. In: HVC. LNCS, vol. 8855, pp. 267–283. Springer (2014). https://doi.org/10.1007/978-3-319-13338-6_20
32. Laarman, A.: Scalable Multi-Core Model Checking. Ph.D. thesis, University of Twente (2014). https://doi.org/10.3990/1.9789036536561
33. Laarman, A.W., Langerak, R., van de Pol, J.C., Weber, M., Wijs, A.J.: Multi-core Nested Depth-First Search. In: ATVA. LNCS, vol. 6996, pp. 321–335. Springer (2011). https://doi.org/10.1007/978-3-642-24372-1_23
34. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's Plenty of Room at the Top: What Will Drive Computer Performance After Moore's Law? Science **368**(6495) (2020). https://doi.org/10.1126/science.aam9744
35. Neele, T., Wijs, A., Bošnački, D., van de Pol, J.: Partial Order Reduction for GPU Model Checking. In: ATVA. LNCS, vol. 9938, pp. 357–374. Springer (2016). https://doi.org/10.1007/978-3-319-46520-3_23
36. Osama, M.: GPU Enabled Automated Reasoning. Ph.D. thesis, Eindhoven University of Technology (2022), ISBN: 978-90-386-5445-4

37. Osama, M., Gaber, L., Hussein, A.I., Mahmoud, H.: An Efficient SAT-Based Test Generation Algorithm with GPU Accelerator. J. Electron. Test. **34**(5), 511–527 (2018). https://doi.org/10.1007/s10836-018-5747-4
38. Osama, M., Wijs, A.: Parallel SAT Simplification on GPU Architectures. In: TACAS. LNCS, vol. 11427, pp. 21–40. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_2
39. Osama, M., Wijs, A.: SIGmA: GPU Accelerated Simplification of SAT Formulas. In: IFM. LNCS, vol. 11918, pp. 514–522. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_29
40. 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
41. Osama, M., Wijs, A.: Artifact for paper: Hitching a Ride to a Lasso: Massively Parallel On-The-Fly LTL Model Checking (2023). https://doi.org/10.5281/zenodo.10425384
42. Osama, M., Wijs, A., Biere, A.: SAT Solving with GPU Accelerated Inprocessing. In: TACAS. LNCS, vol. 12651, pp. 133–151. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_8
43. Osama, M., Wijs, A., Biere, A.: Certified SAT Solving with GPU Accelerated Inprocessing. Formal Methods in System Design, Springer (2023). https://doi.org/10.1007/s10703-023-00432-z
44. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN 2007. LNCS, vol. 4595, pp. 263–267 (2007). https://doi.org/10.1007/978-3-540-73370-6_17
45. Pnueli, A.: The temporal logic of programs. In: SFCS. pp. 46–57. IEEE (1977). https://doi.org/10.1109/SFCS.1977.32
46. de Putter, S., Wijs, A., Zhang, D.: The SLCO Framework for Verified, Model-driven Construction of Component Software. In: FACS. LNCS, vol. 11222, pp. 288–296. Springer (2018). https://doi.org/10.1007/978-3-030-02146-7_15
47. Vardi, M., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS. pp. 332–344. IEEE (1986)
48. 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
49. 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
50. 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
51. 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
52. 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
53. Wijs, A.: BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In: CAV, Part II. LNCS, vol. 9780, pp. 472–493. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_26

54. 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
55. 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
56. Wijs, A., Katoen, J.P., Bošnački, D.: Efficient GPU Algorithms for Parallel Decomposition of Graphs into Strongly Connected and Maximal End Components. Formal Methods Syst. Des. **48**(3), 274–300 (2016). https://doi.org/10.1007/s10703-016-0246-7
57. 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
58. 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
59. 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
60. Youness, H., Osama, M., Hussein, A., Moness, M., Hassan, A.M.: An Effective SAT Solver Utilizing ACO Based on Heterogenous Systems. IEEE Access **8**, 102920–102934 (2020). https://doi.org/10.1109/ACCESS.2020.2999382
61. Youness, H.A., Ibraheim, A., Moness, M., Osama, M.: An Efficient Implementation of Ant Colony Optimization on GPU for the Satisfiability Problem. In: PDP. pp. 230–235. IEEE (2015). https://doi.org/10.1109/PDP.2015.59

Towards Safe Autonomous Driving: Model Checking a Behavior Planner during Development

Lukas König[1(✉)], Christian Heinzemann[1], Alberto Griggio[2], Michaela Klauck[1], Alessandro Cimatti[2], Franziska Henze[3], Stefano Tonetta[2], Stefan Küperkoch[1], Dennis Fassbender[3], and Michael Hanselmann[1]

[1] Robert Bosch GmbH, 70465 Stuttgart, Germany
{lukas.koenig,christian.heinzemann,michaela.klauck,stefan.kueperkoch, michael.hanselmann}@de.bosch.com

[2] Fondazione Bruno Kessler, 38122 Trento, Italy
{griggio,cimatti,tonettas}@fbk.eu

[3] CARIAD SE, 38440 Wolfsburg, Germany
{franziska.henze,dennis.fassbender}@cariad.technology

Abstract. Automated driving functions are among the most critical software components to develop. Before deployment in series vehicles, it has to be shown that the functions drive safely and in compliance with traffic rules. Despite the coverage that can be reached with very large amounts of test drives, corner cases remain possible. Furthermore, the development is subject to time-to-delivery constraints due to the highly competitive market, and potential logical errors must be found as early as possible. We describe an approach to improve the development of an actual industrial behavior planner for the *Automated Driving Alliance* between Bosch and Cariad. The original process landscape for verification and validation is extended with model checking techniques. The idea is to integrate automated extraction mechanisms that, starting from the C++ code of the planner, generate a higher-level model of the underlying logic. This model, composed in closed loop with expressive environment descriptions, can be exhaustively analyzed with model checking. This results, in case of violations, in traces that can be re-executed in system simulators to guide the search for errors. The approach was exemplarily deployed in series development, and successfully found relevant issues in intermediate versions of the planner at development time.

Keywords: Autonomous Driving · Model Checking · Industry Application

1 Introduction

Automated Driving (AD) is an ever-growing research field with the potential to make traffic safer and more efficient. Recently, ISO 21448 on Road Vehicles – *Safety of the Intended Functionality (SOTIF)* specified that the number of unsafe, both known and unknown, scenarios should be minimized [30] and the political goal "Vision Zero" aims to practically eliminate traffic fatalities by 2050 [58]. This demonstrates that there is a high interest in and pressure on research to make automated vehicles (AV) safe. AD in the sense of a (future) product has to comply with a vast variety of safety requirements originating from

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 44–65, 2024.
https://doi.org/10.1007/978-3-031-57249-4_3

domains as diverse as physics, law and ethics [4,21,37,40,58]. Improving safety up to human driving level is already a challenging problem [51]. However, it is assumed that AD needs to vastly exceed the "human" safety benchmark since even very few fatalities caused by automated vehicles are hardly acceptable to the public [2,49,61]. The de facto standard today is that tremendous amounts of test drives are supposed to account for the safety of AVs. However, statistical considerations show that deriving confidence in the safety of an AV solely via test drives might indeed require ludicrous amounts of driving [34,44,62].

In this paper, we describe an approach to improve the development of AD software, adopted within the *Automated Driving Alliance (Alliance)*[1] between Bosch and Cariad. Specifically, we consider a *behavior planner (BP; also, tactical BP)*, that controls the high-level actions of an AV (e. g., accelerating, braking, lane changes) based on the perceived state of the environment (e. g., flow of the surrounding traffic). It is part of a system called *Highway Pilot* which realizes automated driving on highways or highway-like roads. The BP is implemented in C++, and is under active development, undergoing repeated updates, with the addition of new features and improvements. Due to time-to-delivery constraints, the *verification and validation (V&V)* activities are supposed to proceed in parallel to the development, preferably in an "observe only" manner.

We enhance the V&V process by integrating automated formal verification techniques, in particular *model checking (MC)* of infinite-state transition systems [10], within the original development environment. The primary purpose is not (yet) to provide arguments for absolute correctness, but to increase the coverage of known unsafe scenarios, currently provided by test drives and simulation. By exhaustively analyzing a huge range of scenarios, MC is largely insusceptible to human bias and can discover corner cases that may be overlooked otherwise.

We face the challenge that MC must become part of the *continuous integration (CI)* process, hence it must be directly connected to the consecutive versions of the BP. This means that manual modeling is to be avoided, in order to enable a seamless connection between development and V&V. Therefore, we integrate suitable mechanisms for the extraction of the BP logic directly into the development environment. Starting from the C++ code of a BP, we automatically derive a model of the underlying logic including the interface to the surrounding software stack. This model is converted into K2, a low-level imperative-style language of the Kratos2 software model checker [23], which is, in turn, converted into SMV, the input language of the symbolic model checker nuXmv [8].

The scenarios for validation are obtained by composing the model of the BP in closed-loop with an *environment model (EM)*. The EM contains rules about the succession of a highway-like traffic scene in terms of the *possible* behavior of a set of free cars which drive in scope of a distinguished BP-controlled car (*ego*), cf. Fig. 1 [2]. EM and BP are integrated such that the BP receives perception

[1] The Alliance has set out to "[build] a state-of-the-art ADAS software platform for use in all Volkswagen Group brand vehicles – and therefore in *one of the world's biggest vehicle fleets*". More information on BOSCH and CARIAD websites.

[2] The graphic was auto-generated by our explainability toolchain, cf. Sec. 3.1.

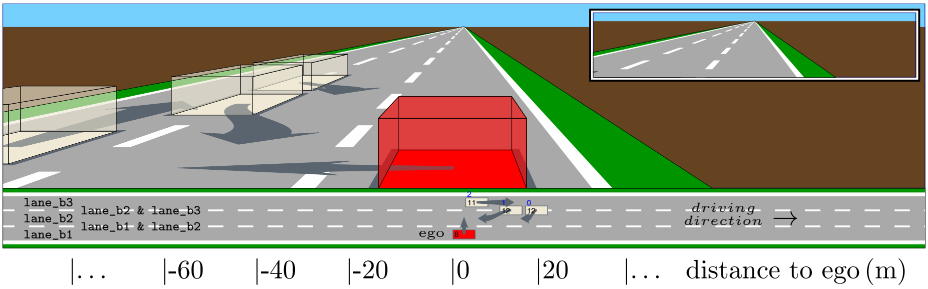


Figure 1: Highway scene illustrating the base scenario. We model three straight lanes unrestricted in longitudinal direction, non-ego positions are calculated relative to ego. Laterally, cars are either on a single lane or between two lanes (during a lane change). Blue numbers: car IDs, black numbers: velocities in $^{m}/_{s}$. Grey arrows: movement from current to next EM iteration.

inputs from the EM and returns an actuation output, which is translated into the movement of the ego. The EM is also written in SMV and can be parametrized to further specify the scenario (e. g., number of non-ego cars, physical features of the cars, driving behavior). The composed model of EM/BP is exhaustively analyzed with nuXmv, which results in a *counterexample (CEX)* in case of the violation of a property, e. g., a collision. The CEX can be re-executed in simulators to guide the search for errors. The approach was exemplarily deployed in series development, resulting in a fully automatic toolchain usable, e. g., in a CI system. The deployment was very successful in that actually relevant issues could be found in intermediate versions of the BP at development time.

Since the full BP code is restricted from publication, we use a mock BP in this publication to illustrate details about the process. Nonetheless, all presented results have been originally obtained with the actual BP used in the Alliance by tracking its development with CI techniques. Though being much simpler, the mock BP is designed to resemble the actual BP in some essential aspects, such that two major bugs found in the actual BP can be reproduced with it. Runtime and performance analyses are performed on data from the actual BP. Using the mock BP, we provide possible fixes to the found bugs in this simpler setup, and show that the model checker then efficiently supplies proof for the now correct functioning of the model of the mock BP, within the simulation by the EM.

Our contributions are: $\langle 1 \rangle$ a self-consistent toolchain, involving automatic processing of the C++ code of a BP, integration with plausible physics and surrounding traffic behavior, MC with nuXmv, as well as, extracting traffic scenes from CEXs for debugging; $\langle 2 \rangle$ presentation and discussion of two safety-relevant issues found in an industrial BP; $\langle 3 \rangle$ by means of re-simulation ensuring that the model soundly captures the real-world system; $\langle 4 \rangle$ analysis of the feasibility of the approach for an actual industrial context, including efficiency.

The remainder of the paper is structured as follows. Sec. 2 provides the basic AD context, as well as the theoretical background required for MC. Sec. 3

describes the methodology underlying the experimental setup. Sec. 4 describes and discusses the experimental results. Sec. 5 lists and assesses literature related to our approach. Sec. 6 provides a conclusion and an outlook to future work.

Additionally, we provide an appendix which is available in an extended version of the paper published as supplementary material. It is intended to facilitate reproducibility, but is not required to follow the presented results. The supplementary material also contains the mock BP and two versions of the EM, as used for the experiments. In an artifact associated to this paper, we provide the full functioning toolchain, i. e., all code and software (except for the Alliance BP) in a state as used for the experiments.

2 Background

The task of driving automatically can typically be segmented according to the classic "sense – plan – act" paradigm [7, 38]. The *sense* part comprises perception of the environment using sensors like camera, lidar or radar, and fusing their measurements into a *model of the environment* (e. g., [41]).This model contains all available information of the AV's surroundings, e. g., lanes to drive on, the state of other traffic participants (e. g., position, velocity, acceleration) and predictions of their motion. This is the basis for planning the motion of the controlled AV. The *plan* part can be divided into three steps [3]: First, a *strategic planner* decides about the global navigation, i. e., the route to follow. Then, the *tactical BP* decides between available maneuvers, e. g., lane following or lane change. Finally, the *trajectory planner* calculates a desired trajectory which eventually results in a sequence of desired accelerations and curvatures. In the *act* part, these signals are forwarded to the actuators, thus accomplishing the actual driving on the road. We focus on the tactical BP which, for MC, is decoupled from the other software modules in the sense – plan – act pipeline.

Model Checking of Infinite-State Symbolic Transition Systems. The system under analysis is derived in several steps from the BP and the EM code, cf. Sec. 3, and finally represented as a symbolic transition system expressed using quantifier-free formulæ in first-order logic modulo theories (for further reading, refer to, e. g., [5]). We work in the setting of many-sorted first-order logic, and we assume the usual first-order notions of interpretation, satisfiability, validity, logical consequence, and theory, as given, e.g., in [17]. Unless otherwise stated, when we talk about a logical formula $\varphi(X)$, we mean that φ is a quantifier-free first-order formula whose free variables are included in the set X, and using symbols from the theory of (linear) arithmetic (with their usual interpretation). For example, $\varphi(\{x_1, x_2\}) := (3x_1 > 0) \wedge (x_2 + x_1 < -5)$. A symbolic transition system $S = \langle X, I, T \rangle$ is a tuple, where X is a set of (state) variables, $I(X)$ is a formula representing the initial states, and $T(X, X')$ is a formula representing the transitions, where X' is the set of variables representing the next state of the system. A state s of a transition system S is an assignment to the state variables X; a path (trace) π of S is a possibly infinite sequence $\pi := (s_0, s_1, \ldots, s_i, \ldots)$

of states s_i such that $I(X)$ is true under the assignment s_0, and $T(X, X')$ is true under the assignment s_{i-1}, s'_i for all $i > 0$ in π, where s' is the assignment obtained by replacing each $x \in X$ with the corresponding $x' \in X'$. We say that a state s is reachable in S if and only if there exists a path π of S such that $s \in \pi$. Given a formula $P(X)$ over the variables X, the invariant verification problem for S and P is the problem of checking if all the reachable states of S satisfy the formula P. In that case, we say that S satisfies P, written as $S \models P$.

Tools. For solving the core MC problem we use nuXmv [8], a state-of-the-art symbolic model checker for both finite- and infinite-state transition systems. It supports the verification of invariant and LTL properties using a combination of efficient algorithms based on Boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT) [11, 13, 24]. Each verification engine can be used unbounded or bounded (in which case only disproving is possible). For details see the documentation [8]. The transition system for nuXmv is defined using an extension of the standard SMV language for finite-state systems (simply SMV in the following). An example of the syntax is reported in Alg. 2 of the appendix.

To automatically derive a nuXmv transition system out of imperative C++ code, Kratos2 can be used as an intermediate step. Kratos2 [23] is a tool for the automatic verification of imperative programs, using nuXmv as its main verification engine. The native language of Kratos2 is a verification language called K2 (similar to Boogie and Why3 [43]) that provides a well-defined and unambiguous formal semantics suitable for verification. An example program in K2 is shown in Alg. 3 of the appendix. The resulting K2 program is then transformed by Kratos2 into SMV and verified with nuXmv.

Parsing C++ code and creating the K2 model out of it is done by a prototypical software called vfm which was developed within the Alliance (cf. artifact).

3 Methodology

This section describes the proposed workflow and justifies how the respective components were chosen to establish an industry-ready setup.

3.1 Overview

Our toolchain consists of several components centered around a BP under analysis, cf. Fig. 2. We impose MC on top of a functioning software and in scope of an established development process. Therefore, the BP's source code, (1) in the figure, is considered largely *immutable*, meaning that we cannot change it to our likings, but developers *can* change it at any time outside of our control.

The BP cannot be checked on its own, since its behavior *within a traffic situation* is of major interest. The entity responsible for providing an initial traffic scene, and keeping track of how it evolves according to the BP's actions, is the EM (3). It is given in SMV language, i. e., we can directly identify it with a transition system $S_E := \langle X_E, I_E, T_E \rangle$. In a first processing step, the BP and the EM are parsed separately to create an intermediate representation (4) which contains an internal description of the BP logic including its interface towards the EM. The respective mapping between EM and BP is provided by the *type abstraction layer (TAL)* (2), which is embedded into the C++ code of the BP via comments (see below). The intermediate representation is translated into a K2 version of the BP logic (6) which, in turn, is translated by Kratos2 into an SMV representation (7; cf. discussion in Sec. 4.3 about making this detour instead of directly translating from C++ to SMV); we identify the SMV representation with the transition system $S_P := \langle X_P, I_P, T_P \rangle$; additionally, the interface information is used to generate the full integrated transition system, called $S_{E \sqcup P}$. On SMV level, it is constructed to include and connect S_E and S_P as two separate modules within a `main` module (5); it is also the place to add specifications to check. This file is then handed to the nuXmv model checker for the actual MC task. Depending on the result, we either terminate the process after MC, if the specifications are fulfilled ("OK"), or otherwise ("FINDING") trigger the creation of a CEX (8), which is a witness of the specification being violated. It provides a trace within the checked transition system $S_{E \sqcup P}$ in the course of which the specification does not hold. In our case, the sequence of variable values along this trace provides information about $\langle 1 \rangle$ the traffic situation in which the violation occurred and $\langle 2 \rangle$ the BP actions in this situation. $\langle 1 \rangle$ is used to generate visualizations of the traffic scene in question, and to further process the resulting scenarios of interest, by using the *Open Scenario 2 (OSC2)* format [1] as underlying representation (9). Figs. 1, 4 and 5 were created using this functionality. $\langle 2 \rangle$ *could*, in principle, be used for a deep analysis of the error, such as tracking back which lines in the original C++ code are involved in the violation. This is planned to be done in future, but is not possible yet.

The retrieval of the transition system to check needs to be flexible enough to adapt to future changes of the BP, which are expected to happen frequently during development. Therefore, the EM should not be customized towards a specific BP, but rather provide a generic interface which works with a variety of BP instances. The toolchain then runs fully automatically by attaching flexibly to different BPs, as long as the EM is "suitable" for them, i. e., all required data is available in the EM's generic interface, agnostic of naming. For example, the BP might require as input its own velocity in a variable called `agent.v` which the EM provides as `ego.v`. The TAL connection can be established by adding an "`aka`" tag as comment to the BP variable `agent.v` (for details cf. the appendix).

Therefore, the remaining manual effort consists of adding or adapting the TAL information when the BP changes. This is expected to be mostly a one-time effort, since once the connection to EM variables is established, it only needs to be changed if new data is added to the interface (*not* on mere re-usage or renaming). This happens occasionally, but is not very common in practice. Only if the EM becomes unsuitable for a developed BP, i. e., when a non-existent signal is requested, this implicates the fairly high effort of adapting the EM.

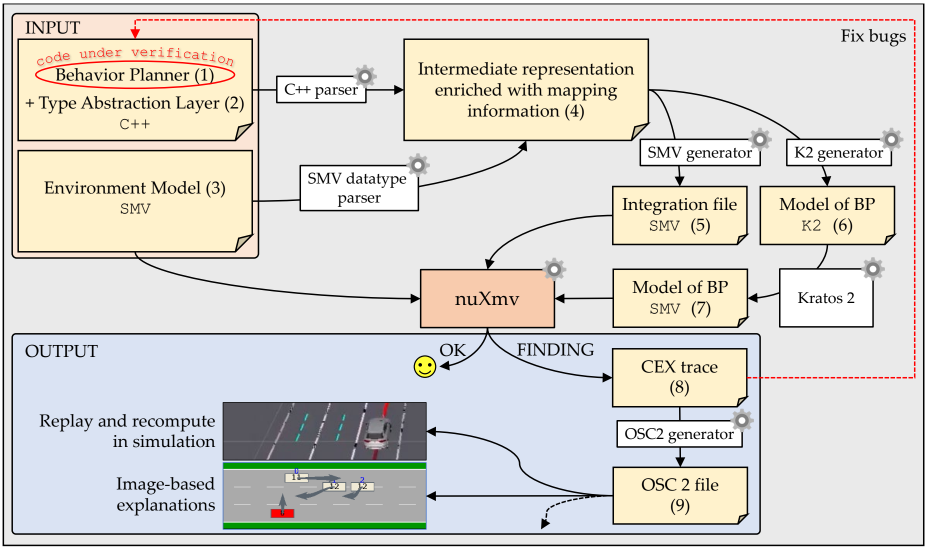


Figure 2: Overview of the presented toolchain. The BP to check (1) is the main input. The EM is given as transition system in native SMV language (3). nuXmv (center) is run on an integration of BP/EM created in several steps (3/5/7). Visualizations and simulations are derived from CEXs (9).

3.2 Environment Model

Since we cannot model-check the full software stack the BP is part of, with perception on one side and actuation on the other, the BP needs to be directly fed with mock data from a simulated environment, and its output needs to be propagated back into this environment in a closed-loop manner. This is accomplished by the EM, which creates an initial state of the environment, and then progresses by supplying the current state as perception input to the BP and deriving the next state based on its output. Therefore, the EM maintains physical states of all agents, and applies laws of physics and possible driving behavior to them.

We assume a *perfect environment*, i. e., perfect knowledge without any sensor uncertainties and perfect behavior without actuatory imprecision. We use a highway-like road model with 3 lanes, allowing traffic only in one direction, as illustrated in Fig. 1. We support an arbitrary number of non-ego vehicles[3], each of which has physical properties such as relative position to ego, velocity and lane association. A vehicle is associated to either a single lane or, during a lane change, to both its source and its target lane (cf. `veh[1]` and `veh[2]` in Fig. 1).

In each verification step, corresponding to 1s, all non-ego vehicles may choose a new acceleration $a \in \{-8, -7, \ldots, 6\}$ $^{\mathrm{m}}/_{\mathrm{s}^2}$, which is used to update the velocity and relative position simultaneously. In addition, each non-ego vehicle may

[3] A generator is used to create an EM with an adjustable number of non-ego cars for each MC run (cf. artifact).

choose to perform a lane change. A lane change is executed in two stages. In the first stage, the non-ego vehicle signals the lane change via turn indicators but is still located on its source lane. The second stage is the transition from source to target lane, i. e., it starts when the lane marking is initially touched until the non-ego vehicle is contained entirely in the target lane. During both stages, the non-ego vehicle may non-deterministically decide to abort the lane change. If the decision to abort is made while being in the second stage, the non-ego vehicle moves back to the source lane. The durations of both stages of the lane change and of the return to the source lane when aborting are chosen non-deterministically from intervals of possible values (cf. section marked "`Begin of lc parameterization`" in the published version of the EM). Thereby, we enable to verify a high variety of lane changes in the MC process, including much tougher ones than expected to usually happen on real roads, with the goal to over-estimate (rather than under-estimate) violations by the planner in the sense discussed in Sec. 4.3. On the other hand, we do prohibit excessively malicious behavior by ensuring that ego is able to prevent a collision by staying in its lane and reacting instantaneously with a deceleration of up to $-8^{\mathrm{m}}/_{\mathrm{s}^2}$.

The ego vehicle tracks objects in its proximity via so-called *gaps*, defined by a front and a rear vehicle on its own and its neighboring lanes, cf. Fig. 3. This data needs to be provided by the EM, as well. Depending on lane availability, there can be up to three gaps, one is always in the ego lane, the other two can be in the left or right lane next to ego, respectively, if available. For each car in the gaps, information such as relative distance to ego, velocity and acceleration are stored which can be used by the BP for decision making. If one of the positions is not filled, e. g., if the next car is out of perception range, this is indicated by a special value. The full SMV code of the EM, as well as a more in-depth description are published as supplementary material.

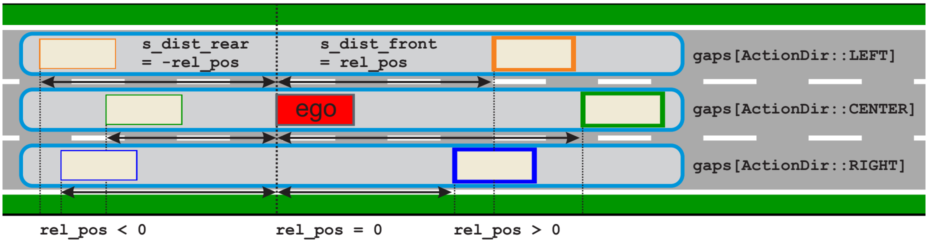


Figure 3: Illustration of gaps tracked for ego. The *gap data structure* provides information about the free space which ego drives in on its own lane or could dive into when performing a lane change, in terms of the two cars limiting this space to the front and rear. A gap contains the IDs, distances, velocities, accelerations etc. of the closest cars to the front and rear on the respective lane.

3.3 The Original and Mock BPs

We consider two BPs, $\langle 1 \rangle$ the actual BP currently under development for the Alliance project, and $\langle 2 \rangle$ the mock BP which is a simplified version of $\langle 1 \rangle$. The code considered for $\langle 1 \rangle$ is an excerpt of the project BP containing the logic for a *lane change decision towards the "fast" lane (LCfast)* (left assumed; as opposed to *LCslow*). It consists of more than 1000 lines of C++ code. The mock planner $\langle 2 \rangle$ is much smaller (about 100 lines) and can be inspected in its entirety as supplementary material. It contains a simple version of the logic for LCfast, logic for LCslow and a simple longitudinal control.

The interface towards the EM is equal for $\langle 1 \rangle$ and $\langle 2 \rangle$ regarding LCfast. It uses the *gap structure* as input and returns as output the decision whether or not to initiate a lane change towards the fast lane. The mock BP requires additional data for the longitudinal control, as well as some signals used to fix the found issues; we mostly disregard these differences in the following, for simplicity. The exact interfaces are given in the appendix.

4 Experiments

Our goal is to demonstrate that the proposed approach can practically guide development processes of a BP in industry. Therefore, we need to show that $\langle 1 \rangle$ our setup derives valuable insights for $\langle a \rangle$ development and/or $\langle b \rangle$ release; $\langle 2 \rangle$ it does this in acceptable runtime, $\langle 3 \rangle$ it does not overly disturb every-day development, and $\langle 4 \rangle$ its results are self-explanatory to developers and V&V experts. In Sec. 4.1 we analyze two major issues found early during development in the Alliance BP $\langle 1a \rangle$. We also comment on the implications of proofs of error-freeness on the model level, which may be of high relevance for possible future release argumentation $\langle 1b \rangle$. Efficiency of bounded and unbounded MC $\langle 2 \rangle$ is analyzed in Sec. 4.2. Sec. 4.3 discusses the results and elaborates qualitatively on the topics $\langle 3 \rangle$ and $\langle 4 \rangle$.

4.1 Disproven Specifications with Counterexamples

The two issues we describe were found by checking for the invariant property `!blamable_crash`. It is true if ego's bounding box never overlaps with the bounding box of any car in front of ego. nuXmv showed in both cases that it does not hold true, which led to the generation of CEXs. They reveal violations of the BP in typical highway-traffic, which we named *Lead Vehicle Occlusion* and *Double Merge*[4]. The CEXs have been further processed to show the succession of the traffic scenes that lead to the violations (Figs. 4 and 5), and to confirm the issues in simulation by using the full underlying software stack (cf. Sec. 4.3).

[4] Note that these issues certainly could have been detected with regular V&V techniques, as well, however, with considerably greater effort.

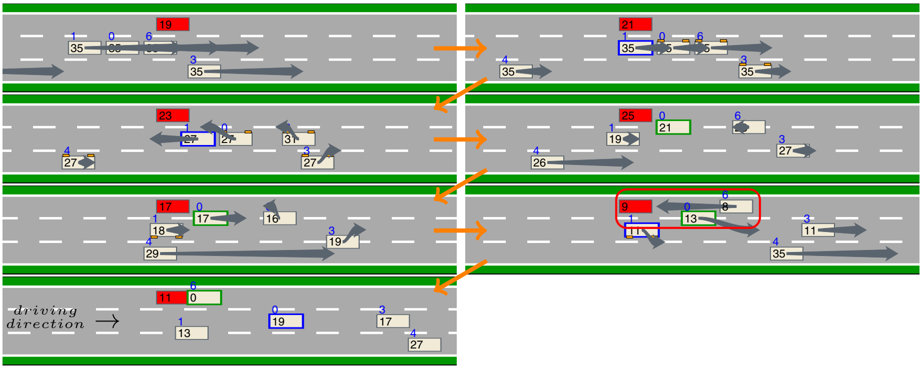


Figure 4: Full CEX trace of crash caused by ego overlooking hard brake ahead due to another car "cutting out". The highlighted portion of the second-to-last image shows how the car in the lead position (0) pulls away ("cuts out") while another car ahead (6) brakes hard, unnoticeable by ego due to the way lead vehicles are tracked. Instead of braking, ego even accelerates, since car 0 also accelerates and departs.

Lead Vehicle Occlusion. Since ego bases all decisions on the *gap structure*, it cannot "see" possible cars in front of the one it finds as front car in one of the gaps. On the other hand, the surrounding traffic is allowed to behave arbitrarily rudely, as long as ego has, upon immediate action, a chance to avoid a crash (cf. Sec. 3.2). Considering this, nuXmv reported the situation illustrated in Fig. 4 as a CEX[5] . The seven snapshots correspond to the full path given in the CEX (i. e., it takes at least 7 steps from an initial to a violating state). The actual crash occurs in the last step, but the underlying wrong decision is made already in the sixth step. Here, car 0 is partially on ego's lane, and is, therefore, considered the lead car in the `CENTER` gap (indicated by a green frame). On the other hand, car 6 is farther ahead, and, at this point invisible to ego (in a logical, *not* a perception sense). The problem arises from car 6 performing a hard brake, while car 0 budges towards the middle lane ("cuts out"). Considering car 0 as lead car, which actually departs to the front, ego misinterprets the scene to clearing up, and accelerates rather than braking, which leads to the crash. The CEX points to a fundamental problem caused by considering only a single lead vehicle per gap. (Indeed any constant number n of lead vehicles won't fix the underlying problem, assuming unrestricted velocities and/or restricted braking capabilities. Imagine a row of $n+1$ cars in front of ego, with the furthest one standing still while the other n cut out. In practice, safety distance needs to be adjusted such that braking in time is possible even in the worst case scenarios.)

Double Merge. This issue is caused by ego observing only cars on neighboring lanes, which can lead to conflicts if cars from two lanes apart change towards

[5] The CEX can be produced with any number of non-ego cars ≥ 2. We choose to show non-minimal examples here to give intuitive insight into the functioning of the EM.

a neighboring lane while ego itself is changing towards this lane. This issue is related to the gap structure, too, but points to the lateral, rather than longitudinal limit of the structure. Fig. 5 shows how the violation unfolds in the course of 6 steps. The actual problematic decision already occurs in step 2 where ego decides to change the lane towards the middle, as displayed by the indicators turned on[6]. At this point, ego waits for two steps before actually starting the lane change. During this whole period, the middle lane appears to be free (although all three non-egos indicate at some point to be starting a lane change, which is actually performed by cars 1 and 2, but aborted by car 0). When ego actually does the lane change, car 1 happens to finish its own and ends up colliding with ego. While it could be argued that the other car could have avoided this situation, too, ego is at least partially to blame for the crash.

As opposed to the Lead Vehicle Occlusion issue, which was more fundamental in nature, this issue is strongly connected to the exact way the BP performs a lane change. Having a more flexible cancellation mechanism, or including cars *between* a neighboring lane and the one next to that into the gaps, could, for example, easily avoid this type of issues.

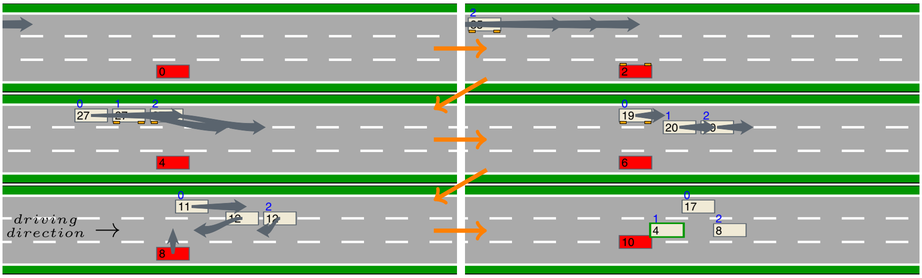


Figure 5: Full CEX trace of crash caused by ego not noticing a car merging towards the middle lane while itself merges there. The issue is caused by ego not looking further than one lane to the left when deciding to change a lane, and delaying the actual lane change after the decision for 2 steps.

Comment on Specifications Proven to Hold. We used unbounded MC for validating the EM by proving many of its desired properties to hold; for example, "ego is never 'forced' into a collision by a non-ego vehicle", and "the vehicles in the gaps are always the ones closest to ego on the respective lane". Also, when fixing the two issues described above, the mock BP can be shown to fulfill the `!blamable_crash` specification, cf. the artifact published with this paper. However, it is currently unclear what such proofs of *absence* of errors on the model level mean for the real system, see discussion in Sec. 4.3.

[6] The actual Alliance BP, whose behavior is shown here, supports lane changes upon driver request, while the mock planner performs lane changes towards the fast lane only to overtake a slower vehicle. Thus, the issue can, for the Alliance BP, be produced with at least one, for the mock BP with at least two non-ego vehicles.

4.2 Runtime Analysis

For the runtime analysis we are particularly interested if the model checker terminates within a time limit "acceptable" for typical aspects of the V&V workflow utilized in the Alliance (and probably similarly in other AD projects). We agreed that for the following standard situations the respective approximate runtimes would be acceptable. For runs performed on each *pull request (PR)*, e. g., as gatekeepers: $\lessapprox$ 2 min. For runs performed during each *nightly job (NI)*: $\lessapprox$ 4 h. For runs performed for a *release (RE)*: 5 d and more, may be acceptable.

Assuming an ever-changing development setting, it seems misguided to compare rigorously clocked runs of specific software versions to find the best EM/BP setup. To present a broader picture, we rather list the average runtimes of all runs made during a fairly mature phase of the study with somewhat changing BP and EM versions. This mature phase is defined as starting at the point from which no major fixes to the EM occurred anymore. We do not analyze the differences in implementations, but deliberately provide by this means a general impression of practicability, in a setting close to how it is expected to emerge in real industrial contexts.

We focus on the checked property `!blamable_crash`, using the Alliance BP (i. e., the expected result is a CEX for all runs; its creation time is included in the listed times) and do not restrict the behavior in any way. Tab. 1 summarizes the runtimes obtained for up to 10 non-ego vehicles with bounded or unbounded MC. We estimate many of the numbers in the table for the sake of a comprehensive, rather than overly exact, image, as they reflect our experience very well. Note, however, that the results only include runs that terminated at all (as opposed to runs that needed to be aborted, usually due to excessively long runtime). This particularly distorts the rows with three and four non-ego cars and unbounded MC, which – especially with earlier versions of the EM – frequently ran for days and weeks without finishing. Considering future changes to the BP, these rows need to be taken cautiously, although we also expect further improvements to the EM, which, in the past, greatly reduced runtime (cf. Sec. 4.3).

The general takeaway is that the overall setup is sufficiently efficient to be used in an industrial context, largely even at PR level. For future release argumentation, using unbounded MC seems to be possible with up to 4 non-ego cars. Note that adding non-ego cars beyond two did not gain any new insight into issues of the BP in all runs so far. Discussions are ongoing which number of non-ego cars is sufficient to reflect all relevant situations on a straight 3-lane highway, in terms of BP logic, with 4–5 being among the highest estimates.

Considering further improvements, the setup seems to be extendable towards more complex road topologies. However, we experienced the runtimes to be fairly volatile. It may well happen that marginal changes to the BP and/or the EM yield a factor of 2 − 3 in runtime (cf., e. g., the high variance for the "2-car unbounded" case). While this is still acceptable for our context in many regards, it needs to be considered for each individual case, cf. also discussion in Sec. 4.3.

Table 1: Runtimes for MC runs clustered along number of non-ego cars and bounded vs. unbounded MC. The "bounded" rows comprise runs that were not all tracked and analyzed exactly, but overall yield a fairly clear picture (runtimes estimated here; as well as for the 2-car unbounded row). The "Suitable for" row is a best guess based on the given numbers.

# Non-egos	MC type	# Finished Runs	Average runtime	Suitable for
1	bounded	> 50	$\lesssim$ 1 min	PR/NI/RE
2	bounded	> 30	$\lesssim$ 1 min	PR/NI/RE
3	bounded	≈ 10	$\lesssim$ 2 min	PR/NI/RE
4	bounded	≈ 10	$\lesssim$ 2 min	PR/NI/RE
5	bounded	≈ 10	$\lesssim$ 5 min	(PR)/NI/RE
10	bounded	2	19.6 min	NI/RE
1	unbounded	6	30 s	PR/NI/RE
2	unbounded	> 50	$\approx$ 1–7 h	(NI)/RE
3	unbounded	2	16.6 h	RE
4	unbounded	1	5.7 d	(RE)

4.3 Discussion

The presented results indicate that MC can be used in a real industrial context to guide the iterative development of a BP. The approach is overall suitable to be used on top of the regular development processes for AD without considerably interfering with them, and to guide this development by detecting safety-relevant issues earlier than with plain testing, thus reducing futile testing time. It is important to recognize the complete lack of human bias in this process. Particularly, the type of scenario is not provided in any way, but all scenarios producible by the respective EM/BP combination are inspected. The toolchain works mostly self-governed by automatically extracting the BP logic from C++, and creating self-explanatory visualizations and test cases out of CEXs.

Current Limitations. *Relevance for the real world:* MC provides either $\langle 1 \rangle$ a proof that the (extracted version of the) BP logic complies with a given specification (in relation to the EM), or $\langle 2 \rangle$ a proof that this is not the case, which is complemented with a CEX. These proofs *relate* to statements about the correctness of the BP behavior, i. e., the BP is supposed to be "correct" $\langle 1 \rangle$ or "incorrect" $\langle 2 \rangle$ w. r. t. to the checked specification when steering an actual car on the road. However, full confidence in these statements requires not only to prove that the EM itself as well as the extraction of the BP logic are correct, but also that the EM reflects real-world traffic, physics and perception/actuation "adequately". Assuming any deviation from the real behavior of the system means that the EM would need to over-estimate violations for $\langle 1 \rangle$ and under-estimate them for $\langle 2 \rangle$. Therefore, it is not possible to provide an EM which simplifies the actual system behavior in any way, and evidently entails only true statements about both correctness and violations of the BP. It may be possible, though, to strengthen one direction up to a level where some notion of confidence in its validity can be derived. Then, it appears reasonable to let the EM over-estimate, rather than under-estimate violations such that error-freeness $\langle 1 \rangle$ is reflected accurately. This is due to the additional information provided by CEXs for $\langle 2 \rangle$. While the proof of error-freeness is a somewhat final statement, false positives of violations *can* be ruled out by re-simulating the CEXs with the actual software stack. This additional check was performed for all the CEXs presented in this paper. The EM is built with the *intent* to rather over-estimate than under-estimate violations, cf. Sec. 3.2. However, at present we do not investigate more profoundly to what extent this is actually accomplished and what exactly, consequently, a proof of correctness implies on BP level.

Number of non-ego cars: According to the results presented in Sec. 4, a natural measure of performance of the approach is the maximum number of non-ego cars that can be processed in a time acceptable for one of the presented use cases. For unbounded MC, this number is currently limited to about 4 non-ego cars, if considering the "release" use case. Using bounded MC, up to 10 non-ego cars can be processed quite efficiently, i. e., easily suitable for a regular "nightly job". These limitations are relativized by $\langle 1 \rangle$ the potential to further improve the EM, and $\langle 2 \rangle$ by arguing that on a straight highway most critical traffic situations involve only few directly involved participants.

Runtime volatility: Another limitation is the high sensitivity of nuXmv to changes in the checked transitions systems, cf. Sec. 4.2. Small differences can result in a factor of $2-3$ in runtime. It needs to be considered for each individual case if this is acceptable. A related issue, which can be problematic in an industrial context, is not knowing the remaining time of a run ("will it finish in a minute or run for another two weeks?"). For bounded MC, this question intensifies to whether the run will finish at all, which it never can if the specification is fulfilled. In productive use, long-enduring runs probably need to be aborted after some time. There is currently no general solution for this problem.

Force to fix bugs right away: The characteristic of nuXmv, to always produce essentially the same CEX as long as an issue is not fixed, is somewhat problematic for practical application. In theory, a discovered issue is supposed to be fixed immediately, before going on discovering and fixing others. In practice, however, it may well happen that an issue is not easily fixable in a solid and process-abiding way while development still must go on in other directions. An "ignore" option is desirable which lets users "skip" a CEX and trigger the generation of "profoundly" new ones. In fact, we were able to present two differing CEXs with the same version of the Alliance BP in Sec. 4.1 only due to the fact that in one case lane changes were prohibited for ego. These sort of tricks can help to work around this issue.

Range of applicable BP types: Technically, we only assume that the BP is written in an imperative language, i. e., C++ for now. Our approach is not limited to deterministic BPs; the presented EM is already non-deterministic, e. g., in modeling the behavior of other cars. However, the presented toolchain is not directly applicable to most AI-based approaches. This is not a structural limitation (a C++ implementation of a deep neural network could, in principle, be fed into our toolchain), but seems infeasible, at today's state of research, due to well-known issues regarding runtime and numerical instability of these approaches in combination with non-probabilistic MC. An extension to probabilistic MC or probability-based models like POMDPs is conceivable, but has so far not been part of our investigations.

Lessons Learned. *A hybrid solution is required:* As pointed out before and in Sec. 5, out-of-the-box solutions (e. g., C model checkers, as well as a pure combination of Kratos2 + nuXmv) can deliver some aspects of the presented toolchain. However, important features like the closed-loop integration of an EM, and the CEX explainability functionality need to be customized for the problem at hand. Particularly, this made it necessary to use a specific C++ parser for the creation of the interface between EM and BP. In future, such a functionality could become a native feature of a model checker.

"Detour" over Kratos2 and SMV is beneficial: As shown in Fig. 2 (page 7), the BP code is first translated from C++ into K2, and from there into SMV, where it is connected to an EM in SMV. Here, two "shortcuts" are thinkable, $\langle 1 \rangle$ the SMV representation of the BP could be generated directly from C++, and $\langle 2 \rangle$ the explicit representation in SMV could be overall omitted by rather implementing the EM in C++, as well. However, the presented path makes full use of the imperative MC functionality provided by Kratos2 and the rich syntax of SMV. Both shortcuts have been tried out in the beginning and abandoned later, since the current setup outperformed them by far.

Explainability can be simple: We experienced our method of extracting traffic scenes from CEXs and further processing them towards visualization and testing as highly effective for $\langle 1 \rangle$ quickly explaining bugs to both BP developers and, during early phases, EM designers, $\langle 2 \rangle$ for re-simulation of the results with the actual full software stack to confirm the MC results, and $\langle 3 \rangle$ for further processing the scenes, for example, for test case generation. Generally, there are a number of further explainability methods to explore, such as translation to natural language [9, 31–33], or, specific to this use case, a further investigation of which code lines of the BP lead to a violation.

General scalability: The approach scaled well across the different versions of the Alliance BP, as well as the mock BP (i. e., runtime differences were insignificant). However, none of the BPs' logic so far contained loops or recursion, which might significantly increase complexity.

Granularity of abstraction is an open question: All presented results were produced with a time scale rasterized to one iteration per second in the EM. This shows that this granularity of abstraction can produce useful findings. Going towards an argumentation of "error-freeness", it needs to be more profoundly inspected what granularity is actually required for which types of statements.

Every-day development is not impaired: In practice, for typical "every-day" changes to the code adapting the tags was simple enough to be correctly done by non-MC-experienced developers. More complex changes were done by an MC expert, or split into work packages. The additional effort was overall tolerable.

5 Related Work

In this section, we discuss related work on formal methods and how our work complements them. In general, formal methods aim to prove safety properties of algorithms theoretically. Redfield et al. give a good overview of the challenges within this field [50]. Formal methods include temporal logic encodings [63], monitoring [45,47], theorem proving [46,52] (see also overview in [48]) and MC [5]. In the following, we will focus on approaches that could be used for the safety assessment of AV. Notably, only few works actually incorporate such methods into industrial production or for the verification of (parts of) products [16,18,29]. A more extensive body of research focuses on theoretically adopting formal methods to the problem landscape given in the automotive industry, cf. overview in [59,64].

A typical practical issue, preventing broader adoption, is the interface between the problem to solve and the theoretical tooling. The input languages for MC are often quite low-level and lacking a lot of features of modern programming languages like object-oriented features, dynamic data structures, etc. [6, 8,15,27,42]. Therefore, many approaches embed MC into modeling approaches based on domain specific languages and translate from such higher-level representations to the model checkers [14,22,26,56]. In our setting, such approaches cannot be applied as the BP under verification is only available in source code. Other industrial MC applications involve a manual translation step of (part of) the code under verification into model checker language [20,35]. However, during the development process in a fast-changing environment this is not feasible. Especially in early development stages, not only the code under analysis, but also interfaces and data structures change rapidly.

For handling cases where the system under verification is only available as source code, several MC approaches taking source code as an input have been developed. Existing MC approaches in this category mostly focus on (subsets) of C code [12,55,60]. Similar to our strategy, most of these approaches translate the model of the code into a suitable mathematical input language for the model checkers [28, 36]. Checking C++ code requires more sophisticated approaches, since object-orientation and other specific C++ concepts introduce additional layers of complexity. One MC method checking C++ code is DIVINE, which also includes, e. g., exceptions [54]. It could be used as an alternative backend for our approach. The Bogor framework [53] provides means for creating software model checkers for object-oriented languages, but is only available for Java.

Verifying the decisions of the BP in different traffic situations requires to represent these in the EM. While we manually implemented the EM in our approach, there exist approaches using ontologies for representing features of an EM [19]. They represent abstract scenes of driving scenarios and use them in logic-based reasoning systems. Such approaches could be combined with our approach in future works for automatically configuring the EM.

Another approach is to directly include all safety constraints into the planning itself [25]. A popular example is to use reinforcement learning and integrate safety constraints into the learned policy [39, 57]. However, reinforcement learning approaches suffer from a strong dependency on the specific environment the agents interact with. Thus, one can never ensure safety in all possible corner cases.

6 Conclusion

In this paper we described the application of automated verification to improve the development of an actual industrial *behavior planner (BP)*. Complementing the regular validation process based on simulation and test drives, we developed a mechanism to automatically extract from C++ code the model of the underlying BP logic. This model can be integrated with a model of the environment (features of the road and the other vehicles), in a closed-loop manner. This allows to deal in a seamless way with multiple versions of the BP, as they occur during development, and to exhaustively analyze a huge variety of scenarios. In case of violations, the model checker is able to produce traces that can be re-executed in simulators of the original system to guide the search for errors. The approach was exemplarily deployed in series development, and successfully detected multiple relevant issues of intermediate versions of the BP at development time.

There are several directions for future activity. First, we will broaden the scope of the environment modeling to more general scenarios. Second, we will investigate the gray area between the exhaustive exploration of a set of scenarios and a general guarantee of correctness in the real world. Finally, we aim at the application of the methodology to other software components. In fact, it is often the case that the development and the validation teams proceed in parallel. In this respect, the context of application of automated model extraction described in this paper can be considered paradigmatic.

Acknowledgements. A. Cimatti, A. Griggio and S. Tonetta acknowledge the support of the project "AI@TN" funded by the Autonomous Province of Trento and of the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU.

References

1. Amid, G.: ASAM OpenSCENARIO V2.0.0. Tech. rep., Association for Standardization of Automation and Measuring Systems (2022)
2. Aptiv, Audi, Baid, BMW, Continental, Daimler, Fca, Here, Infineon, Intel, Volswagen: Safety first for automated driving. Tech. rep. (2019), https://www.aptiv.com/docs/default-source/white-papers/safety-first-for-automated-driving-aptiv-white-paper.pdf, accessed: 25.09.2023
3. Artuñedo, A., Godoy, J., Villagra, J.: A decision-making architecture for automated driving without detailed prior maps. In: 2019 IEEE Intelligent Vehicles Symposium (IV). pp. 1645–1652. Paris, France (2019)
4. Audi AG, Audi Kommunikation: Audi SocAlty Study (2022), https://www.audi.com/content/dam/gbp2/company/research/audi-beyond/2021/AUDI_SocAITy_Study_dgtl_1201_English_small.pdf, accessed: 25.09.2023

5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge, MA, USA (2008)
6. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems. pp. 125–126. QEST 2006, IEEE Computer Society, Los Alamitos, CA, USA (Sep 2006). https://doi.org/10.1109/QEST.2006.59
7. Brooks, R.A.: A robust layered control system for a mobile robot. IEEE Journal on Robotics and Automation **2**(1), 14–23 (1986)
8. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: Computer Aided Verification. CAV 2014 (2014)
9. Cherukuri, H., Ferrari, A., Spoletini, P.: Towards Explainable Formal Methods: From LTL to Natural Language with Neural Machine Translation. In: Gervasi, V., Vogelsang, A. (eds.) Requirements Engineering: Foundation for Software Quality. pp. 79–86. Springer International Publishing, Cham (2022)
10. Cimatti, A., Griggio, A., Mover, S., Roveri, M., Tonetta, S.: Verification modulo theories. Formal Methods in System Design (2023). https://doi.org/10.1007/s10703-023-00434-x
11. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. Formal Methods in System Design **49**(3), 190–218 (2016)
12. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2004. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer, Berlin, Heidelberg (2004)
13. Daniel, J., Cimatti, A., Griggio, A., Tonetta, S., Mover, S.: Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. CAV 2016. Lecture Notes in Computer Science, vol. 9779, pp. 271–291. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_15
14. Daw, Z., Cleaveland, R., Vetter, M.: Integrating model checking and uml based model-driven development for embedded systems. In: Automated Verification of Critical Systems 2013. Electronic Communications of the EASST, vol. 66 (2013). https://doi.org/10.14279/tuj.eceasst.66.888
15. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A storm is coming: A modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 592–600. Springer International Publishing, Cham (2017)
16. Eberhart, C., Dubut, J., Haydon, J., Hasuo, I.: Formal verification of safety architectures for automated driving. In: 2023 IEEE Intelligent Vehicles Symposium (IV). pp. 1–8 (2023). https://doi.org/10.1109/IV55152.2023.10186763
17. Enderton, H.B.: "A Mathematical Introduction to Logic". Academic Press, Boston, MA, USA, 2. edn. (2001)
18. Farrell, M., Bradbury, M., Fisher, M., Dennis, L.A., Dixon, C., Yuan, H., Maple, C.: Using threat analysis techniques to guide formal verification: A case study of cooperative awareness messages. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 471–490. Springer International Publishing, Cham (2019)
19. Fuchs, S., Rass, S., Lamprecht, B., Kyamakya, K.: A Model for Ontology-Based Scene Description for Context-Aware Driver Assistance Systems. In: 1st International ICST Conference on Ambient Media and Systems. Phoenix, AZ, USA (2010). https://doi.org/10.4108/ICST.AMBISYS2008.2869

20. Gardner, R.W., Genin, D., McDowell, R., Rouff, C., Saksena, A., Schmidt, A.: Probabilistic model checking of the next-generation airborne collision avoidance system. In: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC). pp. 1–10 (2016). https://doi.org/10.1109/DASC.2016.7777963
21. Geisslinger, M., Poszler, F., Betz, J., Lütge, C., Lienkamp, M.: Autonomous Driving Ethics: from Trolley Problem to Ethics of Risk. Philosophy & Technology **34**(4), 1033–1055 (2021)
22. Gerking, C., Dziwok, S., Heinzemann, C., Schäfer, W.: Domain-specific model checking for cyber-physical systems. In: 12th Workshop on Model-Driven Engineering, Verification and Validation. pp. 18–27. MoDeVVa 2015, CEUR-WS.org Vol-1514, Ottawa (Sep 2015)
23. Griggio, A., Jonáš, M.: Kratos2: an SMT-Based Model Checker for Imperative Programs. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 423–436. Springer Nature Switzerland, Cham (2023)
24. Griggio, A., Roveri, M.: Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits Systems **35**(6), 1026–1039 (2016). https://doi.org/10.1109/TCAD.2015.2481869
25. Halder, P., Althoff, M.: Minimum-Violation Velocity Planning with Temporal Logic Constraints. In: 2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC). p. 2520–2527. IEEE Press, Macau, China (2022). https://doi.org/10.1109/ITSC55140.2022.9922114
26. Heinzemann, C., Lange, R.: vTSL – a formally verifiable dsl for specifying robot tasks. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 8308–8314. IROS'18, IEEE Computer Society, Madrid, Spain (2018). https://doi.org/10.1109/IROS.2018.8593559
27. Holzmann, G.J.: The model checker spin. Software Engineering, IEEE Transactions on **23**(5), 279 –295 (may 1997). https://doi.org/10.1109/32.588521
28. Holzmann, G.J., H. Smith, M.: Software model checking: extracting verification models from source code†. Software Testing, Verification and Reliability **11**(2), 65–79 (2001). https://doi.org/10.1002/stvr.228
29. Ishigooka, T., Saissi, H., Piper, T., Winter, S., Suri, N.: Practical use of formal verification for safety critical cyber-physical systems: A case study. In: 2014 IEEE International Conference on Cyber-Physical Systems, Networks, and Applications. pp. 7–12 (2014). https://doi.org/10.1109/CPSNA.2014.20
30. ISO/TC 22/SC 32 Electrical and electronic components and general system aspects: ISO 21448:2022 Road vehicles – Safety of the intended functionality (2022), https://www.iso.org/standard/77490.html, accessed: 25.09.2023
31. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A user-study protocol for evaluation of formal verification results and their explanation. arXiv **abs/2108.06376** (2021)
32. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A systematic literature review on counterexample explanation. Information and Software Technology **145**, 1–20 (2022). https://doi.org/10.1016/j.infsof.2021.106800
33. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A user study for evaluation of formal verification results and their explanation at bosch. Empirical Software Engineering **28**(5) (2023)
34. Kalra, N., Paddock, S.M.: Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? Transportation Research Part A: Policy and Practice **94**, 182–193 (2016)

35. Keating, D., McInnes, A., Hayes, M.: An industrial application of model checking to a vessel control system. In: 2011 Sixth IEEE International Symposium on Electronic Design, Test and Application. pp. 83–88 (2011). https://doi.org/10.1109/DELTA.2011.24
36. Keller, C.W., Saha, D., Basu, S., Smolka, S.A.: FocusCheck: A Tool for Model Checking and Debugging Sequential C Programs. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 563–569. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_39
37. Kerner, B.S.: Physics of automated driving in framework of three-phase traffic theory. Physical Review E **97**(4) (2018). https://doi.org/10.1103/PhysRevE.97.042303
38. Kortenkamp, D., Simmons, R.: Robotic Systems Architectures and Programming. In: Siciliano, B., Khatib, O. (eds.) Springer Handbook of Robotics. pp. 187–206. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-30301-5_9
39. Krasowski, H., Zhang, Y., Althoff, M.: Safe Reinforcement Learning for Urban Driving using Invariably Safe Braking Sets. In: 2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC). pp. 2407–2414. Macau, China (2022)
40. Kriebitz, A., Max, R., Lütge, C.: The German Act on Autonomous Driving: Why Ethics Still Matters. Philosophy & Technology **35**(2), 29 (2022). https://doi.org/10.1007/s13347-022-00526-2
41. Krämer, S., Stiller, C., Bouzouraa, M.E.: LiDAR-Based Object Tracking and Shape Estimation Using Polylines and Free-Space Information. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 4515–4522. Madrid, Spanien (2018). https://doi.org/10.1109/IROS.2018.8593385
42. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer (2011)
43. Leino, K., M., R.: Program Proving Using Intermediate Verification Languages (IVLs) like Boogie and Why3. In: Proceedings of the 2012 ACM Conference on High Integrity Language Technology. pp. 25–26. Association for Computing Machinery (2012). https://doi.org/10.1145/2402676.2402689
44. Majzik, I., Semeráth, O., Hajdu, C., Marussy, K., Szatmári, Z., Micskei, Z., Vörös, A., Babikian, A.A., Varró, D.: Towards System-Level Testing with Coverage Guarantees for Autonomous Vehicles. In: Kessentini, M., Yue, T., Pretschner, A., Voss, S., Burgueño, L. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019. pp. 89–94. IEEE, Munich, Germany (2019). https://doi.org/10.1109/MODELS.2019.00-12
45. Mehdipour, N., Althoff, M., Tebbens, R.D., Belta, C.: Formal methods to comply with rules of the road in autonomous driving: State of the art and grand challenges. Automatica **152** (2023). https://doi.org/10.1016/j.automatica.2022.110692
46. Mitsch, S., Ghorbal, K., Vogelbacher, D., Platzer, A.: Formal verification of obstacle avoidance and navigation of ground robots. The International Journal of Robotics Research **36**(12), 1312–1340 (2017). https://doi.org/10.1177/0278364917733549
47. Mitsch, S., Platzer, A.: ModelPlex: verified runtime validation of verified cyber-physical system models. Formal Methods in System Design **49**, 33–74 (2016). https://doi.org/10.1007/s10703-016-0241-z

48. Nawaz, M.S., Malik, M., Li, Y., Sun, M., Lali, M.I.U.: A survey on theorem provers in formal methods (2019)
49. Nees, M.A.: Safer than the average human driver (who is less safe than me)? examining a popular safety benchmark for self-driving cars. Journal of Safety Research **69**, 61–68 (2019)
50. Redfield, S.A., Seto, M.L.: Verification challenges for autonomous systems. In: Lawless, W., Mittu, R., Sofge, D., Russell, S. (eds.) Autonomy and Artificial Intelligence: A Threat or Savior?, pp. 103–127. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-59719-5_5
51. Reid, T., Houts, S., Cammarata, R., Mills, G., Agarwal, S., Vora, A., Pandey, G.: Localization requirements for autonomous vehicles. SAE International Journal of Computer Aided Verification **2**(3), 173–190 (2019). https://doi.org/10.4271/12-02-03-0012
52. Rizaldi, A., Keinholz, J., Huber, M., Feldle, J., Immler, F., Althoff, M., Hilgendorf, E., Nipkow, T.: Formalising and Monitoring Traffic Rules for Autonomous Vehicles in Isabelle/HOL. In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods: 13th International Conference, IFM 2017, Turin, Italy, pp. 50–66. No. 10510 in Lecture Notes in Computer Science, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_4
53. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: A flexible framework for creating software model checkers. In: Proceedings of Testing: Academic and Industrial Conference - Practice And Research Techniques. pp. 3 –22. TAIC PART 2006 (aug 2006). https://doi.org/10.1109/taic-part.2006.5
54. Ročkai, P., Barnat, J., Brim, L.: Model checking C++ programs with exceptions. Science of Computer Programming **128**, 68–85 (2016). https://doi.org/10.1016/j.scico.2016.05.007
55. Schlich, B., Kowalewski, S.: Model checking c source code for embedded systems. International Journal on Software Tools for Technology Transfer **11**(3), 187–202 (2009). https://doi.org/10.1007/s10009-009-0106-5
56. Schmidt, Á., Varró, D.: Checkvml: A tool for model checking visual modeling languages. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003 - The Unified Modeling Language. Modeling Languages and Applications, Lecture Notes in Computer Science, vol. 2863, pp. 92–95. Springer Berlin Heidelberg (Oct 2003). https://doi.org/10.1007/978-3-540-45221-8_8
57. Schmidt, L.M., Kontes, G., Plinge, A., Mutschler, C.: Can You Trust Your Autonomous Car? Interpretable and Verifiably Safe Reinforcement Learning. In: 2021 IEEE Intelligent Vehicles Symposium (IV). pp. 171–178. Nagoya, Japan (2021). https://doi.org/10.1109/IV48863.2021.9575328
58. Schreurs, M., Steuwer, S.: Autonomous Driving - Political, Legal, Social, and Sustainability Dimensions. Autonomes Fahren: Technische, rechtliche und gesellschaftliche Aspekte pp. 151–173 (2015)
59. Selvaraj, Y., Ahrendt, W., Fabian, M.: Verification of decision making software in an autonomous vehicle: An industrial case study. In: Larsen, K.G., Willemse, T. (eds.) Formal Methods for Industrial Critical Systems. pp. 143–159. Springer International Publishing, Cham (2019)
60. Shankar, S., Pajela, G.: A tool integrating model checking into a c verification toolset. In: Bošnački, D., Wijs, A. (eds.) Model Checking Software, Lecture Notes in Computer Science, vol. 9641, pp. 214–224. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-32582-8_15

61. Shariff, A., Bonnefon, J.F., Rahwan, I.: How safe is safe enough? Psychological mechanisms underlying extreme safety demands for self-driving cars. Transportation Research Part C: Emerging Technologies **126**, 1–12 (2021). https://doi.org/10.1016/j.trc.2021.103069
62. Wachenfeld, W., Winner, H.: The Release of Autonomous Vehicles, pp. 425–450. Springer, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-48847-8_21
63. Zhao, T., Yurtsever, E., Paulson, J.A., Rizzoni, G.: Formal Certification Methods for Automated Vehicle Safety Assessment. IEEE Transactions on Intelligent Vehicles **8**(1), 232–249 (2022). https://doi.org/10.1109/TIV.2022.3170517
64. Zhao, T., Yurtsever, E., Paulson, J.A., Rizzoni, G.: Formal certification methods for automated vehicle safety assessment. IEEE Transactions on Intelligent Vehicles **8**(1), 232–249 (2023). https://doi.org/10.1109/TIV.2022.3170517

Enhancing GenMC's Usability and Performance

Michalis Kokologiannakis(✉), Rupak Majumdar, and Viktor Vafeiadis

MPI-SWS, Kaiserslautern, Germany
{michalis,rupak,viktor}@mpi-sws.org

Abstract. GenMC is a state-of-the-art stateless model checker that can verify safety properties of concurrent C/C++ programs under a wide range of memory consistency models, such as SC, TSO, RC11, and IMM. In this paper, we improve the performance and usability of GenMC: we provide a probabilistic estimate of the expected verification cost, we automate the porting of new memory models, and employ caching and other data structure optimizations to improve the tool's performance.

1 Introduction

GenMC [31, 27] is a state-of-the-art stateless model checker that verifies assertion safety of concurrent C/C++ programs in a fully automated ("push-button") fashion. In its core, it implements the TruSt dynamic partial order reduction (DPOR) algorithm [27], which has polynomial space complexity and optimal time complexity: it explores only the consistent executions of the given program and never repeats any work. GenMC also incorporates custom techniques for verifying programs with constructs such as synchronization barriers [30] and loops [29, 28] more effectively.

Despite its solid theoretical foundations, certain parts of GenMC's *implementation* were somewhat neglected, and are addressed as part of this work.

Time Unpredictability (§3): Non-expert users of GenMC were finding it difficult to estimate how long verification will take, and whether it is worth waiting for the verification result or give up. To address this problem, we implement a procedure that produces a probabilistic estimate of the size of the state space so that users can anticipate the total verification cost, and perhaps revise their code as necessary.

Customization Difficulty (§4): Although the TruSt algorithm is parametric in the choice of the underlying *memory consistency model* (MCM) [9], adding support for new MCMs to GenMC was arduous and required human effort. To make the tool more easily customizable, we extend an already existing domain specific language so that users can port new MCMs into the tool completely automatically.

Overall Performance (§5): GenMC used to spend a lot of time repeatedly simulating the execution of LLVM bytecode with an interpreter, which led to non-trivial performance overhead. We improve the tool's performance by caching interpreter results and optimizing other internal data structures.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 66–84, 2024.
https://doi.org/10.1007/978-3-031-57249-4_4

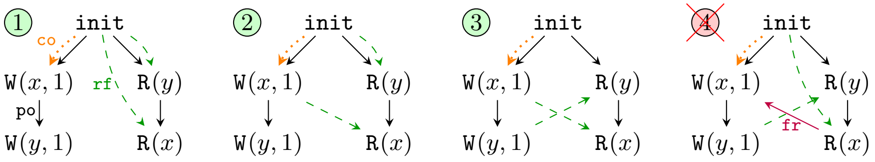


Fig. 1. MP: Three consistent and one inconsistent execution graphs under SC and RA. We omit co edges for variable y to avoid cluttering the presentation.

2 Background

We begin with a brief tour of declarative MCM semantics (§ 2.1) and proceed with a description of GenMC's core model checking algorithm (§ 2.2).

2.1 Semantics of Memory Consistency Models

When dealing with multiple MCMs, it is convenient to use *declarative semantics* to represent program executions as execution graphs. An execution graph comprises a set of nodes corresponding to program instructions (e.g., loads or stores to shared memory), and a set of edges corresponding to various relations among the instructions. Examples of primitive relations used by most MCMs are: the *program order* (po), which orders the instructions of each thread, the *reads-from* (rf) relation, which maps each read to the write it gets its value from, and the *coherence order* (co), which totally orders writes to the same memory location.

The semantics of a program P under a model M is expressed as a set of "consistent" execution graphs, representing an abstract set of program executions that the model allows. Consistency for a given MCM entails satisfying the MCM's consistency predicate.

Consistency predicates are typically expressed in relational algebra. For instance, *sequential consistency* (SC) [35], a standard MCM where threads take turns executing their instructions, demands that $(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})$ be acyclic, where $\mathtt{fr} \triangleq \mathtt{rf}^{-1}; \mathtt{co}$. *Release-acquire* (RA) [33] is a weaker MCM, which demands that $(\mathtt{po} \cup \mathtt{rf})^{+}; (\mathtt{co} \cup \mathtt{fr})^{?}$ be irreflexive.

To illustrate these concepts, consider the MP example below along with the annotated outcome, which corresponds to graph 4 in Fig. 1 and is forbidden under both SC and RA.[1]

$$\begin{array}{l||l} x := 1 & a := y \;/\!/\; 1 \\ y := 1 & b := x \;/\!/\; 0 \end{array} \qquad (\text{MP})$$

Graph 4 is inconsistent according to both SC and RA because of the $\mathtt{po}; \mathtt{rf}; \mathtt{po}; \mathtt{fr}$ cyclic path from $\mathtt{W}(x, 1)$. Intuitively, thread II cannot read a stale value for x after having read thread I's write to y. The other three depicted execution graphs 1, 2, 3 are consistent and correspond to the outcomes where $a = 0 \vee b = 1$.

[1] In all our examples, x, y, z are shared variables, while $a, b, c, ...$ are local.

Algorithm 1 An optimal graph-based DPOR algorithm

```
procedure $\textsc{Verify}(P)$
    $\textsc{Visit}_P(G_\emptyset)$

procedure $\textsc{Visit}_P(G)$
    $a \leftarrow \textsc{AddNextEvent}_P(G)$
    if $a = \bot$ then return "Execution complete"
    if $\textsc{IsErroneous}_{\mathsf{M}}(G)$ then exit("error")
    if $a \in \mathtt{R}$ then
        $\textsc{VisitRFs}_P(G, a)$
    else if $a \in \mathtt{W}$ then
        $\textsc{VisitCOs}_P(G, a)$
        for $r \in G.\mathtt{R}_{\mathtt{loc}(a)} \setminus \mathsf{cprefix}(a)$ do
            if $\neg\textsc{DuplicateRevisit}(G, \langle r, a\rangle)$ then
                $\mathit{Deleted} \leftarrow \{e \in G.\mathtt{E} \mid r <_G e <_G a\} \setminus \mathsf{cprefix}(a)$
                $\textsc{VisitCOs}_P(\mathsf{SetRF}(G \setminus \mathit{Deleted}, r, a), a)$
    else$\textsc{Visit}_P(G)$

procedure $\textsc{VisitRFs}_P(G, a)$
    for $w \in G.\mathtt{W}_{\mathtt{loc}(a)}$ do
        $G' \leftarrow \mathsf{SetRF}(G, a, w)$
        if $\mathsf{consistent}_{\mathsf{M}}(G')$ then $\textsc{Visit}_P(G')$

procedure $\textsc{VisitCOs}_P(G, a)$
    for $w_p \in G.\mathtt{W}_{\mathtt{loc}(a)}$ do
        $G' \leftarrow \mathsf{SetCO}(G, w_p, a)$
        if $\mathsf{consistent}_{\mathsf{M}}(G')$ then $\textsc{Visit}_P(G')$
```

2.2 Dynamic Partial Order Reduction and GenMC

Declarative semantics enable effective automated verification with the "TruSt" model checking algorithm [27]. TruSt is a graph-based *dynamic partial order reduction* (DPOR) algorithm that takes as parameters a program P and a memory consistency model M. It verifies the program by generating all M-consistent graphs of P and checking that they do not contain any errors. For this purpose, we assume the MCM M defines three components:

1. a *causal order*, $\texttt{corder} \subseteq (\texttt{po} \cup \texttt{rf})^+$, prescribing causal dependencies among the instructions;
2. the $\mathsf{consistent}_{\mathsf{M}}(G)$ predicate prescribing when a graph G is consistent; and
3. the $\textsc{IsErroneous}_{\mathsf{M}}(G)$ predicate prescribing whether the graph contains an error (e.g., an assertion violation or a data race).

The core structure of GenMC's DPOR algorithm can be seen in Algorithm 1. In what follows, we provide a high-level overview of the algorithm and refer readers to Kokologiannakis et al. [27] for a more detailed presentation.

Verify generates all possible execution graphs of P by calling Visit on the initial graph $G_\emptyset$ containing only the initialization event `init`.

In turn, Visit generates all program executions incrementally by extending a given graph with one event at a time by calling AddNextEvent(Line 4), which selects some unfinished program thread and runs an interpreter to run the code of that thread until its next event.

Visit returns successfully if no further events can be added (Line 5, i.e., if all threads are finished or stuck) or when an error is encountered (Line 6). Otherwise, the next action that Visit takes depends on the type of a.

If a is a read event (Line 7), Visit recursively explores all its possible `rf` options by calling VisitRFs. The latter iterates over all same-location writes (Line 17) and recursively explores the ones that preserve consistency (Line 19).

If a is a write event (Line 9), similarly to the read case, Visit recursively explores all possible `co` options for it by calling VisitCOs (Line 10). In addition, Visit *revisits* existing same-location reads in G, since they did not have the chance to read from a when they were added. Specifically, for each read r that does not causally precede a (Line 11), Visit checks (Line 12) whether a should revisit r (i.e., that the revisit has not taken place in some other exploration), and if so calls VisitCOs on an appropriately restricted graph G' (Line 14).

Restricting the graph is necessary because the value read by r might affect e.g., the control flow of the corresponding thread. In GenMC, the restricted graph only contains the events that were added before the revisited read r, as well as the ones causally preceding a, effectively creating a graph that models a scenario where a and its prefix were present when r was added. (The way a graph is restricted is important when estimating the program state space in §3.)

Finally, if a has any other event type (Line 15), Visit recursively calls itself.

To conclude our brief presentation of GenMC's algorithm, we reiterate its core properties. Algorithm 1 is optimal (i.e., it explores each consistent execution graph of P exactly once and never embarks into futile explorations), has polynomial memory consumption, and can accommodate arbitrary MCMs subject to the following constraints:

Well-formedness: Consistency does not depend on the order in which events are added to the graph, and, in consistent graphs, `corder` should be acyclic (i.e., an event cannot circularly depend on itself).

Prefix-closedness: Restricting a consistent graph to any `corder`-prefix-closed subset of its events yields a consistent graph.

Extensibility: Adding a `corder`-maximal event to a consistent graph preserves consistency for some choice of `rf`/`co`. Intuitively, for the case that $\texttt{corder} \triangleq (\texttt{po} \cup \texttt{rf})^{+}$, executing a program should never get stuck if a thread has more statements to execute: each read can read from the most recent, same-location write, and each write can be added last in `co`.

3 Estimating the Program State Space

The execution time of Algorithm 1 is difficult to predict because it depends on the number of consistent execution graphs of the program P, which in turn is challenging to estimate without actually generating the consistent graphs. In

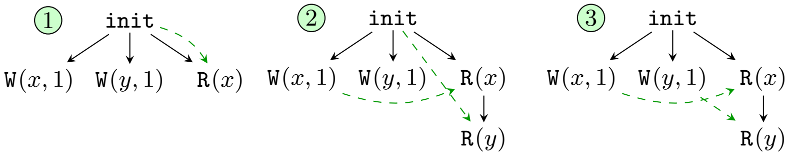


Fig. 2. R+W+W: three consistent execution graphs under SC

particular, the number of possible rfs/cos for a given read/write is not fixed beforehand, and depends on the exploration choices made so far. Moreover, since the state space is often asymmetric, we cannot estimate the remaining exploration options merely by assuming that untaken exploration options (i.e., unexplored rfs or cos) will yield the same number of exploration options as their taken counterparts.

3.1 The Basic Approach

One solution for the problem above is to use a Monte Carlo simulation [23]. In Monte Carlo methods, one generates a number of random samples from a given input domain, performs some computation on each sample, and then aggregates the results. As long as the sampling process is unbiased, the law of large numbers guarantees that the empirical mean of the obtained samples will approximate the expected value of the corresponding random variable.

Applying this idea to our problem, we construct a randomized version of Algorithm 1 that generates a single consistent program execution by adapting VisitRFsand VisitCOsto explore a random rf/co choice for each read/write (picked uniformly at random) instead of all consistent choices. For one execution, we can estimate the size of the search tree by taking the product of the exploration options encountered for each read/write (effectively assuming a symmetric state space). We can then run the modified Visit on $G_\emptyset$ a fixed number of times, and take the mean of the individual estimates as our state-space estimate.

Let us examine how such a method would work with the example below, where there are three graphs to be generated (shown in Fig. 2).

$$x := 1 \;\Big\|\; y := 1 \;\Big\|\; \begin{array}{l}\texttt{if } (x = 1)\\ \quad b := y\end{array} \qquad (\text{W+W+RR})$$

Assuming the algorithm adds events from left to right, the state-space size will be estimated to be either $2 \cdot 2 = 4$ (in samples where the algorithm makes the read of x read 1 and the read of y also appears), or 2 (in samples where the algorithms makes the read of x read 0, and the read of y does not appear). In fact, since the algorithm picks among the rf choices for x uniformly at random, each of these estimations occurs with a similar frequency, thereby yielding an estimated mean that approximates 3 (given a sufficiently large sample).

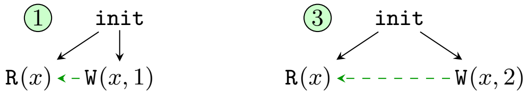


Fig. 3. R+W+W: two possible revisits

3.2 Problems with Revisits

The challenge of applying Monte Carlo method to our setting is achieving an unbiased sampling in the presence of revisits. (Recall from §2 that if a write w is added after a same-location read r, GenMC will also examine the scenario where w revisits r, but only if a certain revisiting condition C holds.)

Blindly applying this revisiting condition when estimating, however, does not work. Since its purpose is to avoid duplication, the probability that C holds for a sequence of random rfs/cos choices is tiny, thereby yielding a biased simulation.

Similarly, one cannot ignore the revisiting condition C and perform revisits with some probability p because this can lead to very long runs. A given revisit R may delete part of the execution graph, which when subsequently re-added might invoke another revisit R', which may in turn delete the write that performed R, and this process can be repeated again and again. To see this, consider the R+W+W program below and the graphs that occur as a result of each of the writes revisiting the read of x, depicted in Fig. 3.

$$a := x \parallel x := 1 \parallel x := 2 \qquad (\text{R+W+W})$$

As can be seen, if $\mathtt{W}(x, 1)$ revisits $\mathtt{R}(x)$ then $\mathtt{W}(x, 2)$ is deleted, and vice versa.

An obvious solution to this issue would be to preclude multiple revisits and e.g., only allow one revisit per sample. Even in this case, however, it is quite difficult to find the probability p with which revisits should take place. Indeed, suppose that once a revisit is performed, the revisiting write and its causal prefix can never be deleted from the execution graph. Alas, in such an approach, the probability that some read r gets revisited by a given write w decreases exponentially as the number of writes that are added after r (but before w) increases, and we again obtain a biased simulation. In the R+W+W above, $\mathtt{R}(x)$ will be revisited by $\mathtt{W}(x, 1)$ with probability p and by $\mathtt{W}(x, 2)$ with probability $(1 - p)p$. (Observe that, even if p is not fixed, the probability that later writes get to revisit a given read r decreases exponentially.)

Finally, precluding revisits altogether is not a viable solution either. Even though such an approach would yield an unbiased simulation, the simulation would only cover a subset of the input domain, as revisits would not be accounted for. In the R+W+W example, precluding revisits means that the state-space size would always be erroneously estimated to be 1.

3.3 Our Solution

Our solution to make the sampling process as unbiased as possible is twofold. First, we keep a *choice map* for each encountered event representing its possible exploration alternatives (e.g., rf or co options). The reason a map is used is to be able to account for revisiting: as possible rf options for a read r might appear after r has been added, we cannot calculate the product of all available explorations options on-the-fly. Instead, we populate the available rf options for each read dynamically, as more rf options become available, and calculate the product at the end of each complete execution.

Second, as far as revisiting is concerned, in order to maintain an unbiased estimation, we do preclude revisits, but mitigate the negative effects of this decision by employing a *custom scheduler* that prioritizes the addition of writes over that of reads (so that reads have as many rfs as possible available when they are added), and chooses uniformly at random when only reads can be added.

Intuitively, the reason this approach works well is that each of the graphs in a program's declarative semantics can be generated incrementally, following `corder` (see §2). What this means is that we can, in principle, generate all graphs in a program's declarative semantics without any backward revisiting. Of course, it can still be the case that a read is added before some of its possible rfs (e.g., if the next available event of all threads is a read), but picking at random among reads guarantees that all rf options will eventually be considered (though, perhaps, some of them not often enough).

A modification of Verify that implements the above approach can be seen in Algorithm 2. The Estimate function obtains a number of samples[2] by calling GetSample (Line 4), and calculates the mean of all estimations (Line 6).

GetSample closely resembles Visit, with the addition of a choice map C that stores consistent exploration options for each event of an execution graph. At each step, GetSample extends the current graph with an event a obtained by our custom scheduler (Line 8), and then performs a case analysis on a's type.

If a is a read, PickRF populates $C[a]$ with all consistent rfs for a (Line 18), and then picks an rf for a uniformly at random (Line 19).

If a is a write, PickCO performs actions similar to the ones taken by PickRF in the read case (Line 14), and upon returning, GetSample also updates the entries of all revisitable reads, recording a as a possible rf. (Observe that no revisiting is performed during the estimation process.)

Finally, at the end of each sample (Line 9), GetSample returns the current estimation $\prod_{e \in G} |C[e]|$.

3.4 Stopping the Sampling Process

Before concluding the presentation of our estimation procedure, let us discuss when that estimation procedure should stop, i.e., when an adequate number of samples has been taken.

While ShouldKeepSampling in Algorithm 2 could in principle return **false** after a fixed number of samples, doing so is often undesirable. For programs with a very small state space, it does not make sense to obtain more samples than the number of consistent executions because that would unnecessarily delay

[2] See § 3.4 for how this number is calculated.

Algorithm 2 Estimating the DPOR state space

```
**procedure** $\textsc{Estimate}(P)$
    $T \leftarrow 0, \mathit{Samples} \leftarrow 0$
    **while** $\textsc{ShouldKeepSampling}(T, \mathit{Samples})$ **do**
        $T \leftarrow T + \textsc{GetSample}_P(G_\emptyset, C_\emptyset)$
        $\mathit{Samples} \leftarrow \mathit{Samples} + 1$
    **print** $T/\mathit{Samples}$

**procedure** $\textsc{GetSample}_P(G, C)$
    $a \leftarrow \textsc{AddNextEvent}_P(G)$
    **if** $a = \bot$ **then return** $\prod_{e \in G} |C[e]|$
    **if** $\textsc{IsErroneous}_{\mathsf{M}}(G)$ **then exit**(“error”)
    **if** $a \in \texttt{R}$ **then**
        $\textsc{PickRF}_P(G, C, a)$
    **else if** $a \in \texttt{W}$ **then**
        $\textsc{PickCO}_P(G, C, a)$
        **for** $r \in G.\texttt{R}_{\texttt{loc}(a)} \setminus \mathsf{cprefix}(a)$ **do** $C[r] \leftarrow C[r] \uplus \{a\}$
    **else** $\textsc{GetSample}_P(G, C)$

**procedure** $\textsc{PickRF}_P(G, C, a)$
    $C[a] \leftarrow \{w \in G \mid \mathsf{consistent}(\mathsf{SetRF}(G, w, a))\}$
    **randomly choose some** $w \in C[a]$
    $\textsc{GetSample}_P(\mathsf{SetRF}(G, w, a), C)$

**procedure** $\textsc{PickCO}_P(G, C, a)$
    $C[a] \leftarrow \{w_p \in G \mid \mathsf{consistent}(\mathsf{SetCO}(G, w_p, w))\}$
    **randomly choose some** $w_p \in C[a]$
    $\textsc{GetSample}_P(\mathsf{SetCO}(G, w_p, a), C)$
```

verification. Similarly, for programs whose state space is completely symmetric, a near-perfect estimate can be achieved with only a few samples.

To deal with such issues, we make ShouldKeepSampling *dynamic*. After a minimum (fixed) number of samples has been taken, ShouldKeepSampling returns **false** if any of the following holds:

- the standard deviation of the current estimated mean M is less than some fixed percentage of M
- the number of executions explored exceeds M
- a maximum (fixed) number of samples has been taken.

While users are free to override and tune the above heuristics, we found them satisfactory in practice, and they did not seem to affect the quality of the produced estimation.

3.5 Evaluation

To evaluate the accuracy of our approach, we estimated the state-space size of various benchmarks, and measured how close the estimation was to the actual

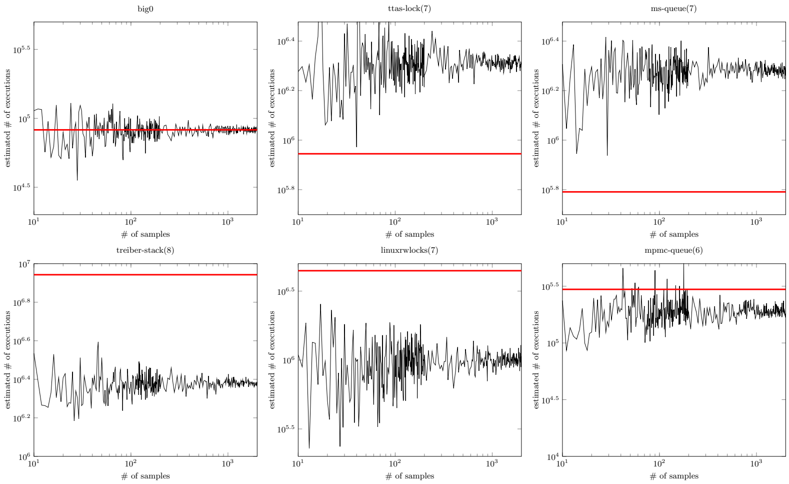


Fig. 4. Estimation accuracy for various benchmarks

```
let sc = po ∪ rf ∪ co ∪ fr
acyclic sc
```

Fig. 5. SC expressed in KAT

size. When selecting tests, we opted for non-symmetric benchmarks (as these are more challenging to estimate), though with a manageable state-space size (so that we can measure how accurate the estimation is). The results we obtained for some representative benchmarks (and a varying sample size) are shown in Fig. 4. The actual state-space size is shown in **red**.

As it can be seen, the state-space estimation is quite useful. It is always within one order of magnitude from the correct number of executions (sometimes under- and sometimes over-approximating the correct value), and converges very quickly: even with only 20 samples we get a fairly accurate estimate of the state space. Finally, estimation is very fast: for instance, it takes about 3 seconds to obtain 2000 samples of `ms-queue`, while verification needs about 1470 seconds.

4 Automatically Porting New MCMs

To port a new MCM into GenMC, one has to define the `corder` relation and implement the consistency checking routine $\mathsf{consistent}_{\mathsf{M}}(G)$ and the MCM-specific error-checking code of $\textsc{IsErroneous}_{\mathsf{M}}(G)$.

Until recently, these routines had to be written manually directly in C++. To some extent, porting new MCMs was automated with Kater [25], a framework for proving metatheoretic properties about MCMs, which can also generate code for checking acyclicity constraints.

```
// Calculation of synchronizes-with
let relseq = [REL] ; ([F] ; po)? ; (rf ; rmw)*
let sw_to_r = relseq ; rf ; [ACQ]
let sw_to_f = relseq ; rf ; po ; [F] ; [ACQ]
let sw = sw_to_r ∪ sw_to_f

// Optimized calculation of happens-before.
// Save the part of ‘hb‘ that does not finish with a reads-from edge
view hb_stable = (po-imm ∪ sw_to_r ; po-imm ∪ sw_to_f)+
let hb = (hb_stable ∪ hb_stable? ; sw_to_r)
assert hb = (po ∪ sw)+

// Coherence : Optimize the checking of irreflexive (hb ; eco)
coherence (hb_stable)

// Sequential consistency order
let eco = (rf ∪ mo ∪ fr)+
let scb = po ∪ rf ∪ mo ∪ fr
let psc = [SC] ; po ; hb ; po ; [SC]
        ∪ [SC] ; ([F] ; hb)? ; scb ; (hb ; [F])? ; [SC]
        ∪ [F∩SC] ; hb ; [F∩SC]
        ∪ [F∩SC] ; hb ; eco ; hb ; [F∩SC]
        ∪ [SC] ; po ; [SC]
acyclic psc

// RC11 error detection
let ww_conflict = [W] ; loc-overlap ; [W]
let wr_conflict = [W] ; loc-overlap ; [R] ∪ [R] ; loc-overlap ; [W]
let conflicting = ww_conflict ∪ wr_conflict
let na_conflict = [NA] ; conflicting ∪ conflicting ; [NA]

// [...]

error RaceNotAtomic   unless  na_conflict ⊆ hb_stable
warning WWRace        unless  [W]; ww_conflict; [W] ⊆ hb_stable
```

Fig. 6. Handling irreflexivity, emptiness, and inclusion constraints

Given an acyclicity constraint of a relation expressible in *Kleene Algebra with Tests* (KAT) [32] (e.g., the `sc` relation of Fig. 5), we can check whether an execution graph G satisfies the acyclicity constraint by taking the product of G with an automaton accepting the (rotational closure of) the language of corresponding to the acyclicity constraint and performing a depth-first search over it. Kater generates code performing such acyclicity checks in linear time in the size of G.

4.1 Beyond Acyclicity Constraints

Although the consistency predicate of SC can be defined as a single acyclicity constraint, more advanced models also check other kinds of constraints, for which Kater could not generate consistency checking code out of the box.

One such model employing different types of constraints is RC11 [34], a fragment of which expressed in KAT can be seen in Fig. 6. It contains three other kinds of constraints, which we discuss below.

First, the `assert` introduces a static constraint about the MCM that is checked at compile-time: it checks that the rewriting of the model that defines

```
Error: Non-atomic race!
Event (2, 4) conflicts with event (1, 3) in graph:
<0, 1> thread_p:
[...]
(1, 4): Wrel (deq.bottom, 1) deque.h:26
[...]
(1, 22): Wrlx (deq.bottom, 4) deque.h:33
[...]
<0, 2> thread_s:
[...]
(2, 3): Racq (deq.bottom, 4) [(1, 22)] deque.h:65
[...]
```

Fig. 7. A GenMC error for a Chase-Lev deque [13]. Events (1, 4) and (2, 3) do not synchronize under C/C++17

`hb` in terms of the `hb_stable` relation is equivalent to the original model which defines `hb` directly. As this constraint can be checked completely statically by Kater, there is no need to generate any code for it.

Second, there is the *coherence* constraint about `hb_stable` dictating that $\texttt{hb\_stable}; (\texttt{rf} \cup \texttt{co} \cup \texttt{fr})^+$ be irreflexive. Such irreflexivity constraints are very common in MCMs and can typically be checked in a very efficient manner. The idea is to represent the `hb_stable` relation using a vector clock. Then, for a read r and a candidate store $s \xrightarrow{\texttt{rf}} r$ in VisitRFs, we check that no other store s' that is `co`-after s is included in r's vector clock[3]. (This would mean that r is aware of a "more recent" store than s.) The total complexity of the generated checks is $\mathcal{O}(N)$ in the size of the graph.

Finally, there are two inclusion constraints introduced by the `error/unless` and the `warning/unless` constructs. Even though these constraints are not technically part of consistency checking but rather of the IsErroneous$_{\mathsf{M}}$ function, they can still be MCM-specific and so the code checking them has to be mechanically generated.

Generally, for inclusion constraints of the form $a \subseteq b$, we have extended Kater to generate code that calculates the reachable states of a and b, and then checks that the reachable states of a are included in those of b. In the special case where b is represented as a vector clock (as is the case with `hb_stable`), the calculation of b's reachable states is spared, and the generated code only checks whether a's reachable states are contained in b's vector clock.

4.2 Experimenting with MCMs

The above extensions to Kater and GenMC made it possible to fully automate the porting of models like SC [35], RA [33], RC11 [34] and IMM [39]. They also

[3] The procedure for VisitCOs is similar and thus omitted for brevity.

made it easier to experiment with changes in these MCMs, often leading to interesting observations.

One such observation occurred when we "upgraded" GenMC's RC11 model to use the slightly different `sw` definition of C/C++17 (see Fig. 6), a change that was not supposed to have any impact for most benchmarks. Surprisingly, we found that certain data-structure implementations inadvertently relied on the old definition, and were deemed incorrect (due to improper synchronization) when using the new one (see Fig. 7 for an example). Subtle issues like this underline the need for tools that support automatic porting of MCMs.

5 Performance Improvements

Let us now turn our attention to GenMC's performance. For a program P with E consistent executions under a model M, the actual verification cost is $E \times C$, where C is the average cost of generating and checking consistency of one program execution. Even if we keep E fixed, engineering optimizations can significantly reduce the cost C and improve the overall performance.

GenMC's infrastructure is split into two (largely independent) components: the runtime environment, which executes the program under test, and the model checker, which undertakes the construction of consistent execution graphs. The cost per execution, C, can thus be attributed either to the runtime environment (especially in cases where the program code is large or contains some expensive computation) or to the execution graph construction and consistency checking.

In what follows, we address these bottlenecks by presenting two engineering optimizations: one on the runtime environment front (§ 5.1), and one on the model checking front (§ 5.2).

5.1 Reducing Runtime Reliance

Recall that Algorithm 1 repeatedly simulates the execution of the program P to generate all its execution graphs. When the number of shared-memory accesses (which are the ones recorded in an execution graph) are only a small percentage of the program code, the execution time is dominated by code unnecessary for verification.

Obviously, it would be really helpful to reduce reliance on the runtime environment. The key insight in doing so is that we can *cache* the events following a given sequence of read values. To make this concrete, consider the following program where N threads employ a single lock (implemented as a CAS loop) to access a hash table HT:

$$\begin{array}{l||c||l} \texttt{while}\,(\neg\texttt{CAS}(\mathit{lock},0,1))\ ; & & \texttt{while}\,(\neg\texttt{CAS}(\mathit{lock},0,1))\ ; \\ \mathit{HT}[i] := \ldots & \ldots & \mathit{HT}[i] := \ldots \\ \mathit{lock} := 0 & & \mathit{lock} := 0 \end{array}$$

Observe that the program above has $N!$ (non-blocked) executions, corresponding to all the ways the threads can access the hash table. However, each CAS

Table 1. Performance impact of label caching on data structures

		GenMC/no-cache	GenMC
	Executions	*Time*	*Time*
`treiber-stack(6)`	720	2.26	1.32
`treiber-stack(7)`	5040	57.20	29.37
`treiber-stack(8)`	40 320	1032.09	581.20
`ms-queue(5)`	120	0.91	0.64
`ms-queue(6)`	720	20.98	13.18
`ms-queue(7)`	5040	445.76	294.51
`buf-ring(2)`	20	0.06	0.03
`buf-ring(3)`	1218	0.65	0.48
`buf-ring(4)`	193 280	353.27	253.35
`ttas-lock(5)`	120	0.20	0.14
`ttas-lock(6)`	720	2.47	1.75
`ttas-lock(7)`	5040	62.17	44.24

operation can read one out of two possible values: 0, indicating that the CAS will succeed, the thread will enter its critical section, and release the lock, or 1, indicating that the CAS acquisition failed, and that the thread will try again.

More generally, the value domain used in concurrent programs is small, and it is thus straightforward to record the encountered values (and the corresponding event sequences) for each thread in a trie structure. Then, whenever the model checker e.g., changes a read's `rf`, it gets the sequence of events that will be added after the read (up until the next read) by checking whether the values that the thread read so far have been cached in the trie.

Besides read events, there are other events with read semantics, and thus need to be treated similarly in the trie. One such case are memory allocation events, which are undertaken by GenMC, as opposed to the runtime environment. If the allocated addresses are not recorded in the trie and distributed anew by the model checker, the cache might end up being inconsistent (e.g., if the addresses of later events depend on the allocated address). A better solution is to ensure a deterministic semantics of memory allocation, where the memory address returned depends only the thread identifier and the previous allocations of the same thread.

Caching interacts with the concurrent nature of GenMC itself. GenMC may use multiple worker threads to parallelize the exploration procedure [27], each of which explores a different (unique) execution graph. To avoid contention between worker threads, we equip each GenMC thread with its own (thread-local) cache.

Evaluation As shown in Table 1, caching the encountered events can yield significant performance benefits: depending on the benchmark, GenMC can be 1.4 to 1.8 times faster when caching is employed. In fact, when caching is employed, 90-99% of the calls to the runtime environment are spared for the benchmarks of Table 1, as the respective value sequences are cached.

The reason these savings do not directly translate to runtime gains is that the benchmarks of Table 1 are standard concurrent data structures, which consist almost exclusively of shared-memory accesses. The only gains in these benchmarks stem from the faster access times the trie provides compared to the runtime environment.

5.2 Optimizing Consistency Checking

Runtime aside, most of the remaining time is spent checking consistency of the constructed graph. When it comes to consistency checks, there are two major performance hindrances: (a) the fact that such checks run always, regardless of the program under test and the constructed execution graph, and (b) the consistency checking code itself.

For the first performance issue, we can observe that the nature of the program might render checking full consistency of a graph unnecessary. Let us consider RC11 (Fig. 6) as an example. A large part of the model is devoted to the handling of SC accesses, via the `psc` relation. However, for a program P that does not contain any SC accesses, checking for `psc` acyclicity is unnecessary, and reducing to the Release-Acquire (RA) [33] fragment would suffice. Analogously, if P solely contains SC accesses, we can reduce the verification problem to the SC one.

Leveraging this insight, we employ GenMC with a *model simplification* criterion: if GenMC can statically determine that a particular program P only uses SC/RA accesses, it will try to verify P under SC/RA.

In a somewhat similar manner, we optimize GenMC to try and re-use relations already calculated in an execution graph, rather than recalculating them from scratch. Relations stored in vector clocks (see §4) readily offer such a prime optimization opportunity. When a relation r to be saved in a vector clock is transitive, instead of recalculating it every time we add an event e, we modify GenMC to only calculate the difference between e and its predecessor in r.

For the second performance issue, we observed that that a lot of time was spent allocating memory and indexing into arrays. We therefore optimized the consistency checking code in the following ways.

- To make access to the graph's primitive relations (`rf`, `co`, `po` and `fr`) faster, we rewrote a large part of the existing GenMC infrastructure to use pointers to fetch relation successors/predecessors (instead of array indices), thus sparing one level of indirection.
- To avoid using extra memory when saving relations, we converted all corresponding data structures to intrusive ones, thus significantly reducing access times and memory allocation calls.
- To avoid re-allocating memory for auxiliary data necessary for intermediate computations (e.g., for storing reachable states during inclusion checks), we provided each GenMC thread with a thread-local copy, thereby minimizing memory allocations.

Table 2. Consistency-intensive benchmarks

		Time/SC		Time/TSO		Time/RC11	
	Executions	Old	New	Old	New	Old	New
`szymanski(2)`	78	0.12	0.09	0.14	0.11	0.31	0.17
`szymanski(3)`	1068	2.33	1.36	2.90	1.89	6.90	3.34
`peterson(3)`	588	0.14	0.08	0.16	0.10	0.45	0.18
`peterson(4)`	7360	2.11	0.97	2.51	1.40	7.81	2.80
`parker(1)`	54	0.02	0.02	0.02	0.02	0.04	0.04
`parker(2)`	6701	1.99	1.21	2.51	1.41	6.76	4.13
`dekker_f(3)`	1344	0.51	0.27	0.62	0.39	2.53	0.62
`dekker_f(4)`	26 797	10.39	6.78	15.25	10.23	38.39	16.74
`fib_bench(5)`	218 243	2.14	1.97	3.06	2.45	7.48	4.08
`fib_bench(6)`	2 363 803	23.54	21.38	33.43	26.59	90.57	46.27
`lamport(2)`	16	0.01	0.01	0.01	0.02	0.01	0.02
`lamport(3)`	9216	2.23	1.35	2.67	1.74	7.15	3.26

Evaluation To see how the optimized consistency checks perform, we compared GenMC with its previous version[4]. The results are shown in Table 2. As it can be seen, these performance optimizations greatly improved the performance of GenMC on a set of benchmarks requiring intensive consistency checks.

6 Related Work

Even though there is a plethora of works in stateless model checking [20, 36, 17, 2, 4, 12, 14, 5, 6, 7, 10, 16, 37, 40], few tools can deal with weak MCMs [1, 3, 38, 42, 11, 24]. The only tool among them that supports more than one MCMs is Nidhugg [1], although it is not parametric in the choice of the MCM, and employs a different algorithm for each supported MCM.

As far as other model checking techniques are concerned, SAT/SMT-based bounded model checking (BMC) techniques have been extended to handle weak MCMs [15, 8, 18]. Among these, Dartagnan [18, 22] stands out, as it is also parametric in the choice of the MCM, and porting new MCMs is also automated.

We are not aware of any other (enumerative) model checker that provides an estimation of the state-space size. Even though there are tools that provide a *progress report* (as opposed to a state space estimation), progress reports are typically not as helpful as a size estimation. Java Pathfinder [41] provides a progress bar by assuming that the state space is symmetric (i.e., nodes at the same level in the exploration tree will have same-size subtrees). While the bar grows monotonically, it may not advance at a steady pace, and be ”stuck” at

[4] For this comparison, we disabled the cache (§ 5.1) and the model simplification criterion (§ 5.2), so as to not unfairly penalize the previous GenMC version.

e.g., 99%. DPOR tools like Concuerror [21] and NIDHUGG [1] provide a progress report based on how many backtracking options have still to be explored. This number does not monotonically decrease, and is thus of limited use.

7 Summary

In this paper, we enhanced the usability of GENMC (and DPOR in general) by: (a) providing a completion estimate based on a Monte Carlo simulation, and (b) completely automating the porting of new MCMs into the tool. In addition, we improved the tool performance using caching and engineering optimizations. We hope that similar techniques will be leveraged by other researchers and developers working on DPOR tools.

Data-Availability Statement The paper replication package is available at [26]. GENMC is available at [19].

Acknowledgements We thank the anonymous reviewers for their feedback. This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., and Sagonas, K.: Stateless model checking for TSO and PSO. In: TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). DOI: 10.1007/978-3-662-46681-0_28
2. Abdulla, P.A., Aronis, S., Jonsson, B., and Sagonas, K.: Optimal dynamic partial order reduction. In: POPL 2014, pp. 373–384. ACM, New York, NY, USA (2014). DOI: 10.1145/2535838.2535845
3. Abdulla, P.A., Atig, M.F., Jonsson, B., and Leonardsson, C.: Stateless model checking for POWER. In: CAV 2016. LNCS, vol. 9780, pp. 134–156. Springer, Heidelberg (2016). DOI: 10.1007/978-3-319-41540-6_8
4. Abdulla, P.A., Atig, M.F., Jonsson, B., and Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. 2(OOPSLA), 135:1–135:29 (2018). DOI: 10.1145/3276505
5. Agarwal, P., Chatterjee, K., Pathak, S., Pavlogiannis, A., and Toman, V.: Stateless Model Checking Under a Reads-Value-From Equivalence. In: Silva, A., and Leino, K.R.M. (eds.) CAV 2021, pp. 341–366. Springer International Publishing, Cham (2021). DOI: 10.1007/978-3-030-81685-8_16
6. Albert, E., Arenas, P., Banda, M.G. de la, Gómez-Zamalloa, M., and Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., and Kunčak, V. (eds.) CAV 2017, pp. 526–543. Springer International Publishing, Cham (2017). DOI: 10.1007/978-3-319-63387-9_26
7. Albert, E., Gómez-Zamalloa, M., Isabel, M., and Rubio, A.: Constrained dynamic partial order reduction. In: Chockler, H., and Weissenbacher, G. (eds.) CAV 2018, pp. 392–410. Springer International Publishing, Cham (2018). DOI: 10.1007/978-3-319-96142-2_24

8. Alglave, J., Kroening, D., and Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). DOI: 10.1007/978-3-642-39799-8_9
9. Alglave, J., Maranget, L., and Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (2014). DOI: 10.1145/2627752
10. Aronis, S., Jonsson, B., Lång, M., and Sagonas, K.: Optimal dynamic partial order reduction with observers. In: TACAS 2018. LNCS, vol. 10806, pp. 229–248. Springer, Heidelberg (2018). DOI: 10.1007/978-3-319-89963-3_14
11. Bui, T.L., Chatterjee, K., Gautam, T., Pavlogiannis, A., and Toman, V.: The Reads-from Equivalence for the TSO and PSO Memory Models. Proc. ACM Program. Lang. 5(OOPSLA) (2021). DOI: 10.1145/3485541
12. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., and Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM Program. Lang. 2(POPL), 31:1–31:30 (2017). DOI: 10.1145/3158119
13. Chase, D., and Lev, Y.: Dynamic circular work-stealing deque. In: SPAA 2005, pp. 21–28. ACM (2005). DOI: 10.1145/1073970.1073974
14. Chatterjee, K., Pavlogiannis, A., and Toman, V.: Value-Centric Dynamic Partial Order Reduction. Proc. ACM Program. Lang. 3(OOPSLA) (2019). DOI: 10.1145/3360550
15. Clarke, E.M., Kroening, D., and Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). DOI: 10.1007/978-3-540-24730-2_15
16. Coons, K.E., Musuvathi, M., and McKinley, K.S.: Bounded Partial-Order Reduction. In: OOPSLA 2013, pp. 833–848. ACM, Indianapolis, Indiana, USA (2013). DOI: 10.1145/2509136.2509556
17. Flanagan, C., and Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005, pp. 110–121. ACM, New York, NY, USA (2005). DOI: 10.1145/1040305.1040315
18. Gavrilenko, N., Ponce-de-León, H., Furbach, F., Heljanko, K., and Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Dillig, I., and Tasiran, S. (eds.) CAV 2019, pp. 355–365. Springer International Publishing, Cham (2019). DOI: 10.1007/978-3-030-25540-4_19
19. MISC
20. Godefroid, P.: Software Model Checking: The VeriSoft Approach. Form. Meth. Syst. Des. 26(2), 77–101 (2005). DOI: 10.1007/s10703-005-1489-x
21. Gotovos, A., Christakis, M., and Sagonas, K.: Test-driven development of concurrent programs using concuerror. In: Rikitake, K., and Stenman, E. (eds.) Erlang 2022 2011, pp. 51–61. ACM (2011). DOI: 10.1145/2034654.2034664
22. Haas, T., Meyer, R., and Ponce de León, H.: CAAT: Consistency as a Theory. Proc. ACM Program. Lang. 6(OOPSLA2) (2022). DOI: 10.1145/3563292
23. Knuth, D.E.: Estimating the Efficiency of Backtrack Programs. Math. Comput. 29(129), 121–136 (1975). DOI: 10.2307/2005469
24. Kokologiannakis, M., Lahav, O., Sagonas, K., and Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. 2(POPL), 17:1–17:32 (2017). DOI: 10.1145/3158105
25. Kokologiannakis, M., Lahav, O., and Vafeiadis, V.: Kater: Automating Weak Memory Model Metatheory and Consistency Checking. Proc. ACM Program. Lang. 7(POPL) (2023). DOI: 10.1145/3571212
26. MISC

27. Kokologiannakis, M., Marmanis, I., Gladstein, V., and Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. 6(POPL) (2022). DOI: 10.1145/3498711
28. Kokologiannakis, M., Marmanis, I., and Vafeiadis, V.: Unblocking Dynamic Partial Order Reduction. In: CAV 2023, pp. 230–250. Springer (2023). DOI: 10.1007/978-3-031-37706-8_12
29. Kokologiannakis, M., Ren, X., and Vafeiadis, V.: Dynamic Partial Order Reductions for Spinloops. In: FMCAD 2021, pp. 163–172. IEEE (2021). DOI: 10.34727/2021/isbn.978-3-85448-046-4_25
30. Kokologiannakis, M., and Vafeiadis, V.: BAM: Efficient Model Checking for Barriers. In: NETYS 2021. LNCS, Springer, Heidelberg (2021). DOI: 10.1007/978-3-030-91014-3_16
31. Kokologiannakis, M., and Vafeiadis, V.: GenMC: A model checker for weak memory models. In: Silva, A., and Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 427–440. Springer, Heidelberg (2021). DOI: 10.1007/978-3-030-81685-8_20
32. Kozen, D.: Kleene Algebra with Tests. ACM Trans. Program. Lang. Syst. 19(3) (1997). DOI: 10.1145/256167.256195
33. Lahav, O., Giannarakis, N., and Vafeiadis, V.: Taming Release-acquire Consistency. In: POPL 2016, pp. 649–662. ACM, St. Petersburg, FL, USA (2016). DOI: 10.1145/2837614.2837643
34. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., and Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017, pp. 618–632. ACM, Barcelona, Spain (2017). DOI: 10.1145/3062341.3062352
35. Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9), 690–691 (1979). DOI: 10.1109/TC.1979.1675439
36. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., and Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: OSDI 2008, pp. 267–280. USENIX Association (2008)
37. Nguyen, H.T.T., Rodríguez, C., Sousa, M., Coti, C., and Petrucci, L.: Quasi-optimal partial order reduction. In: Chockler, H., and Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 354–371. Springer, Heidelberg (2018). DOI: 10.1007/978-3-319-96142-2_22
38. Norris, B., and Demsky, B.: CDSChecker: Checking concurrent data structures written with C/C++ atomics. In: OOPSLA 2013, pp. 131–150. ACM (2013). DOI: 10.1145/2509136.2509514
39. Podkopaev, A., Lahav, O., and Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. 3(POPL), 69:1–69:31 (2019). DOI: 10.1145/3290382
40. Rodríguez, C., Sousa, M., Sharma, S., and Kroening, D.: Unfolding-based Partial Order Reduction. In: CONCUR 2015. LIPIcs, pp. 456–469. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015). DOI: 10.4230/LIPIcs.CONCUR.2015.456
41. Wang, K., Converse, H., Gligoric, M., Misailovic, S., and Khurshid, S.: A Progress Bar for the JPF Search Using Program Executions. ACM SIGSOFT Softw. Eng. Notes 43(4), 55 (2018). DOI: 10.1145/3282517.3282525
42. Zhang, N., Kusano, M., and Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI 2015, pp. 250–259. ACM, New York, NY, USA (2015). DOI: 10.1145/2737924.2737956

Automata and Learning

Scalable Tree-based Register Automata Learning

Simon Dierl[1](✉), Paul Fiterau-Brostean[2](✉), Falk Howar[1](✉),
Bengt Jonsson[2](✉), Konstantinos Sagonas[2,3](✉),
and Fredrik Tåquist[2](✉)

[1] Technical University of Dortmund, Dortmund, Germany
{simon.dierl,falk.howar}@tu-dortmund.de
[2] Uppsala University, Uppsala, Sweden
fiteraup@yahoo.com, {bengt.jonsson,kostis,fredrik.takvist}@it.uu.se
[3] National Technical University of Athens, Athens, Greece

Abstract. Existing active automata learning (AAL) algorithms have demonstrated their potential in capturing the behavior of complex systems (e.g., in analyzing network protocol implementations). The most widely used AAL algorithms generate finite state machine models, such as Mealy machines. For many analysis tasks, however, it is crucial to generate richer classes of models that also show how relations between data parameters affect system behavior. Such models have shown potential to uncover critical bugs, but their learning algorithms do not scale beyond small and well curated experiments. In this paper, we present SL^λ, an effective and scalable register automata (RA) learning algorithm that significantly reduces the number of tests required for inferring models. It achieves this by combining a tree-based cost-efficient data structure with mechanisms for computing short and restricted tests. We have implemented SL^λ as a new algorithm in RALib. We evaluate its performance by comparing it against SL^*, the current state-of-the-art RA learning algorithm, in a series of experiments, and show superior performance and substantial asymptotic improvements in bigger systems.

Keywords: Active automata learning, Register automata

1 Introduction

Model Learning (aka *Active Automata Learning* (AAL) [7,40,50]) infers automata models that represent the dynamic behavior of a software or hardware component from tests. Models obtained through (active) learning have proven useful for many purposes, such as analyzing security protocols [18,19,25,41,45], mining APIs [6], supporting model-based testing [26,47,52] and conformance testing [5]. The AAL algorithms employed in these works are efficient and supported by various domain-specific optimizations (e.g., [31]), but they all generate finite state machine (FSM) models, such as Mealy machines.

For many analysis tasks, however, it is crucial for models to also be able to describe *data flow*, i.e., constraints on data parameters that are passed when the component interacts with its environment, as well as the mutual influence between dynamic behavior and data flow. For instance, models of protocol components must describe how different parameter values in sequence numbers, identifiers, etc. influence the behavior, and vice versa. Existing techniques for

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 87–108, 2024.
https://doi.org/10.1007/978-3-031-57249-4_5

extending AAL to *Extended FSM* (EFSM) models [1,8,10] take several different approaches. Some reduce the problem to inferring FSMs by using manually supplied abstractions on the data domain [1], which requires insight into the control/data dependencies of a system under learning (SUL). Others extend AAL for finite state models by allowing transitions to contain predicates over rich data domains, but cannot generate state variables to model data dependencies between consecutive interactions [13,35]. Finally, there exist extensions of AAL to EFSM models with guards and state variables, such as *register automata* [2,3,10]. While their potential has been shown by being able to uncover critical bugs in e.g., TCP implementations [15,16], their learning algorithms do not scale beyond small and well curated experiments.

We follow the third line of works and address the scalability of register automata (RA) learning algorithms in our work. The main challenge when scaling AAL algorithms is reducing the number of tests that learners perform on a SUL. Generally, these tests are sequences of actions of the form $u \cdot v$, where u is the prefix and v the suffix of the sequence. Tests $u \cdot v$ and $u' \cdot v$ are then used to determine if prefixes u and u' can be distinguished based on the SUL's output triggered by v. When inferring RA models, prefixes are sequences of actions with data values, e.g., $\mathsf{push}(1)\,\mathsf{push}(2)$, and suffixes are sequences of actions with symbolic parameters, e.g., $\mathsf{pop}(p_1)\,\mathsf{pop}(p_2)$, that, when instantiated, can incur a number of tests that is exponential in the length of the suffix for identifying dependencies between prefix values and suffix parameters, e.g., different test outcomes for $(p_1 = 2 \wedge p_2 = 1)$ and $(p_1 = 2 \wedge p_2 = 3)$, and for distinguishing prefixes based on suffixes. To make register automata learning scalable, it is crucial to reduce the use of suffixes in tests along three dimensions: (i) First, it is important to use only *few tests*. (ii) Second, when using suffixes in tests, *shorter suffixes* should be preferred over longer ones. (iii) Third, it is essential to *restrict tests to relevant dependencies* between prefix values and suffix parameters instead of bluntly testing all possible dependencies.

In this paper, we present the SL^λ algorithm for learning register automata which achieves scalability by optimizing the use of tests and suffixes in tests in the three stated dimensions. SL^λ uses a *classification tree* as a data structure, constructs a minimal prefix-closed set of prefixes and a suffix-closed set of *short* and *restricted* suffixes for identifying and distinguishing locations, transitions, and registers. Technically, we adopt the idea of using a classification tree from learners for FSMs [32,33] where it proved very successful for reducing tests. We also adopt the technique of computing short suffixes incrementally in order to keep them short [7,32]. This has not been studied for RAs before and leads to an improved worst case complexity compared to state-of-the-art approaches (Theorem 1). Finally, we show how suffixes can be restricted to relevant data dependencies, which is essential for achieving scalability (Section 4).

We have implemented SL^λ as a new algorithm in the RALib[4] tool [9]. For comparison, we have also implemented in RALib the SL^{CT} algorithm that uses a classification tree but relies on suffixes from counterexamples instead of

[4] RALib is available at https://github.com/LearnLib/ralib.

computing short suffixes from inconsistencies. We evaluate the SL^{λ} algorithm by comparing its performance against the SL^{*} [10] and SL^{CT} algorithms in a series of experiments, confirming that: (i) classification trees scale much better than observation tables for register automata, (ii) using restricted suffixes leads to a dramatic reduction of tests for all compared algorithms, and (iii) computing short suffixes from inconsistencies outperforms using suffixes from counterexamples.

Related Work. For a broad overview of AAL refer to the survey paper of de la Higuera [27] from 2005 and to a more recent paper by Howar and Steffen [28].

Learning beyond DFAs has been investigated for many models aside from register automata. For example, algorithms have been presented for workflow Petri nets [14], data automata [24], generic nondeterministic transition systems [51], symbolic automata [13], one-timer automata [48], and systems of procedural automata [22]. Learning of register automata has been performed by combining a FSM learner with the Tomte front-end [2,3]. A different approach using bespoke RA learning algorithms [30,37] has been implemented in RALib. Active learning algorithms for nominal automata, which extend FSMs to infinite alphabets and infinite sets of states, have also been developed [38]. While the expressivity of nominal DFAs is equivalent to that of deterministic register automata with equality, nominal automata do not represent registers symbolically but through permutations on infinite sets, leading to big models (e.g., for storing some data value twice) and active learning algorithms with a high query complexity.

Applications of AAL are diverse. Active learning enables the generation of behavioral models for software [43,46], e.g. for network protocol implementations [41,53], enabling security analyses and model checking [4,20,21]. It can be used in testing [36,44] and to enable formal analyses [50]. Finally, it can be combined with passive learning approaches to support life-long learning [23]. More theoretical advances include the use of Galois connections to model SUL-oracle mappers [34] and the introduction of apartness [49], to formalize state distinction.

Outline. We present the key ideas in tree-based learning of RA informally in the next section, before providing formal definitions of basic concepts in Section 3. Sections 4 to 6 present the SL^{λ} algorithm, its properties, and the experimental evaluation of its performance. The paper ends with few concluding remarks.

2 Main Ideas

In this section, we introduce the main ideas behind the SL^{λ} algorithm. As illustrating example, we will use a stack of capacity two, which stores a sequence of natural numbers. The stack supports the operations push and pop, both of which take one natural number as a parameter. The operation push(d) succeeds if the stack is not full, i.e., contains at most one element; the operation pop(d) succeeds if the last pushed and not yet popped element is d. Let a *symbol* denote an operation with data value, such as push(1), and let $\mathcal{L}_{Stack}$ denote the prefix-closed language consisting of the words of symbols representing sequences of successful operations. Figure 1 shows an acceptor for $\mathcal{L}_{Stack}$. The initial location

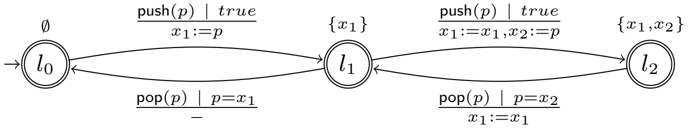


Fig. 1: Register automaton accepting language of stack with capacity two.

l_0 corresponds to an empty stack, location l_1 corresponds to a stack with one element, and l_2 to a location where the stack is full. There is also an implicit sink location for each word that is not accepted by $\mathcal{L}_{Stack}$, e.g. pushing a third element, or popping a non-top element. In each location, registers contain the elements in the stack: for $i = 0, 1, 2$, location l_i has i registers, where the register with the highest index contains the topmost stack element.

The task of the SL^λ algorithm is to learn the acceptor in Fig. 1 in a black-box scenario, i.e., knowing only the operations (push and pop) and the relations that may be used in guards (here tests for equality), by asking two kinds of queries. A *membership query* asks whether a word w is in $\mathcal{L}$; it can be realized by a simple test. An *equivalence query* asks whether a hypothesis RA accepts $\mathcal{L}$; if so, the query is answered by *yes*, otherwise by a *counterexample*, which is a word on which the hypothesis and $\mathcal{L}$ disagree; in a black box setting it is typically approximated by a conformance testing algorithm. Like other AAL algorithms, SL^λ iterates a cycle in which membership queries are used to construct a hypothesis, which is then subject to validation by an equivalence query. If a counterexample is found, hypothesis construction is resumed, etc., until a hypothesis agrees with $\mathcal{L}$.

Classical AAL algorithms that learn DFAs maintain an expanding set of words, S_p, called *short prefixes*, and an expanding set of words, called *suffixes*, which induce an equivalence relation $\equiv$ on prefixes, defined by $u \equiv u'$ iff $uv \in \mathcal{L} \Leftrightarrow u'v \in \mathcal{L}$ for all suffixes v; this allows equivalence classes of prefixes to represent states in a DFA. The SL^λ algorithm maintains a set U of data words called *prefixes*, which is the union of S_p and one-symbol extensions of elements in S_p. Instead of suffixes, SL^λ maintains a set $\mathcal{V}$ of *symbolic suffixes*, each of which is a *parameterized* word, i.e., a word where data values are replaced by parameters $p_1, \ldots, p_m$. For each prefix u, say push(0), and symbolic suffix $\boldsymbol{v}$, say push(p_1)pop(p_2), membership in $\mathcal{L}$ of words of form $u\boldsymbol{v}$ depends on the relation between the data values of u and the parameters p_1, p_2 of $\boldsymbol{v}$, which in SL^λ is represented by a function $\mathcal{L}[u, \boldsymbol{v}]$ with parameters x_1 (representing the data value of u), p_1, and p_2. In this case $\mathcal{L}[u, \boldsymbol{v}](x_1, p_1, p_2)$ is $+$ iff $p_2 = p_1$ and $-$ otherwise. In SL^λ, such functions are represented as decision trees of a specific form. Figure 2 shows the decision tree for the just described function. Note that it checks constraints for parameters one at a time: first the constraint on only p_1 (which is *true*), and thereafter the constraint on p_2 (a comparison with p_1). Two prefixes u, u' are then equivalent w.r.t. $\mathcal{V}$ if $\mathcal{L}[u, \boldsymbol{v}]$ and $\mathcal{L}[u', \boldsymbol{v}]$ are "isomorphic modulo renaming" for all $\boldsymbol{v} \in \mathcal{V}$ (details in Section 4).

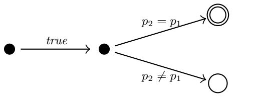


Fig. 2: Decision tree for $\mathcal{L}[u, \boldsymbol{v}](x_1, p_1, p_2)$.

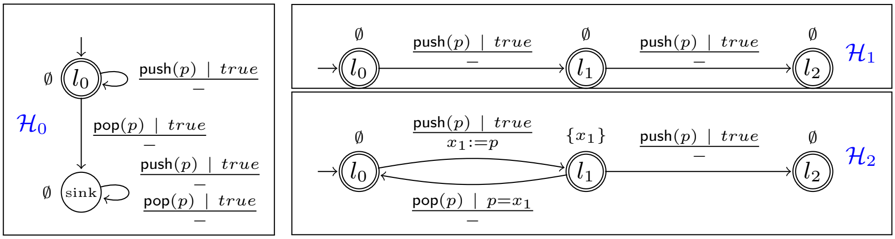


Fig. 3: Three hypotheses constructed by SL^λ: $\mathcal{H}_0$ (left), $\mathcal{H}_1$ and $\mathcal{H}_2$ (right).

Functions of form $\mathcal{L}[u, \boldsymbol{v}]$ are generated by so-called *tree queries*, which perform membership queries for relevant combinations of relations between data values in u and parameters in $\boldsymbol{v}$, and summarize the results in a canonical way. The tree query above requires five membership queries. SL^λ employs techniques for reducing this number by restricting the symbolic suffix; see end of this section.

Initially, S_p and $\mathcal{V}$ contain only the empty sequence ϵ. Since ϵ is a short prefix, one-symbol extensions, push(0) and pop(0), are entered into U. Tree queries are performed for the prefixes in U and the empty suffix, revealing that push(0) is accepted and pop(0) is rejected. Thus, push(0) cannot be distinguished from ϵ, but pop(0) can, so it must lead to a new location, hereafter referred to as the *sink*, which is therefore added to S_p. One-symbol extensions of pop(0), in this case pop(0)push(1) and pop(0)pop(1), are added to U and tree queries for them and the empty suffix are performed, revealing that they cannot be separated from the sink. At this point, we can formulate hypothesis $\mathcal{H}_0$ in Fig. 3(left) from S_p, U, and the computed decision trees.

This hypothesis is then subject to validation. Assume that it finds the counterexample push(0)push(1)push(2), which is accepted by $\mathcal{H}_0$ but rejected by $\mathcal{L}_{Stack}$. Analysis of this counterexample reveals that ϵ and push(0) are inequivalent, since they are separated by the suffix push(p_1)push(p_2) (since the concatenation of ϵ and push(p_1)push(p_2) is accepted for all p_1, p_2 but push(0) · push(1)push(2) is always rejected for all p_1, p_2). It could seem natural to add push(p_1)push(p_2) to $\mathcal{V}$, but SL^λ will not do that, since it follows the principle (from L^λ [29]) that a new prefix in S_p must extend an existing prefix by one symbol, and that a new suffix in $\mathcal{V}$ must prepend one symbol to an existing one. This principle keeps S_p prefix-closed and $\mathcal{V}$ suffix-closed, and aims to avoid inclusion of unnecessarily long sequences. Therefore, instead of adding push(p_1)push(p_2) as a suffix, SL^λ enters the prefix push(0) into S_p, and adds one-symbol extensions of push(0), in this case push(0)push(1) and push(0)pop(1), to U. It notes that push(0)push(1) is inequivalent to both ϵ and push(0), separated by the suffix push(p_1). Again, push(0)push(1) is therefore promoted to a short prefix, and its one-symbol extensions, push(0)push(1)push(2) and push(0)push(1)pop(2), are entered into U. Now, SL^λ is able to add suffixes to $\mathcal{V}$ that separate all prefixes in S_p, by two operations that achieve consistency.

1. The push-extensions of ϵ and push(0)push(1), (i.e., push(0) and push(0)push(1)-push(2)) are separated by the empty suffix, hence these two prefixes are

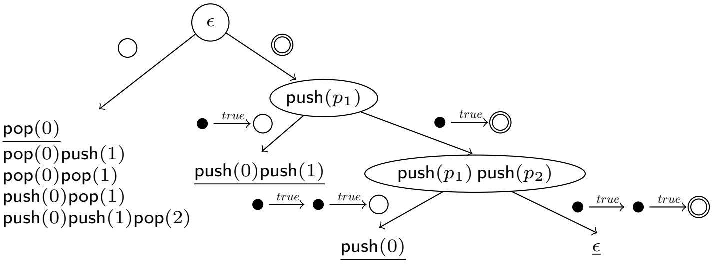


Fig. 4: Classification tree for hypothesis $\mathcal{H}_1$ in Fig. 3. Short prefixes are underlined.

separated by the suffix push(p_1), a one-symbol extension of ϵ which is added to $\mathcal{V}$.

2. The push-extensions of ϵ and push(0) (i.e., push(0) and push(0)push(1)) are separated by the suffix push(p_1), hence ϵ and push(0) are separated by push(p_1)push(p_2), formed by prepending a symbol to the just added suffix push(p_1), which is added to $\mathcal{V}$.

After adding the suffixes, the closedness and consistency criteria are met, producing hypothesis $\mathcal{H}_1$ in Fig. 3(right, top). Assume that the validation of $\mathcal{H}_1$ finds counterexample push(0)pop(0), which is in $\mathcal{L}_{Stack}$, but rejected by $\mathcal{H}_1$. This counterexample reveals that after push(0), the two continuations pop(0) and pop(1) lead to inequivalent locations (separated by suffix ϵ), suggesting to refine the pop(p)-transition after push(0). To this end, $\mathcal{V}$ is extended by a suffix formed by prepending pop(p) to the empty suffix, and a tree query is invoked for $\mathcal{L}[\mathsf{push}(0), \mathsf{pop}(p_1)]$, which is + iff $p_1 = x_1$ and − otherwise. Since $\mathcal{L}[\mathsf{push}(0), \mathsf{pop}(p_1)]$ makes a test for x_1, which represents the data value of push(0), we infer that the data parameter of the push(0)-prefix must be remembered in a register, and that the pop(p)-transition must be split into two with guards $(x_1 \neq p)$ and $(x_1 = p)$. The resulting hypothesis, $\mathcal{H}_2$, is shown in Fig. 3(right, bottom), which is subject to another round of validation; the subsequent hypothesis construction reveals the pop-transitions from l_2 in Fig. 1.

In SL^{λ}, the sets U and $\mathcal{V}$ are maintained in a *classification tree CT*, a data structure that is specially designed to represent how the suffixes in $\mathcal{V}$ partition U into equivalence classes corresponding to locations. This permits an optimization that can elide superfluous membership queries. A classification tree is a decision tree. Each leaf is labeled by a subset of U. Each inner node is labeled by a symbolic suffix $\boldsymbol{v}$ and induces a subtree for each equivalence class w.r.t. $\boldsymbol{v}$, whose leaves contain prefixes in this equivalence class. For example, in Fig. 4, which shows a CT corresponding to hypothesis $\mathcal{H}_1$, the nodes are labeled by the suffixes ϵ, push(p_1) and push(p_1)push(p_2), which separate the leaves into four equivalence classes corresponding to the four locations in Fig. 1. Each edge is labeled by the results of the tree queries for a prefix in its equivalence class and the symbolic suffix of the source node.

Each tree query requires a number of membership queries which may grow exponentially with the length of the suffix. SL^{λ} reduces this number by *restricting* the involved symbolic suffix to induce fewer membership queries, as long as the tree query can still make the separation between prefixes or transitions for which it was invoked. To illustrate, recall that the analysis of the counterexample $\mathsf{push}(0)\mathsf{push}(1)\mathsf{push}(2)$ for $\mathcal{H}_0$ shows that ϵ and $\mathsf{push}(0)$ are inequivalent. To separate these, we need not naïvely use the symbolic suffix $\mathsf{push}(p_1)\mathsf{push}(p_2)$; but we can restrict it by considering only values of p_1 and p_2 that are *fresh*, i.e., different from all other preceding parameters in the prefix and suffix. With this restriction, the suffix can still separate ϵ and $\mathsf{push}(0)$, and the tree query for prefix $\mathsf{push}(0)$ requires only one membership query instead of five.

3 Data Languages and Register Automata

In this section, we review background concepts on data languages and register automata. Our definitions are parameterized on a *theory*, which is a pair $\langle \mathcal{D}, \mathcal{R} \rangle$ where $\mathcal{D}$ is a (typically infinite) domain of *data values*, and $\mathcal{R}$ is a set of *relations* (of arbitrary arity) on $\mathcal{D}$. Examples of theories include: (i) $\langle \mathbb{N}, \{=\} \rangle$, the theory of natural numbers with equality, and (ii) $\langle \mathbb{R}, \{<\} \rangle$, the theory of real numbers with inequality; this theory also allows to express equality between elements. Theories can be extended with constants (allowing, e.g., theories of sums with constants).

Data Languages. We assume a set Σ of *actions*, each with an arity that determines how many parameters it takes from the domain $\mathcal{D}$. For simplicity, we assume that all actions have arity 1; our techniques can be extended to handle actions with arbitrary arities. A *data symbol* is a term of form $\alpha(\mathtt{d})$, where α is an action and $\mathtt{d} \in \mathcal{D}$ is a data value. A *data word* (or simply *word*) is a finite sequence of data symbols. The concatenation of two words u and v is denoted uv, often we then refer to u as a *prefix* and v as a *suffix*. Two words $w = \alpha_1(\mathtt{d}_1) \ldots \alpha_n(\mathtt{d}_n)$ and $w' = \alpha_1(\mathtt{d}'_1) \ldots \alpha_n(\mathtt{d}'_n)$ with the same sequences of actions are *$\mathcal{R}$-indistinguishable*, denoted $w \approx_{\mathcal{R}} w'$, if $R(\mathtt{d}_{i_1}, \ldots, \mathtt{d}_{i_j}) \Leftrightarrow R(\mathtt{d}'_{i_1}, \ldots, \mathtt{d}'_{i_j})$ whenever R is a j-ary relation in $\mathcal{R}$ and $i_1, \cdots, i_j$ are indices among $1 \ldots n$. A *data language* $\mathcal{L}$ is a set of data words that respects $\mathcal{R}$ in the sense that $w \approx_{\mathcal{R}} w'$ implies $w \in \mathcal{L} \Leftrightarrow w' \in \mathcal{L}$. We often represent data languages as mappings from the set of words to $\{+, -\}$, where $+$ stands for *accept* and $-$ for *reject*.

Register Automata. We assume a set of *registers* $x_1, x_2, \ldots$, and a set of *formal parameters* $p, p_1, p_2, \ldots$. A *parameterized symbol* is a term of form $\alpha(p)$, where α is an action and p a *formal parameter*. A *constraint* is a conjunction of negated and unnegated relations (from $\mathcal{R}$) over registers and parameters. An *assignment* is a parallel update of registers with values from registers or the formal parameter p. We represent it as a mapping π from $\{x_{i_1}, \ldots, x_{i_m}\}$ to $\{x_{j_1}, \ldots, x_{j_n}\} \cup \{p\}$, meaning that the value $\pi(x_{i_k})$ is assigned to x_{i_k}, for $k = 1, \ldots, m$. In multiple-assignment notation, this would be written $x_{i_1}, \ldots, x_{i_m} := \pi(x_{i_1}), \ldots, \pi(x_{i_m})$.

Definition 1. *A* register automaton *(RA) is a tuple $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$, where*

- *L is a finite set of* locations*, with $l_0 \in L$ as the* initial location*,*
- *$\mathcal{X}$ maps each location $l \in L$ to a finite set $\mathcal{X}(l)$ of registers,*

- *Γ is a finite set of* transitions*, each of form $\langle l, \alpha(p), g, \pi, l' \rangle$, where*
    - *$l \in L$ is a source location and $l' \in L$ is a target location,*
    - *$\alpha(p)$ is a parameterized symbol,*
    - *g, the* guard*, is a constraint over p and $\mathcal{X}(l)$, and*
    - *π (the* assignment*) is a mapping from $\mathcal{X}(l')$ to $\mathcal{X}(l) \cup \{p\}$, and*
- *λ maps each $l \in L$ to $\{+, -\}$, where $+$ denotes* accept *and $-$* reject*.*

A *state* of a RA $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ is a pair $\langle l, \mu \rangle$ where $l \in L$ and μ is a valuation over $\mathcal{X}(l)$, i.e., a mapping from $\mathcal{X}(l)$ to $\mathcal{D}$. A *step* of $\mathcal{A}$, denoted $\langle l, \mu \rangle \xrightarrow{\alpha(\mathtt{d})} \langle l', \mu' \rangle$, transfers the state of $\mathcal{A}$ from $\langle l, \mu \rangle$ to $\langle l', \mu' \rangle$ on input of the data symbol $\alpha(\mathtt{d})$ if there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle \in \Gamma$ such that (i) $\mu \models g[\mathtt{d}/p]$, i.e., d satisfies the guard g under the valuation μ, and (ii) μ' is defined by $\mu'(x_i) = \mu(x_j)$ if $\pi(x_i) = x_j$, otherwise $\mu'(x_i) = d$ if $\pi(x_i) = p$. A *run* of $\mathcal{A}$ over a data word $w = \alpha(\mathtt{d}_1) \ldots \alpha(\mathtt{d}_n)$ is a sequence of steps of $\mathcal{A}$

$$\langle l_0, \mu_0 \rangle \xrightarrow{\alpha_1(\mathtt{d}_1)} \langle l_1, \mu_1 \rangle \quad \ldots \quad \langle l_{n-1}, \mu_{n-1} \rangle \xrightarrow{\alpha_n(\mathtt{d}_n)} \langle l_n, \mu_n \rangle$$

for some initial valuation μ_0. The run is *accepting* if $\lambda(l_n) = +$ and *rejecting* if $\lambda(l_n) = -$. The word w is *accepted (rejected) by $\mathcal{A}$ under μ_0* if $\mathcal{A}$ has an accepting (rejecting) run over w from $\langle l_0, \mu_0 \rangle$. Define the language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ as the set of words accepted by $\mathcal{A}$. A language is *regular* if it is the language of some RA.

We require a RA to be *determinate*, meaning that there is no data word over which it has both accepting and rejecting runs. A determinate RA can be easily transformed into a deterministic one by strengthening its guards, and a deterministic RA is by definition also determinate. Our construction of RAs in Section 4 will generate determinate RAs which are not necessarily deterministic. RAs have been extended to Register Mealy Machines (RMM) in several works and it has been established how RA learning algorithms can be used to infer models of systems with inputs and outputs [9], which we do, too.

4 The SL^λ Learning Algorithm

In this section, we present the main building blocks of SL^λ before an overview of the main algorithm, followed by techniques for reducing the cost of tree queries (page 13) and for analyzing counterexamples (page 14).

Symbolic Decision Trees. The functions, of form $\mathcal{L}[u, \boldsymbol{v}]$, that result from tree queries, should represent how the language $\mathcal{L}$ to be learned processes instantiations of $\boldsymbol{v}$ after the prefix u. Since SL^λ is intended to construct canonical RAs, it is natural to let these functions have the form of a tree-shaped "mini-RA", which we formalize as *symbolic decision trees* of a certain form.

For a word $u = \alpha_1(\mathtt{d}_1) \ldots \alpha_k(\mathtt{d}_k)$ and a symbolic suffix $\boldsymbol{v} = \alpha'_1(p_1) \ldots \alpha'_m(p_m)$ a *$(u, \boldsymbol{v})$-path* τ is a sequence $g_1, \ldots, g_m$, where each g_i is a constraint over $x_1, \ldots, x_k$ and $p_1, \ldots, p_i$. Define the condition represented by τ, denoted $\mathcal{G}_\tau$, as $g_1 \wedge \cdots \wedge g_m$. A *$(u, \boldsymbol{v})$-tree* T is a mapping from a set $Dom(T)$ of $(u, \boldsymbol{v})$-paths to $\{+, -\}$. Write $\overline{\mathtt{d}}$ for $\mathtt{d}_1, \ldots, \mathtt{d}_k$, $\overline{\mathtt{d}'}$ for $\mathtt{d}'_1, \ldots, \mathtt{d}'_m$, $\overline{x}$ for $x_1, \ldots, x_k$ and $\overline{p}$ for

$p_1, \ldots, p_m$. A $(u, \boldsymbol{v})$-tree T can be seen a function with parameters $\overline{x}, \overline{p}$ to $\{+, -\}$, defined by $T(\overline{x}, \overline{p}) = T(\tau)$ whenever $\tau \in \mathit{Dom}(T)$ and $\mathcal{G}_\tau(\overline{x}, \overline{p})$ holds. That is, for data values $\overline{\mathsf{d}}$ and $\overline{\mathsf{d}'}$ and each $(u, \boldsymbol{v})$-path τ, we have $T(\overline{\mathsf{d}}, \overline{\mathsf{d}'}) = T(\tau)$ whenever $\mathcal{G}_\tau(\overline{\mathsf{d}}, \overline{\mathsf{d}'})$ is true. If $\mathcal{L}$ is a data language, then $\mathcal{L}[u, \boldsymbol{v}]$ is a $(u, \boldsymbol{v})$-tree representing membership in $\mathcal{L}$ in the sense that for any values of $p_1, \ldots, p_m$ we have $\mathcal{L}[u, \boldsymbol{v}](\overline{\mathsf{d}}, \overline{p}) = +$ iff $u\alpha'_1(p_1) \ldots \alpha'_m(p_m) \in \mathcal{L}$, and $\mathcal{L}[u, \boldsymbol{v}](\overline{\mathsf{d}}, \overline{p}) = -$ iff $u\alpha'_1(p_1) \ldots \alpha'_m(p_m) \notin \mathcal{L}$. For example, Fig. 2 shows a $(u, \boldsymbol{v})$-tree where $u = \mathsf{push}(\mathsf{d}_1)$ and $\boldsymbol{v} = \mathsf{push}(p_1)\mathsf{pop}(p_2)$. This tree maps the $(u, \boldsymbol{v})$-path $\mathit{true} \wedge p_2 = p_1$ to $+$ and $\mathit{true} \wedge p_2 \neq p_1$ to $-$. From this, we can determine, e.g., that the word $\mathsf{push}(0)\mathsf{push}(1)\mathsf{pop}(1) \in \mathcal{L}$, but $\mathsf{push}(0)\mathsf{push}(1)\mathsf{pop}(2) \notin \mathcal{L}$.

SL^λ generates $(u, \boldsymbol{v})$-trees $\mathcal{L}[u, \boldsymbol{v}]$ representing the language $\mathcal{L}$ to be learned through so-called *tree queries*, which perform membership queries for values of the data parameters $p_1, \ldots, p_m$ that cover relevant equivalence classes of $\approx_\mathcal{R}$.

From the results of tree queries, we can extract registers and guards in the location reached by a prefix u. Intuitively, the registers must remember the data values of u that occur in some guard in some $\mathcal{L}[u, \boldsymbol{v}]$, and the outgoing guards from the location reached by u can be derived from the initial guards in the trees $\mathcal{L}[u, \boldsymbol{v}]$, since the initial guards represent the constraints that are used when processing the first symbol of $\boldsymbol{v}$. Let $\mathit{mem}_{\boldsymbol{v}}(u)$, the set of *memorable parameters*, denote the set of registers among $\{x_1, \ldots, x_k\}$ that occur on some $(u, \boldsymbol{v})$-path in $\mathit{Dom}(\mathcal{L}[u, \boldsymbol{v}])$. Intuitively, if x_i is a memorable parameter, then the i^{th} data value in u will be remembered in the register x_i in the location reached by u. For example, for the stack in Fig. 1, in the location reached by $\mathsf{push}(0)$ the data value $\mathsf{d}_1 = 0$ is memorable so will be remembered in register x_1. Note that a $(u, \boldsymbol{v})$-tree itself does not have any registers: it only serves to show which registers are needed in the location reached by u in the to-be-constructed automaton. Define $\mathit{mem}_\mathcal{V}(u)$ as $\cup_{\boldsymbol{v} \in \mathcal{V}} \mathit{mem}_{\boldsymbol{v}}(u)$. For a prefix u and symbolic suffix $\boldsymbol{v}$ whose first action is α, let $\mathcal{G}_{\{\boldsymbol{v}\}}(u, \alpha)$ denote the initial guards in the $(u, \boldsymbol{v})$-tree $\mathcal{L}[u, \boldsymbol{v}]$, with p_1 replaced by p. For a set $\mathcal{V}$, let $\mathcal{G}_\mathcal{V}(u, \alpha)$ denote the set of satisfiable conjunctions of guards in $\mathcal{G}_{\{\boldsymbol{v}\}}(u, \alpha)$ for $\boldsymbol{v} \in \mathcal{V}$ with first action α.

Two $(u, \boldsymbol{v})$-trees, T and T', are *equivalent* denoted $T \equiv T'$, if $\mathit{Dom}(T) = \mathit{Dom}(T')$ and $T(\tau) = T'(\tau)$ for each $\tau \in \mathit{Dom}(T)$. For a mapping γ on registers, we define its extension to $(u, \boldsymbol{v})$-paths in the natural way. For a $(u, \boldsymbol{v})$-tree T, we define $\gamma(T)$ by $\mathit{Dom}(\gamma(T)) = \{\gamma(\tau) \ : \ \tau \in \mathit{Dom}(T)\}$ and $\gamma(T)(\gamma(\tau)) = T(\tau)$.

Let $u \equiv_\mathcal{V} u'$ denote that $\mathcal{L}[u, \boldsymbol{v}] \equiv \mathcal{L}[u', \boldsymbol{v}]$, for all symbolic suffixes $\boldsymbol{v} \in \mathcal{V}$. Let $u \simeq^\gamma_\mathcal{V} u'$ denote that γ is a bijection from $\mathit{mem}_\mathcal{V}(u)$ to $\mathit{mem}_\mathcal{V}(u')$ such that for all $\boldsymbol{v} \in \mathcal{V}$ we have $\gamma(\mathcal{L}[u, \boldsymbol{v}]) \equiv \mathcal{L}[u', \boldsymbol{v}]$. Let $u \simeq_\mathcal{V} u'$ denote that $u \simeq^\gamma_\mathcal{V} u'$ for some bijection γ. Intuitively, two words u and u' are equivalent if there is a bijection γ which for each $\boldsymbol{v} \in \mathcal{V}$ transforms $\mathcal{L}[u, \boldsymbol{v}]$ to $\mathcal{L}[u', \boldsymbol{v}]$, Note that in general, when $u \simeq_\mathcal{V} u'$, there can be several such bijections.

Data Structures. During the construction of a hypothesis, the SL^λ algorithm maintains: (i) a prefix-closed set S_p of *short prefixes*, representing locations, (ii) and a set of one-symbol extensions of the prefixes in S_p, representing transitions; we use U to represent the union of S_p and this set, and (iii) a suffix-closed set $\mathcal{V}$ of symbolic suffixes. Each one-symbol extension of form $u\alpha(\mathsf{d})$ is formed to

Algorithm 1: Operations on the Classification Tree.

```
Function $Sift(u, N)$ is
    if $N$ is a leaf then $\mathcal{U} \leftarrow \mathcal{U}[u \mapsto N]$
    else
        Compute $\mathcal{L}[u, suff(N)]$
        if $N$ has child $N'$ with $u \simeq_{\mathcal{V}(N)} rp(N')$ then $Sift(u, N')$
        else
            Create new leaf $N'$ as child of $N$ with $rp(N') = u$
            $Sift(u, N')$

Function $Expand(u)$ is
    $S_p \leftarrow S_p \cup \{u\}$
    for $\alpha \in \Sigma$ do $Sift(u\alpha(\mathtt{d}_u^g), root(CT))$ for each $g \in \mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$

Function $Refine(N, \boldsymbol{v})$ is
    Replace $N$ by an inner node $N'$ with $suff(N') = \boldsymbol{v}$
    for $u \in \mathcal{U}^{-1}(N)$ do $Sift(u, N')$
```

let $\mathtt{d}$ satisfy a specific guard g; we then always choose $\mathtt{d}$ as a *representative data value*, denoted $\mathtt{d}_u^g$, satisfying g after u.

The sets U and $\mathcal{V}$ are maintained in a *classification tree* CT, which is designed to represent how the suffixes in $\mathcal{V}$ partition the set U into equivalence classes corresponding to locations. A classification tree is a rooted tree, consisting of nodes connected by edges. Each inner node is labeled by a symbolic suffix, and each leaf is labeled by a subset of U. To each node N is assigned a representative prefix $rp(N)$ in U. For a node N, let $suff(N)$ its suffix and $\mathcal{V}(N)$ denote the set of symbolic suffixes of N and all its ancestors in the tree. Each outgoing edge from N corresponds to an equivalence class of $\simeq_{\mathcal{V}(N)}$ from which a representative member is chosen as the representative prefix of its target node. Each leaf node N is labeled by a set of data words, which are all in the same equivalence class of $\simeq_{\mathcal{V}(N)}$. Thus, nodes in different leaves are guaranteed to be inequivalent, since they are separated by the symbolic suffixes in $\mathcal{V}(lca(N,\ N'))$, where $lca(N,\ N')$ is the lowest common ancestor node of N and N'. We let $\mathcal{U}$ denote the mapping, which maps each prefix $u \in U$ to the classification tree leaf where it is contained. We also let $\mathcal{V}(u)$ denote $\mathcal{V}(\mathcal{U}(u))$, the suffixes of all ancestors of $\mathcal{U}(u)$. The representative prefix, $rp(N)$, of each leaf node N will induce a location in the RA to be constructed.

The insertion of a new prefix u into the classification tree CT is performed by function *Sift* (cf. Algorithm 1). It traverses the CT from the root downwards. At each internal node N, it checks whether $u \simeq_{\mathcal{V}(N)} rp(N')$ for any child N' of N. If so, it continues the traversal at N', otherwise a new child of N is created as a leaf N with $rp(N) = u$. When reaching a leaf N, the mapping $\mathcal{U}$ is updated to reflect that u has been sifted to N. In the classification tree in Fig. 4, e.g., ϵ is the representative prefix of inner nodes $\mathsf{push}(p_1)$ and $\mathsf{push}(p_2)\mathsf{push}(p_2)$ as it is the first prefix that was sifted down this path. The short prefix $\mathsf{push}(0)$ at the second leaf from right was sifted from the root to $\mathsf{push}(p_1)$ and then to $\mathsf{push}(p_1)\mathsf{push}(p_2)$

Algorithm 2: SL^λ Learning.

```
Initialize $CT$ as inner node $root(CT)$ with suffix $\epsilon$ and $U \leftarrow \emptyset$, $S_p \leftarrow \emptyset$
$Sift(\epsilon, root(CT))$
HYP: repeat
    ▷ Check closedness
    if $exists\ leaf\ N\ for\ which\ \mathcal{U}^{-1}(N) \cap S_p = \emptyset$ then                                   // location
        $Expand(u)$ for some $u \in \mathcal{U}^{-1}(N)$
    if $u \in S_p\ and\ g \in \mathcal{G}_{\mathcal{V}(u)}(u, \alpha)\ but\ u\alpha(\mathtt{d}^g_u) \notin U$ then                   // transition
        $Sift(u\alpha(\mathtt{d}^g_u), root(CT))$
    if $u\alpha(\mathtt{d}) \in U\ s.t.\ mem_{\mathcal{V}(u\alpha(\mathtt{d}))}(u\alpha(\mathtt{d})) \not\subseteq mem_{\mathcal{V}(u)}(u) \cup \{x_{|u]+1}\}$ then   // register
        Let $\boldsymbol{v} \in \mathcal{V}(u\alpha(\mathtt{d}))$ with $mem_{\boldsymbol{v}}(u\alpha(\mathtt{d}))) \not\subseteq (mem_{\mathcal{V}(u)}(u) \cup \{x_{|u]+1}\})$
        $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$
    ▷ Check consistency
    if $u, u' \in \mathcal{U}^{-1}(L) \cap S_p\ with\ u \simeq^{\gamma}_{\mathcal{V}(N)} u'\ for\ leaf\ N\ with$
     $u\alpha(\mathtt{d}^g_u), u'\alpha(\mathtt{d}^{\gamma(g)}_{u'}) \in U\ but\ \mathcal{U}(u\alpha(\mathtt{d}^g_u)) \neq \mathcal{U}(u'\alpha(\mathtt{d}^{\gamma(g)}_{u'}))$ then      // location
        $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$ with $\boldsymbol{v} = suff(lca(u\alpha(\mathtt{d}^g_u),\ u'\alpha(\mathtt{d}^{\gamma(g)}_{u'})))$
    if $g \in \mathcal{G}_{\mathcal{V}(u)}(u, \alpha)\ and\ u\alpha(\mathtt{d}^g_u), u\alpha(\mathtt{d}) \in U\ with\ (u, \mathtt{d}) \vDash g\ but$
     $\mathcal{U}(u\alpha(\mathtt{d}^g_u)) \neq \mathcal{U}(u\alpha(\mathtt{d}))$ then                                        // transition(a)
        $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$ with $\boldsymbol{v} = suff(lca(u\alpha(\mathtt{d}^g_u),\ u\alpha(\mathtt{d})))$
    if $u\alpha(\mathtt{d}^g_u), u\alpha(\mathtt{d}) \in U\ with\ u\alpha(\mathtt{d}^g_u) \not\simeq^{\mathbf{id}}_{\mathcal{V}(u\alpha(\mathtt{d}))} u\alpha(\mathtt{d})$ then               // transition(b)
        $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$ with $\boldsymbol{v}$ s.t. $u\alpha(\mathtt{d}^g_u) \not\simeq^{\mathbf{id}}_{\{\boldsymbol{v}\}} u\alpha(\mathtt{d})$
    if $u, u\alpha \in U\ with\ u \simeq^{\gamma}_{\mathcal{V}(u)} u\ and\ no\ extension\ \gamma'\ of\ \gamma\ with$
     $u\alpha(\mathtt{d}) \simeq^{\gamma'}_{\mathcal{V}(u\alpha(\mathtt{d}))} u\alpha(\mathtt{d})$ then                                         // register
        $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$ with $\boldsymbol{v}$ s.t. $u\alpha(\mathtt{d}) \not\simeq^{\gamma'}_{\{\boldsymbol{v}\}} u\alpha(\mathtt{d})$ for any $\gamma'$
until $closed\ and\ consistent$
$\mathcal{H} \leftarrow$ Hypothesis$(CT)$
if $\exists w \in \Sigma^+\ s.t.\ \mathcal{H}(w) \neq \mathcal{L}(w)$ then $Analyze(w)$ and goto HYP else return $\mathcal{H}$
```

as $\mathsf{push}(0) \simeq_{\{\epsilon\}} \epsilon$ and $\mathsf{push}(0) \simeq_{\{\epsilon,\ \mathsf{push}(p_1)\}} \epsilon$. Since, however, $\mathsf{push}(0) \not\simeq_{\mathcal{V}} \epsilon$ for $\mathcal{V} = \{\epsilon,\ \mathsf{push}(p_1), \mathsf{push}(p_1)\mathsf{push}(p_2)\}$, a new leaf was created and $\mathsf{push}(0)$ was made the representative prefix of the new leaf and a short prefix.

The SL^λ Algorithm. The core of the SL^λ algorithm, shown in Algorithm 2, initializes the classification tree to consist of one (root) inner node, for the empty suffix (which classifies words as accepted or rejected); U and S_p are empty. It then sifts the empty prefix ϵ, thereby entering it into U. Thereafter, Algorithm 2 repeats a main loop in which CT is checked for a number of closedness and consistency properties. Whenever such a property is not satisfied, a corrective update is made by adding information to CT. These corrective updates fall into two categories, carried out by the following functions:

- *Expand* takes a prefix $u \in U$ and makes it into a short prefix. Since each short prefix must have a set of one-symbol extensions in U, the function forms one-symbol extensions of form $u\alpha(\mathtt{d}_u^g)$, which are entered into the classification tree by sifting.

- *Refine* takes a leaf node N and a symbolic suffix $\boldsymbol{v}$; it sifts the prefixes u in N, thereby obtaining $\mathcal{L}[u, \boldsymbol{v}]$ from a tree query. This can either split N into several equivalence classes, refine the initial guards or extend the set of registers in the location represented by N.

Let us now describe the respective corrective updates in Algorithm 2.

Location Closedness is satisfied if each leaf contains a short prefix in S_p. Whenever a leaf N does not contain a short prefix in S_p, one of its prefixes u is chosen for inclusion in S_p by calling *Expand*(u), which adds one-symbol extensions to U.

Transition Closedness is satisfied if for each short prefix u, action α, and initial guard in $\mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$, the extension $u\alpha(\mathtt{d}_u^g)$ is in U. If this is not satisfied, the missing $u\alpha(\mathtt{d}_u^g)$ is added to U by sifting into CT.

Register Closedness is satisfied if for each pair of prefixes u and $u\alpha(\mathtt{d})$ in U, the memorable parameters found for u contain the memorable parameters revealed by the suffixes for $u\alpha(\mathtt{d})$, except for $x_{|u|+1}$, where $|u|$ is the length of u. Register closedness guarantees that in a hypothesis $\mathcal{H}$, values of registers in the location of $u\alpha(\mathtt{d})$ can all be obtained by assignment from the registers in location u and the just received parameter. If it is not satisfied, a suffix $\boldsymbol{v}$ for $u\alpha(\mathtt{d})$ which reveals a missing register is prepended by $\alpha(p_1)$ and added to the suffixes for u, whereafter *Refine*$(\mathcal{U}(u), \boldsymbol{\alpha v})$ will reveal the missing parameter. Here, and in the following, we use $\boldsymbol{\alpha}$ to denote $\alpha(p_1)$, and $\boldsymbol{\alpha v}$ to denote the result $\alpha(p_1)\alpha_2'(p_2)\ldots\alpha_{m+1}'(p_{m+1})$ of prepending $\boldsymbol{\alpha}$ to $\boldsymbol{v} = \alpha_1'(p_1)\ldots\alpha_m'(p_m)$. If possible, we try to choose a shortest $\boldsymbol{v}$, and also restrict the parameters of $\boldsymbol{\alpha v}$ to reduce the cost of the tree query for $\mathcal{L}[u, \boldsymbol{\alpha v}]$.

Location Consistency. Analogously to consistency in the classic L^* algorithm, we split a leaf containing two short prefixes u, u', in case their corresponding extensions are not equivalent, i.e., there is a $g \in \mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$ such that $\mathcal{U}(u\alpha(\mathtt{d}_u^g)) \neq \mathcal{U}(u'\alpha(\mathtt{d}_{u'}^{\gamma(g)}))$. The splitting is done by calling *Refine*$(\mathcal{U}(u), \boldsymbol{\alpha v})$, where $\boldsymbol{v}$ is the symbolic suffix labeling the common ancestor of the leaves of $u\alpha(\mathtt{d}_u^g)$ and $u'\alpha(\mathtt{d}_{u'}^{\gamma(g)})$.

Transition Consistency is satisfied if all one-symbol extensions $u\alpha(\mathtt{d})$ that satisfy some guard g in $\mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$, are sifted to the same leaf as the extension $u\alpha(\mathtt{d}_u^g)$ with the representative data value $\mathtt{d}_u^g$. If not, the guard g should be split by calling *Refine*$(\mathcal{U}(u), \boldsymbol{\alpha v})$, where $\boldsymbol{v}$ is the symbolic suffix labeling the common ancestor of the leaves of $u\alpha(\mathtt{d}_u^g)$ and $u\alpha(\mathtt{d})$. A similar case (*Transition Consistency(b)*) occurs when $u\alpha(\mathtt{d}_u^g)$ and $u\alpha(\mathtt{d})$ are sifted to the same leaf, but are not equivalent under the identity mapping between registers. Also here, the guard g should be split by calling *Refine*$(\mathcal{U}(u), \boldsymbol{\alpha v})$, where $\boldsymbol{v}$ is a shortest suffix under which $u\alpha(\mathtt{d}_u^g) \not\simeq_{\{\boldsymbol{v}\}}^{\mathbf{id}} u\alpha(\mathtt{d})$.

Register Consistency. For some short prefix u with memorable values $mem_{\mathcal{V}(u)}(u)$, there may be symmetries in $\mathcal{L}[u, \boldsymbol{v}]$ for some $\boldsymbol{v} \in \mathcal{V}(u)$, i.e., for some permutation γ on $mem_{\mathcal{V}(u)}(u)$ we have $u \simeq_{\mathcal{V}(u)}^{\gamma} u$. It may be that this symmetry does not exist in the SUL, but we did not yet add a suffix that disproves it . Register consistency checks for the existence of such suffixes by comparing symmetries in u and its continuations $u\alpha(\mathtt{d})$. If a symmetry between data values of u does not exist in $u\alpha$ while one or more of the data values are memorable in $u\alpha$, we can construct a suffix that breaks the symmetry also for u.

Restricted Symbolic Suffixes. To reduce the number of membership queries for tree queries of form $\mathcal{L}[u, \boldsymbol{v}]$, we impose, when possible, restrictions to the parameters of $\boldsymbol{v}$, meaning that $\mathcal{L}[u, \boldsymbol{v}]$ represents acceptance/rejection of $u\boldsymbol{v}$ *only for the suffix parameters that satisfy the imposed restrictions.* An illustration was given at the end of Section 2. A more detailed description appears in the extended version [12] of this paper. Since a restricted symbolic suffix $\boldsymbol{v}'$ represents fewer actual suffixes than an unrestricted one $\boldsymbol{v}$, it has less separating power, so suffixes should only be restricted if their separating power is sufficient. The principles for adding restrictions are specific to the theory; we have implemented them for the theory $\langle \mathbb{N}, \{=\}\rangle$. There, we consider two forms of restrictions on suffix parameters p_i: (i) $\mathit{fresh}(p_i)$, meaning that p_i is different from all other preceding parameters in the prefix and suffix, (ii) $p_i = p_j$, where $j < i$, i.e., p_j is an earlier parameter in the restricted suffix. Let us consider how restricted suffixes arise when prepending an action $\boldsymbol{\alpha}$ to an existing suffix $\boldsymbol{v}$, in a call of form $\mathit{Refine}(\mathcal{U}(u), \boldsymbol{\alpha v})$, in the case that u, $\boldsymbol{\alpha}$, and $\boldsymbol{v}$ are chosen such that $\mathit{mem}_{\boldsymbol{v}}(u\alpha(\mathtt{d})))$ contains a particular memorable parameter. Let us denote the parameters of $\boldsymbol{\alpha v}$ by $p_1, \cdots, p_{|\boldsymbol{v}|+1}$. The restriction of suffix $\boldsymbol{\alpha v}$ is then obtained by

1. letting the parameter of $\boldsymbol{\alpha}$ be fresh if $\mathtt{d}$ is not equal to a previous data value in u, and
2. restricting each parameter p_i with $i > 1$ in $\boldsymbol{\alpha v}$ to be (i) fresh whenever p_{i-1} is fresh in $\boldsymbol{v}$ or the branch taken in $\mathcal{L}[u\alpha(\mathtt{d}), \boldsymbol{v}]$ for fresh p_{i-1} reveals the sought register, and (ii) equal to a previous value p_j in $\boldsymbol{\alpha v}$ if the branch taken in $\mathcal{L}[u\alpha(\mathtt{d}), \boldsymbol{v}]$ for p_{i-1} equal to the corresponding value reveals the sought register.

Hypothesis Construction. We can construct a hypothesis from a closed and consistent classification tree. Location closedness ensures that every transition has a defined source and target location, transition closedness ensures that every transition that is observed by the tree queries we have performed so far, is represented by a prefix, and register closedness ensures that registers exist for all memorable data values in corresponding locations. Location consistency, transition consistency, and register consistency ensure that we can construct a unique (up to naming of locations and registers) determinate register automaton eventhough there may exist multiple short prefixes for one location and symmetries betweeen memorable data values.

We construct the register automaton $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$, where

- L is the set of leaves of CT, and l_0 is the leaf containing the empty prefix ϵ,
- $\mathcal{X}$ maps each location $l \in L$ to $\mathit{mem}_{CT}(u)$, where u is the representative short prefix of the leaf corresponding to l, and

Algorithm 3: Analyze Counterexample.

```
Function $Analyze(w)$ is
    for $|w| \geq i > 0$ do
        for $u \in As(w_{1:i-1})$ do
            Let $u\alpha(\mathtt{d}_u^g) \in U$ represent the last transition of $w_{1:i}$ in $\mathcal{H}$
            Let $\boldsymbol{v} = Acts(w_{i+1:|w|})$ (or $\epsilon$ for $i = |w|$)
            if $u\alpha(\mathtt{d}_u^g) \not\simeq_{\{\boldsymbol{v}\}} u'$ for all $u' \in As(w_{1:i})$ then        // location
                $Expand(u\alpha(\mathtt{d}_u^g))$ and stop analysis of $w$
            if initial guard $g$ in $\mathcal{L}(u, \boldsymbol{\alpha v})$ but no $u\alpha(\mathtt{d}_u^g) \in U$ then        // transition
                $Sift(u\alpha(\mathtt{d}_u^g), root(CT))$ and stop analysis of $w$
```

- $\lambda(l) = +$ if the leaf l is in the accepting subtree of the root, else $\lambda(l) = -$.
- for every location l with short prefix u, action α, and guard g in $\mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$, there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle$, where
  - $l' = \mathcal{U}(u\alpha(\mathtt{d}_u^g))$ is the target location, and
  - π (the *assignment*) is defined by γ for which $u\alpha(\mathtt{d}_u^g) \simeq^{\gamma}_{\mathcal{V}(u\alpha(\mathtt{d}_u^g))} rp(u\alpha(\mathtt{d}_u^g))$

Analysis of Counterexamples. When an equivalence query returns a counterexample w, we process the counterexample as is shown in Algorithm 3. From right to left, we split the counterexample at every index into a location prefix $w_{1:i-1}$, a transition prefix $w_{1:i}$, and a suffix $w_{i+1:|w|}$. We use the location and transition prefixes to find corresponding short prefixes u and prefixes $u\alpha(\mathtt{d}_u^g)$ by tracing $w_{1:i-1}$ and $w_{1:i}$ on the hypothesis. We write $As(w_{1:i})$ for the short prefix corresponding to the location reached by $w_{1:i}$ in a hypothesis and $Vals(w)$ for the sequence of actions of w. We can then distinguish two cases: (1) The word $u\alpha(\mathtt{d}_u^g)$ is inequivalent to all corresponding short prefixes for the suffix of the counterexample. In this case, we make $u\alpha(\mathtt{d}_u^g)$ a short prefix. (2) The tree query $\mathcal{L}(u, \boldsymbol{\alpha v})$ shows a new initial guard. In this case, we add the corresponding (new) prefix $u\alpha(\mathtt{d}_u^g)$ to the set of prefixes. If neither case applies, we continue with the next index. Since w is a counterexample, one of the cases will apply for some index (cf. Lemma 1).

5 Correctness and Complexity

Let us now briefly discuss the correctness and query complexity of SL^λ. The correctness arguments are analogous to the arguments presented for other active learning algorithms. One notable difference to SL^* is that SL^λ establishes register consistency instead of relying on counterexamples for distinguishing symmetric registers. Proofs can be found in the extended version [12] of this paper.

Lemma 1. *A counterexample leads to a new short prefix or to a new prefix.*

This is a direct consequence of Algorithm 3. Using a standard construction that leverages properties of counterexamples (cf. [10, 40]), it can be shown that one of the two cases in the algorithm will trigger for some index of the counterexample.

As long as expanding (or sifting) new prefixes does not trigger a refinement, the current counterexample can be analyzed again, until a refinement occurs.

Lemma 1 establishes progress towards a finite RA for a language $\mathcal{L}$. Let m be the length of the longest counterexample, t the number of transitions, r the maximal number of registers at any location, and n the number of locations in the final model. (t dominates both n and r.)

Theorem 1. *SL^λ infers a RA for regular data language $\mathcal{L}$ with $O(t)$ equivalence queries and $O(t^2\, n^r + tmn\, m^m)$ membership queries for sifting words and analyzing counterexamples.*

$O(t)$ is an improvement over the worst case estimate of $O(tr)$ equivalence queries for SL^* [10]. SL^λ also improves the worst case estimate for membership queries for sifting to $O(t^2\, n^r)$ from $O(t^2 r\, n^r)$ for filling the table in SL^*. For analyzing counterexamples, SL^λ replaces $O(trm\, m^m)$ with $O(tmn\, m^m)$.

6 Evaluation

As mentioned, we have implemented the SL^λ algorithm in the publicly available RALib tool for learning register automata. RALib already implemented the SL^* algorithm [10] that uses an observation table as its data structure. In order to evaluate the effect of analyzing counterexamples as described in Section 4, we have also implemented the SL^{CT} classification tree learning algorithm that uses the same counterexample analysis technique as the SL^* algorithm, i.e., adding suffixes from counterexamples to the classification tree directly. We compare the performance of the SL^λ algorithm against that of SL^* and SL^{CT}. All models, the experimental setup, and infrastructure for executing the experiments are available on the paper's artefact at Zenodo [42] and updated versions on GitHub[5].

Experimental Setup. We use two series of experiments: (1) A black-box learning setup with random walks for finding counterexamples on small models from the Automata Wiki [39] to establish a baseline comparison with other results and to evaluate the impact of using non-minimal counterexamples. In these experiments, we verify with a model checker that the inferred model is equivalent to the SUL and we stop as soon as the correct model is produced by a learning algorithm. (2) A white-box setup with a model checker for finding short counterexamples to analyze the scalability of algorithms on (2a) 24 consecutive hypotheses of the Mbed TLS 2.26.0 server,[6] as well as (2b) sets of randomly generated automata.[7] All results were obtained on a MacBook Pro with an Apple M1 Pro CPU and 32 GB of memory, running macOS version 12.5.1 and OpenJDK version 17.0.8.1.

[5] https://github.com/LearnLib/ralib-benchmarking

[6] We obtained these hypotheses by extending the machinery of DTLS-Fuzzer [17], a publicly available tool for learning state machine models of DTLS implementations.

[7] We used the algorithm of Champarnaud and Paranthoën [11] to enumerate semantically distinct DFAs with a specific alphabet and number of locations. We then replaced the alphabet symbols with RA actions of arity one, and finally replaced a fraction of the transitions with simple gadgets that store and compare data values.

Table 1: Results on AutomataWiki Systems.

SUL					**Resets (Learn)**			**Resets (Total)**			**CounterExs**			**WCT Learn [ms]**			**WCT Test [ms]**		
	$\|Q\|$	$\|\Gamma\|$	$\|X\|$	$\|C\|$	SL^*	SL^λ	SL^{CT}	SL^*	SL^λ	SL^{CT}	SL^*	SL^λ	SL^{CT}	SL^*	SL^λ	SL^{CT}	SL^*	SL^λ	SL^{CT}
channel-frame	5	8	3	2	**11**	**11**	15	**24**	28	32	**1**	2	2	49	43	**36**	295	**289**	294
abp-receiver3	6	10	3	2	489	**88**	466	614	**249**	610	4	4	4	147	**74**	245	**256**	282	264
palindrome	6	15	4	0	479	**358**	476	508	**384**	504	5	5	5	73	52	**51**	406	403	**402**
login	12	19	4	0	436	**244**	433	509	**300**	512	3	**2**	3	86	**54**	67	**301**	303	310
abp-output	30	50	1	2	363	**208**	311	**590**	4 552	6 151	**5**	11	11	**142**	151	696	260	175	**154**
sip	30	72	2	0	487	**233**	345	**934**	3 633	2 772	**9**	15	16	370	**347**	353	194	**149**	160
fifo3	12	16	4	0	29	24	**23**	212	**202**	209	5	5	5	114	**106**	108	**547**	636	563
fifo5	18	24	6	0	66	**55**	60	435	**434**	468	**6**	7	7	1 303	**1 144**	1 451	**575**	600	584
fifo7	24	32	8	0	118	**96**	123	**738**	839	989	**7**	8	9	317 435	**279 888**	346 897	589	591	**583**

Results. Table 1 summarizes the results of the experiments in a black-box learning setup. For every SUL, we report its complexity (in number of locations $|Q|$, transitions $|\Gamma|$, registers $|X|$, and constants $|C|$) and, for each learning algorithm, the number of resets (i.e., tests) during the learning phase, total tests (incl. counterexample search), the number of counterexamples found, and wall clock times (WCT) for learning and testing. In Table 1, all numbers are averages from 20 experiments. It can be seen that the SL^λ algorithm consistently outperforms the other two algorithms w.r.t. the number of tests during learning. As can be expected, the SL^* algorithm requires the fewest counterexamples. Execution times do not show a consistent pattern for these small systems or a clear 'winner' between these three RA learning algorithms, but there is a strong correlation between the number of learner tests and the time that learning requires. Due to this, in most cases, SL^λ is fastest overall.

The SULs of the previous set of experiments were all quite small ($|\Gamma| \leq 72$), and did not show any scalability differences between the three algorithms. Also, with the exception of fifo, the benchmarks were not parametric. In the following experiments, we scale the SULs which are learned.

Figure 5 shows the results of our experiments with DTLS models. For each algorithm, the graphs show the relationship between the number of transitions in each hypothesis model and the number of resets with restricted and unrestricted suffixes (in the first two graphs), the number of counterexamples (3^{rd} graph), and execution times (4^{th} graph). It is evident that, with increasing model complexity, the number of counterexamples grows linearly for all algorithms at roughly the same rate, yet the number of resets grows much more rapidly for SL^* than it does for SL^{CT} and SL^λ. In terms of time performance, the trend is even more pronounced. For SULs with more that 100 transitions, learning times grow significantly worse for SL^* than the other two algorithms, and SL^λ clearly also beats SL^{CT} on even bigger systems.

Finally, Fig. 6 shows the results of the experiments with randomly generated automata. The graphs show how the number of resets scales with the number of locations and actions (left) and the number of registers when using both restricted (center) and unrestricted suffixes (right). The number of resets grows much more rapidly for SL^* than for the other algorithms. Not restricting suffixes leads to a 2–4x increase in resets; notice the different scales on the y-axis.

Overall, the experiments show a clear advantage of SL^λ over table-based RA learning algorithms in terms of the number of resets and execution times for

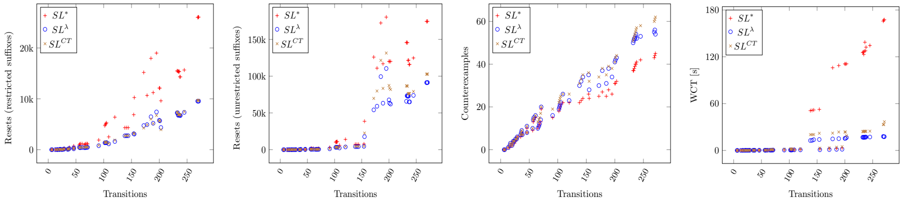


Fig. 5: Number of resets (two leftmost graphs), counterexamples (3^{rd} graph), and wall clock times (4^{th} graph) for inferring models of the Mbed TLS 2.26.0 server.

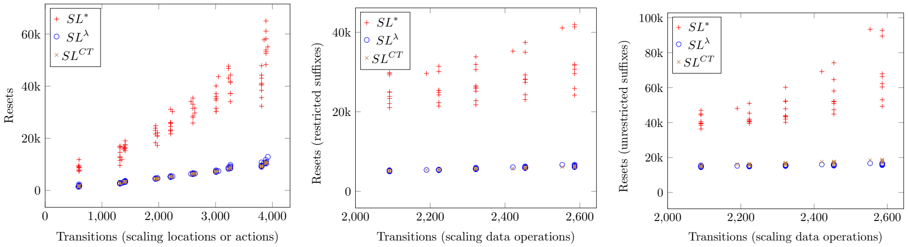


Fig. 6: Resets for inferring models of generated SULs, scaling the number of transitions through locations and actions as well as by increasing the percentage of transitions with data operations using restricted and unrestricted suffixes.

bigger systems. These results confirm the theoretical properties of the algorithms and are consistent with the behavior of AAL algorithms for FSMs.

7 Conclusion

We have presented SL^λ, a scalable tree-based algorithm for register automata learning. SL^λ reduces the membership queries needed for inferring RA models by constructing short restricted suffixes incrementally. This enables active learning in scenarios not feasible with previous algorithms. We prove a reduction in the worst-case number of tests and, via a practical evaluation, show performance improvements on both real-world (i.e., on a complex network protocol) and synthetic models compared to the state-of-the-art RA learning algorithm.

Acknowledgements This research was partially funded by the Swedish Research Council (Vetenskapsrådet), the Swedish Foundation for Strategic Research through project aSSIsT, the Knut and Alice Wallenberg Foundation through project UPDATE, and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) projects 495857894 (STING) and 442146713 (NFDI4Ing). We thank these funding agencies and the TACAS 2024 reviewers for their comments.

References

1. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods in System Design pp. 1–41 (2015). https://doi.org/10.1007/s10703-014-0216-x
2. Aarts, F., Fiterau-Brostean, P., Kuppens, H., Vaandrager, F.: Learning register automata with fresh value generation. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015. LNCS, vol. 9399, pp. 165–183. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-25150-9_11
3. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods. LNCS, vol. 7436, pp. 10–27. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_4
4. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods in System Design **46**(1), 1–41 (Feb 2015). https://doi.org/10.1007/s10703-014-0216-x
5. Aarts, F., Kuppens, H., Tretmans, J., Vaandrager, F.W., Verwer, S.: Learning and testing the bounded retransmission protocol. In: Proceedings of the Eleventh International Conference on Grammatical Inference, ICGI 2012. JMLR Proceedings, vol. 21, pp. 4–18. JMLR.org (2012), http://proceedings.mlr.press/v21/aarts12a.html
6. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proc. 29^{th} ACM Symp. on Principles of Programming Languages. pp. 4–16. ACM (2002). https://doi.org/10.1145/503272.503275
7. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
8. Bollig, B., Habermehl, P., Leucker, M., Monmege, B.: A fresh approach to learning register automata. In: Developments in Language Theory. LNCS, vol. 7907, pp. 118–130. Springer Verlag (2013). https://doi.org/10.1007/978-3-642-38771-5_12
9. Cassel, S., Howar, F., Jonsson, B.: RALib: a LearnLib extension for inferring EFSMs. In: Proceedings of the 4th International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS). pp. 1–8 (2015), https://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf
10. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. **28**(2), 233–263 (2016). https://doi.org/10.1007/s00165-016-0355-5
11. Champarnaud, J.M., Paranthoën, T.: Random generation of DFAs. Theoretical Computer Science **330**(2), 221–235 (Feb 2005). https://doi.org/10.1016/j.tcs.2004.03.072
12. Dierl, S., Fiterau-Brostean, P., Howar, F., Jonsson, B., Sagonas, K., Tåquist, F.: Scalable tree-based register automata learning. arXiv CoRR (Jan 2024). https://doi.org/10.48550/arXiv.2401.14324, Extended version of the TACAS 2024 paper.
13. Drews, S., D'Antoni, L.: Learning symbolic automata. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 10205, pp. 173–189. Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_10
14. Esparza, J., Leucker, M., Schlund, M.: Learning workflow Petri nets. Fundamenta Informaticae **113**(3-4), 205–228 (2011). https://doi.org/10.3233/FI-2011-607

15. Ferreira, T., Brewton, H., D'Antoni, L., Silva, A.: Prognosis: Closed-box analysis of network protocol implementations. In: ACM SIGCOMM 2021 Conference. pp. 762–774. ACM (Aug 2021). https://doi.org/10.1145/3452296.3472938
16. Fiterau-Brostean, P., Howar, F.: Learning-based testing the sliding window behavior of TCP implementations. In: Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS. LNCS, vol. 10471, pp. 185–200. Springer (2017). https://doi.org/10.1007/978-3-319-67113-0_12
17. Fiterau-Brostean, P., Jonsson, B., Sagonas, K., Tåquist, F.: DTLS-Fuzzer: A DTLS protocol state fuzzer. In: 15th IEEE Conference on Software Testing, Verification and Validation. pp. 456–458. ICST 2022, IEEE (Apr 2022). https://doi.org/10.1109/ICST53961.2022.00051
18. Fiterau-Brostean, P., Jonsson, B., Sagonas, K., Tåquist, F.: Automata-based automated detection of state machine bugs in protocol implementations. In: Network and Distributed System Security Symposium. NDSS 2023, The Internet Society (Feb 2023), https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s68_paper.pdf
19. Fiterău-Broştean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS implementations using protocol state fuzzing. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2523–2540. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean
20. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. LNCS, vol. 9780, pp. 454–471. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_25
21. Fiterău-Broştean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F., Verleg, P.: Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 142–151. ACM, New York, NY, USA (Jul 2017). https://doi.org/10.1145/3092282.3092289
22. Frohme, M., Steffen, B.: Compositional learning of mutually recursive procedural systems. International Journal on Software Tools for Technology Transfer **23**(4), 521–543 (Aug 2021). https://doi.org/10.1007/s10009-021-00634-y
23. Frohme, M., Steffen, B.: Never-stop context-free learning. In: Olderog, E.R., Steffen, B., Yi, W. (eds.) Model Checking, Synthesis, and Learning, LNCS, vol. 13030, pp. 164–185. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-91384-7_9
24. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. LNCS, vol. 8044, pp. 813–829. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_57
25. Groz, R., Irfan, M.N., Oriat, C.: Algorithmic improvements on regular inference of software models and perspectives for security testing. In: Proc. ISoLA 2012, Part I. LNCS, vol. 7609, pp. 444–457. Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_41
26. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.D., Weber, H. (eds.) Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer Verlag (Apr 2002). https://doi.org/10.1007/3-540-45923-5_6

27. de la Higuera, C.: A bibliographical study of grammatical inference. Pattern Recognition **38**(9), 1332–1348 (Sep 2005). https://doi.org/10.1016/j.patcog.2005.01.003
28. Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits, LNCS, vol. 11026, pp. 123–148. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_5
29. Howar, F., Steffen, B.: Active automata learning as black-box search and lazy partition refinement. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday. LNCS, vol. 13560, pp. 321–338. Springer (2022). https://doi.org/10.1007/978-3-031-15629-8_17
30. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 7148, pp. 251–266. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_17
31. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Computer Aided Verification, 15th International Conference. LNCS, vol. 2725, pp. 315–327 (Jul 2003). https://doi.org/10.1007/978-3-540-45069-6_31
32. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: A redundancy-free approach to active automata learning. In: Runtime Verification: 5th International Conference, RV 2014, Proceedings. LNCS, vol. 8734, pp. 307–322. Springer (Sep 2014). https://doi.org/10.1007/978-3-319-11164-3_26
33. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press (1994)
34. Linard, A., de la Higuera, C., Vaandrager, F.: Learning unions of k-testable languages. In: Martín-Vide, C., Okhotin, A., Shapira, D. (eds.) Language and Automata Theory and Applications. LNCS, vol. 11417, pp. 328–339. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-13435-8_24
35. Maler, O., Mens, I.E.: Learning regular languages over large alphabets. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference,. LNCS, vol. 8413, pp. 485–499. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_41
36. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: Proceedings of the Ninth IEEE International High-Level Design Validation and Test Workshop. pp. 95–100. IEEE, New York, NY, USA (Nov 2004). https://doi.org/10.1109/HLDVT.2004.1431246
37. Merten, M., Howar, F., Steffen, B., Cassel, S., Jonsson, B.: Demonstrating learning of register automata. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 7214, pp. 466–471. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_32
38. Moerman, J., Sammartino, M., Silva, A., Klin, B., Szynwelski, M.: Learning nominal automata. In: Proc. 44^{th} ACM Symp. on Principles of Programming Languages. pp. 613–625. POPL '17, ACM, New York, NY, USA (Jan 2017). https://doi.org/10.1145/3093333.3009879
39. Neider, D., Smetsers, R., Vaandrager, F.W., Kuppens, H.: Benchmarks for automata learning and conformance testing. In: Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday. LNCS, vol. 11200, pp. 390–416. Springer (2018). https://doi.org/10.1007/978-3-030-22348-9_23

40. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Information and Computation **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021
41. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 193–206. USENIX Association (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter
42. Sagonas, K., Jonsson, B., Howar, F., Dierl, S., Fiterau-Brostean, P., Tåquist, F.: Reproduction artifact for TACAS 2024 paper "Scalable tree-based register automata learning" (Dec 2023). https://doi.org/10.5281/zenodo.10442556
43. Schuts, M., Hooman, J., Vaandrager, F.: Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In: Ábrahám, E., Huisman, M. (eds.) Integrated Formal Methods. LNCS, vol. 9681, pp. 311–325. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_20
44. Shahbaz, M., Groz, R.: Analysis and testing of black-box component-based systems by inferring partial models. Software Testing, Verification and Reliability **24**(4), 253–288 (2014). https://doi.org/10.1002/stvr.1491
45. Shu, G., Lee, D.: Testing security properties of protocol implementations - a machine learning based approach. In: 27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007). IEEE Computer Society (2007). https://doi.org/10.1109/ICDCS.2007.147
46. Sun, J., Xiao, H., Liu, Y., Lin, S.W., Qin, S.: TLV: abstraction through testing, learning, and validation. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 698–709. ACM, New York, NY, USA (Aug 2015). https://doi.org/10.1145/2786805.2786817
47. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: IEEE International Conference on Software Testing, Verification and Validation. pp. 276–287. IEEE Computer Society (Mar 2017). https://doi.org/10.1109/ICST.2017.32
48. Vaandrager, F., Bloem, R., Ebrahimi, M.: Learning Mealy machines with one timer. In: Leporati, A., Martín-Vide, C., Shapira, D., Zandron, C. (eds.) Language and Automata Theory and Applications. LNCS, vol. 12638, pp. 157–170. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-68195-1_13
49. Vaandrager, F., Garhewal, B., Rot, J., Wißmann, T.: A new approach for active automata learning based on apartness. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 13243, pp. 223–243. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_12
50. Vaandrager, F.W.: Model learning. Commun. ACM **60**(2), 86–95 (Jan 2017). https://doi.org/10.1145/2967606
51. Volpato, M., Tretmans, J.: Active learning of nondeterministic systems from an ioco perspective. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. LNCS, vol. 8802, pp. 220–235. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_16
52. Walkinshaw, N., Bogdanov, K., Derrick, J., París, J.: Increasing functional coverage by inductive testing: A case study. In: Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010. LNCS, vol. 6435, pp. 126–141. Springer (2010). https://doi.org/10.1007/978-3-642-16573-3_10

53. Yonesaki, N., Katayama, T.: Functional specification of synchronized processes based on modal logic. In: Proceedings of the 6th Int. Conference on Software Engineering. pp. 208–217. IEEE Computer Society Press (1982). https://doi.org/10.5555/800254.807763

Small Test Suites for Active Automata Learning*

Loes Kruger(✉), Sebastian Junges, and Jurriaan Rot

Institute for Computing and Information Sciences,
Radboud University, Nijmegen, the Netherlands
{loes.kruger,sebastian.junges,jurriaan.rot}@ru.nl

Abstract. A bottleneck in modern active automata learning is to test whether a hypothesized Mealy machine correctly describes the system under learning. The search space for possible counterexamples is given by so-called test suites, consisting of input sequences that have to be checked to decide whether a counterexample exists. This paper shows that significantly smaller test suites suffice under reasonable assumptions on the structure of the black box. These smaller test suites help to refute false hypotheses during active automata learning, even when the assumptions do not hold. We combine multiple test suites using a multi-armed bandit setup that adaptively selects a test suite. An extensive empirical evaluation shows the efficacy of our approach. For small to medium-sized models, the performance gain is limited. However, the approach allows learning models from large, industrial case studies that were beyond the reach of known methods.

1 Introduction

System identification algorithms aim to capture the behavior of a black-box system, often called the *system under learning* (SUL), in a formal model. Among the system identification approaches, *active automata learning* (AAL) [5,23,24] is a popular methodology to extract finite automata from a black-box. AAL has been successfully applied to learn security-critical protocol implementations [15,16,18], legacy code [8,44], smart cards [13], interfaces of data structures [22], embedded control software [46], and (explainable) neural network policies [51].

Modern AAL methods [25,48] are available via mature tool sets [11,26,36] that implement these methods. They are primarily built around Angluin's Minimal Adequate Teacher (MAT) framework [5]. In essence, the theoretically elegant MAT framework requires access to two types of queries. First, an *output query* (OQ) allows to execute a sequence of inputs on the black-box and observe its outputs. Second, an *equivalence query* (EQ) asks whether a hypothesized Mealy machine is indeed equivalent to the SUL. Implementing the equivalence query provides practitioners with an impossibility [35]: *How do we decide whether a learned model is equivalent to the behavior of a black-box?* To overcome this impossibility, practitioners take a more modest stance and only *approximate* equivalence queries. One approach is to randomly sample from all possible input

* This research is partially supported by the NWO grant No. VI.Vidi.223.096.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 109–129, 2024.
https://doi.org/10.1007/978-3-031-57249-4_6

sequences, which leads to a statistical guarantee[1], as pioneered in the context of learning by Angluin [6]. Alternatively, based on ideas pioneered in [14,50], the structure of the hypothesis is used to select a finite set of input sequences to be checked. These sets are *test suites* and the approach is called conformance testing [12]. This paper considers EQs via test suites; for an overview, see Sec. 2.

Challenge: Finding small test suites. Finite test suites can be obtained using the notion of *k-completeness*. In short, k-completeness guarantees equivalence under the assumption that the number of states in the SUL is at most k states larger than the hypothesis. Popular k-complete test are the classical W-method [14,50] and variations thereon, such as Wp [19], HSI [32,41] and Hybrid-ADS [34]; see empirical evaluations in [7]. We call these methods *Access-Step-Identify* (ASI). These are standard in tools like LearnLib [26] and AALpy [36]. However, k-complete test suites such as the W-method grow with $|I|^k$, where $|I|$ is the number of input symbols. Consequently, even for small k, these test suites are prohibitively large.

Our approach for smaller test suites. Towards smaller test suites, we adapt ASI methods and make assumptions on the *shape of the SUL* in relation to the shape of the hypothesis. In particular, we consider several natural assumptions that may occur in real-world systems. For instance, one of these assumptions is that in most states, most inputs either lead to an error-state or are simply discarded. Other assumptions are that certain inputs are used only in the beginning (e.g. in the authentication phase of a protocol), or that the SUL has a component structure where inputs are primarily used together within components. We formalize these assumptions, demonstrate the applicability on industrial benchmarks, and develop a notion of completeness under these assumptions. The resulting test suites are much smaller, as the factor $|I|^k$ is restricted to $|I'|^k$, with $|I'| < |I|$.

Challenge: finding counterexamples as soon as possible. The time to find a counterexample during EQs is the bottleneck in AAL applications [47,52]. To accelerate this process, it is helpful to constrain the search space of possible counterexamples, allowing for a targeted search. Here complete test suites are again helpful, even if they can not be fully executed and can only be approximated through sampling, as implemented for instance in the randomised W-method of LearnLib. Complete test suites then provide a constrained search space that still contains all actual counterexamples. Another relevant aspect for finding counterexamples fast is the order that tests are chosen: an adequate ordering in which counterexamples (empirically) occur early in the test suites is preferred.

Our approach for finding counterexamples faster. In the context of randomised W-methods, pruning input sequences that are not counterexamples yields a larger probability of sampling a counterexample and thus speeds up the procedure. However, for the smaller test suites described earlier, without domain-specific knowledge, we can not be certain that they contain (a larger fraction of) counterexamples, as we do not know whether the underlying assumptions are met. Instead, our idea is to *combine* multiple test suites. We prefer tests from test

[1] Typically, a probably-approximately correct (PAC) guarantee.

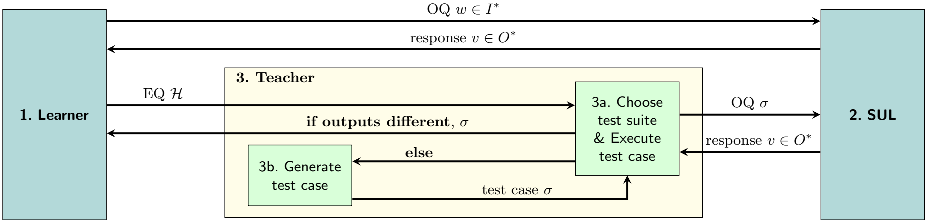


Fig. 1: Interaction between the learner, teacher and SUL in the MAT framework.

suites that led to counterexamples in previous invocations of the EQ during the learning process. We operationalize this idea using multi-armed bandits.

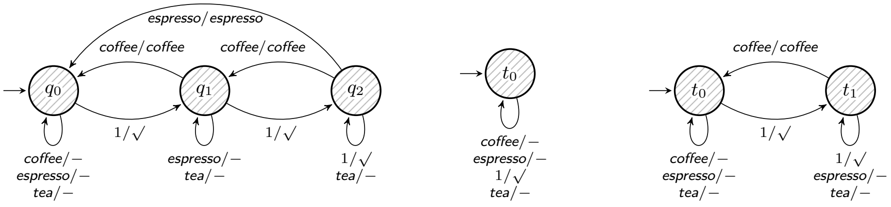
Contributions. In summary, this paper introduces three new test suites that are complete under additional assumptions on the SUL (Sec. 4). We combine these test suites via a multi-armed bandit framework to accelerate finding counterexamples in EQs (Sec. 5). The paper demonstrates performance on scalable self-generated benchmarks, standard benchmarks and industry benchmarks (Sec. 6). The proofs of all theorems, the complete benchmarks results and additional figures can be found in the appendix of the extended version of this paper [30].

2 Overview

We briefly illustrate the interactions in the MAT framework, the W-method, and our approach for generating smaller test suites, using a toy example. Recall that in the MAT framework the learner can pose output queries (OQ) and equivalence queries (EQ). This is depicted in Fig. 1, where EQs are implemented by the teacher. The Mealy machine in Fig. 2a depicts the SUL for a coffee machine with input alphabet $I = \{\textit{coffee}, \textit{espresso}, \textit{tea}, 1\}$. Coffee costs 1 euro, espresso costs 2 euros, and tea never gets dispensed. Via a series of queries, we may obtain the hypothesis in Fig. 2b. The hypothesis is easy to refute with an EQ, e.g., via the counterexample $1 \cdot \textit{coffee}$. After various OQs, we learn the hypothesis in Fig. 2c. A short counterexample that distinguishes the hypothesis $\mathcal{H}_1$ from the SUL $\mathcal{S}$ is

$$\underbrace{1}_{\text{access}} \cdot \underbrace{1 \cdot \textit{coffee}}_{\text{infix}} \cdot \underbrace{\textit{coffee}}_{\text{distinguish}} .$$

The counterexample consists of three parts. We first *access* q_1 and t_1, from which we run an infix that leads to either q_1 or t_0, and then we distinguish both states with *coffee*. Executing input *coffee* from q_1 returns output *coffee* while executing input *coffee* from t_0 returns output $-$. The W-method generates test suites that consist of input words of a similar shape. Concretely, test suites are constructed as $P \cdot I^{\leq k+1} \cdot W$, where P ensures access to the states in the hypothesis, $I^{\leq k+1}$ is the set of sequences of at most $k+1$ arbitrary input symbols, used to step to states in the (larger) SUL, and W contains sequences that help to distinguish states. Test suites constructed in this way tend to contain many input sequences



(a) SUL $\mathcal{S}$ (b) Hyp. $\mathcal{H}_0$ (c) Hyp. $\mathcal{H}_1$

Fig. 2: A coffee machine and two hypotheses which can be generated using AAL.

which do not help to refute the hypothesis. In our example, the W-method test suite with $k = 2$ for $\mathcal{H}_1$ also contains uninformative sequences such as

$$\underbrace{\epsilon}_{\text{access}} \cdot \underbrace{1 \cdot \mathit{espresso} \cdot 1 \cdot}_{\text{infix}} \underbrace{\mathit{coffee}}_{\text{distinguish}} \qquad \text{and} \qquad \underbrace{1}_{\text{access}} \underbrace{\cdot\, \mathit{tea} \cdot \mathit{espresso} \cdot}_{\text{infix}} \underbrace{\mathit{espresso}}_{\text{distinguish}} .$$

A smaller test suite. In hypothesis $\mathcal{H}_1$, *espresso* and *tea* self-loop in all states. The counterexample to refute this hypothesis only requires the inputs *coffee* and 1. It is natural that input *tea* is not necessary to reach new states, as this option is obsolete. This leads us to a test suite for $\mathcal{H}_1$ that excludes the inputs *tea* and *espresso* in the infix. If we generate infixes of length at most 3 ($k = 2$) with the full alphabet, the test suite contains 112 test cases. If we exclude two inputs, only 12 test cases remain.

A set of smaller test suites. The restricted test suites that aim to exclude obsolete inputs can be refined. These restrictions can be adapted for other typical scenarios. Consider, e.g., network protocols that only perform a three-way handshake in the initial phase. In states where the communication protocol is initialized, these inputs are no longer relevant. Likewise, there are often clusters where the same input symbols are relevant. For instance, if a 10 cent coin is a relevant input in some state of a vending machine, then a 50 cent coin is likely also relevant.

Mixing test suites. Restricting the test suites yields the risk of missing counterexamples. While the test suite may be complete under (natural) additional assumptions, in a black-box setting we have no way to check whether these assumptions hold. We therefore present a methodology where various restricted test suites are combined, using multi-armed bandits to select test suites. During learning, the EQs then increasingly use test suites for which the assumptions hold, without the need for advanced knowledge of the SUL.

3 Complete Test Suites

We recall complete test suites and start with preliminaries on Mealy machines.

Definition 3.1. *A* Mealy machine *is a tuple* $\mathcal{M} = (Q, I, O, q_0, \delta, \lambda)$ *with finite sets* Q*,* I *and* O *of* states*,* inputs *and* outputs *respectively; the* initial state $q_0 \in Q$*, the* transition function $\delta\colon Q \times I \to Q$ *and the* output function $\lambda\colon Q \times I \to O$*.*

Below, we also use *partial* Mealy machines; these are defined as above but with $\delta\colon Q \times I \rightharpoonup Q$ and $\lambda\colon Q \times I \rightharpoonup O$ partial functions with the same domain. For a partial function f we write $f(x){\downarrow}$ if $f(x)$ is defined and $f(x){\uparrow}$ otherwise. The transition and output functions are extended to input words of length $n \in \mathbb{N}$ in the standard way, as functions $\delta\colon Q \times I^n \rightharpoonup Q$ and $\lambda\colon Q \times I^n \rightharpoonup O^n$. We abbreviate $\delta(q_0, w)$ by $\delta(w)$. Given $Q' \subseteq Q$ and $L \subseteq I^*$, we write $\Delta^{\mathcal{M}}(Q', L) = \{\delta(q, w) \mid q \in Q', w \in L\}$ for the set of states reached from Q' via words in L, and we let $\Delta^{\mathcal{M}}(L) = \Delta^{\mathcal{M}}(\{q_0\}, L)$. In particular $\Delta^{\mathcal{M}}(I^*)$ is the set of reachable states of $\mathcal{M}$. We use the superscript $\mathcal{M}$ to indicate to which Mealy machine we refer, e.g. $Q^{\mathcal{M}}$ and $\delta^{\mathcal{M}}$. We write $|\mathcal{M}|$ for the number of states in $\mathcal{M}$. A state $q \in Q^{\mathcal{M}}$ is a *sink* if for all $i \in I$, $\delta(q, i) = q$. We denote the set of sinks by Q_{sink}.

Definition 3.2. *Given a language* $L \subseteq I^*$ *and Mealy machines* $\mathcal{H}$ *and* $\mathcal{S}$*, states* $p \in Q^{\mathcal{H}}$ *and* $q \in Q^{\mathcal{S}}$ *are* L-equivalent*, written as* $p \sim_L q$*, if* $\lambda^{\mathcal{H}}(p, w) = \lambda^{\mathcal{S}}(q, w)$ *for all* $w \in L$*. States* p, q *are* equivalent*, written* $p \sim q$*, if they are* I^**-equivalent. The Mealy machines* $\mathcal{H}$ *and* $\mathcal{S}$ *are equivalent, written* $\mathcal{H} \sim \mathcal{S}$*, if* $q_0^{\mathcal{H}} \sim q_0^{\mathcal{S}}$*.*

Conformance testing techniques construct from a current hypothesis $\mathcal{H}$ a suitable test suite $T \subseteq I^*$, to be executed on the (black-box) SUL $\mathcal{S}$. If a test case fails, we know the machines are inequivalent. Ideally, we want a test suite that contains a failing test case for every possible inequivalent Mealy machine. This is called a *complete* test suite. We define completeness in a more generic way than usual to make it easier to add conditions to the set of Mealy machines for which the test suite is complete in subsequent sections.

Definition 3.3. *Given a Mealy machine* $\mathcal{H}$ *and set of Mealy machines* $\mathcal{C}$*, a test suite* $T \subseteq I^*$ *is* complete *for* $\mathcal{H}$ *w.r.t.* $\mathcal{C}$ *if for all* $\mathcal{S} \in \mathcal{C}$*,* $\mathcal{H} \sim_T \mathcal{S}$ *implies* $\mathcal{H} \sim \mathcal{S}$*.*

In general, there are no test suites that are complete w.r.t. the (infinite) set $\mathcal{C}$ containing all (inequivalent) Mealy machines [35]. In practice, we often use k-completeness, where we assume that $\mathcal{C}$ only contains Mealy machines which have at most k states more than the hypothesis.

Definition 3.4. *Let* $\mathcal{H}$ *be a Mealy machines. A test suite* $T \subseteq I^*$ *is* k-complete *for* $\mathcal{H}$ *if it is complete w.r.t.* $\mathcal{C}^k_{\mathcal{H}} = \{\mathcal{S} \mid |\mathcal{S}| - |\mathcal{H}| \leq k\}$*.*

Conformance testing techniques often build k-complete test suites in a structured manner using a *state cover* and a *characterization set*. We give a formal description of a classical k-completeness technique: the W-method [14, 50].

Definition 3.5. *An* access sequence *for* $q \in Q^{\mathcal{H}}$ *is a word* $w \in I^*$ *such that* $\delta^{\mathcal{H}}(w) = q$*. A language* $P \subseteq I^*$ *is a* state cover *if* $\varepsilon \in P$ *and* P *contains an access sequence for every reachable state, i.e.,* $\Delta^{\mathcal{H}}(P) = \Delta^{\mathcal{H}}(I^*)$*.*

Definition 3.6. *A* characterization set *for a Mealy machine* $\mathcal{H}$ *is a non-empty language* $W \subseteq I^*$ *such that* $p \sim_W q$ *implies* $p \sim q$ *for all* $p, q \in Q^{\mathcal{H}}$*.*

Let P be a minimal state cover and W a characterization set for $\mathcal{H}$. Then the W-method, given $k \in \mathbb{N}$, is given by the test suite $T = P \cdot I^{\leq k+1} \cdot W$. The state cover P makes sure all states in $\mathcal{H}$ are reached. The role of the set of infixes in $I^{\leq k+1}$ is to reach states in $\mathcal{S}$. The characterization set W checks if the states reached in $\mathcal{H}$ and $\mathcal{S}$ after reading a word from $P \cdot I^{\leq k+1}$ match. Other ASI methods differ in the computation of the characterization set and the structure of the test suite but are constructed from the same sets P, I, and W.

In the remainder of this section, we prove that the W-method is k-complete [14, 50]. We recall the proof strategy from [34], based on reachability and bisimulations up-to $\sim_L$, in Appendix A of [30]. With minimal changes, this proof also works for other ASI methods. Here, we summarize the approach in two main steps, which we reuse in Sec. 4 to prove completeness for different test suites under additional conditions. The first step concerns reachability in $\mathcal{S}$. We assume that $\mathcal{H}$ is minimal w.r.t. number of states, which is an invariant of active learning algorithms, our intended application. This assumption is only used for k to be correct; alternatively, one can bound the number of states of $\mathcal{S}$ to the sum of k with the number of *inequivalent* states in $\mathcal{H}$.

Lemma 3.7. *Let $\mathcal{H}$ and $\mathcal{S}$ be Mealy machines with $|\mathcal{S}|-|\mathcal{H}| \leq k$ for some integer k, and assume $\mathcal{H}$ is minimal. Moreover, let P be a state cover for $\mathcal{H}$ and W a characterization set for $\mathcal{H}$. Finally, let $L = P \cdot I^{\leq k}$ and $T = P \cdot W$ and suppose that $\mathcal{H} \sim_T \mathcal{S}$. Then L is a state cover for $\mathcal{S}$.*

It is in the above lemma that the assumption $\varepsilon \in P$ is used, to ensure that all states in $\mathcal{S}$ are reached from a state in $\Delta^{\mathcal{S}}(P)$. The second step extends this to actual equivalence of the two Mealy machines.

Lemma 3.8. *Suppose $L \subseteq I^*$ is a state cover for both $\mathcal{H}$ and $\mathcal{S}$. Let W be a characterization set for $\mathcal{H}$, and $T = L \cdot I^{\leq 1} \cdot W$. If $\mathcal{H} \sim_T \mathcal{S}$, then $\mathcal{H} \sim \mathcal{S}$.*

Combining the above two lemmas, we recover k-completeness of the W-method.

Corollary 3.9. *The W-method is k-complete.*

4 Complete Test Suites with Subalphabets

We introduce test suites that are similar to the W-method but have fewer infixes. These test suites are roughly of the form $T = P \cdot I_{sub}^{\leq k+1} \cdot W$, with different choices for $I_{sub} \subseteq I$. If the subalphabet gets smaller, the test suite size always decreases. If we choose I for I_{sub}, we recover the original W-method test suite.

In the following subsections, we provide three new functions, called *experts*, for generating subalphabets. These experts are tailored to perform well for certain Mealy machine structures. For each expert, we provide a parameterized family of Mealy machines for which the expert should work well, and we show they are complete under specific assumptions that strengthen those of k-completeness.

The experts can be embedded in any ASI method. For conciseness, we focus on the W-method. In the definition of expert, the output is a set of subalphabets rather than a single one I_{sub} as described above; this is used in one of the experts.

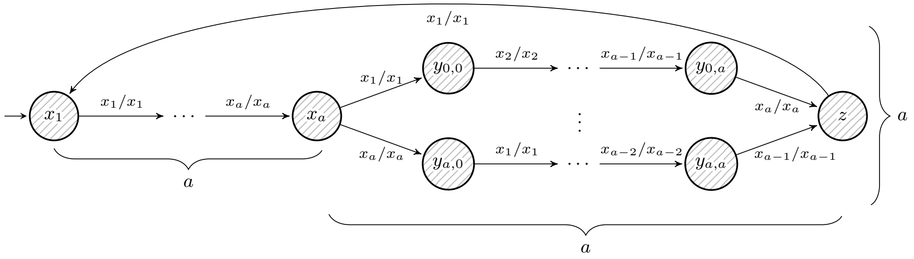


Fig. 3: $\mathsf{ASML}_{a,b}$ models over inputs and outputs $\{x_i \mid 1 \leq i \leq a\} \cup \{y_i \mid 1 \leq i \leq b\}$. Transitions not drawn, including all transitions y_i with $1 \leq i \leq b$, lead to a sink with a unique output.

Definition 4.1. *An* expert *is a function E which takes as arguments a Mealy machine $\mathcal{H}$ and a word $v \in I^*$, and returns a set of subalphabets $I_1, \ldots, I_n$.*

The embedding in the W-method is as follows.

Definition 4.2. *The expert test suite* ETS *for $\mathcal{H}$, expert E and $k \in \mathbb{N}$ is:*

$$\mathsf{ETS}_{E,k}(\mathcal{H}) = \bigcup_{v \in P} (v \cdot (\bigcup_{I_{sub} \in E(\mathcal{H},v)} I_{sub}^{\leq k-1}) \cdot I^{\leq 2} \cdot W)$$

where P is a minimal state cover for $\mathcal{H}$ and W a characterization set.

Before introducing the new experts we define the trivial expert.

Definition 4.3. *The* trivial expert E_{T} *is given by $E_{\mathsf{T}}(\mathcal{H}, q) = \{I^{\mathcal{H}}\}$.*

If P is a minimal state cover and W a characterization set, then $\mathsf{ETS}_{E_{\mathsf{T}},k}(\mathcal{H})$ is given by $P \cdot I^{\leq k-1} \cdot I^{\leq 2} \cdot W$, which is precisely the W-method test suite.

4.1 Active Inputs Expert

Motivation. Mealy machines with many inputs are challenging, even when most inputs induce no interesting behavior, i.e., when most inputs transition to sinks. This challenge is exemplified by the *ASML models* which were first described in [52] and partially made available for the 2019 RERS challenge [27]. The ASML models represent components of lithography systems used at ASML. These models feature many inputs that often lead to a sink state. Model *m135* in particular has approximately 100 inputs that always transition to the sink state with the same output. The Mealy machines $\mathsf{ASML}_{a,b}$ where $a, b \in \mathbb{N}$, displayed in Fig. 3, closely resemble *m135*. The model starts with a spine, then there is a choice between a branches, and the spine inputs are reused in a different order after the choice. There are b distinct inputs that always lead to a sink.

The expert. The *active inputs expert* addresses Mealy machines where there is a significant set of inputs that always lead to the sink state or self-loop. We define the active version of a Mealy machine and then the active inputs expert.

Definition 4.4. *An input $i \in I$ is active in $q \in Q$, if $\delta(q,i) \notin Q_{sink}$ and $\delta(q,i) \neq q$. The* active Mealy machine *of $\mathcal{H} = (Q, I, O, q_0, \delta, \lambda)$ is the partial Mealy machine* $\mathsf{active}(\mathcal{H}) = (Q \setminus Q_{sink}, I', O, q_0, \delta', \lambda')$ *such that*

$$I' = \Big\{ i \in I \mid \exists q \in Q.\ i \textit{ active in } q \Big\},$$

$$\delta'(q,i) = \begin{cases} \delta(q,i) & \textit{if } i \textit{ active in } q, \\ \uparrow & \textit{otherwise}, \end{cases} \quad \textit{and} \quad \lambda'(q,i) = \begin{cases} \lambda(q,i) & \textit{if } \delta'(q,i)\downarrow, \\ \uparrow & \textit{otherwise.} \end{cases}$$

Definition 4.5. *The active inputs expert E_{AI} is given by $E_{\mathsf{AI}}(\mathcal{H}, p) = \{I^{\mathsf{active}(\mathcal{H})}\}$.*

Complexity. The time complexity of E_{AI} is $\mathcal{O}(nk)$, where n is the number of states and k the number of inputs. This is achieved by first determining the set Q_{sink} in $\mathcal{O}(nk)$, and then computing δ' and I' simultaneously in $\mathcal{O}(nk)$.

Completeness. Test suite $\mathsf{ETS}_{E_{\mathsf{AI}},k}$ is complete for the set of Mealy machines which 1) have at most k additional states and 2) where all non-sink states can be reached by a word in the state cover followed by at most k active inputs.

Theorem 4.6. *Suppose $\mathsf{ETS}_{E_{\mathsf{AI}},k}(\mathcal{H})$ uses state cover P. Let $\mathcal{C} = \{\mathcal{S} \in \mathcal{C}^k_{\mathcal{H}} \mid Q^{\mathcal{S}} \setminus Q^{\mathcal{S}}_{sink} \subseteq \Delta^{\mathcal{S}}(P \cdot (I^{\mathsf{active}(\mathcal{H})})^{\leq k})\}$. Then $\mathsf{ETS}_{E_{\mathsf{AI}},k}(\mathcal{H})$ is complete for $\mathcal{C}$.*

The proof follows from Lemma 3.8; the hypotheses make sure that a variant of Lemma 3.7 holds. The above theorem applies in particular, if $\mathcal{H}$ is minimal, for the restriction of $\mathcal{C}^k_{\mathcal{H}}$ to those Mealy machines $\mathcal{S}$ where all non-sink states are reachable by the sub-alphabet generated by E_{AI}.

The active inputs expert performs well on $\mathsf{ASML}_{a,b}$ once the spine is learned because it will not generate infixes with inputs that always lead to the sink state. In the empirical evaluation performed in Sec. 6, it can be observed that E_{AI} requires significantly fewer symbols to learn $\mathsf{ASML}_{a,b}$ compared to E_{T}.

4.2 Future Expert

Motivation. Real-world systems often contain an 'initialization phase' where inputs like *start* or *login* are used that are not used later in the system. Fig. 4 shows the family of Mealy machines $\mathsf{TCP}_{a,b}$ inspired by the TCP models [16]. The models of TCP clients contain two distinct phases: the three-way handshake and the connected part. After the three-way handshake, some inputs are never active again. $\mathsf{TCP}_{a,b}$ has the same two phases. For the last few hypotheses that arise during learning, all inputs will be active. Therefore, E_{AI} will generate the same ETS as E_{T}. E_{AI} is too coarse here because, at different parts of the system, different sets of inputs are active.

The expert. The *future expert* generates a subalphabet for each state in the hypothesis. This subalphabet contains all inputs that are active from that state onwards, within a given number of steps. Bounding the number can be useful in large models, and avoids that we end up with the complete alphabet if the Mealy machine is strongly connected.

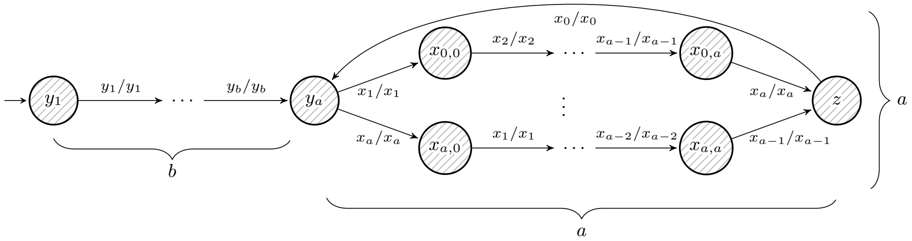


Fig. 4: $\mathsf{TCP}_{a,b}$ models over inputs and outputs $\{x_i \mid 1 \leq i \leq a\} \cup \{y_i \mid 1 \leq i \leq b\}$. Transitions not shown lead to a sink with a unique output.

Definition 4.7. *The future expert* E^l_{F}, *is given for* $l \in \mathbb{N}$ *by* $E^l_{\mathsf{F}}(\mathcal{H}, v) = \{I_{v,l}\}$ *where* $I_{v,l} = \{i \mid \exists q \in \Delta^{\mathsf{active}(\mathcal{H})}(v \cdot I^{\leq l-1}) \wedge \delta^{\mathsf{active}(\mathcal{H})}(q, i) \downarrow\}$.

Complexity. The time complexity $\mathcal{O}(n(n + n|I|))$ can be achieved for E_{F} with a bounded BFS for each state.

Completeness. For E^l_{F}, we have the following completeness result.

Theorem 4.8. *Suppose* $\mathsf{ETS}_{E^l_{\mathsf{F}},k}(\mathcal{H})$ *uses state cover* P. *Let* $\mathcal{C} = \{\mathcal{S} \in \mathcal{C}^k_{\mathcal{H}} \mid Q^{\mathcal{S}} \setminus Q^{\mathcal{S}}_{sink} \subseteq \bigcup_{v \in P} \Delta^{\mathcal{S}}(v \cdot I^{\leq k}_{v,l})\}$. *Then* $\mathsf{ETS}_{E^l_{\mathsf{F}},k}(\mathcal{H})$ *is complete for* $\mathcal{C}$.

E_{F} performs well on $\mathsf{TCP}_{a,b}$ once the spine is learned because the subalphabet for states after y_a does not contain y-symbols, contrary to subalphabet from E_{T}. Sec. 6 shows that E^l_{F} often outperforms the trivial expert E_{T}.

4.3 Components Expert

Motivation. In some systems, sets of inputs are often used together. For example, after entering a username you often enter a password as well. It is possible that the set of inputs that are used together occur at multiple places in the system. Fig. 5 shows Mealy machines $\mathsf{SSH}_{a,b}$, loosely inspired by OpenSSH [18]. The OpenSSH model contains three phases: the key exchange, the authentication, and then the connection phase where re-keying is possible. For the family of Mealy machines $\mathsf{SSH}_{a,b}$, we assume there is a fixed set of possible keys and the key exchange and re-keying uses the same key-specific inputs, i.e., the inputs for the key exchange of key k are the relevant inputs for re-keying with key k.

The expert. The component expert generates subalphabets based on sets of states and is defined as follows.[2]

Definition 4.9. *Let* g *be a function that takes a Mealy machine* $\mathcal{H}$ *and returns a set of subsets of* Q, *referred to below as* components. *The* component expert E^g_{C} *with parameter* g *is defined s.t.* $E^g_{C}(\mathcal{H}, p) = \{I_X \mid X \in g(\mathcal{H})\}$ *where* $I_X = \{i \mid \exists q, q' \in X.\delta(q, i) = q'\}$.

[2] E^l_{F} can be seen as a refined form of the E^g_{C} which returns the subalphabets I_X with X consisting of the states reachable in at most l steps from state p.

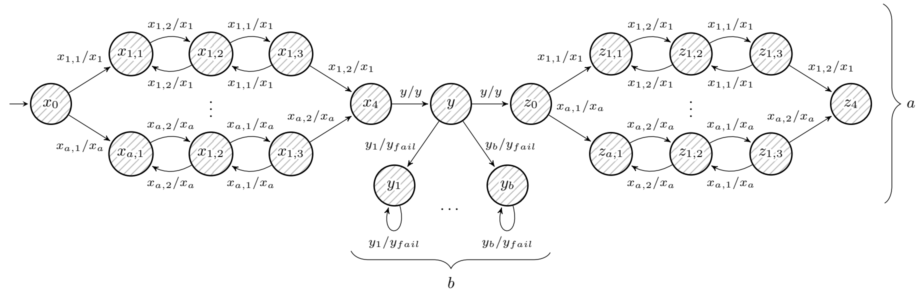


Fig. 5: $\mathsf{SSH}_{a,b}$ models over inputs $\{x_{i,j} \mid 1 \le i \le a, j = 1, 2\} \cup \{y_i \mid 1 \le i \le b\} \cup \{y\}$ and the outputs $\{x_i \mid 1 \le i \le a\} \cup \{y, y_{fail}\}$. Transitions not shown lead to a sink with a unique output.

Finding components. Finding a suitable subroutine g to determine components from a hypothesis is a non-trivial task. One relatively easy method for finding components is to compute the strongly connected components (SCCs). However, if the system can be reset at any state, then the complete model is an SCC and the components expert reduces to the trivial expert. Therefore, SCCs are often too strict. Another possibility is to utilize algorithms used in graph theory to decompose graphs into subgraphs. We propose to use Newman's algorithm for detecting community structure [39] to identify components. The algorithm outputs sets of states with high transition density between states within the group. It starts with singleton communities and then greedily joins communities based on the maximal change in modularity, as long as it is positive. The modularity value mod(c) for component c is:

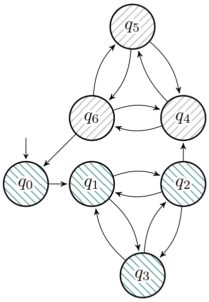


Fig. 6: Example with colored communities.

$$\mathrm{mod}(c) = \frac{\text{\#edges staying in c}}{\text{\#edges}} - \frac{\text{\#outgoing edges of c} \cdot \text{\#incoming edges of c}}{\text{\#edges}^2}$$

Example 4.10. We illustrate Newman's algorithm on Fig. 6. Initially, $\mathrm{mod}(\{q_1\}) = 0 - \frac{2 \cdot 3}{15^2} \approx -0.027, \mathrm{mod}(\{q_3\}) = 0 - \frac{2 \cdot 2}{15^2} \approx -0.018$. The difference between the initial modularity and the modularity of $\{q_1, q_3\}$ $(2 - \frac{4 \cdot 5}{15^2} = 0.0444)$ is the highest possible change in modularity. We thus merge communities $\{q_1\}$ and $\{q_3\}$. Likewise, we then merge $\{q_1, q_3\}$ and $\{q_2\}$. After several steps we get to the final communities $\{q_0, q_1, q_2, q_3\}$ and $\{q_4, q_5, q_6\}$.

To apply Newman's algorithm, the subroutine g transforms $\mathsf{active}(\mathcal{H})$ to a directed graph $G = (Q, E)$ where $E = \{(q, q') \mid q, q' \in Q \wedge \exists i \in I. \delta(q, i) = q'\}$ and then applies Newman's algorithm on G.

Complexity. The time complexity of E^g_C is $\mathcal{O}(g + nk)$ where g is the complexity of the subroutine. The subterm $\mathcal{O}(nk)$ originates from the active transformation. With Newman's algorithm, the total complexity is in $\mathcal{O}(n(n + n|I|))$ [39].

Completeness. $\mathsf{ETS}_{E^g_\mathsf{C},k}$ is k-complete if all non-sink states in the SUL can be reached from a state p in the hypothesis with at most k inputs from some I_X.

Theorem 4.11. *Suppose* $\mathsf{ETS}_{E^g_\mathsf{C},k}(\mathcal{H})$ *uses state cover* P*. Let* $\mathcal{C} = \{\mathcal{S} \in \mathcal{C}^k_\mathcal{H} \mid Q^\mathcal{S} \setminus Q^\mathcal{S}_{sink} \subseteq \bigcup_{X \in g(\mathcal{H})} \Delta^\mathcal{S}(P \cdot I^{\leq k}_X)\}$*. Then* $\mathsf{ETS}_{E^g_\mathsf{C},k}(\mathcal{H})$ *is complete for* $\mathcal{C}$*.*

$E^{\text{Newman}}_\mathsf{C}$ performs well on $\mathsf{SSH}_{a,b}$ once the key exchange and authentication phase have been learned because the subalphabet mostly contains symbols that belong together and allows discovery of a whole new key exchange component. Ideally, $\{x_{i,1}, x_{i,2}, x_{i,3}\}$, $\{z_{i_1}, z_{i_2}, z_{i,3}\}$ for $1 \leq i \leq a$ and $\{y, y_0, ..., y_b\}$ form components for $\mathsf{SSH}_{a,b}$. In our experiments, Newman's algorithm sometimes finds slightly bigger components.

5 Test Case Prioritization

To establish equivalence, *all* tests in a complete test suite need to be executed and their order is then irrelevant. However, to find a counterexample, we only need to execute tests until we hit that counterexample. This means that different orderings lead to significant performance changes [7]. In this section, we first describe the state-of-the-art in (ordered) test suites. We then create new, ordered test suites, that combine the ETS's from Sec. 4 adaptively.

5.1 Randomised Test Suites

Test suites are often stored in a tree-like data structure. The straightforward ordering iterates over this tree to process the test cases deterministically. However, a variety of deterministic orderings for P, I, W are all (on average) outperformed by randomised methods that do a better job in diversification [7, Ch. 4]. State-of-the-art randomised test suite generation methods are described in [34, 46] and make use of a geometric distribution to determine the length of the infix. We present a simpler[3] and more generic variation: Given an expert e and a distribution μ over natural numbers, the *randomised* ETS $S_{e,\mu}$ is a distribution over words $v \cdot i \cdot w \in P \cdot I^* \cdot W$ such that:

$$S_{e,\mu}(v \cdot i \cdot w) = \frac{\mu(l)}{|\mathsf{ETS}_{e,l}|} \qquad \text{for } |i| = l \tag{1}$$

Informally, (1) indicates that the probability of sampling a test case with infix length l is the probability of sampling infix length l from distribution μ and then uniformly sampling a test case from $\mathsf{ETS}_{e,l}$.

[3] The randomised Hybrid-ADS method in [46] first exhausts $P \cdot I^2 \cdot W$ and then generates test cases from $P \cdot I^2 \cdot I^* \cdot W$ where the length of the infix is described by a geometric distribution.

Algorithm 1 Instantiated EXP3 Algorithm for Test Case Generation

```
procedure MAB_EQ($\mathcal{H}$, weights)
    while true do
        probs ← UpdateProbs(weights)                              ▷ Eq. 2
        $e$ ← sample expert proportional to probs
        $\sigma$ ← sample next test case from $S_{e,\mu}$                 ▷ Eq. 1
        $v \leftarrow \lambda^{\mathcal{H}}(q_0^{\mathcal{H}}, \sigma)$
        $v' \leftarrow$ OutputQuery($\sigma$)
        weights ← UpdateWeights(probs, weights, $e, v \neq v'$)      ▷ Eq. 3
        if $v \neq v'$ return Some($\sigma$), weights
```

For any μ with infinite support, the generated test suite is infinite. Thus, randomised ASI methods are test case prioritizations over infinite test suites $P \cdot I^* \cdot W$. Still, randomised ASI methods often find counterexamples faster than k-complete ASI methods [4, 20]. To ensure k-completeness in randomised ASI methods, we need extra bookkeeping to determine whether the right tests have been executed and we can only guarantee that we execute these tests in the limit.

5.2 Multi-Armed Bandits

We want to use all experts from Sec.4 to generate test cases. A naive solution is to determine a static distribution that describes how often an expert should be selected for generating a test case. However, it is unclear how such a distribution should be determined. Instead, we use so-called multi-armed bandits to dynamically update the distribution over available experts using information from previous testing rounds. We refer to this algorithm as the Mixture of Experts. The multi-armed bandits problem was first described by Robbins [43] and is a classic reinforcement learning problem. We instantiate the EXP3 algorithm for adversarial multi-armed bandits [9]. Intuitively, our instantiation prioritizes test cases by better performing experts. We embed the Mixture of Expert algorithm in the MAT framework from Fig. 1 and list the pseudocode in Algorithm 1.

Algorithm 1 is used with randomised ASI-methods. The algorithm is called with a hypothesis $\mathcal{H}$ and weights. The parameter weights indicates how good an expert is and is initialized to 1 for each enabled expert. The algorithm uses the set of enabled experts E, constant k, distribution μ, and exploration parameter γ as global parameters. The exploration parameter determines how often we choose an expert at random. In Algorithm 1, each iteration of the loop represents the generation of one test case. In each iteration, we first update the distribution probs for each expert $i \in E$ using Eq. 2.

$$\mathsf{probs}(i) \leftarrow (1-\gamma) \cdot \frac{\mathsf{weights}(i)}{\Sigma_{j \in E}\mathsf{weights}(j)} + \frac{\gamma}{|E|} \tag{2}$$

Next, we sample an expert from probs and sample a test case from $S_{e,\mu}$. We determine the output of the test case on $\mathcal{H}$ and $\mathcal{S}$ and update the weights for chosen expert e using Eq. 3 if $v \neq v'$, otherwise the weights remain the same.

$$\mathsf{weights}(e) \leftarrow \mathsf{weights}(e) \cdot \mathit{exp}\left(\frac{\gamma}{\mathsf{probs}(i) \cdot |E|}\right) \tag{3}$$

If $v \neq v'$, then we have found a counterexample and the weights value for the chosen expert significantly increases. Consequently, the expert is more likely to be chosen to generate test cases in the next rounds. Finally, if $v \neq v'$, we return the counterexample. Otherwise, generate a new test case.

6 Experimental Evaluation

In this section, we empirically investigate the performance of our implementation of Algorithm 1 in comparison with a state-of-the-art baseline. The source code and all benchmarks are available online[4] [29]. We investigate the performance on four benchmark sets with varying complexities in the first three experiments:

RQ1: How does Algorithm 1 scale on the models from Figs. 3, 4, and 5?
RQ2: How does Algorithm 1 compare to the state-of-the-art on industrial benchmarks from the RERS challenge [27]?
RQ3: How does Algorithm 1 perform on the standard automata wiki [38] benchmark suite and randomly generated Mealy machines?

In Experiment 3, we additionally consider an alternative non-randomised version of the presented algorithm which is not feasible to apply to the benchmarks of Experiment 2 given the worse performance of non-randomised test suites. Experiment 4 provides an in-depth analysis of runs on two benchmarks from the RERS challenge. Detailed benchmark results can be found in Appendix C of [30].

Experimental Setup We have extended the $L^{\#}$ learning library [48] with the multi-armed bandits approach described in Sec. 5. We compare our implementation instantiated with different experts. We write $\mathsf{MoE}(*)$ to refer to our key contribution, using the Mixture of all Experts, i.e., $\mathsf{MoE}(E_{\mathsf{T}}, E_{\mathsf{AI}}, E_{\mathsf{F}}^{k}, E_{\mathsf{C}}^{\text{Newman}})$. The exploration parameter γ used in Algorithm 1 is set to 0.2 (determined by grid search) and the number of hypothesis states before we start sampling experts to 5. We evaluate within a MAT framework as in Figure 1. Our contributions can be paired with any learning algorithm in the MAT framework. We use $L^{\#}$ [48], as this is a recent learning algorithm. We sample test cases from $S_{E_{\mathsf{T}},\mu}$ as our *baseline*. More precisely, we use randomised Hybrid-ADS, as formulated in [34, Ch. 1], as conformance testing technique. For both the baseline and algorithm 1, the μ in Eq. (1) is instantiated as follows: Let geom be the geometric distribution with mean 2, then randomised Hybrid-ADS generates $S_{e,\mu}$ as in Eq. (1), where $\mu(x) = \mathsf{geom}(x)$ if $x > 3$, $\mu(3) = {}^7\!/_8$, and $\mu(x) = 0$ otherwise. These hyperparameters are chosen to match [20]. We run Experiments 1 and 3 with 30 seeds, and Experiments 2 and 4 with 50 seeds. In Experiments 1, 3 and 4 we evaluate the performance based on the total number of symbols and resets which is the

[4] https://gitlab.science.ru.nl/sws/lsharp/-/tree/testingstrategy

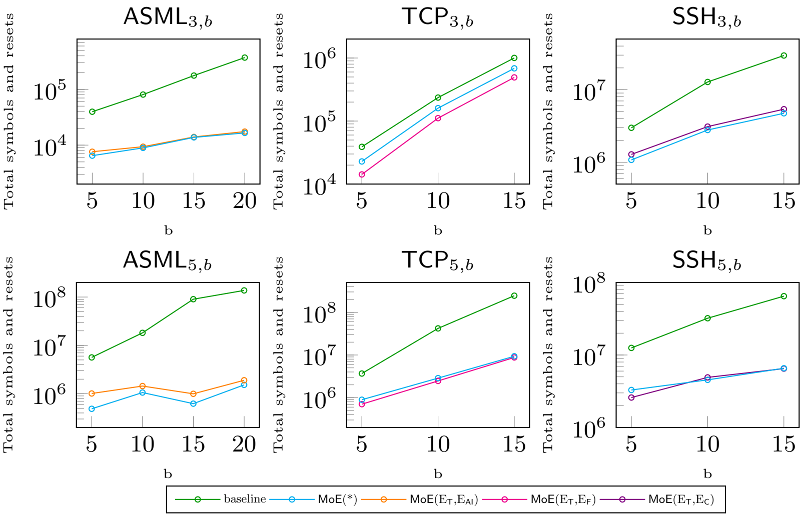


Fig. 7: Results Experiment 1.

sum of the length of all test cases plus the number of test cases sent to the SUT. Additional plots based on only the symbols or only the resets can be found in Appendix D of [30].

Experiment 1 We evaluate the performance on the benchmark families $\mathsf{ASML}_{a,b}$, $\mathsf{TCP}_{a,b}$, and $\mathsf{SSH}_{a,b}$, for several choices of a and b. In all models, increasing a leads to a general increase in difficulty, while b adds the number of 'irrelevant' inputs. Beyond the baseline and $\mathsf{MoE}(*)$, we include for each family the associated experts discussed in Sec. 4, to validate that they indeed perform well on these families. Thus, for $\mathsf{ASML}_{a,b}$ we run $\mathsf{MoE}(E_{\mathsf{T}}, E_{\mathsf{AI}})$, for $\mathsf{TCP}_{a,b}$ we run $\mathsf{MoE}(E_{\mathsf{T}}, E_F^k)$, and for $\mathsf{SSH}_{a,b}$ we run $\mathsf{MoE}(E_{\mathsf{T}}, E_{\mathsf{C}}^{\text{Newman}})$.

Results. Fig. 7 plots the results, distinguishing six cases. Each column reflects another benchmark family. The top row shows the values for the parameterized models with $a = 3$, while the bottom row shows the values for the parameterized models with $a = 5$. In each figure, the x-axis reflects the value of b. The y-axis (log scale) shows the total number of symbols and resets to learn and test a model. The y-axis is different for all subplots.

Discussion. From the plot, we observe that the baseline is outperformed by the other algorithms. Interestingly, the performance of $\mathsf{MoE}(*)$ and the algorithm belonging to the parameterized model is often comparable. Increasing a leads to an increase in the total number of symbols and resets, which illustrates the scalability of the parameterized models. Increasing the value b has more influence on the baseline than the other algorithms, as expected.

Experiment 2 We compare $\mathsf{MoE}(*)$ to the baseline on the ASML benchmarks introduced in the RERS challenge [27]. We consider 23 models with 25-289 states

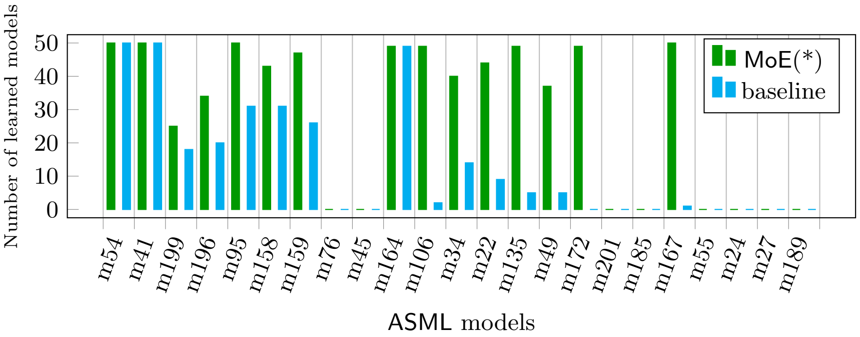


Fig. 8: Results Experiment 2.

and 10-177 inputs. We skip models with less than 15 states because MoE needs time to learn which expert works best. The ASML models frequently do not terminate within a timeout of an hour [52]. Therefore, we set a maximal symbol budget. The SUL rejects new OQs once the budget is depleted.

Results. Fig. 8 lists different models sorted by the number of transitions. For each model, we show how often out of 50 seeds an algorithm learns the model within a symbol budget of 10^8. We provide a similar figure with half the budget in Appendix D of [30].

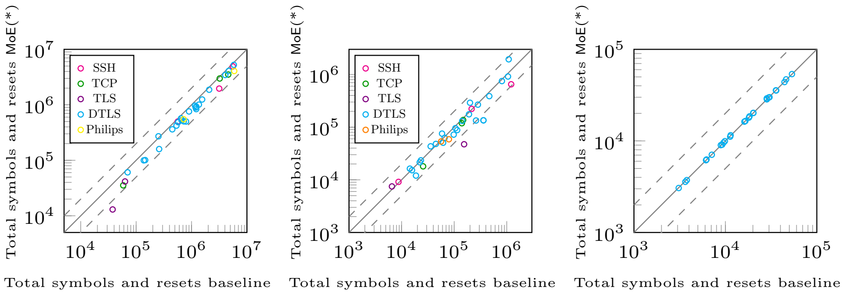
Discussion. From the plot, we observe that MoE($*$) learns the model more often than the baseline. The MoE($*$) algorithm can learn 12 models with at least 80% of the seeds while the baseline only learns 3 models with at least 80% of the seeds. The same pattern can be observed for half the budget.

Experiment 3 We consider the protocol implementations used in [17, 48] (38 models, 15-133 states, 7-22 inputs) and randomly generated models (27 models, 20-60 states, 11-31 inputs). For the standard benchmarks, we perform the experiment with the randomised ETS, as used in the other experiments, and the deterministically ordered ETS from Sec. 5.1 with $k = 2$.

Results. Fig. 9 shows the number of symbols and resets needed to learn and test a model (log-scaled). The y-axis shows MoE($*$) and the x-axis shows the baseline. The diagonal solid lines correspond to using the same number of symbols and resets, the dotted lines indicate a factor two difference. Points in the right triangle indicate that MoE($*$) used fewer symbols and resets than the baseline.

Discussion From Fig. 9a, we observe that MoE($*$) slightly outperforms the baseline in the k-complete test suite setting. From Fig. 9b, we observe that the performance of MoE($*$) leads to slightly better results than the baseline. The performance is comparable for the randomly generated models (Fig. 9c).

Experiment 4 We analyze runs of MoE($*$) and the baseline for models *m159* and *m189* to provide insights on the behavior of the algorithms.



(a) ETS, $k = 2$ (b) Randomised ETS (c) Randomly generated

Fig. 9: Results Experiment 3.

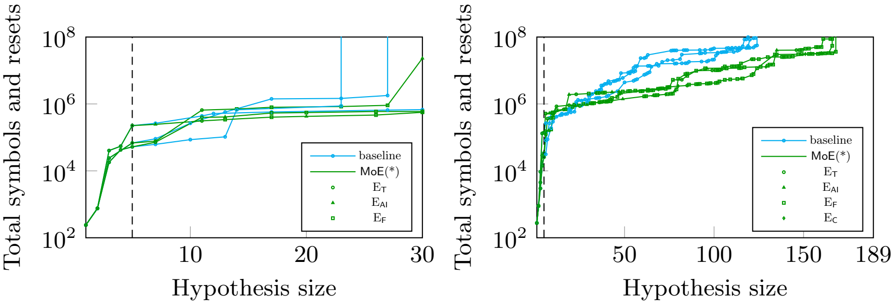


Fig. 10: Results Experiment 4 for *m159* (left) and *m189* (right).

Results. Fig. 10 shows the runs of the first 3 seeds for *m159* and *m189*. Each data point at (x, y) in the subplots represents one hypothesis, with x states, that was learned using a total of y symbols (notice the log scale). The green (or blue) lines correspond to runs with the baseline (or MoE($*$)). The different markers for MoE($*$) indicate which expert was used to generate the counterexample. The vertical lines extending to 10^8 indicate that the algorithm ran out of budget before learning the correct model.

Discussion. In line with Experiment 2, we see that more runs lead to learning the full model using MoE($*$). The plots use the number of states as a rough progress measure. Based on this progress measure, we see that the difference is negligible for small hypothesis sizes, but for larger hypotheses, the difference is substantial. For *m159*, we observe that the baseline runs out of budget before all states have been found, whereas the MoE($*$) is able to learn the correct model within the budget (using the smaller test suites). In *m189*, we observe a significant divergence in progress. On average, the future expert is most used to find counterexamples.

7 Related Work

Test suites. The use of conformance testing [12] is standard in automata learning [47] and goes back to [40]. There are several recent evaluations comparing sample-based conformance testing techniques [4, 7, 20]; these comparisons are orthogonal to the current paper. Another idea is to use mutation testing [3]. Mutation testing performs well on small models (< 100 states) but [3] notices that this technique is computationally too expensive for large models.

Increasing the alphabet size. Instead of reducing the alphabet size for a more guided counterexample finding, a common theme is to use abstraction refinement [23, 47] during learning to iteratively refine the alphabet. Bobaru et al. [10] learn models using abstractions of components to later show that a property holds or is violated. Additionally, Vaandrager and Wißman [49] formally describe the relation between high-level state machines and low-level models using abstraction refinement.

Using the automata structure. A recent trend is gray-box automata learning, which assumes partial information on the SUL and aims to exploit this information. In particular, learning algorithms addressing various types of composition (sequential, parallel, product) have been investiged [2,31,33,37]. However, all these techniques adapt the learning algorithm, not the testing algorithm, as in the current paper. Furthermore, while the results in Sec. 4 are similar to a gray-box setting, the idea in Sec. 5 is that this work leads to better performance in the strict black-box setting, as highlighted by the experiments.

Algorithm selection. Machine learning for algorithm selection is an active area of research, see e.g., [1, 28] and has been applied successfully, e.g., in the context of SAT checking [21]. In formal methods, multi-armed bandits framework has been used, e.g., to prioritize SMT solver over others [42] or to guide falsification processes for hybrid systems [53]. In automata learning, bandits have recently been applied to select between different oracles for answering output queries [45].

8 Conclusion

In this paper, we introduced smaller test suites for conformance testing that preserve the typical completeness guarantees under natural assumptions on the learned system. The paper demonstrates that a combination of these test suites and a multi-armed bandit formulation significantly accelerates modern active automata learning, even when the assumptions do not hold. Natural extensions include adding additional small test suites, designing variations of the presented experts to, for example, handle parallel components [31, 37], and using a multi-armed bandit to select the essential parameter k. Furthermore, our approach paves the way for using similar assumptions to those made for the completeness of the expert test suites in other aspects of active automata learning.

References

1. Salisu Mamman Abdulrahman, Pavel Brazdil, Jan N. van Rijn, and Joaquin Vanschoren. Speeding up algorithm selection using average ranking and active testing by introducing runtime. *Mach. Learn.*, 107(1):79–108, 2018.
2. Andreas Abel and Jan Reineke. Gray-box learning of serial compositions of mealy machines. In *NFM*, volume 9690 of *LNCS*, pages 272–287. Springer, 2016.
3. Bernhard K. Aichernig and Martin Tappler. Efficient active automata learning via mutation testing. *J. Autom. Reason.*, 63(4):1103–1134, 2019.
4. Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. Benchmarking combinations of learning and testing algorithms for active automata learning. In *TAP@STAF*, volume 12165 of *LNCS*, pages 3–22. Springer, 2020.
5. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
6. Dana Angluin. Queries and concept learning. *Mach. Learn.*, 2(4):319–342, 1987.
7. Kousar Aslam. *Deriving behavioral specifications of industrial software components*. PhD thesis, Eindhoven University of Technology, 2021.
8. Kousar Aslam, Loek Cleophas, Ramon R. H. Schiffelers, and Mark van den Brand. Interface protocol inference to aid understanding legacy software components. *Softw. Syst. Model.*, 19(6):1519–1540, 2020. URL: https://doi.org/10.1007/s10270-020-00809-2.
9. Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, 2002.
10. Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, and Dimitra Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *CAV*, volume 5123 of *LNCS*, pages 135–148. Springer, 2008.
11. Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In *CAV*, volume 6174 of *LNCS*, pages 360–364. Springer, 2010.
12. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005.
13. Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. Automated reverse engineering using lego®. In *WOOT*. USENIX Association, 2014.
14. Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.
15. Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium*, pages 193–206. USENIX Association, 2015.
16. Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *CAV (2)*, volume 9780 of *LNCS*, pages 454–471. Springer, 2016.
17. Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *USENIX Security Symposium*, pages 2523–2540. USENIX Association, 2020.
18. Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *SPIN*, pages 142–151. ACM, 2017.
19. Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Software Eng.*, 17(6):591–603, 1991.

20. Bharat Garhewal and Carlos Diego Nascimento Damasceno. An experimental evaluation of conformance testing techniques in active automata learning. In *MODELS*, pages 217–227. IEEE, 2023.
21. Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Automated configuration and selection of SAT solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 481–507. IOS Press, 2021.
22. Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Inferring semantic interfaces of data structures. In *ISoLA (1)*, volume 7609 of *LNCS*, pages 554–571. Springer, 2012.
23. Falk Howar and Bernhard Steffen. Active automata learning in practice - an annotated bibliography of the years 2011 to 2016. In *Machine Learning for Dynamic Software Analysis*, volume 11026 of *LNCS*, pages 123–148. Springer, 2018.
24. Malte Isberner. *Foundations of active automata learning: an algorithmic perspective*. PhD thesis, Technical University Dortmund, Germany, 2015.
25. Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV*, volume 8734 of *LNCS*, pages 307–322. Springer, 2014.
26. Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib - A framework for active automata learning. In *CAV (1)*, volume 9206 of *LNCS*, pages 487–495. Springer, 2015.
27. Marc Jasper, Malte Mues, Alnis Murtovi, Maximilian Schlüter, Falk Howar, Bernhard Steffen, Markus Schordan, Dennis Hendriks, Ramon R. H. Schiffelers, Harco Kuppens, and Frits W. Vaandrager. RERS 2019: Combining synthesis with real-world models. In *TACAS (3)*, volume 11429 of *LNCS*, pages 101–115. Springer, 2019.
28. Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evol. Comput.*, 27(1):3–45, 2019.
29. Loes Kruger, Sebastian Junges, and Jurriaan Rot. Small Test Suites for Active Automata Learning: Supplemental Material, December 2023. doi:10.5281/zenodo.10437793.
30. Loes Kruger, Sebastian Junges, and Jurriaan Rot. Small test suites for active automata learning, 2024. arXiv:2401.12703.
31. Faezeh Labbaf, Jan Friso Groote, Hossein Hojjat, and Mohammad Reza Mousavi. Compositional learning for interleaving parallel automata. In *FoSSaCS*, volume 13992 of *LNCS*, pages 413–435. Springer, 2023.
32. Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Software Eng.*, 20(2):149–162, 1994.
33. Joshua Moerman. Learning product automata. In *ICGI*, volume 93 of *Proceedings of Machine Learning Research*, pages 54–66. PMLR, 2018.
34. Joshua Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. PhD thesis, Radboud University, 2019.
35. Edward F Moore et al. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.
36. Edi Muskardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. AALpy: An active automata learning library. In *ATVA*, volume 12971 of *LNCS*, pages 67–73. Springer, 2021.
37. Thomas Neele and Matteo Sammartino. Compositional automata learning of synchronous systems. In *FASE*, volume 13991 of *LNCS*, pages 47–66. Springer, 2023.

38. Daniel Neider, Rick Smetsers, Frits W. Vaandrager, and Harco Kuppens. Benchmarks for automata learning and conformance testing. In *Models, Mindsets, Meta*, volume 11200 of *LNCS*, pages 390–416. Springer, 2018.
39. Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.
40. Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7(2):225–246, 2002.
41. Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. Nondeterministic state machines in protocol conformance testing. In *Protocol Test Systems*, volume C-19 of *IFIP Transactions*, pages 363–378. North-Holland, 1993.
42. Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. Medleysolver: Online SMT algorithm selection. In *SAT*, volume 12831 of *LNCS*, pages 453–470. Springer, 2021.
43. Herbert Robbins. Some aspects of the sequential design of experiments. 1952.
44. Mathijs Schuts, Jozef Hooman, and Frits W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In *IFM*, volume 9681 of *LNCS*, pages 311–325. Springer, 2016.
45. Ameesh Shah, Marcell Vazquez-Chanlatte, Sebastian Junges, and Sanjit A. Seshia. Learning formal specifications from membership and preference queries. *CoRR*, abs/2307.10434, 2023.
46. Wouter Smeenk, Joshua Moerman, Frits W. Vaandrager, and David N. Jansen. Applying automata learning to embedded control software. In *ICFEM*, volume 9407 of *LNCS*, pages 67–83. Springer, 2015.
47. Frits W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, 2017.
48. Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. In *TACAS (1)*, volume 13243 of *LNCS*, pages 223–243. Springer, 2022.
49. Frits W. Vaandrager and Thorsten Wißmann. Action codes. In *ICALP*, volume 261 of *LIPIcs*, pages 137:1–137:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
50. MP Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, 1973.
51. Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 5244–5253. PMLR, 2018.
52. Nan Yang, Kousar Aslam, Ramon R. H. Schiffelers, Leonard Lensink, Dennis Hendriks, Loek Cleophas, and Alexander Serebrenik. Improving model inference in industry by combining active and passive learning. In *SANER*, pages 253–263. IEEE, 2019.
53. Zhenya Zhang, Ichiro Hasuo, and Paolo Arcaini. Multi-armed bandits for boolean connectives in hybrid system falsification. In *CAV (1)*, volume 11561 of *LNCS*, pages 401–420. Springer, 2019.

Mata: A Fast and Simple Finite Automata Library

David Chocholatý, Tomáš Fiedor, Vojtěch Havlena, Lukáš Holík(✉), Martin Hruška, Ondřej Lengál, and Juraj Síč

Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
`holik@fit.vutbr.cz`

Abstract. Mata is a well-engineered automata library written in C++ that offers a unique combination of speed and simplicity. It is meant to serve in applications such as string constraint solving and reasoning about regular expressions, and as a reference implementation of automata algorithms. Besides basic algorithms for (non)deterministic automata, it implements a fast simulation reduction and antichain-based language inclusion checking. The simplicity allows a straightforward access to the low-level structures, making it relatively easy to extend and modify. Besides the C++ API, the library also implements a Python binding.

The library comes with a large benchmark of automata problems collected from relevant applications such as string constraint solving, regular model checking, and reasoning about regular expressions. We show that Mata is on this benchmark significantly faster than all libraries from a wide range of automata libraries we collected. Its usefulness in string constraint solving is demonstrated by the string solver Z3-Noodler, which is based on Mata and outperforms the state of the art in string constraint solving on many standard benchmarks.

1 Introduction

We introduce a new finite automata library Mata[1]. It is intended to be used in applications where automata languages are manipulated by set operations and queries, presumably in a tight loop where automata are iteratively combined together using the classical as well as special-purpose constructions. Examples are applications like *string constraint solving* algorithms such as [11,24,22,1,10,3,71], processing of regular expressions [28,49], *regular model checking* (e.g., [16,15,58,26,13,81,6]), or *decision procedures for logics* such as WS1S or quantified Presburger arithmetic [20,80,43,12]. The solved problems are computationally hard, often beyond the PSPACE-completeness of basic automata problems such as language inclusion. Efficiency is hence a primary concern. Achieving speed in applications requires, on one hand, fast implementation of basic automata algorithms (union, intersection, complement, minimization or size reduction, determinization, emptiness/inclusion/equivalence/membership test, parsing of regular expressions) and, on the other hand, access to low-level primitives to implement diverse application-specific algorithms and optimizations that often build on a tight integration with the application environment. Moreover, processing of regular expressions and, even more so, string constraint solving are areas of active research, with constantly evolving algorithms, heuristics, and optimizations. An automata library hence needs flexibility, extensibility, easy access to the low-level data structures, and

[1] `https://github.com/VeriFIT/mata`

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 130–151, 2024.
https://doi.org/10.1007/978-3-031-57249-4_7

ideally a low learning curve, which is important when involving students in academic research and utilizing limited resources of small research teams.

Fast and simple are therefore our two main requirements for the library. An additional third requirement is a well-engineered infrastructure and a good set of benchmarks and tests, important for effective research and reliable deployment. MATA is therefore built around a data structure for the transition relation of a non-deterministic automaton that is a compromise between simplicity and speed. It represents transitions explicitly, as triples of a sources state, a single symbol, and a target state. This contrasts with various flavors of symbolic representation of transition relation used in advanced automata implementations in order to handle large or infinite alphabets (e.g. Unicode in processing of texts, or bit vectors in reasoning about LTL, arithmetic, or WS1S). However, in the applications we consider, working internally with large alphabets can essentially always be avoided by preprocessing (mainly by *mintermization*, aka factorization of the alphabet). The simplicity of an explicit representation then seems preferable. It allows to use a data structure specifically tailored for computing post-images of tuples and sets of states in automata algorithms: a source state-indexed array, storing at each index the transitions from that source state in a two layered structure, with the first layer divided and ordered by symbols, and the second layer ordered by target states. The data structure seems to be unique among the existing libraries and yields an exceptional performance.

MATA currently provides basic functionality, basic automata operations and tests, parsing of regexes and automata in a textual format, and mintermization. From the more advanced algorithms for working with non-deterministic automata, it implements antichain-based inclusion checking [35], and simulation-based size reduction based on the advanced algorithm of [65,4,48]. The inclusion check appears to be by a large margin the fastest implementation available, and together with the tree automata library VATA [59], MATA is the only library with an implementation of a simulation algorithm of the second generation originating from [65,21] (the second generation algorithms combine partition-relation pairs to manipulate preorders that were handled explicitly by the first generation algorithms such as [44,52]). MATA is implemented in C++, uses almost exclusively the STL library for its data structures, and has no external dependencies[2] This makes it relatively easy to learn and integrate with other software projects. It is a well-engineered project at GitHub, with modern test and quality of code assurance infrastructure. Besides the C++ API, it provides a Python binding for fast prototyping and easy experimenting, for instance using interactive Jupyter notebooks.

We evaluated its speed in, to our best knowledge, so far the most comprehensive comparison of automata libraries. We compare with 7 well-known automata libraries on a large benchmark of problems from domains close to MATA's designation, mainly string constraint solving, processing regular expressions, and regular model checking. MATA consistently outperforms all other libraries, from several times to orders of magnitude.

That MATA is a good fit for string constraint solving is demonstrated by its central role in the string solver Z3-NOODLER, which implements the algorithms of [11,24], and outperforms the state of the art on many standard benchmarks (see [25] for details).

[2] Although, at the moment, it uses the BDD library CUDD [70] in mintermisation and the regular expression parser from RE2 [41]. The code from these projects is, however, contained within MATA. Moreover, the connection to CUDD is not tight and we plan to remove it in the future.

Our contributions can be summarised by the following three points:

1. Mata, a fast, simple, and well-engineered automata library, well suited for application in string constraint solving and regex processing, in research and student projects, as well as in industrial applications.
2. An extension of a benchmark of automata problems from string constraint solving, processing regular expressions, regular model checking, and solving arithmetic constraints.
3. A comparison of a representative sample of well-known automata libraries against the above benchmark, demonstrating the superior performance of Mata.

2 Related Work

In this overview of automata algorithms and implementations, we focus on the technology relevant to Mata, i.e., automata used as a symbolic representation of sets of words and manipulated mainly by set operations. We omit automata technology made for other purposes, such as regular pattern matching, which concentrates on the membership test.

Automata techniques. The most textbook-like approach is to keep finite automata deterministic (the so-called DFA), which has the advantage of simple algorithms and data structures. Essentially all classical problems reduce to product construction, determinization by subset construction, final state reachability test, and minimization (by Hopcroft's [50], Moore's [62], Brzozowski's [19], or Huffman's [51] algorithms). The obvious drawback is the susceptibility to state explosion in determinization.

An alternative is to determinize automata only when necessary (e.g., only before complementing). Non-determinism may bring up to exponential savings in automata sizes and modern algorithms for *nondeterministic finite automata* (NFA) can in practice avoid the exponential worst-case cost of problems like the language inclusion test.

Namely, a major breakthrough in working with NFAs were the antichain-based algorithms for testing language universality and inclusion of NFA first introduced (to the best of our knowledge) in [74] and later rediscovered in [82]. They dramatically improve practical efficiency of the subset construction by subsumption pruning (discarding larger sets). They were later extended with simulation [5,35] (and generalized to numerous other kinds of automata and problems). A principally similar is the bisimulation up-to congruence technique of [14], which optimizes the NFA language equivalence test. Although experimental data in various works are somewhat contradictory, the more systematic studies so far found antichain-based algorithms more efficient [39,38].

NFAs require more involved reduction methods than DFAs, such as those based on simulation [65,21,44,52,48] or bisimulation [76,64,46]. Simulation reduces significantly more but is much more costly. The algorithms for computing simulation of the *second generation* [65,21], which use the so-called partition-relation pairs to represent preorders on states, are practically much faster than the *first generation algorithms* [44,52].

Representations of the transition relation. In order to handle automata over large or infinite alphabets, such as Unicode or bit vectors, some implementations of automata represent transitions symbolically. Transitions may be annotated by sets of symbols represented as BDDs, logical formulae, intervals of numbers, etc. The most systematic approach to this has been taken in works on *symbolic automata* [78,32,33], where the symbol predicates may be taken from any *effective Boolean algebra* (essentially a countable set closed under Boolean operations). Some libraries, such as Spot [36], Owl [57], or Mosel [55] use BDDs to compactly represent sets of symbols on transitions. Even more compact are the symbolic representations of the transition relation used in Mona [43] and in the symbolic version of the tree automata library Vata [59], where all transitions starting at a state are represented as a single multi-terminal BDDs with the target states in the leaves (the paths represent symbols). Although symbolic representation may offer new optimization opportunities [32] and give more generality, it also brings complexity and overhead. Adapting the known algorithms may be nontrivial [32,46] to the point of being a difficult unsolved problem (such as the fast computation of simulation relation of [65,21]). In our application area, working with large alphabets can mostly be avoided in preprocessing, for instance by means of a priori mintermization (partitioning the alphabet into groups of symbols indistinguishable from the viewpoint of the input problem). The simplicity and transparency of explicit representation of transitions then seems preferable.

Alternating automata. Alternating automata (AFA) received attention recently in the context of string solving and regex processing [79,28,45,40]. They allow to keep automata operations implicit up to the point of the PSPACE-complete emptiness test, which can be solved by clever heuristics (e.g. [79,28,45,82,38,30]). Available implementations were recently compared with selected NFA libraries [38] and neither approach dominated. AFA are, however, often not a viable alternative since adapting complex algorithms from, e.g., string solving to AFA typically requires to redesign the entire algorithm from scratch (as, e.g., in [45,79]).

String solving and SMT solvers. String constraint solving is currently the primary application target of Mata. Mata is already a basis of an efficient string solver Z3-Noodler [25] and a number of other string solvers could perhaps benefit from its performance, especially those that already use automata as a primary data structure, e.g. [23,3,10,1]. Besides, SMT string constraint solvers can also be used to reason about regular properties, though the results of [38] suggest that their efficiency is not on par with dedicated fast automata libraries.

Automata libraries. We give overview of known automata libraries with a focus on those that we later include in our experimental comparison in Section 6.

The Brics [63] automata library is often considered a baseline in comparisons. It implements both NFA and DFA, where each state keeps the set (implemented as a hash map) of transitions, which are represented symbolically using character ranges. It is written in Java and relatively optimized.

The Automata.net library [77], written in C#, implements symbolic NFA parameterized by an effective Boolean algebra. The transition relation (as well as its inverse) are implemented as a hash map from states to the dynamic array of transitions from a given state, each transition annotated with a predicate over the algebra. We use it in our comparison with the algebra of BDDs. AUTOMATA.NET has been developed for a long time and has accumulated a number of novel techniques (e.g., an optimized minimization [31]).

MONA [43], written in C, is a famous optimized implementation of deterministic automata used for deciding WS1S/WSkS formulae. To handle DFA with complex transition relations over large alphabets of bit vectors, MONA uses a compact fully symbolic representation of the transition relation: a single MTBDD for all transitions originating in a state, with the target states in its leaves. MONA can represent only a DFA, hence every operation implicitly determinizes its output.

VATA [59], written in C++, implements non-deterministic tree automata. It can be used with NFA, too as they are a special case of tree automata. It is relatively optimized and features fast implementation of the antichain-based inclusion checking [15,47] (which for NFA boils down to the inclusion check of [35]) and the second generation simulation computation algorithm of [48].

AWALI [60] is a library that targets weighted automata and transducers over an arbitrary semiring. To implement the transition relation, it keeps a vector of transitions and for each state s two vectors: one keeps the indices of transitions leaving s and the other one the indices of transitions entering s.

AUTOMATALIB [53] is a Java automata library and the basis of the automata learning framework LearnLib [54]. It focuses on DFAs and implements their transition relation as a flattened 2D matrix that maps the source state and symbol to the target state.

AUTOMATA.PY [37] is written in Python. It defines the transition relation in a liberal way, as any mapping from source states to a mapping of symbols to a target state (DFA) or to a set of target states (NFA).

FADO [7] is a Python library written with efficiency in mind. It uses a similar structure as AUTOMATA.PY, but more specific, with the transition as a Python dictionary (a hash map), and states represented as numbers used as indices into an array.

There is a number of other automata libraries that we do not include into our comparison since they seem similar to the included ones or we were not able to use them. The C alternative of BRICS [61] and the Java implementation of symbolic NFA of [29] are in our experiment covered by AUTOMATA.NET and BRICS. ALASKA [34] contains interesting implementations of antichain-based algorithms, but is no longer maintained nor available. LASH [12] is a long-developed tool for arithmetic reasoning based on automata, with an efficient core automata library, written in C. Its transition relation is an array indexed by states, where every state is associated with a symbol-target ordered list of transitions. LASH uses partial symbolic representation – it encodes symbols as sequence of binary digits. The comparison with MONA in [56] on automata benchmark originating from arithmetic problems placed its performance significantly behind MONA. It seems to no longer be maintained, and we were not able to run it on our benchmarks.

There is also a number of implementations of automata over infinite words, for instance SPOT [36], OWL [57], or GOAL [75], which are in their nature close to the finite word automata libraries (SPOT and OWL are optimized and use BDDs on transition edges similarly as AUTOMATA.NET), but implement different algorithms.

Mata evolved from a prototype implementation eNfa used in the comparison of AFA emptiness checkers as a baseline implementation of classical automata [38]. Surprised by its performance, we decided to turn it into a serious widely usable library. Current Mata is much more mature and efficient than the eNfa of [38].

3 Preliminaries on Finite Automata

Words and alphabets. An *alphabet* is a set Σ of *symbols/letters* (usually denoted $a, b, c, \ldots$) and the set of all words over Σ is denoted as Σ^*. The *concatenation* of words u and v is denoted by $u \cdot v$. The *empty word*, the neutral element of concatenation, is denoted by ϵ ($\epsilon \notin \Sigma$).

Finite automata. A *(nondeterministic) finite automaton (NFA)* over an alphabet Σ is a tuple $\mathcal{A} = (Q, post, I, F)$ where Q is a finite set of *states*, $post\colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is a *symbol-post function*, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A *run* of $\mathcal{A}$ over a word $w \in \Sigma^*$ is a sequence $p_0 a_1 p_1 a_2 \ldots a_n p_n$ where for all $1 \leq i \leq n$ it holds that $a_i \in \Sigma \cup \{\epsilon\}$, $p_i \in post(p_{i-1}, a_i)$, and $w = a_1 \cdot a_2 \cdots a_n$. The run is *accepting* if $p_0 \in I$ and $p_n \in F$, and the language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of all words for which $\mathcal{A}$ has an accepting run. $\mathcal{A}$ is called *deterministic (DFA)* if $|I| \leq 1$, $|post(q, \epsilon)| = 0$, and $|post(q, a)| \leq 1$ for each $q \in Q$ and $a \in \Sigma$. A state is *useful* if it belongs to some accepting run, else it is *useless*. An automaton with no useless states is *trimmed*. A state is *reachable* if it appears on a run starting at an initial state. In Mata, we further use $post(q) = \{(a, post(q, a)) \mid post(q, a) \neq \emptyset\}$ to denote the *state-post* of q. We call symbol-post and state-post the *post-image functions*. We also use $q \text{-}a\text{→} p$ where $p \in post(q, a)$ to denote *transitions*. The set of all transitions of $\mathcal{A}$ is called the *transition relation* of $\mathcal{A}$ and we denote it by Δ.

Automata operations. In this paragraph we assume automata without ϵ transitions. The *subset construction* generates from $\mathcal{A}$ the DFA $(Q^{\subseteq}, post^{\subseteq}, I^{\subseteq}, F^{\subseteq})$ where $Q^{\subseteq} = \mathcal{P}(Q)$, $I^{\subseteq} = \{I\}$, $F^{\subseteq} = \{S \in Q^{\subseteq} \mid S \cap F \neq \emptyset\}$, and where $post^{\subseteq}(S, a) = \bigcup_{s \in S} post(s, a)$. The automaton for *complement* is obtained from it by complementing $F^{\subseteq}$, i.e., the set of final states is given as $Q^{\subseteq} \setminus F^{\subseteq}$. The *intersection* of two automata $\mathcal{A}_1 = (Q_1, post_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, post_2, I_2, F_2)$ is implemented by their product $(Q_1 \times Q_2, post^{\times}, I_1 \times I_2, F_1 \times F_2)$ where $post^{\times}((q, r), a) = post_1(q, a) \times post_2(r, a)$. A sensible implementation of course only computes the reachable parts of the product and the subset construction. The *union* $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ is obtained by disjointly uniting all components of $\mathcal{A}_1$ and $\mathcal{A}_2$. Similarly, the *concatenation* $L(\mathcal{A}_1).L(\mathcal{A}_2)$ is the automaton $(Q_1 \uplus Q_2, post_1 \uplus post_2 \uplus post', I_1, F')$ where $\uplus$ denotes the disjoint union, $post'(q, a) = \{r \mid q \in F_1 \land \exists s \in I_2\colon r \in post_2(s, a)\}$ is the connecting symbol-post and F' is F_2 if $I_2 \cap F_2 = \emptyset$ and $F_1 \cup F_2$ otherwise (this construction avoids introducing ϵ-transitions). Note that we omit superscript of symbol-post function when it is clear from the context.

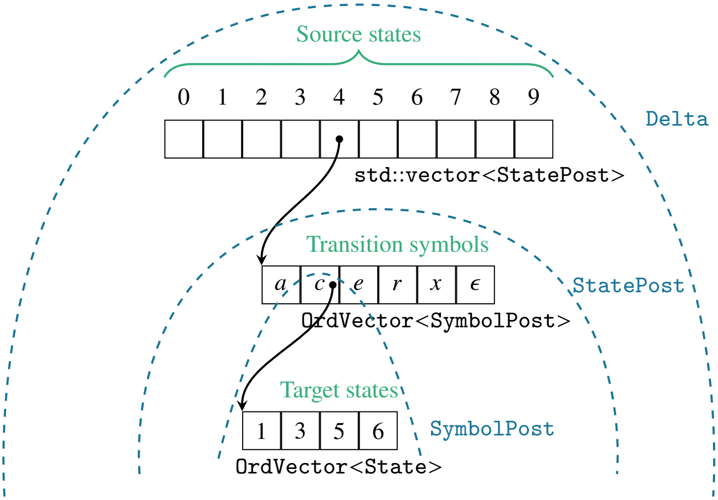


Fig. 1: The transition relation.

4 The Architecture of Mata

We explain in this section the implementation techniques that make Mata efficient on a wide range of automata operations.

4.1 Automata Representation

States and transition symbols are unsigned integers (starting from 0). This makes it easy to store information about them in a state-/symbol-indexed vectors. A frequently used low-level data structure is `OrdVector`, a set of ordered elements implemented as an ordered array (with `std::vector` as the underlying data structure). It has constant time addition and removal of the largest element (`push_back` and `pop_back`), linear union, intersection, and difference (by variants of merging), good memory locality and fast iteration through elements, logarithmic lookup (by binary search), but a slow insertion and removal (`insert` and `erase`) at other than the last position, as the elements on the right of the modified position must be shifted. Many Mata algorithms utilize the constant time handling of the largest element in, e.g., synchronized traversal of multiple `OrdVector` containers. Initial and final states are kept in sparse sets [18], with fast iteration through elements and constant lookup, insertion, and removal.

Data structure for the transition relation. The main determinant of Mata is its three-layered data structure `Delta` for the transition relation. It is implemented as a vector `post` where, for every state q, `post`$[q]$ is of the type `StatePost`, representing $post(q)$ as an `OrdVector` of objects of the type `SymbolPost`, each in turn representing one $post(q, a)$

by storing the symbol a and an OrdVector of the target states. The SymbolPosts in OrdVector are ordered by their symbols[3]. A visualization of Delta is shown in Fig. 1.

The weak point of Delta is inherited from OrdVector: slow insert or erase of a specific transition (these operations are, however, used scarcely in the considered scenarios). Its strength is mainly fast iteration through the post-image of a state, of a pair of states in the product construction, and of a set of states in the subset construction.

4.2 Automata Operations

Generating post-images in subset construction. In the subset construction, each iteration through $post(S)$ for a set of states S is keeping an array of iterators, one into each $post(q)$ for all $q \in S$. Every iteration shifts the iterators to the right, to $post(q, b)$ where b is the closest from above to the current global minimal symbol a, and returns $post(S, a)$ as the union of all $post(q, a)$'s pointed to by the iterators. No searching in vectors is needed. The entire iteration through all $post(S)$'s makes the iterators in the SymbolPosts traverse their respective vectors only once.

Constructing the transitions leading from S while iterating through $post(S)$ is done by appending to OrdVectors, without a need to insert at internal positions of vectors. The iteration through the SymbolPosts is ordered by symbol, hence each newly created transition from the macrostate S has a larger symbol than all the previously created ones. The symbol-post therefore belongs at the end of the OrdVector of symbol-posts of $post(S)$, where it is push_backed. Since the resulting automaton is deterministic, the vectors of targets are singletons, and their creation does not require insert either.

Generating post-images in product construction. Similarly as in the subset construction above, iterating through $post((q, r))$ in the product construction is done by synchronous iteration through $post(q)$ and $post(r)$ from the smallest common symbol to the largest. In each step, the iteration returns the Cartesian product of the targets in the symbol-posts. Unlike the subset construction, adding the corresponding transitions from (q, r) to the product automaton sometimes does need an insert into the vector of targets. It is however not that frequent: Newly discovered product states are assigned the so far highest numbers, so these are added to the target vectors by push_back. The insert may hence be needed only when creating a non-deterministic transition to a state discovered earlier.

Storing sets and pairs of states in the subset and product construction. OrdVector is also used to map generated sets in the subset to the identities of generated states. The map uses a hash table (std::unordered_map) where values are OrdVectors. The product construction uses either a two-dimensional array to map pairs of states to product states (for smaller automata) or a vector pro_map of hash tables, where the identity of the product state (q, r) is found in the hash map prod_map[q] under the key r.

[3] MATA supports ϵ-transitions and some operations can work with them internally. We represent ϵ as the symbol with the highest possible number, hence SymbolPost with ϵ is always the last one in the vectors of SymbolPosts in Delta. The ϵ is therefore easy to be accessed in, e.g., ϵ-transition elimination. Some operations also support several ϵ-like symbols (e.g., ϵ_1, $\epsilon_2, \ldots$), which are convenient in some algorithms in string solving [11,24] or can play a role of different synchronization symbols, etc.

Emptiness test and trimming. Emptiness test and trimming are used frequently and must be fast. Mata's emptiness test is just a state space exploration that utilizes the fast iteration through post-images of a state.

Trimming consists of two steps: (1) identification of useful states and (2) removal of states that are not useful. Identification of useful states must, besides forward exploration to identify reachable states, identify states that reach a final state. A naive solution would be a backward exploration from final states. `Delta` is, however, not well suited for backward search and although reverting it is doable, its cost is not negligible either. We therefore use a smarter solution, which uses a simplification of the non-recursive Tarjan's algorithm [73] to discover strongly connected components (SCCs). Tarjan's algorithm is essentially a depth-first exploration augmented to identify the SCCs. To identify useful states, on finding an SCC with a final state, we mark the entire SCC as useful together with all states on the path to that SCC, which is readily stored on the depth-first search stack. The cost of computing useful states is then similar to the cost of a single depth-first exploration, which is indeed negligible.

Removal of useless states then needs to be done in a `Delta`-friendly way. The naive approach that removes useless states and transitions incident with them one by one would be extremely slow due to the need of searching and calling `erase` in the `OrdVectors` of `Delta`. Instead, we perform the whole removal and related operations in a single pass through `Delta`. Before the pass begins, first, we create a map `renaming` mapping each useful state to its new name (the trimming also renames the states in order to have the remaining states form a consecutive sequence). During the pass, the following operations need to be performed: (i) in the outermost loop, each useful state q in `Delta` is moved to index `renaming`$[q]$, (ii) in every vector of target states, each useful target is moved to the left in the target vector by that many positions, as there were smaller useless states before it, and (iii) while doing that, the target state q is renamed to `renaming`$[q]$.

Union and concatenation. Mata is relatively slow in operations that copy or create large parts of automata, such as non-deterministic union or concatenation, or simple copying of an automaton. This is perhaps due to the imperfect memory locality (the three layers of vectors in `Delta`) and the need to copy every single transition (unlike, e.g., symbolic automata with BDDs on transitions, where the BDDs may be shared). Mata has, however, in-place variants of union and concatenation, which do not copy `Delta`, but only append the `post` vectors and rename the target states in the appended part, which is fast. The price for the speed is the loss of the original automata, but they are in many use cases not needed (as, e.g., in inductive constructions of automata from regular expressions or formulae).

Antichain-based inclusion checking. Mata implements the antichain-based inclusion checking of [35]. Given the inclusion problem $L(\mathcal{A}) \subseteq L(\mathcal{B})$, the algorithm explores the space of the product of $\mathcal{A}$ and the subset construction on $\mathcal{B}$, consisting of pairs (q, S) with q being a state of $\mathcal{A}$ and S being a set of states of $\mathcal{B}$. In particular, it searches, on the fly, for a reachable pair (q, S) with a final q and a non-final S, which would be a witness non-inclusion. The algorithm optimizes the search by *subsumption pruning*—discarding states (q, S) if another (q, S') with $S \subseteq S'$ has been found. Our implementation uses the infrastructure for computing post-images of product and subset construction discussed above. The reached pairs (q, S) are stored in a state-q-indexed vector `incl_map` of collections of sets S. The sets are again represented as `OrdVectors`. On reaching a pair (q, S), all sets S' stored in `incl_map`$[q]$ are tested for inclusion with S. If $S \supseteq S'$, then S is dropped, and if $S \subseteq S'$, then S' is removed from `incl_map`$[q]$ (as well as other sets S'' such that $S \subseteq S''$) and S is added to `incl_map`$[q]$. A large speed-up is sometimes obtained by prioritizing exploration of pairs (q, S) with S being of a small size. A smaller set means a better chance to subsume other pairs, to reach a witness of non-inclusion, and to generate other pairs with small sets. The algorithm then explores a much smaller state space.

Simulation. MATA uses an implementation of a fast algorithms for computing simulation, namely, the algorithm from [65], which was adapted from Kripke structures to automata in [4], and later further optimized in [48]. The implementation originates in VATA [59].

Low-level API. The API of MATA contains an interface for accessing the most low-level features needed to implement algorithms in the style described above. For instance, the API provides iterators over transitions of Δ in the form of triples $q \xrightarrow{a} r$, iterators through *moves* (pairs (a, r) such that $q \xrightarrow{a} r \in \Delta$) of a state q, or generic *synchronized iterators*, which allow a simultaneous iteration in a set of vectors used in union and in computing the post-image in the product and subset construction. Since the main data structures are not complicated and have simple invariants, programming with them on the low level is possible even for an outsider. This low-level MATA API is, for instance, used in the string solver Z3-NOODLER. [25] presents a detailed comparison of Z3-NOODLER with the state of the art in string solving. Its exceptional performance on regex and word equation-heavy constraints is to a large degree due to MATA.

5 Infrastructure of MATA

MATA comes with the following tools and features to make using, developing, and extending it convenient.

Python interface. MATA provides an easy-to-use Python interface, making it a full-fledged automata library for Python projects. It is available on the official Python package repository[4] and can be installed easily using the `pip` package manager:

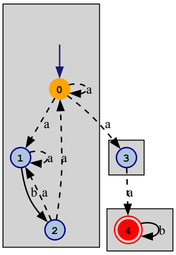
```
$ pip install libmata
```

An example of using the MATA Python binding is shown in Fig. 2. The interface is implemented using the optimizing static compiler Cython wrapping the C++ MATA calls and covers all important parts of the C++ functionality. This low-level interaction with the optimized C++ code keeps the Python code fast. To show the capabilities of the interface and to provide material for easy onboarding, MATA also contains several Jupyter notebooks with examples of how to use it.

[4] https://pypi.org/project/libmata/

```
from libmata import nfa, alphabets, parser, plotting
aut1 = parser.from_regex(’((a+b)*a)*’)
aut2 = parser.from_regex(’aab*’)
con_aut = nfa.nfa.concatenate(aut1, aut2).trim()
plotting.store()[’alphabet’] = \
    alphabets.OnTheFlyAlphabet.from_symbol_map({’a’:97, ’b’:98})
e_h = [
    (lambda aut, e: e.symbol == 98, {’color’:’black’}),
    (lambda aut, e: e.symbol == 97, {’style’:’dashed’,’color’:’black’})
]
n_h = [
    (lambda aut, q: q in aut.final_states,
        {’color’:’red’,’fillcolor’:’red’}),
    (lambda aut, q: q in aut.initial_states,
        {’color’: ’orange’, ’fillcolor’: ’orange’}),
]
plotting.plot(con_aut, with_scc=True,
              node_highlight=n_h, edge_highlight=e_h)
```

(a) An example of using MATA from Python.



(b) The output.

Fig. 2: An example of a Python interface for MATA. The code (a) loads automata from regular expressions (a, b are transition symbols; ⋆, and + represent iterations: 0 or more, and 1 or more, respectively), concatenates them, and displays the trimmed concatenation using the conditional formatting with the output in (b).

.mata *format and parsing.* MATA brings its own automata format. The main features of the format are extensibility to cover various types of automata, human-readability, yet still high level of compactness. Each .mata file consists of automata definitions. The first line of the definition describes the type of the automaton, together with the alphabet. The format supports both explicit and symbolic (bit vector) alphabets. For a symbolic alphabet, symbols are encoded as formulae over atomic propositions, where the parser of .mata implements *mintermization* (partitioning the alphabet into groups of symbols indistinguishable from the viewpoint of the input problem), which transforms it into an explicit alphabet with the symbols representing the minterms. The following lines contain a sequence of key-values statements that set particular traits of the automaton, such as initial or final states. The rest of the definition is a list of transitions. Examples of automata in .mata format are shown in Fig. 3.

```
@NFA-explicit
%Initial q0 q1
%Final q1
q0 a48 q1
q0 a52 q1
q1 a48 q1
```

(a) NFA with explicit alphabet.

```
@NFA-bits
%Initial q1
%Final q2 q1 q0
q0 ((!a0 | !a1) & a2) q2
q1 (a0 & a1 & !a2) q0
q2 ((a0 & a1) | a2) q1
```

(b) NFA with symbolic alphabet.

Fig. 3: Examples of NFAs in the .mata format.

Other than the introduced format, MATA can also parse automata from regular expressions using the parser from the regex matcher RE2 [41]. This means that MATA can handle even complex syntax used in real-world regular expressions.

Continuous integration. We implement continuous integration via GitHub Actions. In particular, actions automatically build the library including the Python binding on MacOS and Ubuntu, check for warnings, code quality and run unit tests together with the code coverage. The actions are triggered after each commit, and the checks are mandatory for merging branches to the main branch, and can also be run locally.

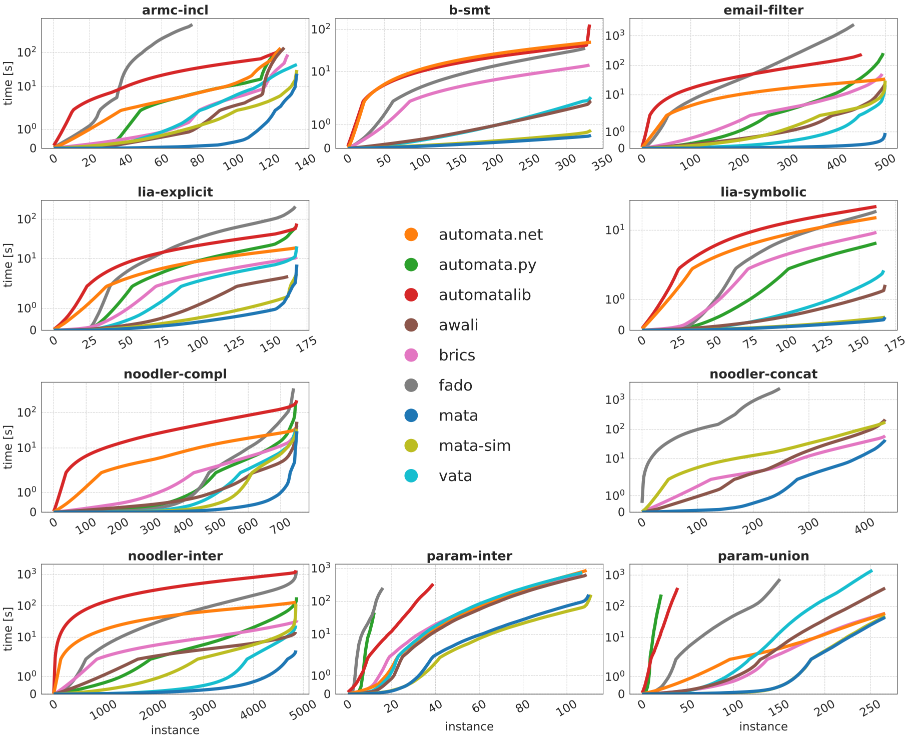


Fig. 4: Cactus plot showing cumulative run time per benchmark. The time axis is logarithmic.

6 Experimental Evaluation

We compared Mata against 7 selected libraries discussed in Section 2: Vata [59], Brics [63], Awali [60], Automata.net [77], AutomataLib [53], FAdo [7], and Automata.py [37],[5] on a benchmark of basic automata problems from string constraint solving, reasoning about regular expressions, regular model checking, and a few examples from solving arithmetic formulae. Most of the benchmark problems are taken from earlier works [38,30,40,28], but we added new problems from string constraint solving and solving quantified linear integer arithmetic (LIA).

We mainly aim to demonstrate the efficiency of the basic data structures and implementation techniques of Mata. This is best seen on standard constructions, where all libraries implement the same high-level algorithm, such as product, subset construction, or reachability test within complementation, intersection, emptiness test, etc. We then also showcase the efficiency of more advanced algorithms implemented only in Mata and Vata, the antichain-based inclusion test and simulation reduction.

[5] We also tried to compare with Mona, but using it as a standalone library that would parse automata in our format turned to be problematic. We were getting many inconsistent results and so we decided to drop it from the comparison.

Table 1: Statistics for the benchmarks. We list the number of timeouts (TO), average time on solved instances (*Avg*), median time over all instances (*Med*), and standard deviation over solved instances (*Std*), with the best values in **bold**. The times are in milliseconds unless seconds are explicitly stated. We use ~0 to denote a value close to zero.

	armc-incl (136)				**b-smt** (384)				**email-filter** (500)				**lia-explicit** (169)				**lia-symbolic** (169)			
	TO	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*
Mata	**0**	**174**	**2**	1 s	**0**	**1**	**1**	1	**0**	**1**	**~0**	9	**0**	42	**6**	356	**0**	**2**	**2**	6
Awali	7	1 s	17	3 s	**0**	6	6	4	**0**	46	4	162	6	**21**	21	16	**0**	8	7	14
Vata	**0**	324	43	577	**0**	7	7	10	**0**	42	2	322	**0**	121	51	671	1	11	10	11
Automata.net	9	1 s	125	3 s	**0**	148	153	30	**0**	69	66	30	**0**	113	117	49	6	103	107	33
Brics	5	659	34	2 s	4	43	43	19	6	103	17	280	**0**	66	62	63	6	55	60	33
AutomataLib	10	843	669	1 s	7	390	126	3 s	48	516	390	521	**0**	458	285	1 s	6	164	173	52
FAdo	58	8 s	22 s	10 s	9	109	112	67	64	6 s	1 s	11 s	1	1 s	727	2 s	6	135	149	105
Automata.py	10	913	133	3 s	334	24	TO	15	4	520	19	2 s	1	372	167	894	6	35	35	25

	noodler-compl (751)				**noodler-conc** (438)				**noodler-inter** (4872)				**param-inter** (267)				**param-union** (267)			
	TO	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*
Mata	**0**	**39**	**~0**	401	**0**	**100**	**10**	286	**0**	**~0**	**~0**	3	**156**	**1 s**	TO	4 s	**0**	**166**	**7**	326
Awali	**0**	73	2	638	**0**	490	55	1 s	6	3	1	7	157	6 s	TO	7 s	**0**	1 s	81	3 s
Vata	**0**	57	2	296			-		2	4	**~0**	22	159	7 s	TO	8 s	14	6 s	270	12 s
Automata.net	**0**	53	39	110			-		**0**	26	24	9	157	8 s	TO	10 s	**0**	220	47	314
Brics	**0**	47	8	190	**0**	136	35	204	**0**	7	3	21	159	6 s	TO	6 s	**0**	223	50	307
AutomataLib	**0**	293	143	793			-		17	276	216	675	227	8 s	TO	13 s	227	10 s	TO	15 s
FAdo	10	646	5	4 s	189	10 s	25 s	13 s	10	271	52	2 s	250	15 s	TO	20 s	115	5 s	12 s	11 s
Automata.py	3	263	5	2 s			-		5	38	3	353	254	4 s	TO	6 s	245	11 s	TO	16 s

Benchmarks. We use the following benchmark sets.

- **b-smt [38]** contains 384 instances of boolean combinations of regular properties, obtained from SMT formulae over the theory of strings. These include difficult handwritten problems containing membership in regular expressions extended with intersection and complement from [71] and emptiness problems from Norn [2,3] and SyGuS-qgen benchmarks, collected in SMT-LIB [9,67,68].
- **email-filter [38]** contains 500 inclusion checks of the form $r_5 \subseteq r_1 \wedge r_2 \wedge r_3 \wedge r_4$ obtained analogously as in [30]. Each r_i is one of the 75 regexes[6] from RegExLib [66], selected so that $r_1 \wedge r_2 \wedge r_3 \wedge r_4 \wedge r_5$ is not empty. Similar kind of these problems is solved in spam-filtering: one tests whether a new filter r_5 adds anything new to existing filters.
- **param-inter [38]** contains 4 sets of parametric intersection problems from [40] and 2 sets from [28]. In total, this includes 267 problems. The parameter controls the size of the regex or the number of regexes to be combined. **param-union** is the variant of the benchmark that performs union instead of intersection.
- **armc-incl [38]** contains 136 language inclusion problems derived from runs of an abstract regular model checker of [15] (verification of the bakery algorithm, bubble sort, and a producer-consumer system).

[6] `https://github.com/lorisdanto/symbolicautomata/blob/master/benchmarks/src/main/java/regexconverter/pattern%4075.txt`

Table 2: Relative speedup of Mata on instances where both libraries finished.

	Awali	Vata	Automata.net	Brics	AutomataLib	FAdo	Automata.py
armc-incl	27.52	1.86	29.73	16.98	21.44	4839.55	23.22
b-smt	3.7	4.52	89.64	26.13	236.36	70.16	24.47
email-filter	25.07	22.59	37.19	55.3	273.35	9999.29	282.41
lia-explicit	2.22	2.88	2.69	1.57	10.89	85.17	25.38
lia-symbolic	3.46	4.65	51.82	27.99	82.47	67.54	17.97
noodler-compl	1.85	1.45	1.37	1.22	7.44	137.53	15.58
noodler-conc	4.87	-	-	1.36	-	1979.56	-
noodler-inter	4.02	6.42	33.98	9.04	371.23	363.49	51.51
param-inter	5.36	7.3	7.27	6.49	1.43	2148.64	58.85
param-union	8.61	51.77	1.33	1.34	833.69	1618.04	5860.62

lia consists of 169 complementation problems created during the run of Amaya [8], a tool for deciding linear integer arithmetic (LIA) formulae using an automata-based decision procedure of [17]. The formulae are taken from *UltimateAutomizer* [42] and *tptp* [72] benchmarks, collected in SMT-LIB [9,69]. The transition relation in Amaya is represented symbolically using BDDs; in our experiments we tested both symbolic representation (in **lia-symbolic**) and explicit representation (in **lia-explicit**), where explicit symbols are bit vectors represented by the BDDs.

noodler consists of instances created during the run of the string solver Z3-Noodler [11,24,25] on the regex-heavy benchmark AutomatArk [10] from SMT-LIB [9,67]. We collected 751 complementation, 438 concatenation, and 4,872 intersection problems in **noodler-compl**, **noodler-conc**, and **noodler-inter** respectively.

Experimental setup. We converted all benchmarks into a common textual automata format (the `.mata` format, see Section 5), and wrote dedicated parsers or conversions for all the libraries. The conversion and parsing are not included in the run times since the parsers are not optimized and the typical use cases do not require parsing every input automaton from a textual format. From some of the benchmarks, we excluded small units of examples where the conversion failed. We measure only the time needed for carrying out the specified operations on automata already parsed into each library's internal data structures. Automata in all benchmarks but **lia** and those coming from regexes, **email-filter**, **b-smt**, **param-inter**, and **param-union**, had small or moderate alphabet sizes (all below 100 symbols, except **noodler-inter** with up to 252 symbols). The explicit automata from LIA solving (**lia-explicit**) have at most 1,024 symbols (corresponding to 10 bits).[7] After performing mintermization on automata with symbolic representation (**lia-symbolic**), the number of symbols was reduced to at most 30, and mintermization runs on automata from regular expressions returned alphabets with at most 80 symbols.

[7] It should be noted that these LIA problems are by no means representative of typical LIA formulae, which could generate much larger alphabets and transition relations that require some sort of symbolic representation.

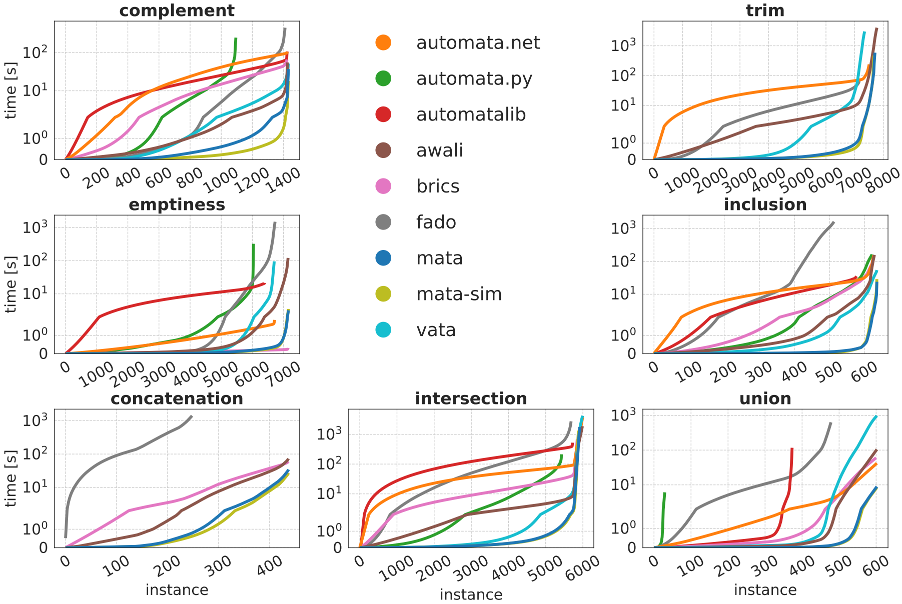


Fig. 5: Cactus plot showing cumulative run time per operation. The time axis is logarithmic.

Results. We summarize the results of each benchmark in cactus plots in Fig. 4 (displaying cumulative run times of benchmarks, with the instances ordered by their run time) and Table 1. Table 2 shows relative speedups of Mata over each library on problem instances that both libraries finished in time. We also present statistics for individual automata operations across the entire benchmark in Fig. 5 and Table 3. We do not show the performance of Mata's Python interface in the plots and tables as it is matches that one of Mata. All examples were run in six parallel jobs on Fedora GNU/Linux 38 with an Intel Core 3.4 GHz processor and 20 GiB RAM with 60 s timeout.

Mata consistently outperforms all other libraries on all benchmarks and in all operations, up to few exceptions. It is sometimes matched or outperformed by Automata.net and Brics in union and concatenation operation (on **param-union** and **noodler-conc**). Brics and Automata.net are sometimes faster since they may be able to share parts of the representation (such as BDDs on the transitions) between the automata operands and the union/concatenation, while Mata copies the entire data structure (and the memory locality of `Delta`, with its three layers of vectors, is not perfect). Brics appears particularly fast in emptiness checking since it implicitly trims the automata, after which the emptiness test becomes a trivial query on emptiness of the set of states. The cost of the emptiness check is thus hidden in the cost of other operations (we do not state statistics from trimming for Brics for this reason). Brics and Automata.net also have a smaller average time in constructing the complements in **lia-symbolic**, due to a few high run times of Mata on examples that have many transitions per a pair of states. Solving these examples, and generally examples generated from solving LIA, is indeed

Table 3: Statistics for the operations on solved instances. We list the average time (*Avg*), median time (*Med*), and standard deviation (*Std*), with the best values in **bold**. The times are in milliseconds. Note that only the operations that the given library finished within the timeout are counted, hence the numbers are significantly biased in favour of libraries that timeouted more (the harder benchmarks are no counted in), and should be red in the context of Table 1 and the cactus plots. We use ~0 to denote a value close to zero.

	complement			**concatenation**			**emptiness**			**inclusion**			**intersection**			**trim**			**union**		
	Avg	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*
Mata	**25**	**1**	315	**78**	**8**	235	**~0**	**~0**	2	**37**	**~0**	576	295	**~0**	3 s	76	**~0**	828	**14**	**~0**	45
Awali	38	2	462	166	22	402	17	**~0**	138	250	2	2 s	312	**~0**	2 s	516	**~0**	4 s	173	**~0**	527
Vata	36	3	294		-		14	**~0**	130	85	1	374	699	**~0**	4 s	408	**~0**	3 s	2 s	**~0**	5 s
Automata.net	73	59	89		-		**~0**	**~0**	~0	245	43	1 s	621	14	4 s	31	9	165	69	6	163
Brics	46	24	140	136	35	204	**~0**	**~0**	~0	204	10	1 s	115	4	1 s		-		99	2	232
AutomataLib	75	31	657		-		3	2	5	60	42	102	91	59	748		-		311	2	3 s
FAdo	320	3	2 s	6 s	10 s	10 s	223	**~0**	2 s	3 s	84	8 s	479	48	3 s	**10**	3	70	1 s	84	6 s
Automata.py	226	25	2 s		-		53	**~0**	1 s	263	6	1 s	**39**	2	479		-		203	TO	377

a case for symbolic representation of transitions, and it is currently not a primary target of Mata. However, Mata is still much faster than any other library on mintermised versions of the same examples. AutomataLib is faster in some parametric intersection examples because of its implicit determinization, which in some particular examples returns much smaller automata. When the other libraries are made to determinize, they behave analogously, and Mata again solves most examples and takes the least time. Still, on all operations except emptiness, Mata is the fastest overall, and on emptiness it is by far the fastest from libraries that actually do solve the emptiness problem. Mata has especially efficient inclusion test, and trimming, an operation which is usually needed very frequently, is also a strong point of Mata's performance.

Mata's simulation reduction (Mata-Sim in the results) does not help much when the time for computing the simulation is counted in, as seen in Fig. 4. Simulation reduction is indeed costly, and our eager strategy of reducing all automata is probably sub-optimal. The run times of complement, however, show a considerable speedup after automata are reduced, and Mata-Sim solves some complement and also parametric intersection examples that no other library can.

Overall, Mata appears significantly faster than all the libraries we have tried, with the closest competitor being often more than an order of magnitude slower.

Threats to validity. Our results must be taken with a grain of salt as the experiment contains an inherent room for error. Mainly, not knowing every library intimately, we might have missed the most optimal solutions, and our parsers of the `.mata` format might be building the internal data structures of the libraries in a sub-optimal way. The experiment was also running in parallel on a server with limited resources, which might lead to fluctuations in run times We are, however, confident that our main conclusions are well justified.

7 Conclusions and Future Work

We have introduced a new automata library Mata, explained its principles, and evaluated its performance. Mata is not the most general or feature-full library. Libraries such as Awali or Automata.net are much more complex and comprehensive, are more widely applicable, either to various symbolic representations of automata or to automata with registers, while still being impressively efficient. Mata, however, does what it is meant to do better than all the other libraries: solve examples from string solving, regular expression processing, and regular model checking much faster, while staying simple and transparent, easily extensible and applicable to projects.

We continue working on Mata's set of features as well as its efficiency. We plan to extend Mata with transducers, add support for registers that could handle, e.g., counting in regular expressions, and experiment with the poor man's symbolic representation of bit vector alphabets represented as sequences of bits (used in Lash [12]), so that Mata can be used adequately in applications such as solving WS1S and arithmetic formulae. We believe that the efficiency of the basic data structures discussed here can be much improved by focusing on the low-level performance. Custom data structures, specialised memory management, improvement in memory locality, and, generally, the class of optimizations used in BDD packages, could shift Mata's performance much further.

Acknowledgments

This work has been supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 23-07565S, and the FIT BUT internal project FIT-S-23-8151.

Data Availability Statement

An environment with the tools and data used for the experimental evaluation in the current study is available at [27].

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: Proc. of FMCAD'18. IEEE (2018)
2. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 150–166. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_10, https://doi.org/10.1007/978-3-319-08867-9_10
3. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Computer Aided Verification. pp. 462–469. Springer International Publishing, Cham (2015)

4. Abdulla, P.A., Bouajjani, A., Holík, L., Kaati, L., Vojnar, T.: Computing simulations over tree automata. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 93–108. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_8, https://doi.org/10.1007/978-3-540-78800-3_8
5. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Proc. of TACAS'10. LNCS, vol. 6015. Springer (2010)
6. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004 - Concurrency Theory. pp. 35–48. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
7. Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: Fado and guitar: Tools for automata manipulation and visualization. In: Maneth, S. (ed.) Implementation and Application of Automata. pp. 65–74. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
8. authors, A.: Amaya (2023), https://github.com/MichalHe/amaya
9. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
10. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An SMT solver for regular expressions and linear arithmetic over string length. In: Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 289–312. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_14, https://doi.org/10.1007/978-3-030-81688-9_14
11. Blahoudek, F., Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Word equations in synergy with regular constraints. In: Proc. of FM'23. Springer (2023)
12. Boigelot, B., Latour, L.: Counting the solutions of Presburger equations without enumerating them. Theoretical Computer Science **313**(1), 17–29 (2004). https://doi.org/https://doi.org/10.1016/j.tcs.2003.10.002, https://www.sciencedirect.com/science/article/pii/S0304397503005322, implementation and Application of Automata
13. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification. pp. 223–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
14. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Proc. of POPL'13. ACM (2013)
15. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Proc. of CIAA'08. Springer (2008)
16. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3114, pp. 372–386. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_29, https://doi.org/10.1007/978-3-540-27813-9_29
17. Boudet, A., Comon, H.: Diophantine equations, Presburger arithmetic and finite automata. In: Kirchner, H. (ed.) Trees in Algebra and Programming — CAAP '96. pp. 30–43. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
18. Briggs, P., Torczon, L.: An efficient representation for sparse sets. ACM Lett. Program. Lang. Syst. **2**(1–4), 59–69 (mar 1993). https://doi.org/10.1145/176454.176484, https://doi.org/10.1145/176454.176484

19. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Proc. of Symposium on Mathematical Theory of Automata (1962)
20. Büchi, J.R.: Weak Second-Order Arithmetic and Finite Automata, pp. 398–424. Springer New York, New York, NY (1990). https://doi.org/10.1007/978-1-4613-8928-6_22, https://doi.org/10.1007/978-1-4613-8928-6_22
21. Cécé, G.: Foundation for a series of efficient simulation algorithms. In: Proc. of LICS'17. IEEE (2017)
22. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proc. of POPL'18 (2018)
23. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. of POPL'19 (2019)
24. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Solving string constraints with lengths by stabilization. Proc. ACM Program. Lang. **7**(OOPSLA2) (oct 2023). https://doi.org/10.1145/3622872
25. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-noodler: An automata-based string solver. In: Proc. of TACAS'24. LNCS, Springer (2024)
26. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 76–83. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102244, https://doi.org/10.23919/FMCAD.2017.8102244
27. Chocholatý, D., Fiedor, T., Havlena, V., Holík, L., Hruška, M., Lengál, O., Síč, J.: A replication package for reproducing the results of paper "Mata: A fast and simple finite automata library" (Oct 2023). https://doi.org/10.5281/zenodo.10044515, https://doi.org/10.5281/zenodo.10044515
28. Cox, A., Leasure, J.: Model checking regular language constraints. CoRR **abs/1708.09073** (2017)
29. D'Antoni, L.: A symbolic automata library, https://github.com/lorisdanto/symbolicautomata
30. D'Antoni, L., Kincaid, Z., Wang, F.: A symbolic decision procedure for symbolic alternating finite automata. Electronic Notes in Theoretical Computer Science **336** (2018)
31. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Proc. of POPL'14. ACM (2014)
32. D'Antoni, L., Veanes, M.: Minimization of symbolic tree automata. In: Proc. of LICS'16. ACM (2016)
33. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 47–67. Springer International Publishing, Cham (2017)
34. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.F.: Alaska. In: Proc. of ATVA'08. Springer (2008)
35. Doyen, L., Raskin, J.: Antichain algorithms for finite automata. In: Proc. of TACAS'10. LNCS, Springer (2010)
36. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Gbaguidi Aisse, A., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 174–187. Springer International Publishing, Cham (2022)
37. Evans, C.: Automata (2023), https://github.com/caleb531/automata
38. Fiedor, T., Holík, L., Hruska, M., Rogalewicz, A., Síc, J., Vargovčík, P.: Reasoning about regular properties: A comparative study. In: Pientka, B., Tinelli, C. (eds.) Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy,

July 1-4, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14132, pp. 286–306. Springer (2023). https://doi.org/10.1007/978-3-031-38499-8_17, https://doi.org/10.1007/978-3-031-38499-8_17

39. Fu, C., Deng, Y., Jansen, D.N., Zhang, L.: On equivalence checking of nondeterministic finite automata. In: Proc. of SETTA'17. LNCS, Springer (2017)
40. Gange, G., Navas, J.A., Stuckey, P.J., Søndergaard, H., Schachte, P.: Unbounded model-checking with interpolation for regular language constraints. In: Proc. of TACAS'13. LNCS, Springer (2013)
41. Google: Re2. https://github.com/google/re2
42. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 36–52. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
43. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Proc. of TACAS '95. LNCS, vol. 1019. Springer (1995)
44. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. of FOCS. IEEE (1995)
45. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proc. of POPL'18 **2** (2018)
46. Holík, L., Lengál, O., Síč, J., Veanes, M., Vojnar, T.: Simulation algorithms for symbolic automata. In: Lahiri, S.K., Wang, C. (eds.) Proc. of ATVA'18. Springer (2018)
47. Holík, L., Lengál, O., Šimáček, J., Vojnar, T.: Efficient inclusion checking on explicit and semi-symbolic tree automata. In: Proc. of ATVA'11. LNCS, Springer (2011)
48. Holík, L., Šimáček, J.: Optimizing an LTS-simulation algorithm. Computing and Informatics **29**(6+), 1337–1348 (2010), https://arxiv.org/abs/2307.04235
49. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: PLDI'09. ACM (2009)
50. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Tech. rep., Stanford University, Stanford, CA, USA (1971)
51. Huffman, D.: The synthesis of sequential switching circuits. Journal of the Franklin Institute **257**(3) (1954)
52. Ilie, L., Navarro, G., Yu, S.: On NFA reductions. In: Theory Is Forever: Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday. Springer (2004)
53. Isberner, M., Howar, F., Steffen, B.: AutomataLib, https://learnlib.de/projects/automatalib/
54. Isberner, M., Howar, F., Steffen, B.: The open-source learnlib. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 487–495. Springer International Publishing, Cham (2015)
55. Kelb, P., Margaria, T., Mendler, M., Gsottberger, C.: MOSEL: A sound and efficient tool for M2L(Str). In: Grumberg, O. (ed.) Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1254, pp. 448–451. Springer (1997). https://doi.org/10.1007/3-540-63166-6_45, https://doi.org/10.1007/3-540-63166-6_45
56. Klaedtke, F.C.: Automata-based decision procedures for weak arithmetics. Ph.D. thesis, University of Freiburg, Freiburg im Breisgau, Germany (2004), http://freidok.ub.uni-freiburg.de/volltexte/1439/index.html
57. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis. pp. 543–550. Springer International Publishing, Cham (2018)

58. Legay, A.: T(O)RMC: A tool for (ω)-regular model checking. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification. pp. 548–551. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
59. Lengál, O., Šimáček, J., Vojnar, T.: VATA: A library for efficient manipulation of non-deterministic tree automata. In: Proc. of TACAS'12. LNCS, vol. 7214. Springer (2012)
60. Lombardy, S., Marsault, V., Sakarovitch, J.: Awali, a library for weighted automata and transducers (version 2.0) (2021), software available at http://vaucanson-project.org/Awali/2.0/
61. Lutterkort, D.: libfa, https://augeas.net/libfa/
62. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies. Volume 34. Princeton University Press, Princeton (1956)
63. Møller, A., et al.: Brics automata library, https://www.brics.dk/automaton/
64. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing **16**(6) (1987)
65. Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Information and Computation **208**, 1–22 (2010)
66. RegExLib.com: The Internet's first Regular Expression Library. http://regexlib.com/
67. SMT-LIB: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S (2023)
68. SMT-LIB: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA (2023)
69. SMT-LIB: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/LIA (2023)
70. Somenzi, F.: CUDD: CU decision diagram package release 3.0.0 (2015)
71. Stanford, C., Veanes, M., Bjørner, N.S.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Proc. of PLDI'21. ACM (2021)
72. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)
73. Tarjan, R.E.: Depth-first search and linear graph algorithms (working paper). In: 12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, USA, October 13-15, 1971. pp. 114–121. IEEE Computer Society (1971). https://doi.org/10.1109/SWAT.1971.10, https://doi.org/10.1109/SWAT.1971.10
74. Tozawa, A., Hagiya, M.: XML schema containment checking based on semi-implicit techniques. In: Ibarra, O.H., Dang, Z. (eds.) Implementation and Application of Automata, 8th International Conference, CIAA 2003, Santa Barbara, California, USA, July 16-18, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2759, pp. 213–225. Springer (2003). https://doi.org/10.1007/3-540-45089-0_20, https://doi.org/10.1007/3-540-45089-0_20
75. Tsay, Y.K., Chen, Y.F., Tsai, M.H., Wu, K.N., Chan, W.C.: Goal: A graphical tool for manipulating büchi automata and temporal formulae. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 466–471. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
76. Valmari, A.: Simple bisimilarity minimization in O(m log n) time. Fundamenta Informaticae **105**(3) (2010)
77. Veanes, M.: A .NET automata library, https://github.com/AutomataDotNet/Automata
78. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Proc. of ICST'10. IEEE (2010)
79. Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R.: String analysis via automata manipulation with logic circuit representation. In: Proc. of CAV'16. LNCS, vol. 9779. Springer (2016)

80. Wolper, P., Boigelot, B.: An automata-theoretic approach to Presburger arithmetic constraints (extended abstract). In: Mycroft, A. (ed.) Proc. of SAS'95. LNCS, vol. 983. Springer (1995)
81. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification. pp. 88–97. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
82. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.: Antichains: A new algorithm for checking universality of finite automata. In: Proc. of CAV'06. LNCS, vol. 4144. Springer (2006)

Software Verification

Accelerated Bounded Model Checking Using Interpolation Based Summaries

Mayank Solanki[1], Prantik Chatterjee[1](✉), Akash Lal[2], and Subhajit Roy[1]

[1] Indian Institute of Technology Kanpur, Kanpur, India
{smayank,prantik}@cse.iitk.ac.in
[2] Microsoft Research, Bangalore, India
akashl@microsoft.com, subhajit@iitk.ac.in
https://www.cse.iitk.ac.in,
https://www.microsoft.com/en-us/research/lab/microsoft-research-india

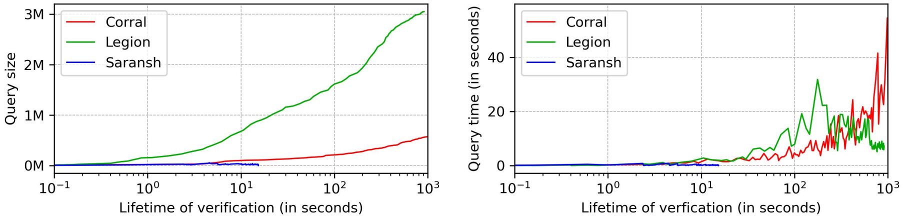
Abstract. We propose a novel lazy bounded model checking (BMC) algorithm, *Trace Inlining*, that identifies relevant behaviors of the program to compute *partial proofs* as procedural summaries. Whenever procedures are reused in other contexts, Trace Inlining attempts to construct safety proofs using these summaries. If the current summaries are sufficient to complete the proof, it gains both in solving times and smaller encodings. If the summaries are found to be insufficient, they are automatically refined for future use. The partial proofs are enabled by a sequence of alternating underapproximation and overapproximation rounds until the program verification condition is found to be unsatisfiable. We evaluate our Trace Inlining algorithm on real-world benchmarks consisting of Windows and Linux device drivers. Our results show that the proposed algorithm is able to solve 12% additional benchmarks that were unsolved by state-of-the-art lazy BMC solvers Corral and Legion. Further, Trace Inlining is 6× faster than Corral and 3× faster than Legion in terms of verification time. The virtual best of all three verifiers is 4× faster than the virtual best of Corral and Legion, implying that our technique significantly improves on what is possible today.

Keywords: Software Verification · Bounded Model Checking · Dynamic Inlining · Interpolation

1 Introduction

Bounded model checking (BMC) has remained a popular verification methodology for being able to sidestep the problem of discovering inductive invariants. A prominent framework to solving BMC instances is to reduce them to *hierarchical programs*—both bounded as well as unbounded verification problems can be reduced to reachability on hierarchical programs [20]. For bounded programs, the hierarchical program can be obtained by unrolling loops and unfolding recursive procedure to a given bound. Unbounded programs need appropriate annotations

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 155–174, 2024.
https://doi.org/10.1007/978-3-031-57249-4_8



(a) Query size v/s verification lifetime (b) Query time v/s verification lifetime

Fig. 1: Comparison of query size and SMT solver query time against the lifetime of a verification instance (x-axis is in log scale)

for the loops and procedure calls that can allow the removal of backedges in loops and recursive calls to yield a hierarchical program. Corral [19] and Legion [11] are the current state-of-the-art BMC engines that adopt this methodology. Corral, at present, drives the Static Driver verification (SDV) [4] framework within Microsoft for verification of Windows device drivers.

We propose a new algorithm, *Trace Inlining*, that leverages the modularity of software towards a new algorithm for bounded model checking of hierarchical programs. Trace Inlining identifies sub-regions of a program, and alternates between overapproximations and underapproximations, to either find a counterexample or converge to a proof of that region. Whenever a proof of safety of a region is found, it uses the Craig's Interpolation Theorem [15] to summarize this proof in the form of procedure summaries. We refer to such proofs on a subset of the program as *partial proofs*. The learnings from these partial proofs (procedure summaries) are then re-used in other regions where the same procedures appear. In certain cases, because the contexts in which these summaries are computed differ from the contexts in which they are used, the summaries may not always be enough to reach a complete proof. In such cases, *Trace Inlining* continues its search and keeps *refining* summaries to continuously make them stronger.

We develop a tool, Saransh, based on the above algorithm. We compare Saransh against the two state-of-the-art BMC verifiers, Corral [19] and Legion [11]. Legion is reported to be approximately 1.9× faster than Corral and the virtual best of Corral + Legion (running both of them in parallel and taking whichever finishes first) turns out to be 2.9× faster than Corral on bounded verification of real life Windows and Linux device driver benchmarks. Trace Inlining differs from these algorithms in multiple aspects:

- Nature of modeling. In terms of operation, Corral only performs overapproximation, while Legion only uses underapproximation. Trace Inlining, not only uses both overapproximation and underapproximation, it also uses procedure summaries as an effective modeling instrument.

- Learning. None of Corral and Legion are able to learn from procedures invoked from multiple contexts. Trace Inlining learns procedure summaries from *partial proofs* and uses them effectively in different contexts.
- VC size. The VC sizes in both Corral and Legion increase monotonically, and eventually become quite large, thereby increasing the risk of running out of memory. As Trace Inlining is able to summarize large regions of the code via relatively short summaries, the verification conditions from Trace Inlining remain significantly smaller. Also, due to repeated summarization, the size of the VCs grow non-monotonically.
- Theorem prover query times. While the VC size is an important parameter, VC size and SMT solver times are not always strongly correlated. We experimentally validate that with Trace Inlining, the SMT solver query times also remain low over the verification lifetime.

To illustrate the above points, consider Figure 1 that shows how the size of the verification condition (Figure 1a) as well as the SMT solver time (Figure 1b) evolves during the course of verification, for Corral, Legion and Saransh. (This comparison is made on a single device driver benchmark from the SDV suite [25].) Note that the verification lifetime (on x-axis) is plotted in log scale. We see that while both Corral and Legion grow the VC monotonically, the growth of the VC is non-monotonic for Saransh—in fact, it remains almost constant over the complete verification lifetime. Furthermore, notice that VC size is not correlated with solver time: although the VC size of Legion grows faster than Corral, its query times are faster than Corral. The query solving time for Saransh remains significantly lower than both the other tools. Thus, Saransh outperforms both Corral and Legion in terms of the VC size and query solving times, allowing it to scale much better.

In our empirical evaluation over a set of challenging benchmarks, Saransh solves 12% additional benchmarks that were solved by neither Corral nor Legion. Saransh reduces the PAR2 score by 36% as compared to Corral and 11% when compared to Legion. In terms of the cumulative verification time, Saransh is 6× faster than Corral and 3× faster than Legion on the benchmarks that were solved by all three of the verifiers. Even when compared to the virtual best of Corral + Legion (running both of them in parallel and taking the time of whichever one finishes first), Saransh turns out to be 1.75× faster. Further, the virtual best of Corral + Legion + Saransh is 4× faster than Corral + Legion (the current state-of-the-art as shown in [11]).

Our work is orthogonal to other work that use interpolants (like[1, 22, 23, 24]) as we leverage interpolation to accelerate BMC for refuting properties rather than performing full verification of a system. Thus, our technique does not require *inductiveness* of any invariants that it finds, it only requires constructing valid abstractions (or over-approximate summaries). Furthermore, we have instantiated our algorithm in a lazy inlining-based setting for verification of hierarchical programs that is more particular to Corral and Legion. These lazy inlining-based tools are known to perform better than static inlining based techniques (like CBMC [18] and ESBMC [17]) that compute a full inling of the whole procedure rather than use overapproximation/underapproximation to lazily inline parts of the program that are relevant to the property to be proven.

This paper makes the following contributions:

- We propose a new verification algorithm, *Trace Inlining*, that uses both overapproximation and underapproximation of the program to compute *partial proofs*, and extracts procedural summaries from them for faster and scalable bounded model checking;
- We instantiate our ideas in a tool, SARANSH;
- We conduct detailed experiments to evaluate SARANSH. Our results show that SARANSH is able to solve 12% additional benchmarks and provides a 6× speed up over CORRAL and 3× speed up over LEGION.

We have provided an artifact containing all the benchmarks and prebuilt binaries of SARANSH, CORRAL *and* LEGION, *which is available at https://zenodo.org/records/10440854.* SARANSH *is open source and the source code is available at https://github.com/mayanksolanki393/corral-traceInlining/tree/traceInterpolation.*

2 Preliminaries

2.1 Language Model

We consider a simple programming language consisting of multiple procedures, procedure calls, assignment statements (⟨var⟩ := ⟨expr⟩) and branching statements (if-then and if-then-else). A procedure call can accept multiple parameters and return multiple values. This language does not provide any loop statements, but loops can be realized using recursive procedure calls. Any variable type and expression is allowed, as long as it is supported by an SMT theory. The language additionally allows *assume*, *assert* and *havoc* statements. This language is Turing complete; programs from languages like C can be compiled to it.

2.2 Verification Problem

Given a program $\mathcal{P}$, we attempt to answer if $\mathcal{P}$ can fail, i.e., if it has any execution that can reach an *assert* φ statement where φ evaluates to false. If so, $\mathcal{P}$ is UNSAFE, else it is SAFE. Our algorithm operates by running a counterexample-guided abstraction refinement (CEGAR) loop in an underapproximate abstraction of the program. It alternates between underapproximation and overapproximation, computing procedural summaries in the process to guide verification. We describe some relevant background that is required to understand our algorithm.

Overapproximating a procedure call. A procedure callsite $r_1, r_2, \ldots r_n =$ call $proc(x_1, x_2, \ldots, x_k)$ is *overapproximated* by replacing the callsite with a *havoc* $r_1, r_2, ..., r_n$ statement. A *havoc* statement assigns non-deterministic values to its arguments.

Blocking a procedure call. A callsite, $r_1, r_2, \ldots, r_n = \text{call}\, proc(x_1, x_2, \ldots, x_k)$, is *blocked* by replacing the callsite with *assume false*, i.e., any execution in the original program that invoked the callsite is now infeasible, and hence, this transformation creates an *underapproximation* of the program.

Inlining a procedure call A callsite $r_1, r_2, ..., r_n = \text{call}\, proc(x_1, x_2, ..., x_k)$ can be *inlined* in the caller p by duplicating the complete body of callee *proc* within the procedure p, at its callsite.

Computing procedure summaries A procedure summary is a logical formula over the (formal) input-output variables of a procedure that is an over-approximation of the set of possible behaviours of the procedure. We crucially rely on Craig's interpolation theorem for computing summaries.

Theorem 1 (Craig's Interpolation Theorem [15]). *Given an Unsatisfiable logical formula of the form $A \wedge B$, there always exists a logical formula I such that $A \Rightarrow I$, $I \wedge B$ is Unsatisfiable, and I only contains variables in the intersection of the symbols of A and B.*

We show in §4 the applicability of interpolation for computing procedure summaries. Roughly, when A is a procedure and B is the context in which it is invoked, then an interpolant is an overapproximation of the procedure ($A \Rightarrow I$) that is sufficient in the current context ($I \wedge B$ is unsatisfiable) and that I only uses variables in the interface between A and B, i.e., the input and output variables.

Inlining tree In our algorithm, a procedure callsite can be in one of these four states: *Inlined*, *Blocked*, *Overapproximated*, or *Summarized*. We use an *inlining tree* data structure to maintain this information during the course of verification. The root of the inlining tree is the entry procedure of the program. A node p_L inside the tree denotes a callsite: of a procedure p at location L within its caller.

In the context of bounded model checking, given a bound d on the number of allowed recursive invocations, the inlining tree cannot grow unbounded. This is because any path in the tree can contain at most d occurrences of a procedure, and the set of procedures is naturally fixed.

We can think of a given inlining tree as defining an abstraction of a program. In fact, this abstraction can be written down as a single-procedure program that can be constructed from the entry procedure by following the status of its callsites and taking the appropriate action (e.g., inlining it or blocking it, etc.).

2.3 Notations

For a procedure `foo()`, we use $\texttt{foo}, \overset{\wedge}{\texttt{foo}}$, $\overset{\vee}{\texttt{foo}}$ and $\overset{\sim}{\texttt{foo}}$ to denote its inlined, overapproximated (havoc `foo`), underapproximated (block `foo`) and summarized state, respectively. We denote an inlining tree as a list of callsites with their respective states [3].

[3] We use the program to disambiguate the list representation of the inlining tree.

```
int main():
  int x, r, c
  assume x > 0
  havoc c
  if (c == 1)
    r := foobar(x)
  else if (c == 2)
    r := barbaz(x)
  else
    r := bazfoo(x)
  assert r > 0
```

```
int foobar(int x):
  int r, bool c1
  havoc c1
  if (c1)
C1: r := foo(x)
  else
C2: r := bar(x)
  return r
```

```
int barbaz(int x):
  int r, bool c2
  havoc c2
  if (c2)
C3: r := bar(x)
  else
C4: r := baz(true, x)
  return r
```

```
int bazfoo(int x):
  int r, bool c3
  havoc c3
  if (c3)
C5: r := baz(false, x)
  else
C6: r := foo(x)
  return r
```

```
int foo(int x):
  int r, w
  havoc w
  if (w > 0)
    r := x + w
  else
    r := x
  return r
```

```
int bar(int x):
  int r
  r := 2 * x + 1
  return r
```

```
int baz(bool op, int x):
  int r
  if (op)
    r := x + 1
  else
    r := x * 2
  return r
```

Fig. 2: Motivating Example

3 Overview

We now attempt to verify programs written in the language described in §2.1. We assume that programs may have multiple procedure calls but a well-defined unique entry procedure.

Verification Oracle. We assume the availability of a verification oracle $\mathcal{V}$. Given an inlining tree, `Tree`, a verification query $\mathcal{V}$(`Tree`) returns `Verified` if the abstraction defined by `Tree` does not lead to an assertion failure; else it returns a counterexample trace T as a sequence of instructions that lead to the assertion failure. The oracle $\mathcal{V}()$ operates by constructing a symbolic encoding of the inlining tree (referred to as the *verification condition*), and using a theorem prover to check if an assertion failure is feasible. We show how such a verification oracle can be constructed in §4.1.

Figure 2 shows our motivating example. We will reduce it to a single procedure (inlining tree) and use our verification oracle $\mathcal{V}$. For completeness, our reduction to a single procedure must ensure that each procedure appears in *every possible context* in which the procedure can possibly be invoked.

3.1 Trace Inlining (Our Proposal)

Trace Inlining starts by inlining the entry procedure, `main()`, and overapproximating the procedures called by it, i.e. `foobar()`, `barbaz()` and `bazfoo()`. The Verification Oracle($\mathcal{V}$) returns the following counterexample trace.

$$\mathcal{V}(\langle \texttt{main}, \overset{\wedge}{\texttt{foobar}}, \overset{\wedge}{\texttt{barbaz}}, \overset{\wedge}{\texttt{bazfoo}} \rangle) = \texttt{Error}, [\texttt{main}, \overset{\wedge}{\texttt{foobar}}]$$

As `foobar()` is overapproximated, this error can be a false positive. Hence, we refine the inlining tree by inlining `foobar`. This is similar to Overapproximation Refinement. However, instead of calling the verifier on this refined tree, *Trace Inlining also blocks `barbaz()` and `bazfoo()`.* This constrains the verifier to either validate or refute the current (interprocedural) abstract trace, rather than switching to a different one. The procedures called by `foobar()`, i.e., $\texttt{foo}_{C1}$ and $\texttt{bar}_{C2}$ are overapproximated. $\mathcal{V}$, now, returns the following counterexample:

$$\mathcal{V}(\langle \texttt{main}, \texttt{foobar}, \overset{\wedge}{\texttt{foo}}_{C1}, \overset{\wedge}{\texttt{bar}}_{C2}, \overset{\vee}{\texttt{barbaz}}, \overset{\vee}{\texttt{bazfoo}} \rangle) = \texttt{Error}, [\texttt{main}, \texttt{foobar}, \overset{\wedge}{\texttt{foo}}_{C1}]$$

The new counterexample is a refinement of the previous trace as all other paths in the program had been blocked. We inline the overapproximated $\texttt{foo}_{C1}\texttt{()}$ and block $\texttt{bar}_{C2}\texttt{()}$ (to deny the execution path through it), thereby forcing a decision on the current trace. $\mathcal{V}$ returns Verified for this inlining which is a *partial proof*: the verifier has proved that paths that only pass through `main`, `foobar`, and $\texttt{foo}_{C1}$ cannot cause an error. Trace Inlining now attempts to *learn* from this proof so that it can apply it in other contexts. It uses Theorem 1 to compute the summary of `foo()` (see Figure 3a; details of computing these summaries are provided in §4.1). One of the possible summaries that it could learn is `foo(x)` : ret(`foo`) $>=$ `x`. Note that, as discussed in §2.2, this summary is both *sound* as well as *sufficient* to verify the current inlining tree, i.e., given that the original inlining tree was verified, replacing the inlining of `foo` by its summary will maintain the same decision on the verifier—this is indeed the case considering that in both the execution paths of `foo()`, ret(`foo`) is always greater than or equal to x.

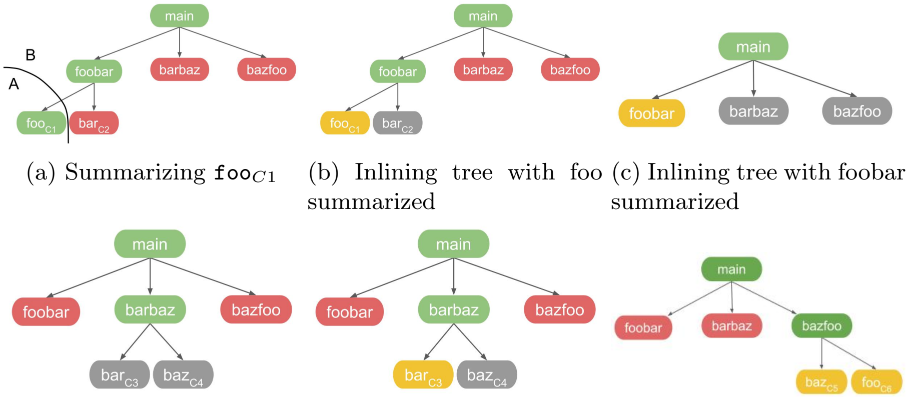
Hence, we *replace* the inlining of `foo()` with its summary. This current inlining tree, if passed to $\mathcal{V}$, will still return Verified. However, the verification is not complete, as multiple paths have been blocked.

Trace Inlining follows a last-in-first-out (LIFO) strategy[4] for unblocking procedures. Here, it unblocks and overapproximates $\texttt{bar}_{C2}$ (see Figure 3b). $\mathcal{V}$, now, returns the following counterexample:

$$\mathcal{V}(\langle \texttt{main}, \texttt{foobar}, \overset{\sim}{\texttt{foo}}_{C1}, \overset{\wedge}{\texttt{bar}}_{C2}, \overset{\vee}{\texttt{barbaz}}, \overset{\vee}{\texttt{bazfoo}} \rangle) = \texttt{Error}, [\texttt{main}, \texttt{foobar}, \overset{\wedge}{\texttt{bar}}_{C2}]$$

Trace Inlining progresses in this manner, computing the summary of $\texttt{bar}_{C2}$, and backtracking further to compute the summary of `foobar` (see Figure 3c).

[4] the LIFO strategy can exploit the stack-based interface provided by modern incremental SMT solvers



(a) Summarizing $\texttt{foo}_{C1}$ (b) Inlining tree with foo summarized (c) Inlining tree with foobar summarized

(d) Inlining tree with `barbaz()` inlined (e) Summary of `bar()` reused to summarize $\texttt{bar}_{C3}$ (f) Summary of `bazfoo` inlined

Fig. 3: Inlining trees during verification. We depict Green for *Inlined*, Red for *Blocked*, Gray for *Overapproximated*, and Yellow for *Summarized* callsites.

Here, the summary of `foobar` is a sufficient summary of the *complete inlining tree* rooted at `foobar`. This reduces the inlining tree, leading to faster theorem proving times and a lesser memory footprint. A verification query on this inlining tree may make `barbaz` appear in the counterexample trace:

$$\mathcal{V}(\langle \texttt{main}, \overset{\sim}{\texttt{foobar}}, \overset{\wedge}{\texttt{barbaz}}, \overset{\wedge}{\texttt{bazfoo}} \rangle) = \texttt{Error}[\texttt{main}, \overset{\wedge}{\texttt{barbaz}}]$$

As `barbaz()` is overapproximated, we inline it while blocking `foobar()` and `bazfoo()`. At the same time, the procedures called by `barbaz()`, i.e $\texttt{bar}_{C3}$ and $\texttt{baz}_{C4}$, are overapproximated (see Figure 3d). However, as the summary of `bar()` is already available, we assert its summary (see Figure 3e). For this inlining, $\mathcal{V}$ returns a counterexample with the trace containing $\texttt{baz}_{C4}$ (along with some other inlined callsites).

$$\mathcal{V}(\langle \texttt{main}, \overset{\vee}{\texttt{foobar}}, \texttt{barbaz}, \overset{\sim}{\texttt{bar}}, \overset{\wedge}{\texttt{baz}}, \overset{\vee}{\texttt{bazfoo}} \rangle) = \texttt{Error}[\texttt{main}, \texttt{barbaz}, \overset{\wedge}{\texttt{baz}}_{C4}]$$

Note that, if $\texttt{bar}_{C3}$ was not available, this callsite would have been in the overapproximated state. Hence, the counterexample [`main`, `barbaz`, $\texttt{bar}_{C3}$] would also be possible. However, with the summary of `bar` asserted at C3, the above counterexample is not feasible. This shows how the learnings from our *partial proof* were useful in a different context. The savings would have compounded had $\texttt{bar}_{C3}$ been an interior node as this summary would have saved the inlining of the (possibly large) sub-tree rooted at it.

The algorithm now inlines $\texttt{baz}_{C4}$ and blocks $\texttt{bar}_{C3}$:

$$\mathcal{V}(\langle \texttt{main}, \overset{\vee}{\texttt{foobar}}, \texttt{barbaz}, \overset{\vee}{\texttt{bar}}_{C3}, \texttt{baz}_{C4}, \overset{\vee}{\texttt{bazfoo}} \rangle) = \texttt{Verified}$$

This allows Trace Inlining to compute a summary for `baz`; a possible summary is $\texttt{baz(op, x)} : \texttt{op} \implies (\mathit{ret}(\texttt{baz}) >= \texttt{x} + 1)$.

The algorithm continues by backtracking to `barbaz` and computing its summary. Then, it would encounter an error trace with `bazfoo`; as a result `bazfoo` will get inlined. At this point, the summaries of both the methods it invokes, i.e. baz_{C5} and foo_{C6}, are available, and will be asserted (see Figure 3f). Now, invoking the verifier on this inlining tree produces the following result:

$$\mathcal{V}(\langle \texttt{main}, \check{\texttt{foobar}}, \check{\texttt{barbaz}}, \texttt{bazfoo}, \tilde{\texttt{baz}}, \tilde{\texttt{foo}} \rangle) = \texttt{Error}, [\texttt{main}, \texttt{bazfoo}, \tilde{\texttt{baz}}_{C5}]$$

This seems surprising as the counterexample contains no overapproximated callsite, but rather one that is summarized. This is because the summary of `baz`, *although it was sufficient in the context of where it was computed, it is not sufficient in the context where it is now applied*[5]. A closer examination of the current summary of `baz` reveals that it was invoked in `barbaz` with the first parameter as *true*, and hence, the summarization could witness only the true branch within `baz`. In the current context, the false branch of `baz` is exposed, which renders the previous summary insufficient.

Trace Inlining fixes this problem by computing a new summary for `baz` in this context; a possible summarization is:

$$\texttt{baz(op, x)} : (\neg \texttt{op} \implies (\mathit{ret}(\texttt{baz}) >= 2 * \texttt{x}))$$

However, this summary may not be sufficient in the previous context. To obtain a summary that is sufficient for *all contexts seen so far*, we conjoin this summary to the existing summary. Running $\mathcal{V}$ with the new summary gives `Verified`. Hence, now the summary of `bazfoo()` can be computed, and swapped for its inlining. Finally, the algorithm again makes progress by unblocking `foobar` and `barbaz`, and moving them to the summarized state.

$$\mathcal{V}(\langle \texttt{main}, \tilde{\texttt{foobar}}, \tilde{\texttt{barbaz}}, \tilde{\texttt{bazfoo}} \rangle) = \texttt{Verified}$$

As none of the procedures are blocked, we declare the program Safe.

4 Algorithm

4.1 Symbolic Encoding

Our verification oracle translates an input program into a *passified program* consisting of only assume statements and call statements. A passified program does not have global variables. This program is then translated to a logical formula, referred to as a *verification condition* (VC). The VC is *unsatisfiable* if and only if the program does not violate any assertion in the program (when $\mathcal{V}()$ returns `Verified`). If satisfiable, the model is a concrete execution trace of the program

[5] the soundness of the summaries in all contexts is guaranteed by Criag's Interpolation Theorem

$$
\begin{aligned}
& b_{L4} \\
\wedge\, b_{L4} &\implies (b_{L5} \wedge f(4) == 5) \\
&\quad \vee (b_{L6} \wedge f(4) == 6) \\
\wedge\, b_{L5} &\implies c_1 \wedge c_{foo} \\
&\quad \wedge (b_7 \wedge f(5) == 7) \\
\wedge\, b_{L6} &\implies \neg c_1 \wedge c_{bar} \\
&\quad \wedge (b_7 \wedge f(6) == 7) \\
\wedge\, b_{L7} &\implies ret(\texttt{foobar}) == r_{foobar}
\end{aligned}
$$

(a) pVC of procedure `foobar()` defined in Figure 2

$$
\begin{gathered}
c_{foobar} \implies pVC(foobar) \wedge \\
c_{foo} \implies pVC(foo) \wedge \\
(r_{foobar} == r_{foo}) \wedge (x_{foobar} == x_{foo}) \wedge \\
(c_{bar} == false)
\end{gathered}
$$

(b) VC of `foobar()` with `foo()` inlined and `bar()` blocked

$$
\begin{gathered}
c_{foobar} \implies pVC(foobar) \wedge \\
c_{foo} \implies r_{foo} > x_{foo} \wedge \\
(r_{foobar} == r_{foo}) \wedge (x_{foobar} == x_{foo})
\end{gathered}
$$

(c) VC of `foobar()` with `foo()` summarized

Fig. 4: Overapproximation, Inlining, Blocking and Summarization in VC

that ends in a violation of some assertion. [19] provides detailed description on building such verification conditions.

Let $pVC(p)$ refer to the *partial* verification condition for pertaining to only a method p. To handle multi-procedure programs, $\mathcal{V}$ should be able to perform *overapproximation*, *inlining*, *blocking* and *summarization* for procedure calls. We discuss how this can be done with the *Verfication Condition* below.

- Overapproximating a call-site. For a procedure p that calls, say, a procedure q, the call is overapproximated by default in $pVC(p)$ because the corresponding return variables of the call to q are left unconstrained in $pVC(p)$.
- Inlining a procedure p is achieved by renaming all symbols in $pVC(p)$ to fresh symbols, say `renamed(pVC(p))`, and conjoining ($c_i \implies$ `renamed(pVC(p))`) to the verification condition of the caller, where c_i is the call-site variable of the call to p. We add new constraints that bind the formal and actual parameters to have the same value; the return values of the procedure are also bound to appropriate variables in the caller. An example of inlining is shown in Figure 4b, where the call to `foo()` from `foobar()` has been inlined.
- Blocking of a procedure call is achieved by setting the respective call-site variable to false, because it disallows any feasible execution to invoke the respective call. An example of blocking is also shown in Figure 4b where the call to $\texttt{bar}_{C2}$`()` from `foobar()` has been blocked.
- Summarizing a call-site is similar to inlining, except that instead of using the partial VC of the callee, its summary is instantiated and asserted instead: ($c_i \implies$ `instantiated(Summary(p))`). For a summary $foo(x_1, \ldots, x_n)$: $\varphi(x_1, \ldots, x_n, ret(foo))$, `instantiated()` adds equality constraints between the formal arguments $x_1, \ldots, x_n$ and the actual arguments in the VC of the caller, assigns $ret(foo)$ to the respective variable in VC of the caller, and

renames all the other symbols in $VC(foo)$ to fresh symbols. For example, Figure 4c shows the summary of foo() i.e. $foo(x) : ret(foo) > x$ asserted for call-site $\texttt{foo}_{C1}$.

4.2 The Trace Inlining Algorithm

As we have seen before, in an inlining Tree, every call-site is in one of four possible states: *overapproximated*, *inlined*, *blocked* or *summarized.* We define a few helper functions to explain our algorithm.

- Tree.inline(callsite) transitions a callsite in the Tree from *overapproximated* or *summarized* state to inlined state.
- Tree.block(callsite, track=true) transitions a callsite in the Tree from *overapproximated* or *summarized* state to *blocked* state. When track is set to $false$ the Tree doesnot track the callsite as a blocked callsite.
- Tree.overApproximate(callsite) transitions a callsite in the Tree from *inlined* or *blocked* state to *overapproximated* state.
- Tree.summarize(callsite, SummDB) transitions a callsite in the Tree from *overapproximated* state to *summarized* state. Note that summaries can get updated over time. Therefore, one additional responsibility of this method is to replace any stale summary of callsite in Tree with the latest version available in the summary database SummDB.
- Tree.computeSummary(callsite) returns the summary of callsite.
- Tree.getOpen() returns the set of callsites in *overapproximated* or *summarized* state in Tree.
- Tree.getInlined() returns the set of callsites in *inlined* state in Tree.
- Tree.getBlocked() returns the set of callsites in *blocked* state in Tree. This method would not return the callsites that were blocked with track set to $false$.
- procedureName(callsite) returns the name of the procedure corresponding to callsite.
- recursionDepth(callsite) returns the recursion depth of callsite.
- computeSummaries(Tree, SummarizableCallsites, SummDB) computes the summary of each callsites in SummarizableCallsites set and stores it in SummDB. If the summary of a procedure is already present, then it is conjuncted with the newly computed summary.

Algorithm 1 is the Trace Inlining algorithm. It takes as input a program $\mathcal{P}$ with the entry procedure main and a Verification Oracle $\mathcal{V}$. The algorithm starts by initializing multiple variables:

- Tree, initialized at Line 1, represents an inlining tree. It is initialized by inlining main. Note that all the procedure callsites in main would be in the overapproximated state in Tree, as discussed in §4.1.
- SummDB is the summary map, initialized to be empty in Line 2. Note that summaries are stored against the name of a procedure, not against a particular callsite. This allows procedure summaries computed in one part of the inlining tree to be reused in other parts of the tree.

Algorithm 1: Trace Inlining Algorithm

```
   Input  : Program $\mathcal{P}$ with starting procedure main, Verification Oracle $\mathcal{V}$,
            Max Recursion Depth RecursionLimit
   Output: Safe, or Unsafe(with trace)
 1 Tree ← ⟨ main ⟩                                              // Inlining tree
 2 SummDB ← { }                              // Map from procedure name to summary
 3 Context ← [ ]                           // Stack to track the state of callsites
 4 HitRecursionLimit ← $false$
 5 while $true$ do
 6     outcome, $T$ ← $\mathcal{V}$ (Tree)
 7     if outcome == Verified then
 8         if Tree.getBlocked() == ∅ then
 9             if HitRecursionLimit then
10                 return Safe (Within recursion bounds)
11             else
12                 return Safe
13         LastInlined, LastBlocked ← Context.pop()
14         computeSummaries(Tree, LastInlined, SummDB)
15         foreach callsite $in$ LastInlined ∪ LastBlocked do
16             Tree.overApproximate(callsite)
17     else
18         // Are all the callsites in $T$ inlined?
19         if $T \setminus$ Tree.getInlined() == ∅ then
20             return Unsafe, $T$
21         OpenCallsites ← Tree.getOpen()
22         foreach callsite ∈ OpenCallsites do
23             if recursionDepth(callsite) > RecursionLimit then
24                 HitRecursionLimit ← $true$
25                 Tree.block(callsite, $false$)
26             if callsite ∈ $T$ then
27                 Tree.inline(callsite)
28             else
29                 Tree.block (callsite)
30         Context.push (⟨ $T$, OpenCallsites $\setminus$ $T$ ⟩)
31     foreach callsite ∈ Tree.getOpen() do
32         Tree.summarize(callsite, SummDB)
```

- Context, initialized at Line 3, is a stack that keeps track of the callsites that were inlined or blocked in prior iterations. Each entry in Context is a tuple of the sets of inlined and blocked callsites.
- HitRecursionLimit initialized at Line 4, is a boolean variable to track if the algorithm has hit the recursion limit.

After initialization, the first iteration of the algorithm starts with a call to $\mathcal{V}$ (Line 6). If $\mathcal{V}$ returns a counterexample, at Line 19 we check if the counterexample only consists of *inlined* callsites. If so, we return UNSAFE along with the counterexample. Otherwise, all the `overapproximated/summarized` callsites in the counterexample are inlined and others are blocked (Lines 21 to 29). Also, the inlined and the blocked callsites are pushed on the `Context` stack (Line 30). At this point if the algorithm find any callsites that have crossed the recursion depth, those callsites are blocked at Line 25. Also, the `HitRecursionLimit` is set to *true* indicating that we have hit the recursion bound during verification.

If $\mathcal{V}$ returns *Verified*, we check if there are any blocked callsites in `Tree` (Line 8) (The callsites blocked due recursion depth are not considered). If not, then we return SAFE or SAFE (WITHIN RECURSION BOUND) depending on the value of `HitRecursionLimit`. Otherwise, we pop `Context` to get the callsites that were last inlined/blocked (Line 13). Next, we invoke *computeSummaries* (Line 14) to update `summDB` with the summaries of all inlined callsites of the last iteration.

After summarization, the inlined and blocked callsites are again set to the overapproximated state in `Tree` (Line 16).

The algorithm makes progress on each iteration of the loop at Lines 31 to 32. For an iteration where $\mathcal{V}$ returns a counterexample, this results in inlining of callsites on the counterexample (Line 27) and for the iteration where $\mathcal{V}$ returns *Verified*, this results in the summarization of callsites for which a summary was computed/updated in the last iteration.

4.3 Summaries of recursive procedures

For recursive programs, an inlining based algorithm may never terminate as possible callsites in the program is infinite. BMC handles this by limiting the inlining of recursive callsites up to a fixed (user-defined) depth D. Any callsite with a depth more than D is considered to be blocked. Hence, a verified verdict by our algorithm is for a bounded version of the original program (unless there are no recursive procedures in the program). Recursion creates one additional complication for our algorithm. The summary computed for a recursive procedure at depth d_i is not an overapproximation of the procedure at depth d_j when $j < i$. To see this, consider a procedure `f` that calls itself. A call to `f` at depth D is infeasible. A call to `f` at depth $D - 1$ is to a version of `f` that cannot call itself, etc. Thus, the higher the depth at which `f` is called, the fewer are its set of behaviors. Hence, we create separate procedure summaries for different depths; a summary computed for a procedure at a certain depth can only be used at a callsite at the same depth.

4.4 Proof of Soundness

Soundness of Trace Inlining algorithm follows when a summary is asserted for an inlined call-tree. Given an inlining $I = \langle proc_1, ..., proc_i, ..., proc_n \rangle$, if $proc_i$ is replaced by its summary to create $I' = \langle proc_1, ..., summary(proc_i), ..., proc_n \rangle$, then, verification of $\mathcal{V}(I')$ must imply verification of $\mathcal{V}(I)$ (Craig Interpolation theorem). Further, as a new summary is computed by conjoining the newly generated interpolant to the previous summary, the summaries computed for a procedure grow monotonically stronger and remain sufficient to prove summaries of all the contexts witnessed so far.

5 Experiments

We implement the *Trace Inlining* algorithm in our tool, SARANSH, and evaluate it on challenging Windows and Linux device driver benchmarks. The Windows device drivers are available as the SDV benchmark suite from Microsoft [25]. These benchmarks exercise all features of the C language such as loops and recursion (up to a bounded depth), pointers, arrays etc. SDV compiles the C source code of a device driver to Boogie [8] and then instruments it to add assertions that check a driver property[6].

We also use Linux device drivers benchmarks available as part of the SV-COMP [6] benchmark suite. We use SMACK [27] to compile these drivers to Boogie and instrument the properties to be verified.

We compare SARANSH against the CEGAR based model checker CORRAL [19] and the proof-guided model checker LEGION [11]. Both of them perform verification by bounded model checking over hierarchical programs via dynamic inlining [20]. Tools that use static inlining (such as CBMC [18]) do not scale well [20]. CORRAL and LEGION have been extensively tuned for verifying device drivers. Therefore, these tools, on the device driver benchmarks, were a strong baseline for our work. We used Z3v4.5 [16] as the underlying SMT solver for all of the verifiers. We extract summaries by computing interpolants using the Z3 solver (summary extraction is described in [13]). We used the default setting of a fixed random seed for Z3 after verifying that the choice of the random seed does not have any statistically significant impact on the results reported in this paper. For empirical evaluation, we only select hard benchmarks on which CORRAL requires more than 200 seconds to solve (similar setting is used in [11, 12]). For each verification task, we set a time budget of 2 hours and a recursion bound of 3 for each tool [11]. Each verification instance was given access to a single core and 32 GB of RAM.

5.1 SARANSH vs CORRAL vs LEGION

We compare SARANSH against CORRAL and LEGION on the number of instances they can solve and the total time taken to solve those instances. We also report the *PAR2 score* that is used in software verification competitions to rank verifiers. The PAR2 score is defined as follows

$$\text{PAR2Score(tool)} = \sum_{i=1}^{n} \begin{cases} \mathit{VerifTime}_i & \text{if the i-th benchmark is verified} \\ 2 * \mathit{timeout} & \text{otherwise} \end{cases}$$

[6] A reference manual to the Boogie language is available at https://boogie-docs.readthedocs.io/en/latest/LangRef.html.

Table 1: Summary of the comparison between tools across all benchmarks

Tool	Time (hours)	#Solved	PAR-2 (hours)
Corral	143	342	2159.84
Legion	152	492	1568.19
Saransh	54	512	1390.5
Corral+Saransh	77	567	1193.51
Corral+Legion	136	527	1412.93
Saransh+Legion	86	619	994.96
Corral+Legion+Saransh	75	630	939.68

where n is the total number of benchmarks, $VerifTime_i$ denotes the time taken by a tool to verify the i-th benchmark and *timeout* is the time budget allotted for verifying each benchmark. A lower PAR2 score is indicative of a better tool.

Table 1 reports the number of benchmarks solved within the allotted time of 2 hours by each verifier. Out of 846 benchmarks, Corral solves 342 benchmarks in 143 hours and Legion solves 492 benchmarks in 152 hours, whereas, Saransh solves 512 benchmarks in only 54 hours. *Hence,* Saransh *solves more benchmarks than any of the tools in lesser time.* Not surprisingly, Saransh has the lowest PAR2 score among the three verifiers. Saransh reduces the PAR2 score by 36% as compared to Corral and by 11% when compared to Legion. Out of the 512 benchmarks solved by Saransh, 214 (25%) were unsafe, 55 (6%) were safe and 243 (28%) were safe within the recursion bound. We did not observe any correlation between performance and safe/unsafe.

Speed up Figure 5 presents the speed up achieved by Saransh over Corral and Legion on the benchmarks which were solved all three of the verifiers. There were 263 such instances. Corral took 98 hours to solve these 263 instances and Legion required 51 hours, whereas Saransh needed only 16 hours, i.e., Saransh is more than 6× faster than Corral and 3× faster than Legion.

Lazy inlining techniques such as Corral and Legion monotonically grow the VC by incrementally adding more regions of relevant code, until either a proof or a counterexample is found. In contrast, Saransh explores individual execution traces by blocking all traces which are disjoint to the current trace. Once a trace has been proven, summaries are learned and reused while proving other traces. Modular programs, i.e., where the same functions are called multiple times across paths, the opportunity to reuse summaries is high and therefore Saransh outperforms Corral and Legion. In contrast, programs where the same functions are not called often in other paths, Saransh incurs the cost of computing the summaries but the opportunity to reuse them is low. Therefore, in these instances, Corral and Legion outperforms Saransh.

5.2 Virtual Best of Saransh, Corral and Legion

From the experiments, we observe that Saransh demonstrates a complementary behavior to both Corral and Legion, i.e., Saransh can easily solve multiple

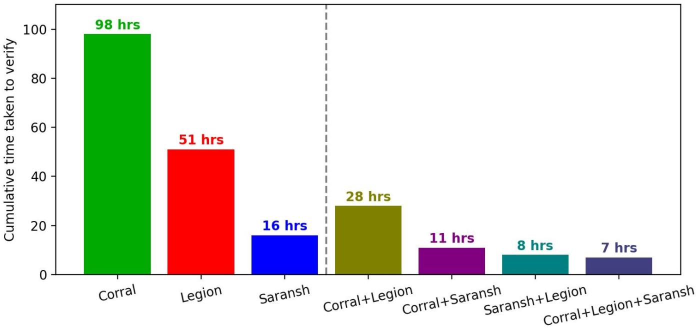


Fig. 5: Cumulative time taken by each verifier to solve benchmarks.

benchmarks that are hard for Corral and Legion and vice versa as well. This complementary nature inspires us to consider the virtual best of Saransh, Corral and Legion in order to reap the benefits from all three of the verifiers. A virtual best of multiple verifiers involves running the verifiers in parallel and picking the outcome of whichever verifier finishes the verification task first. We denote the virtual best of a combination of verifiers with a '+' symbol. For example, we denote the virtual best of Corral, Legion and Saransh as Corral + Legion + Saransh.

Table 1 depicts that Corral + Legion solves 527 benchmarks, whereas Corral + Saransh solves 567 benchmarks and Legion + Saransh solves 619 benchmarks. Finally, the virtual best of Corral + Legion + Saransh solves 630 benchmarks in total. Hence, introducing Saransh in the verifier combination results in verification of an additional 103 benchmarks as compared to the state-of-the-art Corral + Legion strategy [11].

Table 1 further demonstrates the cumulative time taken to solve benchmarks by each of the virtual best combinations. The virtual best of Corral + Legion solves 527 benchmarks in 136 hours, whereas Corral + Saransh solves 567 benchmarks in 77 hours and Legion + Saransh solves 619 benchmarks in 86 hours. Finally, the virtual best of all three verifiers Corral + Legion + Saransh takes only 75 hours to solve 630 benchmarks. Corral + Legion + Saransh also has the lowest PAR2 score out of all the combinations.

Speed up Figure 5 presents the cumulative time taken by each of the virtual best verifiers to solve the 263 benchmarks which were solved by all the verifiers. Corral + Legion, the current state-of-the-art [11], took 28 hours to solve these benchmarks, whereas Corral + Saransh required 11 hours and Saransh + Legion required 8 hours for the same. Finally, the virtual best of all three verifiers Corral + Legion + Saransh required only 7 hours to solve these 263 benchmarks, i.e., introducing Saransh in the virtual best resulted in a 4× speed up over Corral + Legion. Note that, Saransh *by itself solved these* *benchmarks in only* 16 *hours and it is nearly* 1.75× *faster than the state-of-the-art virtual best* CORRAL + LEGION.

The results conclusively demonstrate the effectiveness of SARANSH in verifying real-world code.

6 Related Work

BMC [14] is a popular technique owing to its ability of finding property violations within a user defined bound. It is primarily applied for refuting properties as opposed to proving them. In [22, 23], the authors show that Craig interpolants can be combined with SAT-based BMC to derive the termination condition for symbolic unbounded model checking. [24] extends this idea to verify infinite state sequential programs. Li et al. [21] further improves this idea by combining abstraction refinement and interpolation guided unbounded model checking. Alberti et al. [2, 3] extend the idea of lazy abstraction with interpolation to programs containing arrays of unknown length. Vizel et al. [30] propose an algorithm that imitates BDD-based symbolic model checking by computing a sequence of interpolants for performing full verification. Cabodi et al. [9] revisits the idea of utilizing interpolation sequences for full verification by computing a chain of interpolants which provides a tighter integration with the abstraction refinement strategy. UFO [1] utilizes a combination of abstract interpretation and interpolation for program verification: starting the verification with abstract interpretation, they refine the counterexamples by using interpolants.

Our work is orthogonal to the work discussed above as we leverage interpolation to accelerate BMC for refuting properties rather than performing full system verification via a search for inductive invariants.

Caniart et al. [10] used interpolants to accelerate model checking strategies at the refinement stage to rule out multiple spurious counterexamples at once. However, this technique can only be applied to lazy interpolant guided model checking techniques rather than any CEGAR technique. Sery et al. [28, 29] leverages interpolants to compute function summaries instead of using complete function bodies to ease the burden on BMC techniques. The function summaries are computed after a successful verification run and can be used subsequently. Compared to this, we compute function summaries dynamically and use it in the same verification run itself. Beyer et al. [7] combines abstraction, CEGAR and interpolation based model checking techniques to perform explicit value analysis on program variables as opposed to our strategy which operates on abstraction of callsites and leverages interpolants to derive function summaries to be used in callsite refinements at each CEGAR step. Pick et al. [26] designed an approach to leverage function summaries in presence of mutual recursion. Interpolants have also been used in other settings: Bavishi et al. [5] attempt localization and repair of program faults by leveraging interpolants derived from passing test cases. Chatterjee et al. [12] take an orthogonal direction to scaling BMC by employing a distributed strategy to stratified inlining that leverages proofs of unsatisfiability as a heuristic.

Bibliography

[1] Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction-and interpolation-based software verification. In: Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24. pp. 672–678. Springer (2012)

[2] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Safari: Smt-based abstraction for arrays with interpolants. In: Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24. pp. 679–685. Springer (2012)

[3] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: An extension of lazy abstraction with interpolation for programs with arrays. Formal Methods in System Design **45**, 63–109 (2014)

[4] Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In: Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004. Proceedings 4. pp. 1–20. Springer (2004)

[5] Bavishi, R., Pandey, A., Roy, S.: To be precise: regression aware debugging. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA. ACM (2016)

[6] Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III (2019)

[7] Beyer, D., Löwe, S.: Explicit-state software model checking based on cegar and interpolation. In: Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 16. pp. 146–162. Springer (2013)

[8] Boogie: An intermediate verification language, `https://boogie-docs.readthedocs.io/en/latest/` (accessed on June 2022).

[9] Cabodi, G., Nocco, S., Quer, S.: Interpolation sequences revisited. In: 2011 Design, Automation & Test in Europe. pp. 1–6. IEEE (2011)

[10] Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. 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. 428–442. Springer (2008)

[11] Chatterjee, P., Meda, J., Lal, A., Roy, S.: Proof-guided underapproximation widening for bounded model checking. In: Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I. pp. 304–324. Springer (2022)

[12] Chatterjee, P., Roy, S., Diep, B.P., Lal, A.: Distributed bounded model checking. In: FMCAD. pp. 47–56 (2020)

[13] Chockler, H., Ivrii, A., Matsliah, A.: Computing interpolants without proofs. In: Hardware and Software: Verification and Testing: 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers 8. pp. 72–85. Springer (2013)

[14] Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal methods in system design **19**, 7–34 (2001)

[15] Craig, W.: Linear reasoning. a new form of the herbrand-gentzen theorem. The Journal of Symbolic Logic **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593

[16] 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)

[17] Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: Esbmc 5.0: an industrial-strength c model checker. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 888–891 (2018)

[18] Kroening, D., Tautschnig, M.: Cbmc–c bounded model checker: (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20. pp. 389–391. Springer (2014)

[19] Lal, A., Qadeer, S.: Powering the static driver verifier using Corral. In: FSE (2014)

[20] Lal, A., Qadeer, S.: Dag inlining: a decision procedure for reachability-modulo-theories in hierarchical programs. ACM SIGPLAN Notices **50**(6), 280–290 (2015)

[21] Li, B., Somenzi, F.: Efficient abstraction refinement in interpolation-based unbounded model checking. In: Tools and Algorithms for the Construction and Analysis of Systems: 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-April 2, 2006. Proceedings 12. pp. 227–241. Springer (2006)

[22] McMillan, K.L.: Interpolation and sat-based model checking. In: Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15. pp. 1–13. Springer (2003)

[23] McMillan, K.L.: Applications of craig interpolants in model checking. In: Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 11. pp. 1–12. Springer (2005)

[24] McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18. pp. 123–136. Springer (2006)

[25] Microsoft: Static Driver Verifier Benchmarks, `https://github.com/boogie-org/sdvbench`

[26] Pick, L., Fedyukovich, G., Gupta, A.: Unbounded procedure summaries from bounded environments. In: Verification, Model Checking, and Abstract Interpretation: 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17–19, 2021, Proceedings 22. pp. 291–324. Springer (2021)

[27] Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: CAV (2014)

[28] Sery, O., Fedyukovich, G., Sharygina, N.: Funfrog: Bounded model checking with interpolation-based function summarization. In: Automated Technology for Verification and Analysis: 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings 10. pp. 203–207. Springer (2012)

[29] Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: Hardware and Software: Verification and Testing: 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers 7. pp. 160–175. Springer (2012)

[30] Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: 2009 Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (2009)

Weakest Precondition Inference for Non-Deterministic Linear Array Programs

Sumanth Prabhu S[1,2](✉), Deepak D'Souza[2](✉), Supratik Chakraborty[3](✉), R Venkatesh[1](✉), and Grigory Fedyukovich[4](✉)

[1] Tata Consultancy Services Research, Pune, India
r.venky@tcs.com
[2] Indian Institute of Science, Bengaluru, India
sumanth.prabhu@tcs.com, deepakd@iisc.ac.in
[3] Indian Institute of Technology, Bombay, India
supratik@cse.iitb.ac.in
[4] Florida State University, Tallahassee, USA
grigory@cs.fsu.edu

Abstract. *Precondition inference* is an important problem with many applications. Existing precondition inference techniques for programs with arrays have limited ability to find and prove the weakest preconditions, especially when programs have non-determinism. In this paper, we propose an approach to overcome the limitation. As the problem is uncomputable in general, our approach targets a special class of programs called linear array programs that are commonly encountered in practical applications and have been studied before. We also focus on a class of quantified formulas for pre- and postconditions that suffice to specify program properties in many applications. Our approach uses two novel techniques called *Structural Array Abduction* (SAA) and *Specialized Maximality Checking* (SMC). SAA is an abduction-based technique used to infer quantified preconditions and necessary inductive invariants. SMC proves that an inferred precondition is the weakest by finding an under-approximated program and solving the complement verification problem on it using SAA. When inconclusive, it attempts to weaken the precondition. Our approach can infer (and also prove) the weakest preconditions for a range of benchmarks relatively quickly, and outperforms competing techniques.

1 Introduction

Precondition inference is concerned with finding a set of initial states from which all terminating executions of a given program reach states satisfying a given postcondition. The *weakest* precondition refers to the largest such set of initial states. The weakest precondition can be used as a contract on a library function's input, for run-time argument value checks, as a summary in compositional verification, and in many more applications [2,11,12,24,46,47,52,53].

Finding the weakest precondition, especially in the presence of unbounded loops and data structures like arrays, is challenging and uncomputable in general. To show that a precondition is valid requires reasoning about all possible

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 175–195, 2024.
https://doi.org/10.1007/978-3-031-57249-4_9

executions of loops. Almost always this necessitates the inference of adequate *inductive invariants*. However, automatic invariant inference is an equally difficult problem, and in the case of array programs, the required invariants are often quantified formulas, adding to the difficulty of reasoning about them. Moreover, existing invariant inference techniques [22, 28, 30, 32, 41] rely on a precondition being provided by the user. This makes it difficult to use such techniques directly in our problem setting, where preconditions are not available to begin with.

Even if we are able to find a precondition for a given program and postcondition, proving that the precondition is the weakest presents significant technical challenges. Specifically, we need to prove that adding any new state to the set of initial states represented by the precondition results in an execution that terminates in a state violating the postcondition. To find such a proof, existing quantified precondition inference techniques assume the program to be deterministic, i.e., from every initial state, there is a unique program execution [49, 53]. However, it often becomes necessary to use non-deterministic features when modeling programs, thereby admitting multiple possible executions starting from the same initial state. Such non-deterministic features may be needed to model user input, non-deterministic functions, external functions, or when programs are abstracted. Hence, assuming that all programs are deterministic significantly restricts the applicability of existing techniques for finding weakest preconditions.

We propose a novel technique for inferring weakest preconditions for a class of terminating non-deterministic programs that manipulate arrays, with respect to postconditions expressed in a rich language of formulas. Specifically, we target the class of *linear array programs*, defined formally in Section 3. This includes programs used in many practical applications, and the literature describes several verification techniques for this class of programs [7, 8, 40]. However, existing techniques for weakest precondition inference either apply to deterministic linear array programs, or deal with non-determinism in simpler classes of programs. Our work fills this gap, making it possible to infer weakest preconditions for linear array programs with non-determinism.

The proposed technique works in the *infer-check-weaken* framework [1,27,49, 50, 54]. It first infers a precondition along with adequate inductive invariants. A maximality check follows to see whether the precondition is weakest. If the check yields a negative answer, the precondition is weakened. This loop continues until the weakest precondition is found. In this framework, our core contributions are Structural Array Abduction (SAA) for inferring preconditions and associated invariants, and Specialized Maximality Checking (SMC) for proving that the inferred precondition is maximal (or weakest).

At a high level, SAA "guesses" candidate preconditions and inductive invariants as (quantified) formulas, and checks their correctness using an SMT solver. Since quantified formulas over arrays are challenging to reason about even with state-of-the-art SMT solvers, the guessing has to be done carefully. SAA uses abductive inference for this purpose. First, it constructs an abduction query to find what property of array elements at the start of a loop iteration will result in a desired property after the iteration. The array property thus inferred is then combined with a *range formula* [22], which is a predicate representing the boundary between indices of the array that are processed and those that are yet to be processed. A set of rules guide the construction of appropriate abduction queries and range formulas.

Though SAA is effective in finding weak preconditions, it is not guaranteed to find the weakest precondition. SMC is used to check whether a precondition is indeed the weakest. This amounts to determining whether for every initial state that violates the precondition, there is a terminating execution that results in a state violating the postcondition. To accomplish this, SMC uses the insight that every execution of a non-deterministic program is also an execution of an *under-approximation* of the original program obtained by suitably restricting the non-determinism in control flows (i.e., `if` statements). Specifically, the existence of inductive invariants for *the complement verification problem*, i.e., under-approximated program with complemented pre-and postconditions, proves that the inferred precondition is indeed the weakest for the given (terminating) program and postcondition. SMC uses SAA to find an under-approximated program and its inductive invariants. When SAA fails, SMC weakens the precondition from a set of candidates obtained in a syntax-guided way, like in [22].

Our technique is implemented in a tool called MaxPrANQ. It takes constrained Horn clauses (CHCs) as input, which is a convenient way to model and reason about programs symbolically (details in Sec 3.2). On a challenging set of 66 precondition inference tasks, our tool inferred the weakest precondition for all 66 and automatically proved 59 of them to be the weakest. In comparison, the state-of-the-art tool PreQSyn [49] could only solve 2/66 benchmarks, and P-Gen [53] did not find a precondition for any of them. To further gauge the difficulty level of reasoning about our benchmarks, we tried using two state-of-the-art inductive invariant inference tools, FreqHorn [22] and Spacer [30], to simply prove the correctness of the preconditions inferred by MaxPrANQ. Neither FreqHorn nor Spacer could however complete the task for the entire set of 66 benchmarks in the given time. This shows that even proving the correctness of the weakest preconditions was difficult for our benchmarks, let alone inferring the preconditions automatically.

The primary contributions of our paper are:

1. SAA: a method for finding preconditions, inductive invariants, and stronger guard conditions for non-deterministic linear array programs.
2. SMC: a method for checking if a precondition is the weakest and, when inconclusive, weakening it.
3. MaxPrANQ: a tool for finding the weakest preconditions, with witnesses of validity (inductive invariants) and maximality.

The rest of the paper has following sections: Sect 2 has a running example, Sect 3 provides necessary background, Sect 4 gives an overview of our algorithm, SAA and SMC descriptions are in Sect 5 and Sect 6, resp., Sect 7 gives evaluation details, Sect 8 has related work, and limitations and future work are in Sect 9.

```
int N = nondet(); // N ≥ 0
int A[N], B[N], C[N];
// pre(N, A, B, C);
for (int i = 0; i < N; i++)
  if (nondet()) C[i] = i;
  else A[i] = C[i];
assert(∀j.0≤j<N ⟹ A[j]=B[j]);
```

(a) Program

```
int N = nondet();
int A[N], B[N], C[N];
// ¬pre
assume(∃j.0≤j<N ∧ (A[j]≠B[j] ∨ B[j]≠C[j]));
for (int i = 0; i < N; i++)
  if (A[i]≠B[i]) C[i] = i; //new guard
  else A[i] = C[i];
assert(∃j.0≤j<N ∧ A[j]≠B[j]);//¬post
```

(b) Maximality Proof

Fig. 1: A non-deterministic array program and its maximality proof.

2 A Running Example

Fig. 1a shows a non-deterministic program with a postcondition that requires a universally quantified weakest precondition. The program has three arrays: A, B, and C, each of parametric size N. For each array index i, the program chooses non-deterministically whether to write i to the i-th element of C or copy the i-th element of C into the corresponding index of A. The postcondition, as stated in the `assert`, requires that the arrays A and B have the same content. Our goal is to infer the weakest precondition (denoted by `pre`) over A, B, C, and N under which the program satisfies the postcondition.

Existing weakest precondition inference techniques [49, 53] diverge for non-deterministic programs like the one in Fig. 1a. For instance, P-Gen [53] fails to find a precondition, and PreQSyn [49] fails to prove that the precondition it finds is the weakest in 200 seconds. This is because they either fail to generalise a set of initial states to a quantified precondition or, when they do, they cannot prove it to be the weakest for non-deterministic programs. In contrast, SAA finds the precondition: $\forall j.\, 0 \leq j < N \implies (A[j] = B[j] \wedge B[j] = C[j])$ (details in Sect 5.3), and SMC proves this to be the weakest precondition, all within a few seconds.

To prove maximality, SMC finds an under-approximated program, as shown in Fig. 1b. In this program, the non-determinism in the `if` statement is restricted by a new guard: $A[i] \neq B[i]$. Furthermore, the `assume` condition is the complement of the precondition inferred by SAA earlier, and the condition in the `assert` is also complemented. The existence of an adequate inductive invariant for this program (which in turn can be found by SAA) proves that all its executions from every initial state violating the inferred precondition result in states violating the given postcondition as the program is terminating. In other words, the inferred precondition is indeed the weakest for the program and postcondition in Fig. 1a.

3 Background

3.1 Linear Array Programs

Fig 2 shows a grammar for linear array programs over a set of integer and array variables, $\mathcal{V}$ and $\mathcal{A}$, respectively. In the figure, $v \neq i \in \mathcal{V}$, $a \in \mathcal{A}$, $i \in \mathcal{V}$ is a fixed

$$\begin{array}{rl} \mathit{program} & \rightarrow \mathit{stmts} \\ \mathit{stmts} & \rightarrow \mathit{assign} \;\big|\; \mathit{forloop} \;\big|\; \mathit{stmts};\ \mathit{stmts} \\ \mathit{forloop} & \rightarrow \mathtt{for}(i = 0;\ i < u;\ i = i + 1)\ \{\mathit{assign}\} \end{array} \qquad \begin{array}{rl} \mathit{assign} & \rightarrow v = t(\mathcal{V}, \mathcal{A}) \;\big|\; a[i] = t(\mathcal{V}, \mathcal{A}) \\ & \big|\; \mathtt{if}(::\ g_1(\mathcal{V}, \mathcal{A}) \rightarrow\ \{\mathit{assign}\} \\ & \qquad \cdots \\ & \qquad ::\ g_n(\mathcal{V}, \mathcal{A}) \rightarrow\ \{\mathit{assign}\}) \\ & \big|\; \mathit{assign};\ \mathit{assign} \end{array}$$

Fig. 2: Linear Array Programs

loop counter, $u \in \mathcal{V} \cup \mathbb{Z}$, t is a linear arithmetic expression, and each g_i (or guard) is a boolean combination of linear expressions over $\mathcal{V}$ and $\mathcal{A}$, with $\bigvee_{i=1}^{n} g_i = \top$. The `if` statement is a set of guarded assignments. When such an `if` statement is executed, exactly one guard that evaluates to true in the current program state is non-deterministically chosen and the corresponding assignment statement is executed[5]. A program is *non-deterministic* if there are program states in which more than one guard of an `if` could evaluate to true.

Let P be a linear array program over $\mathcal{V}$ and $\mathcal{A}$. A pre/postcondition for P is a formula of the form $\forall x.\, R(x, \mathcal{V}) \implies Q(x, \mathcal{V}, \mathcal{A})$ or $\exists x.\, R(x, \mathcal{V}) \wedge Q(x, \mathcal{V}, \mathcal{A})$, where $x \notin \mathcal{V}$ is an integer variable, R is a linear predicate over x and $\mathcal{V}$ that represents a range of indices of array(s), and Q is a linear predicate over $\mathcal{V}$ and elements of array(s) in $\mathcal{A}$, the latter being accessed only through linear index expressions in x. As an example, $\forall x.\, (0 \leq x \leq N) \implies (C[x] \leq B[x])$ qualifies for a pre/postcondition, where $N \in \mathcal{V}$ and $C, B \in \mathcal{A}$. Following standard Floyd-Hoare logic, we say a pair of conditions (ψ, ρ) is a valid pre- and postcondition pair for P, if every execution of P that begins in a state satisfying ψ ends in a state satisfying ρ.

The *weakest precondition inference* problem we consider is: given a linear array program P and a postcondition ρ, find the weakest precondition ψ such that (ψ, ρ) forms a valid pre- and postcondition pair for P.

Weakest precondition inference for linear array programs is undecidable in general [49]. Therefore, we cannot hope for an algorithm that infers weakest preconditions in all cases. Nevertheless, many practical and useful programs can be modeled as linear array programs (see for example [7,8,40]). This motivates us to design techniques for finding weakest preconditions that work well for a large subclass of linear array programs.

3.2 Modeling Linear Array Programs as CHCs

In recent years, it is becoming popular to represent a program and its pre- and postcondition as a system of first-order logic (FOL) formulas with uninterpreted relations, called *constrained Horn clauses* (CHCs) [10,19,29,33,35–37,43,45]. In CHCs, the uninterpreted relations represent invariants and the goal is to find interpretations for them. We will consider the task of precondition inference as a CHC-solving task, with the missing precondition represented by a relation.

[5] The usual `if-then-else` statement is easily represented as two guarded assignments.

$$\boldsymbol{pre}(N,A,B,C) \land i = 0 \Longrightarrow \boldsymbol{inv}_1(i,N,A,B,C) \quad (C1)$$

$$\boldsymbol{inv}_1(i,N,A,B,C) \land i<N \land C'= store(C,i,i) \land i' = i+1 \Longrightarrow \boldsymbol{inv}_1(i',N,A,B,C') \quad (C2)$$

$$\boldsymbol{inv}_1(i,N,A,B,C) \land i<N \land A'= store(A,i,C[i]) \land i' = i+1 \Longrightarrow \boldsymbol{inv}_1(i',N,A',B,C) \quad (C3)$$

$$\boldsymbol{inv}_1(i,N,A,B,C) \land \neg(i<N) \land \neg(\forall j.\, 0 \le j < N \Longrightarrow A[j]=B[j]) \implies \bot \quad (C4)$$

Fig. 3: CHC system for the program from Fig. 1a.

Definition 1. *A CHC is a formula in a FOL $\mathcal{L}$ (linear integer arithmetic with arrays in this paper) over a set of relations $\mathcal{R}$ with one of the following forms:*

$$\varphi(\vec{x}_0) \Longrightarrow \boldsymbol{r}_0(\vec{x}_0) \quad (1)$$

$$\bigwedge_{0\le i\le k} \boldsymbol{r}_i(\vec{x}_i) \land \varphi(\vec{x}_0,\ldots,\vec{x}_{k+1}) \Longrightarrow \boldsymbol{r}_{k+1}(\vec{x}_{k+1}) \quad (2)$$

$$\bigwedge_{0\le i\le k} \boldsymbol{r}_i(\vec{x}_i) \land \varphi(\vec{x}_0,\ldots,\vec{x}_k) \Longrightarrow \bot \quad (3)$$

where, for every i, $\boldsymbol{r}_i \in \mathcal{R}$[6], and $\vec{x}_i$ represents the vector of variables $(x_1,\ldots,x_{a_{\boldsymbol{r}_i}})$, where $a_{\boldsymbol{r}_i}$ is the arity of $\boldsymbol{r}_i$. φ, called a *constraint*, is an $\mathcal{L}$-formula in conjunctive normal form without uninterpreted relations. CHCs of type (1) are called *facts*[7], of type (2) *inductive*, and of type (3) *queries*. Note that each CHC has a leading quantification over $\vec{x}$ (e.g. $\forall \vec{x}_0 \ldots \vec{x}_{k+1}$ for type (2)) that is implicit in the paper.

For a CHC C, we use the following notations: *body*(C) (resp. *head*(C)) denotes the left (resp. right) side of the implication in C, *rels*() denotes the relations from $\mathcal{R}$ that appear in *body*(C), or *head*(C), and *args*() denotes the variables in *body*(C), or *head*(C). We assume the constraint φ of a CHC C can be partitioned into two formulas: *assign*(C) and *guard*(C), denoting the assignment statement and control-flow guard conditions (if any). A *system* of CHCs S is a finite set of CHCs. For any system S, if there is a CHC C with $|rels((body(C)))| \ge 1$, then S is *non-linear*, otherwise *linear.*

We assume the input CHC system is induced by a linear array program with $n \ge 0$ sequential loops. In particular, it is a linear CHC system over $\mathcal{R} = \{\boldsymbol{pre}, \boldsymbol{inv}_1, \ldots, \boldsymbol{inv}_n\}$, where $\boldsymbol{pre}$ denotes the precondition, and each $\boldsymbol{inv}_i$ denotes an inductive invariant for the i-th sequential loop.

Example 1. A linear system of CHCs induced by the program from Fig 1a is shown in Fig 3. In the system, the precondition is represented by the relation $\boldsymbol{pre}$ and the inductive invariant by $\boldsymbol{inv}_1$. $C1$ is the initialization CHC with $\boldsymbol{pre}$. The two CHCs $C2$ and $C3$ correspond to non-deterministic writes in the loop, while $C4$ is the query CHC, which has the `assert` condition. It is worth noting that interpretations for $\boldsymbol{pre}$ and $\boldsymbol{inv}_1$ that make each CHC valid gives a

[6] $\boldsymbol{r}_i$'s in each form are not necessarily distinct.

[7] The input CHC system will not have facts but they manifest in Algorithm 4.

precondition and an adequate inductive invariant. For example,

$$\begin{aligned} \boldsymbol{pre} &\mapsto \lambda N, A, B, C. \forall j. 0 \le j < N \implies (A[j] = B[j] \wedge B[j] = C[j]) \\ \boldsymbol{inv}_1 &\mapsto \lambda N, A, B, C, i. \forall j. 0 \le j < i \implies A[j] = B[j] \wedge \\ &\qquad\qquad \forall j. i \le j < N \implies (A[j] = B[j] \wedge B[j] = C[j]) \end{aligned}$$

A *map of interpretations* $\mathcal{M}$ for $\mathcal{R}$ assigns to each relation symbol $\boldsymbol{r} \in \mathcal{R}$ an interpretation of the form $\lambda x_1 \cdots \lambda x_{a_r}. \psi(x_1, \ldots, x_{a_r})$, where ψ is a $\mathcal{L}$-formula. We use the notation $\mathcal{M}[\boldsymbol{r}]$ to denote the interpretation for $\boldsymbol{r}$ by $\mathcal{M}$. For a formula α and a map $\mathcal{M}$ for $\mathcal{R}$, we write $\alpha[\mathcal{M}/\mathcal{R}]$ to denote the formula obtained by replacing each atomic formula of the form $\boldsymbol{r}(t_1, \ldots, t_{a_r})$ in α by $\mathcal{M}[\boldsymbol{r}](t_1, \ldots, t_{a_r})$.

Solution to CHCs A *solution* to a CHC C is a map $\mathcal{M}$ for $\mathcal{R}$ such that the formula $(\mathit{body}(C) \implies \mathit{head}(C))[\mathcal{M}/\mathcal{R}]$ is valid; in this case, we say C is satisfiable. $\mathcal{M}$ is a solution to a system S if it satisfies all the CHCs in S; in this case, we say S is *satisfiable.*

Let S be a system of CHCs induced by a program P and a postcondition ρ. If $\mathcal{M}$ is a solution to S, then $(\mathcal{M}[\boldsymbol{pre}], \rho)$ forms a valid pre/postcondition for P.

3.3 Abductive Inference

The core method used in SAA for inference is *abduction.* Given a formula $(\boldsymbol{r}(\vec{x}) \wedge \alpha(\vec{y})) \implies \beta(\vec{y})$, where $\boldsymbol{r}$ represents a relation, α (hypothesis) and β (conclusion) are formulas without relations, and the variables in $\vec{x}$ are also present in $\vec{y}$, the problem of abduction is to find an interpretation $\lambda x_1 \cdots \lambda x_{a_r}. \psi$ to $\boldsymbol{r}$ such that:

$$\psi(\vec{x}) \wedge \alpha(\vec{y}) \not\Longrightarrow \bot \quad \text{and} \quad \psi(\vec{x}) \wedge \alpha(\vec{y}) \implies \beta(\vec{y})$$

Example 2. Consider the abduction problem $(\boldsymbol{r}(x) \wedge y = 42) \implies (x - y > 0)$. The maximal solution for the problem is $\boldsymbol{r} \mapsto \lambda x. x > 42$.

A given abduction problem can have multiple solutions. SAA seeks the maximal solution. There are techniques, like quantifier elimination, to find maximal solutions [16], but they are limited to non-array theories. To overcome this, range abduction [49] proposes a suitable array-to-integer abstraction, which SAA also uses.

Non-linear CHCs have more than one relation in *body*, requiring an extension of the abduction problem called *multi-abduction.* In multi-abduction, interpretations to multiple relations need to be inferred. SAA encounters non-linear CHCs while searching for maximality proofs, which involve the guard and inductive invariant relations. To solve the multi-abduction problem, SAA uses the technique from [1] after performing the array-to-integer abstraction from [49].

Example 3. The following is a multi-abduction problem: $(\boldsymbol{r}_1(A, i) \wedge \boldsymbol{r}_2(B, i) \wedge C[i] = 42) \implies (A[i] + B[i] > C[i])$. A maximal solution is $\boldsymbol{r}_1 \mapsto \lambda A, i. A[i] > 42$ and $\boldsymbol{r}_2 \mapsto \lambda B, i. B[i] \ge 0$.

Algorithm 1: WEAKESTPRE(S)

```
   Input: $S$ – a system of non-deterministic CHCs over
          $\mathcal{R} = \{\boldsymbol{pre}, \boldsymbol{inv}_1, \ldots, \boldsymbol{inv}_n\}$
   Output: $\langle\{weakest, unknown\}, \mathcal{M}[\boldsymbol{pre}]\rangle$
1  $\langle res, \mathcal{M}\rangle \leftarrow \mathrm{SAA}(S, \varnothing)$;
2  while $res$ do
3      $\langle G, \Gamma\rangle \leftarrow$ GETSPLCHCS$(S, \mathcal{M})$;
4      $\langle max, \_\rangle \leftarrow \mathrm{SAA}(G, \Gamma)$;
5      if $max$ then return $\langle weakest, \mathcal{M}[\boldsymbol{pre}]\rangle$ ;
6      $\langle res, \mathcal{M}\rangle \leftarrow$ WEAKEN$(S, \mathcal{M})$;
7  return $\langle unknown, \mathcal{M}[\boldsymbol{pre}]\rangle$;
```

4 Inferring Weakest Preconditions

An overview of our weakest precondition inference algorithm is in Algorithm 1. The algorithm takes as input a CHC system S over $\{\boldsymbol{pre}, \boldsymbol{inv}_1, \ldots \boldsymbol{inv}_n\}$. It first computes a solution $\mathcal{M}$ to S using SAA (line 1). Though this solution gives a precondition (as the solution will have interpretations to $\boldsymbol{pre}, \boldsymbol{inv}_1, \ldots \boldsymbol{inv}_n$), it is not guaranteed to be the weakest. Hence, in a loop, the algorithm performs maximality checking and weakening (lines 2 to 6). When the maximality check succeeds, the solution is guaranteed to be the weakest (Theorem 1); hence the algorithm returns the current precondition $\mathcal{M}[\boldsymbol{pre}]$ (line 5). Otherwise, the algorithm assumes the maximality check is inconclusive and tries to find a weakening (line 6). The algorithm progresses and continues the same loop if a weakening is found. When the weakening is inconclusive, the loop terminates, and the current precondition is returned without a maximality guarantee (line 7).

SAA takes a CHC system, which is either a precondition inference task (S), or the encoding of maximality check (G) with additional non-CHC constraints (Γ). It finds a solution $\mathcal{M}$ to the CHC system that also satisfies the additional constraints.

Algorithm 1 proves the maximality of a precondition by encoding a CHC system G and non-CHC constraints Γ, together called *specialized CHCs* (line 3). G has the same set of CHCs as the input CHC system S except the following: 1) the relation $\boldsymbol{pre}$ is replaced by the formula $\neg\mathcal{M}[\boldsymbol{pre}]$, 2) the postcondition is the negation of the postcondition in S, and 3) new *guard relations*: $\boldsymbol{g}_{Ci}, \ldots, \boldsymbol{g}_{Cj}$ are added to *body* of CHCs: $Ci, \ldots Cj$ corresponding to non-deterministic `if` conditions. Thus, G is a CHC system over the invariant relations of S and new guard relations. A solution to G gives stronger `if` conditions and inductive invariants for the complement pre- and postcondition. The non-CHC constraints in Γ make sure the disjunction of $\boldsymbol{g}_{Ci}, \ldots, \boldsymbol{g}_{Cj}$ is $\top$; thus ensuring the interpretations for them are not too strong.

When SAA fails to find a solution to G, Algorithm 1 calls WEAKEN. At a high level, WEAKEN enumerates candidate preconditions obtained in a syntax-guided way like [22] and then tries to find inductive invariants using SAA again.

$$(\exists j.\, 0 \leq j < N \wedge (A[j] \neq B[j] \;\vee\; B[j] \neq C[j])) \;\wedge\; i = 0 \Longrightarrow \boldsymbol{inv}_1(i, N, A, B, C)$$

$$\boldsymbol{inv}_1(i,N,A,B,C) \wedge \boldsymbol{g}_{C2}(i,N,A,B,C) \wedge i < N \wedge C' = store(C, i, i) \wedge i' = i+1 \Longrightarrow \boldsymbol{inv}_1(i',N,A,B,C')$$

$$\boldsymbol{inv}_1(i,N,A,B,C) \wedge \boldsymbol{g}_{C3}(i,N,A,B,C) \wedge i < N \wedge A' = store(A, i, C[i]) \wedge i' = i+1 \Longrightarrow \boldsymbol{inv}_1(i',N,A',B,C)$$

$$\boldsymbol{inv}_1(i,N,A,B,C) \wedge \neg(i < N) \wedge \neg(\exists j.\, 0 \leq j < N \wedge A[j] \neq B[j]) \implies \bot$$

$$\top \implies \boldsymbol{g}_{C2}(i,N,A,B,C) \vee \boldsymbol{g}_{C3}(i,N,A,B,C)$$

Fig. 4: Specialized CHCs for the CHCs from Example 1.

Example 4. Fig. 4 shows the specialized CHCs for the CHC system from Fig. 3 and the precondition from Example 1 (viz. $\forall j.\, 0 \leq j < N \implies (A[j] = B[j] \wedge B[j] = C[j])$). This system has following changes: 1) In the first CHC, the relation $\boldsymbol{pre}$ is replaced by the complement of the precondition, 2) The next two CHCs have $\boldsymbol{g}_{C2}, \boldsymbol{g}_{C3}$ in *body*, 3) In the fourth CHC, the postcondition is complemented, and 4) The last constraint is a non-CHC constraint that makes sure $\boldsymbol{g}_{C2} \vee \boldsymbol{g}_{C3}$ is $\top$.

Theorem 1 (Soundness of Algorithm 1[8]). *For a system S, if Algorithm 1 terminates with "weakest" then $\mathcal{M}[\boldsymbol{pre}]$ is the weakest precondition for S.*

5 Structural Array Abduction

Structural Array Abduction (SAA) solves CHCs. In Algorithm 1, SAA solves program-induced CHCs to identify preconditions, specialized CHCs for maximality proofs, and CHCs with candidate weakened preconditions to find invariants.

5.1 Algorithm Description

SAA aims to find interpretations to $\boldsymbol{pre}$ and $\boldsymbol{inv}$ of the following form:

$$\bigwedge \big(\forall x.\, R(x, \mathcal{V}) \implies Q(x, \mathcal{V}, \mathcal{A})\big) \quad \text{or} \quad \bigvee \big(\exists x.\, R(x, \mathcal{V}) \wedge Q(x, \mathcal{V}, \mathcal{A})\big) \tag{4}$$

Similar to the postcondition, here, R is a linear predicate over x and $\mathcal{V}$ that represents a range of indices of array(s), and Q is a linear predicate over $\mathcal{V}$ and elements of array(s) in $\mathcal{A}$, the latter being accessed only through linear index expressions in x. Such a form is sufficient to represent inductive invariants for a large class of array programs, as observed in existing works [22, 28, 30, 31, 38].

A relatively complete guessing algorithm involves enumerating all candidate solutions in the form of 4 and then checking them using an SMT solver. However, given the large number of candidate solutions and the inherent challenge that quantified formulas with arrays pose for SMT solvers, SAA brings a novel improvement. It narrows down the search by guessing likely candidate solutions using a logical method, as presented in Algorithm 2.

[8] Proofs are in [48].

Algorithm 2: SAA (S, Γ)

```
  Input: $S$ – set of CHCs over $\mathcal{R} \cup \mathcal{R}_g$, $\Gamma$ – non-CHC constraints over $\mathcal{R}_g$
  Output: $(\{\top, \bot\}$ – result, $\mathcal{M}$ – solution to $S$ that also satisfies $\Gamma)$
1 while $\exists C \in S.$ CheckSAT$(\neg(body(C) \implies head(C))[\mathcal{M}/\mathcal{R}])$ do
2     if $C$ is not fact then
3         $\mathcal{M} \leftarrow$ ArrayAbduce$(S, C, \mathcal{M})$;
4     else
5         $\mathcal{M} \leftarrow$ WeakenFact$(S, \mathcal{M})$;
6     if $\mathcal{M}$ is unchanged then $\mathcal{M} \leftarrow$ NextCandidate() ;
7 $res \leftarrow$ CheckSAT$(\Gamma)$;
8 return$(res, \mathcal{M})$;
```

Algorithm 2 begins with an initial candidate solution, e.g., $\forall \boldsymbol{r} \in \mathcal{R}.\ \mathcal{M}[\boldsymbol{r}] = \top$ in our implementation, and checks whether it is a solution to all CHCs. If not, the algorithm attempts to make the candidate a solution to the failed CHC mainly through abduction-based strengthening (line 3), or heuristics-based weakening if the CHC is a *fact* (line 5). If neither strengthening nor weakening results in a change to the candidate, the algorithm proceeds to the next candidate in the fixed form. When a candidate is found to be a solution, it is checked for additional constraints in Γ.

The abduction-based strengthening method is presented in Algorithm 3. It seeks new interpretations for the relations in *body* of a CHC, which can be $\boldsymbol{pre}$, $\boldsymbol{inv}$, or $\boldsymbol{g}_C$, that imply the interpretation for the relation in *head* of the CHC. This constitutes the abduction problem, as defined in Sec 3.3. However, existing abduction solvers cannot be used directly as they do not support quantified formulas with arrays. Hence, in Algorithm 3, Q and R from the fixed form (4) are determined separately and then combined into a quantified formula.

To find Q, the algorithm constructs an abduction query based on the rules provided in Table 1. In the abduction query, the hypothesis (α) is the assignment formula present in the constraint of the CHC (line 1), and the conclusion (β) is derived from the table based on the type of the CHC (line 2). Since the query contains array terms, which are not supported by existing abduction solvers, they are replaced by integer terms in a manner similar to the approach presented in [49] (e.g., $A[i]$ is replaced by a new integer variable a_i). Subsequently, the query is solved using an integer abduction solver to obtain a maximal solution (line 4). When the CHC has a guard relation $\boldsymbol{g}_C$ in its *body*, an additional abduction query is constructed to find interpretations for the other guard relations $\boldsymbol{g}_{C'}$ (line 6). Finally, integer terms in the solutions of the abduction queries are mapped back to corresponding array terms (line 7).

SAA uses the concept of range formulas as described in [22] to determine R. In the context of linear array programs, these range formulas can take the form of $0 \leq j < u$, $0 \leq j < i$, and $i \leq j < u$[9], where j is a free variable and u

[9] In [22], these are referred as *Range*, *progressRange*, and *regressRange*, respectively.

```
Algorithm 3: ArrayAbduce($S, C, \mathcal{M}$)
  Input: $S$ – set of CHCs over $\mathcal{R} \cup \mathcal{R}_{\boldsymbol{g}}$, $C$ – CHC in $S$ where $rels(body(C))$ is
         $\{\boldsymbol{r}\}$, or $\{\boldsymbol{r}, \boldsymbol{g}_C\}$, where $\boldsymbol{r}$ is either $\boldsymbol{pre}$ or some $\boldsymbol{inv}$ from $\mathcal{R}$, and
         $\boldsymbol{g}_C \ldots \boldsymbol{g}_{C'}$ are from $\mathcal{R}_{\boldsymbol{g}}$ for the same control-flow condition, $\mathcal{M}$ –
         mapping from $\mathcal{R} \cup \mathcal{R}_g$ to predicates
  Output: $\mathcal{M}'$ – updated $\mathcal{M}$ with new interpretations to $\boldsymbol{r}$ and $\boldsymbol{g}_C \ldots \boldsymbol{g}_{C'}$
 1  $\alpha \leftarrow assign(C) \wedge \mathcal{M}[\boldsymbol{g}_C]$;
 2  $\beta \leftarrow$ Get$\beta(C, \mathcal{M})$                    // cf. Tab 1;
 3  Transform $\alpha$ and $\beta$ to integer formulas;
 4  $\langle Q, Q_{\boldsymbol{g}_C} \rangle \leftarrow$ AbdSolver($\boldsymbol{r}(\vec{x}_{\boldsymbol{r}}) \wedge \boldsymbol{g}_C(\vec{x}_{\boldsymbol{g}_C}) \wedge \alpha \implies \beta$);
 5  if $Q_{\boldsymbol{g}_C} \neq \bot$ then
 6      $\langle Q_{\boldsymbol{g}_{C' \neq C}} \rangle \leftarrow$ AbdSolver($\bigvee_{C' \neq C} \boldsymbol{g}_{C'}(\vec{x}_{\boldsymbol{g}_{C'}}) \implies \neg Q_{\boldsymbol{g}_C}$);
 7  Transform $Q$, $Q_{\boldsymbol{g}_C} \ldots Q_{\boldsymbol{g}_{C'}}$ to array formulas;
 8  $\langle R, j \rangle \leftarrow$ Get$R(S, C)$                    // cf. Tab 1;
 9  if universally quantified then $\mathcal{M}[r] \leftarrow \mathcal{M}[r] \wedge \forall j.\, R \implies Q$ ;
10  else $\mathcal{M}[r] \leftarrow \mathcal{M}[r] \vee \exists j.\, R \wedge Q$ ;
11  $\mathcal{M}[\boldsymbol{g}_C] \leftarrow Q_{\boldsymbol{g}_C} \ldots \mathcal{M}[\boldsymbol{g}_{C'}] \leftarrow Q_{\boldsymbol{g}_{C'}}$ ;
12  return $\mathcal{M}$;
```

is the upper bound of the loop (cf. Fig. 2). From these formulas, a suitable R is selected based on the type of the CHC in Table 1.

The resulting R and Q are appropriately combined into a quantified formula and conjoined (or disjoined in the case of existential quantification) with the existing interpretation (lines 9 and 10). The guard relations are updated by the non-quantified formulas $Q_{\boldsymbol{g}_C}, \ldots, Q_{\boldsymbol{g}_{C'}}$.

Table 1 provides a set of rules for determining R and β for all types of CHCs that Algorithm 3 may encounter. These CHCs include: 1) *precond*: the initialization CHC with $\boldsymbol{pre}$ in *body*, 2) *query*: the CHC with postcondition, 3) *intra*: CHCs representing potentially non-deterministic updates within a loop, and 4) *inter*: CHCs occurring between two loops. For example, when a CHC C falls into the *precond* category, β is the Q present in $\mathcal{M}[rels(head(C))]$ corresponding to the range $i \leq j < u$, and formula R is $0 \leq j < u$. We give an intuition to these rules while illustrating our technique in the following section.

When a *fact* CHC is unsatisfiable, SAA uses heuristic-based weakening for the *head* relation (Algorithm 2, line 5). This method generates a candidate set of Q formulas using the syntax of *body* of the CHC and combines it with $i \leq j < u$ to get a quantified formula.

Theorem 2. *If the input CHC system S has a solution in the form of 4, then Algorithm 2 will find it provided all the SMT checks return a result.*

5.2 Distinguishing SAA With Closely-Related Techniques

SAA uses the concept of range formulas from [22] and array to integer abduction technique from range abduction [49]. Nevertheless, there are notable differences:

$$\frac{\boldsymbol{pre} \in \mathit{rels}(\mathit{body}(C))}{\beta \leftarrow Q(i \leq j < u, \mathcal{M}[\mathit{rels}(\mathit{head}(C))]) \quad R \leftarrow 0 \leq j < u}\ \mathit{precond} \qquad \frac{\mathit{rels}(\mathit{head}(C)) = \varnothing}{\beta \leftarrow Q(0 \leq j < u, \rho) \quad R \leftarrow 0 \leq j < i}\ \mathit{query}$$

$$\frac{\mathit{rels}(\mathit{head}(C)) \subseteq \mathit{rels}(\mathit{body}(C))}{\beta \leftarrow Q(0 \leq j < i, \mathcal{M}[\mathit{rels}(\mathit{head}(C))]) \quad R \leftarrow i \leq j < u}\ \mathit{intra}$$

$$\frac{\mathit{rels}(\mathit{head}(C)) \cap \mathit{rels}(\mathit{body}(C)) = \varnothing \text{ and } \mathit{rels}(\mathit{head}(C)) \neq \varnothing}{\beta \leftarrow Q(i \leq j < u, \mathcal{M}[\mathit{rels}(\mathit{head}(C))]) \quad R \leftarrow 0 \leq j < i}\ \mathit{inter}$$

Table 1: Rules for all CHC types to construct formulas R and Q in Algorithm 3.

1) While [22] relies on preconditions to infer invariants, SAA is capable of inferring invariants even in the absence of preconditions. 2) Both [22] and range abduction can't handle nonlinear CHCs resulting from guarded relations, which SAA support by using multi-abduction. 3) In our experiments, we observed that range abduction tends to generate stronger preconditions compared to SAA. 4) Range abduction performs two abduction queries for each CHC, whereas SAA requires only one. 5) Range abduction uses the Houdini algorithm [23] for weakening, which is not necessary for SAA.

5.3 Illustration

Consider the CHCs from Fig. 3. For these CHCs, the range formulas are: $0 \leq j < N$, $0 \leq j < i$, and $i \leq j < N$, as the upper bound u of the loop is N.

The algorithm begins with $\mathcal{M}[\boldsymbol{pre}] = \mathcal{M}[\boldsymbol{inv}_1] = \top$. But, the *query* CHC (C4) is unsatisfiable as $\mathcal{M}[\boldsymbol{inv}_1]$ is too weak. So, SAA tries to find a strengthening for $\boldsymbol{inv}_1$ using abduction. Recall that the postcondition (ρ) is $\forall j.\, 0 \leq j < N \implies A[j] = B[j]$. While ρ itself can make this CHC satisfiable, it might be too strong for other CHCs with $\boldsymbol{inv}_1$. Therefore, the rule for *query* in the table suggests to consider R as $0 \leq j < i$, and β as $A[j] = B[j]$ from ρ, corresponding to the range $0 \leq j < N$. The abduction query $(\boldsymbol{inv}_1(A, B, C, i, N) \wedge \top) \implies A[j] = B[j]$ yields Q as $A[j] = B[j]$. Combining R and Q results in:

$$\mathcal{M}[\boldsymbol{inv}_1] \overset{\text{cand}}{\mapsto} \forall j.\, (0 \leq j < i) \implies A[j] = B[j]$$

Next, an *intra* CHC $C2$ is unsatisfiable. This is due to the absence of restrictions on the values of A and B in the range $i \leq j < N$ within $\boldsymbol{inv}_1$. One way to fix this is to find a Q in the range $i \leq j < N$ that implies $A[j] = B[j]$. This approach aligns with the rule for *intra* CHC, where β is $A[j] = B[j]$ corresponding to the range $0 \leq j < i$ of $\mathcal{M}[\boldsymbol{inv}_1]$, and R is $i \leq j < N$. Further, $\mathit{assign}(C2)$ is $C'[j] = j$ (primed variables denote updated variables), resulting in the following abduction query: $(\boldsymbol{inv}_1(A, B, C, i, N) \wedge C'[j] = j) \implies A[j] = B[j]$. This query yields $A[j] = B[j]$ as Q. Combining R and Q into a quantified formula, and conjoining it with $\mathcal{M}[\boldsymbol{inv}_1]$ gives:

$$\begin{aligned} \mathcal{M}[\boldsymbol{inv}_1] \overset{\text{cand}}{\mapsto} & \forall j.\, (0 \leq j < i) \implies A[j] = B[j] \wedge \\ & \forall j.\, (i \leq j < N) \implies A[j] = B[j] \end{aligned}$$

In the third iteration, another *intra* CHC $C3$ fails the check. Here, $\mathit{assign}(C3)$ is $A'[j] = B[j]$ and β is $A'[j] = B[j]$, resulting in the following abduction query: $(\boldsymbol{inv}_1(A, B, C, i, N) \wedge A'[j] = C[j]) \implies A'[j] = B[j]$. This query yields $B[j] = C[j]$ as Q. Combining this with R, which is $i \leq j < N$, results in:

$$\mathcal{M}[\boldsymbol{inv}_1] \overset{\text{cand}}{\mapsto} \forall j. (0 \leq j < i) \implies A[j] = B[j] \wedge \\ \forall j. (i \leq j < N) \implies (A[j] = B[j] \wedge B[j] = C[j])$$

Subsequently, a *precond* CHC, $C1$, fails the check. These CHCs have an initialization of the counter variable (i.e., $i = 0$), rendering the formula within the range $0 \leq j < i$ trivially $\top$. Therefore, the rule for *precond* CHC selects β from the other range, i.e., β is $A[j] = B[j] \wedge B[j] = C[j]$ from the range $i \leq j < N$ of $\mathcal{M}[\boldsymbol{inv}_1]$. This leads to the following abduction query: $(\boldsymbol{pre}(A, B, C, i, N) \wedge \top) \implies (A[j] = B[j] \wedge B[j] = C[j])$, which yields Q as $A[j] = B[j] \wedge B[j] = C[j]$. Further, R is $0 \leq j < N$, resulting in:

$$\mathcal{M}[\boldsymbol{pre}] \overset{\text{cand}}{\mapsto} \forall j. (0 \leq j < N) \implies (A[j] = B[j] \wedge B[j] = C[j]).$$

The algorithm terminates as the candidate $\mathcal{M}$ is a solution.

6 Specialized Maximality Checking

While SAA effectively infers precondition, it may not always be the weakest. To check for maximality, a specialized CHC system (G and Γ) is generated in Algorithm 1 using the method GetSplCHCs, which is described in this section. This section also covers the method to weaken a precondition.

6.1 GetSplCHCs method

The GetSplCHCs method constructs a new CHC system G by iterating over all the CHCs in the input system S while performing the following:

1. Replacing $\boldsymbol{pre}$ with $\neg\mathcal{M}[\boldsymbol{pre}]$ and the postcondition ρ with $\neg\rho$, and
2. For each relation $\boldsymbol{inv}_i$, if there exist two *intra* CHCs C and C' with $\mathit{guard}(C) \wedge \mathit{guard}(C') \not\Longrightarrow \bot$, then for each *intra* CHC C of $\boldsymbol{inv}_i$, a new relation $\boldsymbol{g}_C(\mathit{args}(\mathit{body}(C))$ is added to $\mathit{body}(C)$.

Example 5. The CHC system from Fig. 3, has *intra* CHCs $C2$ and $C3$ with $\mathit{guard}(C2) = \mathit{guard}(C3) = i < N$, so $\mathit{guard}(C) \wedge \mathit{guard}(C') \not\Longrightarrow \bot$. As a result, two new relations $\boldsymbol{g}_{C2}$ and $\boldsymbol{g}_{C3}$ are introduced into $\mathit{body}(C2)$ and $\mathit{body}(C3)$, leading to the CHC system shown in Fig. 4. For this system, SAA finds: $\boldsymbol{g}_{C2} \mapsto A[i] \neq B[i]$ and $\boldsymbol{g}_{C3} \mapsto A[i] = B[i]$, along with an invariant for $\boldsymbol{inv}_1$.

A solution to G can result in interpretations for guard relations that block all executions (e.g., $\bot$). To prevent this, the following non-CHC constraint (Γ) will be introduced:

```
Algorithm 4: Weaken($S, \mathcal{M}$)
  Input: $S$ – set of CHCs over $\mathcal{R}$, $\mathcal{M}$ – mapping for $\mathcal{R}$
  Output: $\langle\{\top, \bot\}, \mathcal{M}\rangle$ – $\top$ indicates $\mathcal{M}$ has a weakened pre and $\bot$ indicates
          failure
  $\Delta \leftarrow$ CandidatePrecond($S, \mathcal{M}$);
  for $\delta \in \Delta$ do
      $S' \leftarrow$ replace pre by $\delta$ in $S$;
      $(res, \mathcal{M}') \leftarrow \mathrm{SAA}(S', \varnothing)$;
      if $res$ then return $(\top, \mathcal{M}')$ where $\mathcal{M}'[\boldsymbol{pre}] = \delta$ ;
  return $\langle\bot, \mathcal{M}\rangle$;
```

$$\top \implies \bigvee_{1 \le j \le m} \big(\boldsymbol{g}_{C_j}(args(body(C_j)) \wedge guard(C_j))\big)$$

Theorem 3. *If a CHC system S induced by a program P has a solution $\mathcal{M}$, and its specialized CHCs (G and Γ) are satisfied, then $\mathcal{M}[\boldsymbol{pre}]$ is the weakest precondition of P.*

6.2 Weakening Procedure

When SAA is inconclusive on the specialized CHCs, the precondition $\mathcal{M}[\boldsymbol{pre}]$ is weakened, as shown Algorithm 4.The algorithm begins by computing a set of potential candidate preconditions Δ. We assume that this set is computed in a syntax-guided fashion like in [22] (can also be provided by the user). Only candidate preconditions that are strictly weaker than $\mathcal{M}[\boldsymbol{pre}]$ are taken into consideration. For each such candidate $\delta \in \Delta$, SAA is invoked to infer inductive invariants by passing the CHC system S with the relation $\boldsymbol{pre}$ replaced by δ. This process continues till success or all the candidates have been exhausted. Whenever the method succeeds, the precondition is weaker by construction.

7 Evaluation

Implementation Our algorithm is implemented in a tool called MaxPrANQ on top of the HornSpec framework [50]. The tool takes as input a set of CHCs with preconditions and invariants represented as uninterpreted relations. It returns the weakest precondition, along with proof of validity (viz. inductive invariants) and maximality (viz. specialized CHCs and its solution). It uses Z3 [14] to solve SMT queries. Quantifier elimination is done by model-based projection [3,20].

Research Questions We evaluate MaxPrANQ on the following questions:

RQ1 Can MaxPrANQ find weakest preconditions for a range of benchmarks?
RQ2 How well does MaxPrANQ perform in comparison with state-of-the-art techniques?
RQ3 How challenging is it for existing techniques to infer invariants for our benchmarks even with the preconditions being *given*?

Benchmarks and Configuration We use 66 precondition inference tasks with universal quantified postconditions. While we initially intended to use the precondition inference benchmarks from [53], none of them had quantified postconditions. Hence, we derived our benchmarks from existing *verification tasks* of [22] that had quantified postconditions. Specifically, we considered multiple loop benchmarks from [22], where the first loop is an initialization loop, and the other loops perform various array update operations. We then removed the first loop so that a non-trivial quantified precondition would need to be synthesized. Overall, we consider 26 multiple loop benchmarks from [22]. Since a majority of the 26 benchmarks were deterministic, we added non-deterministic guards to the update operations and introduced a similar update operation in the other branch . To further test our tool, we adapted these benchmarks to 40 more benchmarks by using common array update operations and postconditions. We performed the experiments on a Ubuntu machine with 2.5 GHz processor and 16 GB memory. A timeout of 200 seconds was given to all the tools.

Tools for comparison We compare our tool against PreQSyn [49], an abduction-based precondition inference tool, and P-Gen [53], a predicate abstraction based tool. Additionally, we compare against the CHC solvers that can generate quantified inductive invariants: FreqHorn [22], a SyGuS based tool, and Spacer [30] (Z3 v4.8.10), an extension of PDR for quantified formulas.

RQ1 MaxPrANQ found and automatically proved 59/66 weakest preconditions. For the remaining 7, it found the weakest precondition, but couldn't prove it automatically due to failure in finding a solution to the specialized CHCs. Overall, it solved 125 CHC systems – 66 universal and 59 existential quantification. The time taken was less than 30 seconds on all except one benchmark. Details are in Fig 5 and [48].

RQ2 PreQSyn found and automatically proved 2/66 weakest preconditions. On the remaining 64, it found preconditions for 56 but could not prove; for 8, it did not find a precondition. To compare with our maximality checking, we provided the 56 preconditions generated by PreQSyn to our SMC module. Out of 56, SMC proved 52 to be the weakest, where 7 were weakened before proving. We observe that PreQSyn's maximality checking is unsuitable for non-deterministic programs, and its preconditions are not always the weakest.

P-Gen did not find a precondition for any of the 66 benchmarks. Its output was not a precondition on 41, and on the rest it was stuck in the refinement loop. Our experiments conclude that P-Gen's inference engine is unable to generalize and find quantified preconditions when postconditions are quantified. Hence, our technique complements P-Gen's capability of finding preconditions for quantifier-free postconditions.

RQ3 The reader may wonder whether the benchmarks themselves are easier to solve, given the limited availability of weakest precondition tools for non-deterministic programs. We experimentally demonstrate that this is not the case by passing the CHCs *with* preconditions generated by MaxPrANQ to state-of-the-art CHC solvers for arrays: FreqHorn and Spacer. Out of 66 benchmarks, FreqHorn found inductive invariants for 56 and Spacer found 34. In comparison, MaxPrANQ found preconditions and invariants for all the 66 benchmarks.

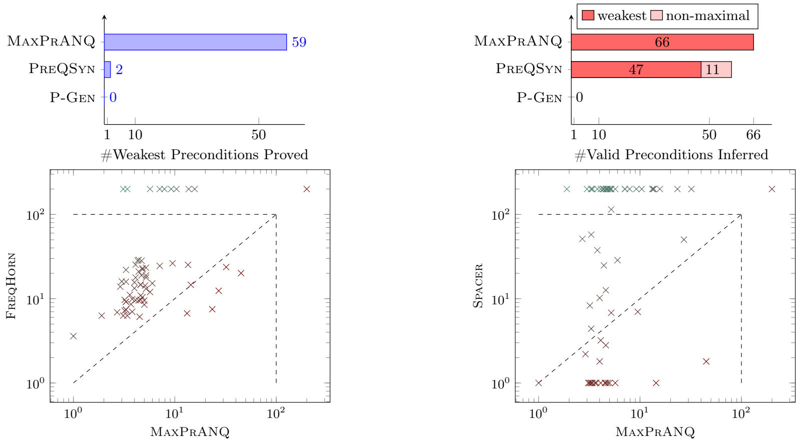


Fig. 5: The bar graphs show the # weakest proven and valid preconditions inferred by the tools. The scatter plots show the time taken by the tools for invariant inference.

8 Related Work

The problem of precondition inference has received considerable attention [2,11,12,24,46,47,52,53]. In particular, for programs with arrays, closely related works include [12,49,53]. The work in [12] infers preconditions by abstract interpretation, [53] by CEGAR based predicate abstraction, and [49] by range abduction. Compared to [12], we don't need a predefined abstract domain. We work in a framework similar to [53] and [49], but they target deterministic programs. Their maximality check assumes that from a precondition only one execution reaches the postcondition, which is not the case for non-deterministic programs. The novelty of our SAA algorithm, compared to range abduction [49], is in how it constructs abduction queries using a set of rules based on the structure of the CHCs, and support for non-linear CHCs. Range abduction, on the other hand, creates two abduction queries and employs the Houdini technique [23], which can generate stronger preconditions, as observed in our experiments.

Precondition inference is closely related to the problem of invariant inference. For inferring universally quantified invariants, several techniques have been proposed. The main methods include predicate abstraction [32,41], abstract interpretation [28], PDR [30], and syntax guided synthesis [22]. These techniques are crucially dependent on given preconditions, which are missing in our setting. Without preconditions, they generate trivial solution like $\bot$.

The validity of a precondition can be established by techniques that do not explicitly generate inductive invariants. Such techniques include array smashing [5], converting to array-free nonlinear CHCs [44], over-approximating unknown bound of loops to a smaller known bound [40], accelerating entire transition relations [6], using CHC transformation [4, 34], induction based techniques [7–9] and trace logic based techniques [25]. These techniques are useful for assertion checking and not directly for precondition inference, however.

CHCs are widely used to symbolically encode different synthesis tasks [18, 21,50,51,55]. However, none of these works handle CHCs with arrays. SAA uses abduction that has been used for programs without arrays to infer invariants [16, 17], preconditions [15,26], and specifications [1,50,54].

The concept of SMC resembles angelic verification [13,42], but differs in how it is solved. Angelic verification neither guarantees maximality nor computes inductive invariants, and uses user supplied specifications. A recent work [27] proposes a reduction of maximality checking to finding termination proofs for CHC systems with integers. In comparison, SMC reduces to finding inductive invariants and guards by exploiting the fact that the programs are terminating.

9 Limitations and Future Work

Usage of Theorem Prover: The preconditions and invariants guessed by SAA in our evaluation are in a fragment of the theory of one-dimensional arrays and linear integer arithmetic that state-of-the-art SMT solvers support reasonably well. However, in a general case, SAA might generate a challenging precondition/invariant for our SMT solver. In such instances, MaxPrANQ logs a failure and switches to another precondition/invariant. To handle such cases, we plan to complement our SMT solver by an automated theorem prover like Vampire [39]. This can also help us to handle preconditions with alternating quantification.

Non-linear CHC Support: The multi-abduction done in SAA can help in handling non-linear CHCs, which can encode programs with recursive functions. For this, the range analysis in SAA has to be tweaked to determine a loop counter-like variable for recursive functions, which we target for future work.

Termination and Compositional Verification: The assumption of terminating programs helps in proving maximality by inferring invariants and stronger guards. Relaxing this assumption would require a more complex maximality checking. Finally, an immediate future work is to integrate this technique in an existing verifier to scale it compositionally.

Acknowledgement

We would like to thank Kumar Madhukar for providing feedback on our paper.

References

1. A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *POPL*, pages 789–801. ACM, 2016.
2. A. Astorga, P. Madhusudan, S. Saha, S. Wang, and T. Xie. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 775–787, 2019.
3. N. Bjørner and M. Janota. Playing with quantified satisfaction. In *LPAR (short papers)*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015.
4. N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *International Static Analysis Symposium*, pages 105–125. Springer, 2013.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation*, pages 85–108. Springer, 2002.
6. M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *Computer Aided Verification*, pages 157–172. Springer Berlin Heidelberg, 2009.
7. S. Chakraborty, A. Gupta, and D. Unadkat. Verifying array manipulating programs by tiling. In *International Static Analysis Symposium*, pages 428–449. Springer, 2017.
8. S. Chakraborty, A. Gupta, and D. Unadkat. Verifying array manipulating programs with full-program induction. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2020.
9. S. Chakraborty, A. Gupta, and D. Unadkat. Diffy: Inductive reasoning of array programs using difference invariants. In *CAV (2)*, volume 12760 of *Lecture Notes in Computer Science*, pages 911–935. Springer, 2021.
10. A. Champion, T. Chiba, N. Kobayashi, and R. Sato. Ice-based refinement type discovery for higher-order functional programs. In *TACAS, Part I*, volume 10805 of *LNCS*, pages 365–384. Springer, 2018.
11. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.
12. P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 150–168. Springer, 2011.
13. A. Das, S. K. Lahiri, A. Lal, and Y. Li. Angelic verification: Precise verification modulo unknowns. In *CAV, Part I*, volume 9206 of *LNCS*, pages 324–342. Springer, 2015.
14. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
15. I. Dillig and T. Dillig. Explain: a tool for performing abductive inference. In *International Conference on Computer Aided Verification*, pages 684–689. Springer, 2013.
16. I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456. ACM, 2013.

17. M. Echenim, N. Peltier, and Y. Sellami. Ilinva: Using abduction to generate loop invariants. In *FroCoS*, volume 11715 of *LNCS*, pages 77–93. Springer, 2019.
18. P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan. Horn-ICE learning for synthesizing invariants and contracts. *PACMPL*, 2(OOPSLA):131:1–131:25, 2018.
19. G. Fedyukovich, M. B. S. Ahmad, and R. Bodík. Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs. In *PLDI*, pages 572–585. ACM, 2017.
20. G. Fedyukovich, A. Gurfinkel, and A. Gupta. Lazy but Effective Functional Synthesis. In *VMCAI*, volume 11388 of *LNCS*, pages 92–113. Springer, 2019.
21. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*, pages 170–178. IEEE, 2018.
22. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Quantified Invariants via Syntax-Guided Synthesis. In *CAV, Part I*, volume 11561 of *LNCS*, pages 259–277. Springer, 2019.
23. C. Flanagan and K. R. M. Leino. Houdini: an Annotation Assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.
24. T. Gehr, D. Dimitrov, and M. Vechev. Learning commutativity specifications. In *International Conference on Computer Aided Verification*, pages 307–323. Springer, 2015.
25. P. Georgiou, B. Gleiss, and L. Kovács. Trace logic for inductive loop reasoning. In *FMCAD*, pages 255–263. IEEE, 2020.
26. R. Giacobazzi. Abductive analysis of modular logic programs. In *ILPS*, volume 94, pages 377–391, 1994.
27. Y. Gu, T. Tsukada, and H. Unno. Optimal chc solving via termination proofs. *Proceedings of the ACM on Programming Languages*, 7(POPL):604–631, 2023.
28. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 235–246, 2008.
29. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework. In *CAV*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.
30. A. Gurfinkel, S. Shoham, and Y. Vizel. Quantifiers on demand. In *ATVA*, volume 11138 of *LNCS*, pages 248–266, 2018.
31. T. A. Henzinger, T. Hottelier, L. Kovács, and A. Rybalchenko. Aligators for arrays (tool paper). In *Logic for Programming, Artificial Intelligence, and Reasoning: 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings 17*, pages 348–356. Springer, 2010.
32. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. *ACM SIGPLAN Notices*, 39(1):232–244, 2004.
33. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, volume 7436 of *LNCS*, pages 247–251. Springer, 2012.
34. H. Hojjat and P. Rümmer. The ELDARICA Horn Solver. In *FMCAD*, pages 158–164. IEEE, 2018.
35. T. Kahsai, R. Kersten, P. Rümmer, and M. Schäf. Quantified heap invariants for object-oriented programs. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 368–384. EasyChair, 2017.
36. T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. Jayhorn: A framework for verifying Java programs. In *CAV, Part I*, volume 9779 of *LNCS*, pages 352–358. Springer, 2016.

37. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *ACM*, pages 222–233. ACM, 2011.
38. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.
39. L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.
40. S. Kumar, A. Sanyal, R. Venkatesh, and P. Shah. Property checking array programs using loop shrinking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 213–231, Cham, 2018. Springer International Publishing.
41. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 267–281. Springer, 2004.
42. S. K. Lahiri, A. Lal, S. Gopinath, A. Nutz, V. Levin, R. Kumar, N. Deisinger, J. Lichtenberg, and C. Bansal. Angelic checking within static driver verifier: Towards high-precision defects without (modeling) cost. In *FMCAD*, pages 169–178. IEEE, 2020.
43. Y. Matsushita, T. Tsukada, and N. Kobayashi. RustHorn: CHC-Based Verification for Rust Programs. In *ESOP*, volume 12075 of *LNCS*, pages 484–514. Springer, 2020.
44. D. Monniaux and L. Gonnord. Cell morphing: From array programs to array-free horn clauses. In *Static Analysis*, pages 361–382. Springer Berlin Heidelberg, 2016.
45. D. Mordvinov and G. Fedyukovich. Verifying Safety of Functional Programs with Rosette/Unbound. *CoRR*, abs/1704.04558, 2017. `https://github.com/dvvrd/rosette`.
46. Y. Moy. Sufficient preconditions for modular assertion checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 188–202. Springer, 2008.
47. S. Padhi, R. Sharma, and T. Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 51(6):42–56, 2016.
48. S. Prabhu, D. D'Souza, S. Chakraborty, R. Venkatesh, and G. Fedyukovich. Weakest precondition inference for non-deterministic linear array programs (extended version). 2024. `https://doi.org/10.6084/m9.figshare.25050077`.
49. S. Prabhu, G. Fedyukovich, and D. D'Souza. Maximal Quantified Precondition Synthesis for Linear Array Loops. In *ESOP*, volume TBA of *LNCS*, page TBA. Springer, 2024. to appear.
50. S. Prabhu, G. Fedyukovich, K. Madhukar, and D. D'Souza. Specification Synthesis with Constrained Horn Clauses. In *PLDI*, pages 1203–1217. ACM, 2021.
51. S. Prabhu, K. Madhukar, and R. Venkatesh. Efficiently learning safety proofs from appearance as well as behaviours. In *SAS*, volume 11002 of *LNCS*, pages 326–343. Springer, 2018.
52. S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 295–306, 2008.
53. M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *European Symposium on Programming*, pages 451–471. Springer, 2013.
54. Z. Zhou, R. Dickerson, B. Delaware, and S. Jagannathan. Data-driven abductive inference of library specifications. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021.

55. H. Zhu, S. Magill, and S. Jagannathan. A data-driven CHC solver. In *PLDI*, pages 707–721. ACM, 2018.

Automated Software Verification of Hyperliveness

Raven Beutner(✉)

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
raven.beutner@cispa.de

Abstract. Hyperproperties relate multiple executions of a program and are commonly used to specify security and information-flow policies. Most existing work has focused on the verification of k-safety properties, i.e., properties that state that all k-tuples of execution traces satisfy a given property. In this paper, we study the automated verification of richer properties that combine universal and existential quantification over executions. Concretely, we consider $\forall^k\exists^l$ properties, which state that for all k executions, there exist l executions that, together, satisfy a property. This captures important non-k-safety requirements, including hyperliveness properties such as generalized non-interference, opacity, refinement, and robustness. We design an automated constraint-based algorithm for the verification of $\forall^k\exists^l$ properties. Our algorithm leverages a sound-and-complete program logic and a (parameterized) strongest postcondition computation. We implement our algorithm in a tool called `ForEx` and report on encouraging experimental results.

Keywords: Hyperproperties · Program Logic · Hoare Logic · Symbolic Execution · Constraint-based Verification · Predicate Transformer · Refinement · Strongest Postcondition · Underapproximation.

1 Introduction

Relational properties (also called hyperproperties [21]) move away from a traditional specification that considers all executions of a system in isolation and, instead, relate *multiple* executions. Hyperproperties are becoming increasingly important and have shown up in various disciplines, perhaps most prominently in information-flow control. Assume we are given a program $\mathbb{P}$ with high-security input h, low-security input l, and public output o, and we want to formally prove that the output of $\mathbb{P}$ does not leak information about h. One way to ensure this is to verify that $\mathbb{P}$ behaves deterministically in the low-security input l, i.e., if the low-security input is identical across *two* executions, so is $\mathbb{P}$'s output.

The above property is a typical example of a 2-safety property stating a requirement on all pairs of traces. More generally, a k-safety property requires that *all* k-tuples of executions, together, satisfy a given property. In the last decade, many approaches for the verification of k-safety properties have been proposed, based, e.g., on model-checking [55,33,31], abstract interpretation [43,41,4,44], symbolic execution [30], or program logics [7,56,28,60,49].

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 196–216, 2024.
https://doi.org/10.1007/978-3-031-57249-4_10

```
if (h > l) then
  o = l + ★ℕ
else
  x = ★ℕ
  if (x > l) then
    o = x
  else
    o = l
```

Fig. 1: Example program

However, for many relational properties, the implicit *universal* quantification found in k-safety properties is too restrictive. Consider the simple program in Figure 1 (taken from [12]), where $\star_{\mathbb{N}}$ denotes the nondeterministic choice of a natural number. This program clearly violates the 2-safety property discussed above as the nondeterminism influences the final value of o. Nevertheless, the program does not leak any information about the secret input h. To see this, assume the attacker observes some fixed low-security input-output pair (l, o), i.e., the attacker observes everything except the high-security input. The key observation is that (l, o) is possible for any possible high-security input, i.e., for every value of h, there *exists* some way to resolve the nondeterminism such that (l, o) is the observation made by the attacker. This information-flow policy – called generalized non-interference (GNI) [45] – requires a combination of universal and existential reasoning and thus cannot be expressed as a k-safety property.

FEHTs. In this paper, we study the automated verification of such (functional) $\forall^*\exists^*$ properties. Concretely, we consider specifications in a form we call Forall-Exist Hoare Tuples (FEHT) (also called *refinement quadruples* [5] or *RHLE triples* [26]), which have the form

$$\langle \Phi \rangle \mathbb{P}_1 \circledast \cdots \circledast \mathbb{P}_k \sim \mathbb{P}_{k+1} \circledast \cdots \circledast \mathbb{P}_{k+l} \langle \Psi \rangle,$$

where $\mathbb{P}_1, \ldots, \mathbb{P}_{k+l}$ are (possibly identical) programs and Φ, Ψ are first-order formulas that relate $k + l$ different program runs. The FEHT is valid if for *all* $k + l$ initial states that satisfy Φ, and for *all* possible executions of $\mathbb{P}_1, \ldots, \mathbb{P}_k$ there *exist* executions of $\mathbb{P}_{k+1}, \ldots, \mathbb{P}_{k+l}$ such that the final states satisfy Ψ. For example, GNI can be expressed as $\langle l_1 = l_2 \rangle \mathbb{P} \sim \mathbb{P} \langle o_1 = o_2 \rangle$, where l_1 and o_1 (resp. l_2 and o_2) refer to the value of l and o in the first (resp. second) program copy. That is, for *any* two initial states σ_1, σ_2 with identical values for l (but possibly different values for h), and *any* final state σ_1' reachable by executing $\mathbb{P}$ from σ_1, there *exists* some final state σ_2' (reachable from σ_2 by executing $\mathbb{P}$) that agrees with σ_1' in the value of o. The program in Figure 1 satisfies this FEHT. In the terminology of Clarkson and Schneider [21], GNI is a *hyperliveness* property, hence the name of our paper. Intuitively, the term hyperliveness stems from the fact that – due to the existential quantification in FEHTs – GNI reasons about the existence of a particular execution. Similar to the definition of liveness in temporal properties [2], we can, therefore, satisfy GNI by *adding* sufficiently many execution traces [22].

Verification Using a Program Logic. For *finite*-state hardware systems, many automated verification methods for hyperliveness properties (e.g., in the form of FEHTs) have been proposed [20,38,15,33,13,14,22]. In contrast, for *infinite*-state software, the verification of FEHTs is notoriously difficult; FEHTs mix quantification of different types, so we cannot employ purely over-approximate reasoning principles (as is possible for k-safety). Most existing approaches for software verification, therefore, require substantial user interaction, e.g., in the form of a custom Horn-clause template [57], a user-provided abstraction [12], or a deductive proof strategy [26,5]. See Section 6 for more discussion.

In this paper, we put forward an automatic algorithm for the verification of FEHTs. Our method is rooted in a novel *program logic*, which we call Forall-Exist Hoare Logic (FEHL) (in Section 3). Similar to many program logics for k-safety properties [56,19], our logic focuses on one of the programs involved in the verification at any given time (by, e.g., symbolically executing one step in one of the programs) and thus lends itself to automation. We show that FEHL is sound and complete (relative to a complete proof system for over- and under-approximate unary Hoare triples).

Automated Verification. Our verification algorithm – presented in Section 4 – then leverages FEHL for the analysis of FEHTs. During this analysis, the key algorithmic challenge is to find suitable instantiations for nondeterministic choices made in existentially quantified executions. Our algorithm avoids a direct instantiation and instead treats the outcome of the nondeterministic choice *symbolically*, allowing an instantiation at a later point in time. Formally, we define the concept of a *parametric assertion.* Instead of capturing a set of states, a parametric assertion defines a function that maps concrete values for a set of parameters (in our case, the nondeterministic choices in existentially quantified programs whose concrete instantiations we have postponed) to sets of states. Our algorithm then recursively computes a *parametric postcondition* and delegates the search for appropriate instantiations of the parameters to an SMT solver. Crucially, our algorithm only explores a restricted class of program alignments (as guided by FEHL). Therefore, the resulting constraints are ordinary (first-order) SMT formulas, which can be handled using off-the-shelf SMT solvers.

Implementation and Experiments. We implement our algorithm in a tool called `ForEx` and compare it with existing approaches for the verification of $\forall^*\exists^*$ properties (in Section 5). As `ForEx` can resort to highly optimized off-the-shelf SMT solvers, it outperforms existing approaches (which often rely on custom solving strategies) in many benchmarks.

2 Preliminaries

Programs. Let $\mathcal{V}$ be a set of program variables. We consider a simple (integer-valued) programming language generated by the following grammar.

$$\mathbb{P}, \mathbb{Q} := \texttt{skip} \mid x = e \mid \texttt{assume}(b) \mid \texttt{if}(b, \mathbb{P}, \mathbb{Q}) \mid \texttt{while}(b, \mathbb{P}) \mid \mathbb{P} \fatsemi \mathbb{Q} \mid x = \star$$

where $x \in \mathcal{V}$ is a variable, e is a (deterministic) arithmetic expressions over variables in $\mathcal{V}$, and b is a (deterministic) boolean expression. `skip` denotes the program that does nothing; $x = e$ assigns x the result of evaluating e; $\texttt{assume}(b)$

assumes that b holds, i.e., does not continue execution from states that do not satisfy b; $\texttt{if}(b, \mathbb{P}, \mathbb{Q})$ executes $\mathbb{P}$ if b holds and otherwise executes $\mathbb{Q}$; $\texttt{while}(b, \mathbb{P})$ executes $\mathbb{P}$ as long as b holds; $\mathbb{P} \fatsemi \mathbb{Q}$ executes $\mathbb{P}$ followed by $\mathbb{Q}$; and $x = \star$ assigns x some nondeterministically chosen integer. For an arithmetic expression e, we write $\mathit{Vars}(e) \subseteq \mathcal{V}$ for the set of all variables used in the expression.

We endow our language with a standard operational semantics operating on states $\sigma : \mathcal{V} \to \mathbb{Z}$. Given a program $\mathbb{P}$, we write $[\![\mathbb{P}]\!](\sigma, \sigma')$ whenever $\mathbb{P}$ – when executed from state σ – *can* terminate in state σ'. Our semantics is defined as expected, and we give a full definition in [10].

Given program states $\sigma_1 : \mathcal{V} \to \mathbb{Z}$ and $\sigma_2 : \mathcal{V}' \to \mathbb{Z}$ with $\mathcal{V} \cap \mathcal{V}' = \emptyset$, we write $\sigma_1 \oplus \sigma_2 : (\mathcal{V} \cup \mathcal{V}') \to \mathbb{Z}$ for the combined state, that behaves as σ_1 on $\mathcal{V}$ and as σ_2 on $\mathcal{V}'$. For $i \in \mathbb{N}$, we define $\mathcal{V}_i := \{x_i \mid x \in \mathcal{V}\}$ as a set of indexed program variables.

Assertions. An assertion Φ is a first-order formula over variables in $\mathcal{V}$ (or in the relational setting over $\bigcup_{i=1}^{k} \mathcal{V}_i$ for some k). Given a state σ, we write $\sigma \models \Phi$ if σ satisfies Φ. We assume that assertions stem from an arbitrarily expressive background theory such that every set of states can be expressed as a formula. This allows us to sidestep the issue of *expressiveness* in the sense of Cook [23] (see, e.g., [50,60,56] for similar treatments).

Hyperliveness Specifications. Our verification algorithm targets specifications that combine universal and existential quantification, similar to *RHLE triples* [26] and *refinement quadruples* [5]:

Definition 1. *A Forall-Exist Hoare Tuple (FEHT) has the form*

$$\langle \Phi \rangle \mathbb{P}_1 \circledast \cdots \circledast \mathbb{P}_k \sim \mathbb{P}_{k+1} \circledast \cdots \circledast \mathbb{P}_{k+l} \langle \Psi \rangle,$$

where Φ, Ψ are assertions over $\bigcup_{i=1}^{k+l} \mathcal{V}_i$, and $\mathbb{P}_1, \dots, \mathbb{P}_{k+l}$ are programs over variables $\mathcal{V}_1, \dots, \mathcal{V}_{k+l}$, respectively. The FEHT is valid *if for all states $\sigma_1, \dots, \sigma_{k+l}$ (with domains $\mathcal{V}_1, \dots, \mathcal{V}_{k+l}$, respectively) and $\sigma'_1, \dots, \sigma'_k$ such that $\bigoplus_{i=1}^{k+l} \sigma_i \models \Phi$ and $[\![\mathbb{P}_i]\!](\sigma_i, \sigma'_i)$ for all $i \in [1, k]$, there exist states $\sigma'_{k+1}, \dots, \sigma'_{k+l}$ such that $[\![\mathbb{P}_i]\!](\sigma_i, \sigma'_i)$ for all $i \in [k+1, k+l]$ and $\bigoplus_{i=1}^{k+l} \sigma'_i \models \Psi$.*

That is, we quantify universally over initial states for all $k + l$ programs (under the assumption that they, together, satisfy Φ) and also universally over executions of $\mathbb{P}_1, \dots, \mathbb{P}_k$. Afterward, we quantify *existentially* over executions of $\mathbb{P}_{k+1}, \dots, \mathbb{P}_{k+l}$ and require that the final states of all $k + l$ executions, together, satisfy the postcondition Ψ. A relational property usually refers to $k+l$ executions of the *same* program $\mathbb{P}$ (operating on variables in $\mathcal{V}$); we can model this by using α-renamed copies $\mathbb{P}_{\langle 1 \rangle}, \dots, \mathbb{P}_{\langle k+l \rangle}$ where each $\mathbb{P}_{\langle i \rangle}$ is obtained from $\mathbb{P}$ by replacing each variable $x \in \mathcal{V}$ with $x_i \in \mathcal{V}_i$. FEHTs capture a range of important properties, including e.g., non-inference [46], opacity [61], GNI [45], refinement [59], software doping [16], and robustness [18]. It is easy to see that FEHTs can also express (purely universal) k-safety properties over programs $\mathbb{P}_1, \dots, \mathbb{P}_k$ as $\langle \Phi \rangle \mathbb{P}_1 \circledast \cdots \circledast \mathbb{P}_k \sim \epsilon \langle \Psi \rangle$, where ϵ denotes the empty sequence of programs.

$$\texttt{($\forall$-Reorder)}\quad \frac{\vdash \langle \Phi \rangle \overline{\chi_{\forall_2}} \circledast \overline{\chi_{\forall_1}} \sim \overline{\chi_\exists} \langle \Psi \rangle}{\vdash \langle \Phi \rangle \overline{\chi_{\forall_1}} \circledast \overline{\chi_{\forall_2}} \sim \overline{\chi_\exists} \langle \Psi \rangle} \qquad \texttt{($\forall$-Skip-I)}\quad \frac{\vdash \langle \Phi \rangle \mathbb{P} \fatsemi \texttt{skip} \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle}{\vdash \langle \Phi \rangle \mathbb{P} \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle} \qquad \texttt{($\forall$-Skip-E)}\quad \frac{\vdash \langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle}{\vdash \langle \Phi \rangle \texttt{skip} \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle}$$

$$\texttt{($\forall$-If)}\quad \frac{\begin{array}{c}\vdash \langle \Phi \land b \rangle \mathbb{P}_1 \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle \\ \vdash \langle \Phi \land \neg b \rangle \mathbb{P}_2 \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle\end{array}}{\vdash \langle \Phi \rangle \texttt{if}(b, \mathbb{P}_1, \mathbb{P}_2) \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle} \qquad \texttt{($\forall$-Step)}\quad \frac{\begin{array}{c}\vdash \{\Phi\} \mathbb{P}_1 \{\Phi'\} \\ \vdash \langle \Phi' \rangle \mathbb{P}_2 \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle\end{array}}{\vdash \langle \Phi \rangle \mathbb{P}_1 \fatsemi \mathbb{P}_2 \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle} \qquad \texttt{($\exists$-Step)}\quad \frac{\begin{array}{c}\vdash [\Phi] \mathbb{P}_1 [\Phi'] \\ \vdash \langle \Phi' \rangle \overline{\chi_\forall} \sim \mathbb{P}_2 \circledast \overline{\chi_\exists} \langle \Psi \rangle\end{array}}{\vdash \langle \Phi \rangle \overline{\chi_\forall} \sim \mathbb{P}_1 \fatsemi \mathbb{P}_2 \circledast \overline{\chi_\exists} \langle \Psi \rangle}$$

$$\texttt{(Done)}\quad \vdash \langle \Phi \rangle \epsilon \sim \epsilon \langle \Phi \rangle \qquad \texttt{($\forall$-Assume)}\quad \frac{\vdash \langle \Phi \land b \rangle \mathbb{P} \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle}{\vdash \langle \Phi \rangle \texttt{assume}(b) \fatsemi \mathbb{P} \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle} \qquad \texttt{($\exists$-Assume)}\quad \frac{\Phi \Rightarrow b \qquad \vdash \langle \Phi \rangle \overline{\chi_\forall} \sim \mathbb{P} \circledast \overline{\chi_\exists} \langle \Psi \rangle}{\vdash \langle \Phi \rangle \overline{\chi_\forall} \sim \texttt{assume}(b) \fatsemi \mathbb{P} \circledast \overline{\chi_\exists} \langle \Psi \rangle}$$

$$\texttt{($\forall$-Choice)}\quad \frac{\vdash \langle \exists x.\, \Phi \rangle \mathbb{P} \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle}{\vdash \langle \Phi \rangle x = \star \fatsemi \mathbb{P} \circledast \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle} \qquad \texttt{($\exists$-Choice)}\quad \frac{x \not\in \mathit{Vars}(e) \qquad \vdash \langle (\exists x.\, \Phi) \land x = e \rangle \overline{\chi_\forall} \sim \mathbb{P} \circledast \overline{\chi_\exists} \langle \Psi \rangle}{\vdash \langle \Phi \rangle \overline{\chi_\forall} \sim x = \star \fatsemi \mathbb{P} \circledast \overline{\chi_\exists} \langle \Psi \rangle}$$

Fig. 2: Selection of core proof rules of FEHL

3 Forall-Exist Hoare Logic

The verification steps of our constraint-based algorithm (presented in Section 4) are guided by the proof rules of a novel program logic operating on FEHTs, which we call Forall-Exist Hoare Logic (FEHL).

3.1 Core Rules

We depict a selection of core rules in Figure 2; a full overview can be found in [10]. We write $\overline{\chi_\forall}$ (resp. $\overline{\chi_\exists}$) to abbreviate a list $\mathbb{P}_1 \circledast \cdots \circledast \mathbb{P}_k$ of programs that are universally (resp. existentially) quantified. Rule `(∀-Reorder)` allows for the reordering of universally quantified programs; `(∀-Skip-I)` rewrites a program $\mathbb{P}$ into $\mathbb{P} \fatsemi$ `skip`; `(∀-Skip-E)` removes a single `skip`-instruction; and `(Done)` derives a FEHL with an empty program sequence. Using `skip`-insertions and reordering (and the analogous rules for existentially quantified programs), we can always bring a program in the form $\mathbb{P}_1 \fatsemi \mathbb{P}_2$, targeted by the remaining rules. Rule `(∀-If)` embeds the branching condition of a conditional into the preconditions of both branches. Rules `(∀-Step)` and `(∃-Step)` allow us to resort to unary reasoning over parts of the program. These rules make the multiplicity of techniques developed for unary reasoning (e.g., symbolic execution [40] and predicate transformers [27]) applicable to the verification of hyperproperties in the form of FEHTs. For universally quantified programs of the form $\mathbb{P}_1 \fatsemi \mathbb{P}_2$, `(∀-Step)` requires an auxiliary assertion Φ' that should hold after *all* executions of $\mathbb{P}_1$ from Φ. We can express this using the standard (non-relational) Hoare triple (HT) $\{\Phi\}\mathbb{P}_1\{\Phi'\}$ [37]. The second premise then ensures that the remaining

$$\texttt{(Loop-Counting)}\quad \frac{\begin{array}{c} k \geq 1,\ B \geq 1 \\ c_1, \ldots, c_{k+l} \in [1, B] \\ \mathbb{I}_1, \ldots, \mathbb{I}_{B+1} \\ \Phi \Rightarrow \mathbb{I} \\ \mathbb{I} \Rightarrow \bigwedge_{i=2}^{k+l}(b_1 \leftrightarrow b_i) \\ \mathbb{I} = \mathbb{I}_1 = \mathbb{I}_{B+1} \end{array} \qquad \begin{array}{c} \left[\vdash \left\langle \mathbb{I}_j \wedge \bigwedge_{i=1 \mid c_i \geq j}^{k+l} b_i \right\rangle \mathop{\circledast}_{i=1 \mid c_i \geq j}^{k} \mathbb{P}_i \sim \mathop{\circledast}_{i=k+1 \mid c_i \geq j}^{k+l} \mathbb{P}_i \left\langle \mathbb{I}_{j+1} \wedge \bigwedge_{i=1 \mid c_i > j}^{k+l} b_i \right\rangle \right]_{j=1}^{B} \\ \vdash \left\langle \mathbb{I} \wedge \bigwedge_{i=1}^{k+l} \neg b_i \right\rangle \mathop{\circledast}_{i=1}^{k} \mathbb{Q}_i \circledast \overline{\chi_\forall} \sim \mathop{\circledast}_{i=k+1}^{k+l} \mathbb{Q}_i \circledast \overline{\chi_\exists} \left\langle \Psi \right\rangle \end{array}}{\vdash \langle \Phi \rangle \mathop{\circledast}_{i=1}^{k} \texttt{while}(b_i, \mathbb{P}_i) \fatsemi \mathbb{Q}_i \circledast \overline{\chi_\forall} \sim \mathop{\circledast}_{i=k+1}^{k+l} \texttt{while}(b_i, \mathbb{P}_i) \fatsemi \mathbb{Q}_i \circledast \overline{\chi_\exists} \langle \Psi \rangle}$$

Fig. 3: Counting-based loop rule for FEHL

FEHT (after $\mathbb{P}_1$ has been executed) holds. For existentially quantified programs, we, instead, employ an underapproximation. In (∃-Step), we, again, execute $\mathbb{P}_1$ but use an *Under-Approximate Hoare triple* (UHT) $[\Phi]\mathbb{P}_1[\Phi']$. The UHT $[\Phi]\mathbb{P}_1[\Phi']$ holds if for all states σ with $\sigma \models \Phi$, there *exists* a state σ' such that $[\![\mathbb{P}_1]\!](\sigma, \sigma')$ and $\sigma' \models \Phi'$.

Remark 1. UHTs behave similar to *Incorrectness Triples* (ITs) [50,58] in that they reason about the existence of a particular set of executions. The key difference is that ITs reason backward (all states in Φ' are reachable from some state in Φ), whereas UHTs reason in a forward direction (all states in Φ can reach Φ'). See, e.g., *Lisbon Triples* [47, §5] and *Outcome Triples* [62] for related approaches. We will later show that FEHL is complete when equipped with some complete proof system for UHTs (cf. Theorem 2). In [10], we show that there exists at least one complete proof system for UHTs. △

For assume statements, (∀-Assume) strengthens the precondition by the assumed expression b; any state that does not satisfy b causes a (universally quantified) execution to halt and renders the FEHT vacuously valid. In contrast, (∃-Assume) assumes that all states in Φ satisfy b; if any state in Φ does not satisfy b, the FEHT is invalid. Likewise, the handling of a nondeterministic assignment $x = \star$ differs based on whether we consider a universally quantified or existentially quantified program. In the former case, (∀-Choice) removes all knowledge about the value of x within the precondition by quantifying x existentially (thus enlarging the precondition). In the latter (existentially quantified) case, we can, in a forward-style execution, choose *any* concrete value for x. (∃-Choice) formalizes this intuition: we first invalidate all knowledge about x and then assert that $x = e$ for some arbitrary expression e that does not depend on x. In our automated analysis (cf. Section 4), we use (∃-Choice), but – instead of fixing some concrete value (or expression) at application time – we postpone the concrete instantiation by treating the value *symbolically.*

3.2 Asynchronous Loop Reasoning

A particular challenge when reasoning about relational properties is the alignment of loops. In FEHL, we propose a novel counting-based loop rule that sup-

$$\mathbb{P}_1 := \left\{\begin{array}{l} \texttt{y_1 = x_1}\fatsemi \\ \texttt{while (y_1 > 0)} \\ \quad \texttt{y_1 = y_1 - 1}\fatsemi \\ \quad \texttt{x_1 = 4 * x_1} \end{array}\right.$$

$$\mathbb{P}_2 := \left\{\begin{array}{l} \texttt{y_2 = 2 * x_2}\fatsemi \\ \texttt{while (y_2 > 0)} \\ \quad \texttt{z_2 = $\star$}\fatsemi \\ \quad \texttt{y_2 = y_2 - z_2}\fatsemi \\ \quad \texttt{x_2 = 2 * x_2} \end{array}\right.$$

(a)

$$\left\langle \begin{array}{c} \mathbb{I}_1 \land \\ y_1 > 0 \land \\ y_2 > 0 \end{array} \right\rangle \begin{array}{l} \texttt{y_1 = y_1 - 1}\fatsemi \\ \texttt{x_1 = 4 * x_1} \end{array} \sim \begin{array}{l} \texttt{z_2 = $\star$}\fatsemi \\ \texttt{y_2 = y_2 - z_2}\fatsemi \\ \texttt{x_2 = 2 * x_2} \end{array} \left\langle \begin{array}{c} \mathbb{I}_2 \land \\ y_2 > 0 \end{array} \right\rangle$$

(b)

$$\left\langle \begin{array}{c} \mathbb{I}_2 \land \\ y_2 > 0 \end{array} \right\rangle \epsilon \sim \begin{array}{l} \texttt{z_2 = $\star$}\fatsemi \\ \texttt{y_2 = y_2 - z_2}\fatsemi \\ \texttt{x_2 = 2 * x_2} \end{array} \left\langle \mathbb{I}_3 \right\rangle$$

(c)

Fig. 4: In Figure 4a, we depict two example programs. In Figures 4b and 4c, we give two intermediate FEHT verification obligations (cf. Example 1).

ports asynchronous alignments while still admitting good automation. Consider the rule **(Loop-Counting)** (in Figure 3), which assumes $k \geq 1$ universally and l existentially quantified loops. The rule requires a loop invariant $\mathbb{I}$ that **(1)** is implied by the precondition ($\Phi \Rightarrow \mathbb{I}$), **(2)** ensures simultaneous termination of all loops ($\mathbb{I} \Rightarrow \bigwedge_{i=2}^{k+l}(b_1 \leftrightarrow b_i)$), and **(3)** is strong enough to establish the postcondition for the program suffixes $\mathbb{Q}_1, \ldots, \mathbb{Q}_{k+l}$ executed after the loops. The key difference from a simple synchronous traversal is that, in each "iteration", we execute the bodies of the loops for possibly different numbers of times. Concretely, **(Loop-Counting)** asks for natural numbers $c_1, \ldots, c_{k+l}$ (ranging between 1 and some arbitrary upper bound B), and – starting from the invariant $\mathbb{I}$ – we execute each $\mathbb{P}_i$ c_i times. Crucially, we need to make sure that each $\mathbb{P}_i$ will execute *at least* c_i times, i.e., the guard b_i holds after each of the first $c_i - 1$ executions. In particular, we cannot naïvely analyze c_i copies of $\mathbb{P}_i$ composed via $\fatsemi$ as this might introduce additional executions of $\mathbb{P}_i$ that would not happen in $\texttt{while}(b_i, \mathbb{P}_i)$. To ensure this, **(Loop-Counting)** demands $B+1$ intermediate assertions $\mathbb{I}_1, \ldots, \mathbb{I}_{B+1}$. In the jth iteration (for $1 \leq j \leq B$), we (symbolically) execute – from $\mathbb{I}_j$ – all loop bodies $\mathbb{P}_i$ that we want to execute at least j times (i.e., all loop bodies $\mathbb{P}_i$ where $c_i \geq j$). We require that **(1)** the postcondition $\mathbb{I}_{j+1}$ is derivable, and **(2)** the guards of all loops that we want to execute *more* than j times (i.e., loops where $c_i > j$) evaluate to true.

Example 1. Consider the two example programs $\mathbb{P}_1, \mathbb{P}_2$ in Figure 4a and the FEHT $\langle x_1 = x_2 \rangle \mathbb{P}_1 \sim \mathbb{P}_2 \langle x_1 = x_2 \rangle$. To see that this FEHT is valid, we can, in each loop iteration, always choose $z_2 = 1$. In this case, $\mathbb{P}_1$ quadruples the value of x_1 for x_1 times and $\mathbb{P}_2$ doubles the value of x_2 for $2x_2$ times, which, assuming $x_1 = x_2$, computes the same result ($x_1 = x_2 \rightarrow 4^{x_1}x_1 = 2^{2x_2}x_2$). Verifying this example automatically is challenging as both loops are executed a different number of times, so we cannot align the loops in lockstep. Likewise, computing independent (unary) summaries of both loops requires complex non-linear reasoning.

Instead, `(Loop-Counting)` enables an asynchronous alignment: After applying `(`$\forall$`-Step)` and `(`$\exists$`-Step)`, we are left with precondition $x_1 = x_2 \land y_2 = 2y_1$. We use `(Loop-Counting)` and align the loops such that every loop iteration in $\mathbb{P}_1$ is matched by *two* iterations in $\mathbb{P}_2$, which allows us to use a simple (linear) invariant. We set $c_1 := 1, c_2 := 2$ and define $\mathbb{I} := x_1 = x_2 \land y_2 = 2y_1$, $\mathbb{I}_1 := \mathbb{I}_3 := \mathbb{I}$, and $\mathbb{I}_2 := x_1 = 2x_2 \land y_2 = 2y_1 + 1$. Note that $\mathbb{I}$ implies the desired postcondition ($x_1 = x_2$). To establish that $\mathbb{I}$ serves as an invariant, we need to discharge the two proof obligations depicted in Figures 4b and 4c. The obligation in Figure 4b (corresponding to iteration $j = 1$) establishes that **(1)** $\mathbb{I}_2$ is a provable postcondition after executing both loop bodies from $\mathbb{I}_1$ and **(2)** that the loop in $\mathbb{P}_2$ will execute at least one more time, i.e., $y_2 > 0$. We can easily discharge this FEHT using `(`$\forall$`-Step)`, `(`$\exists$`-Step)`, and `(`$\exists$`-Choice)` by choosing z_2 to be 1 (note that if $y_2 = 2y_1$ and $y_2 > 0$, then $y_2 - 1 > 0$). The obligation in Figure 4c corresponds to iteration $j = 2$, where we only execute the body of $\mathbb{P}_2$. We can, again, easily discharge this FEHT using `(`$\exists$`-Step)` and `(`$\exists$`-Choice)` (again, choosing z_2 to be 1). △

3.3 Soundness and Completeness

We can show that our proof system is sound and complete:

Theorem 1 (Soundness). *Assume that* $\vdash \{\cdot\} \cdot \{\cdot\}$ *and* $\vdash [\cdot] \cdot [\cdot]$ *are sound proof systems for HTs and UHTs, respectively. If* $\vdash \langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$ *then* $\langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$ *is valid.*

Theorem 2 (Completeness). *Assume that* $\vdash \{\cdot\} \cdot \{\cdot\}$ *and* $\vdash [\cdot] \cdot [\cdot]$ *are complete proof systems for HTs and UHTs, respectively. If* $\langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$ *is valid then* $\vdash \langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$.

Completeness follows easily by making extensive use of *unary* reasoning via (U)HTs, similar to the completeness-proof of relational Hoare logic for k-safety properties [49]. In fact, `(`$\forall$`-Step)`, `(`$\exists$`-Step)`, `(Done)` along with the reordering rules `(`$\forall$`-Reorder)`, `(`$\forall$`-Skip-I)`, and `(`$\forall$`-Skip-E)` (and their analogous counterparts for existentially quantified programs) already suffice for completeness (see [10]). In the following, we leverage the *soundness* of FEHL's rules to guide our automated verification.

4 Automated Verification of Hyperliveness

Our automated verification algorithm for FEHTs follows a *strongest postcondition* computation, as is widely used in the verification of non-relational properties [1,36,51] and k-safety properties [56,19]. However, due to the inherent presence of *existential* quantification in FEHT, the strongest postcondition does, in general, not exist. For example, both $\langle \top \rangle \epsilon \sim x = \star \langle x = 1 \rangle$ and $\langle \top \rangle \epsilon \sim x = \star \langle x = 2 \rangle$ are valid but $\langle \top \rangle \epsilon \sim x = \star \langle x = 1 \land x = 2 \equiv \bot \rangle$ is clearly not. Instead, our algorithm uses the proof rules of FEHL and treats the concrete value for nondeterministic choices in existentially quantified executions symbolically. I.e., we view the outcome as a fresh variable (called a *parameter*) that can be instantiated later. This idea of instating nondeterminism at a later point in time has already found successful application in many areas, such as existential variables in Coq or symbolic execution [40]. Our analysis brings these techniques to the realm of hyperproperty verification, which we show to yield an effective automated verification algorithm. In the following, we formally introduce parametric assertions and postconditions (in Section 4.1) and show how we can compute them using the rules of FEHL (in Sections 4.2 and 4.3).

4.1 Parametric Assertions and Postconditions

We assume that $\mathfrak{P} = \{\mu_1, \dots, \mu_n\}$ is a set of *parameters*. In FEHTs, we use assertions (formulas) over $\bigcup_{i=1}^{k+l} \mathcal{V}_i$, which we interpret as sets of (relational) states. A parametric assertion generalizes this by viewing an assertion as a function mapping *into* sets of (relational) states. Formally, a *parametric assertion* is a pair $(\varXi, \mathcal{C})$ where $\varXi$ is a formula over $\bigcup_{i=1}^{k+l} \mathcal{V}_i \cup \mathfrak{P}$ (called the *function-formula*), and $\mathcal{C}$ is a formula over $\mathfrak{P}$ (called the *restriction-formula*).

Given a function-formula $\varXi$ (over $\bigcup_{i=1}^{k+l} \mathcal{V}_i \cup \mathfrak{P}$) and a *parameter evaluation* $\kappa : \mathfrak{P} \to \mathbb{Z}$, we define $\varXi[\kappa]$ as the formula over $\bigcup_{i=1}^{k+l} \mathcal{V}_i$ where we fix concrete values for all parameters based on κ. We can thus view $\varXi$ as a function mapping each parameter evaluation κ to the set of states encoded by $\varXi[\kappa]$. During our (forward style) analysis, we will use parameters to postpone nondeterministic choices in existentially quantified programs. Intuitively, for every parameter evaluation κ (i.e., any retrospective choice of the nondeterministic outcome), $\varXi[\kappa]$ should describe the reachable states (i.e., strongest postcondition) under those specific outcomes. However, not all concrete values for the parameters are valid in the sense that they correspond to nondeterministic outcomes that result in actual executions. To mitigate this, a parametric assertion $(\varXi, \mathcal{C})$ includes a restriction-formula $\mathcal{C}$ (over $\mathfrak{P}$) which *restrict the domain* of the function encoded by $\varXi$, i.e., we only consider those parameter evaluations that satisfy $\mathcal{C}$.

Example 2. Before proceeding with a formal development, let us discuss parametric assertions informally using an example. Let $\mathbb{P}_1 := x = \star \fatsemi \texttt{assume}(x \geq 9)$ and $\mathbb{P}_2 := y = \star \fatsemi \texttt{assume}(y \geq 2)$ and assume we want to prove the FEHT $\langle \top \rangle \mathbb{P}_1 \sim \mathbb{P}_2 \langle x = y \rangle$. To verify this tuple in a principled way, we are interested in potential postconditions $\varPsi$, i.e., assertions $\varPsi$ such that $\langle \top \rangle \mathbb{P}_1 \sim \mathbb{P}_2 \langle \varPsi \rangle$ is valid. For example, both $\varPsi_1 = x \geq 9 \land y = 2$ and $\varPsi_2 = x \geq 9 \land y = 3$ are valid postconditions, but – as already seen before – there does not exist a strongest assertion. Instead, we capture *multiple* postconditions using the parametric assertion $(\varXi, \mathcal{C})$ where $\varXi := x \geq 9 \land y = \mu$ and $\mathcal{C} := \mu \geq 2$ for some fresh parameter $\mu \in \mathfrak{P}$; we say $(\varXi, \mathcal{C})$ is a *parametric postcondition* for $(\top, \mathbb{P}_1, \mathbb{P}_2)$ (cf. Definition 2). Intuitively, we have used the parameter μ instead of assigning some fixed integer to y. For every concrete parameter evaluation $\kappa : \{\mu\} \to \mathbb{Z}$ such that $\kappa \models \mathcal{C}$, formula $\Xi[\kappa]$ defines the reachable states when using $\kappa(\mu)$ for the choice of y. Observe how formula $\mathcal{C} = \mu \geq 2$ restricts the possible set of parameter values, i.e., we may only choose a value for y such that `assume`$(y \geq 2)$ holds. △

Definition 2. *A* parametric postcondition for $(\Phi, \mathbb{P}_1, \dots, \mathbb{P}_{k+l})$ *is a parametric assertion* $(\Xi, \mathcal{C})$ *with the following conditions. For all states* $\sigma_1, \dots, \sigma_{k+l}$*, and* $\sigma'_1, \dots, \sigma'_k$ *such that* $\bigoplus_{i=1}^{k+l} \sigma_i \models \Phi$ *and* $[\![\mathbb{P}_i]\!](\sigma_i, \sigma'_i)$ *for all* $i \in [1, k]$ *and any parameter evaluation* κ *such that* $\kappa \models \mathcal{C}$ *the following holds:* ***(1)*** *There exist states* $\sigma'_{k+1}, \dots, \sigma'_{k+l}$ *such that* $\bigoplus_{i=1}^{k+l} \sigma'_i \models \Xi[\kappa]$*, and* ***(2)*** *For every* $\sigma'_{k+1}, \dots, \sigma'_{k+l}$ *such that* $\bigoplus_{i=1}^{k+l} \sigma'_i \models \Xi[\kappa]$ *we have* $[\![\mathbb{P}_i]\!](\sigma_i, \sigma'_i)$ *for all* $i \in [k+1, k+l]$*.*

Condition **(1)** captures that no parameter evaluation may restrict universally quantified executions, i.e., if we fix any parameter evaluation κ and reachable final states for the universally quantified programs, $\Xi[\kappa]$ remains satisfiable. This effectively states that $\Xi[\kappa]$ *over-approximates* the set of executions of universally quantified programs. Condition **(2)** requires that all executions of existentially quantified programs allowed under a particular parameter evaluation are also valid executions, i.e., for any fixed parameter evaluation κ, $\Xi[\kappa]$ *under-approximates* the set of executions of the existentially quantified programs.

We can use parametric postconditions to prove FEHTs:

Theorem 3. *Let* $(\Xi, \mathcal{C})$ *be a parametric postcondition for* $(\Phi, \mathbb{P}_1, \dots, \mathbb{P}_{k+l})$*. If*

$$\forall_{x \in \mathcal{V}_1 \cup \dots \cup \mathcal{V}_k} x.\, \exists_{\mu \in \mathfrak{P}} \mu.\ \ \mathcal{C} \ \wedge \ \forall_{x \in \mathcal{V}_{k+1} \cup \dots \cup \mathcal{V}_{k+l}} x.\, (\Xi \Rightarrow \Psi)$$

holds, then the FEHT $\langle \Phi \rangle \mathbb{P}_1 \circledast \dots \circledast \mathbb{P}_k \sim \mathbb{P}_{k+1} \circledast \dots \circledast \mathbb{P}_{k+l} \langle \Psi \rangle$ *is valid.*

Here, we universally quantify over final states in $\mathbb{P}_1, \dots, \mathbb{P}_k$ and existentially quantify over parameter evaluations that satisfy $\mathcal{C}$ (recall that $\mathcal{C}$ only refers to $\mathfrak{P}$). The choice of the parameters can thus depend on the final states of universally quantified programs (as in the semantics of FEHTs). Afterward, we quantify (again universally) over final states of $\mathbb{P}_{k+1}, \dots, \mathbb{P}_{k+l}$ and state that if Ξ holds, so does the postcondition Ψ.

Example 3. Consider the FEHT and parametric postcondition from Example 2. Following Theorem 3, we construct the SMT formula $\forall x.\, \exists \mu.\, \mu \geq 2 \wedge \forall y.\, \big((x \geq 9 \wedge y = \mu) \Rightarrow x = y\big)$. This formula holds; the FEHT is valid. △

Note that $(\Xi, \bot)$ is always a parametric postcondition: no parameter evaluation satisfies $\bot$, so the conditions in Definition 2 are vacuously satisfied. However, $(\Xi, \bot)$ is useless when it comes to proving FEHTs via Theorem 3.

4.2 Generating Parametric Postconditions

Algorithm 1 computes a parametric postcondition based on the proof rules of FEHL from Section 3. As input, Algorithm 1 expects a formula Φ over $\bigcup_{i=1}^{k+l} \mathcal{V}_i \cup \mathfrak{P}$ – think of Φ as a precondition already containing some parameters – and two program lists $\overline{\chi_\forall}$ and $\overline{\chi_\exists}$. It outputs a parametric postcondition.

Algorithm 1 Parametric postcondition generation for FEHT verification

```
1  def genpp($\Phi$,$\overline{\chi_\forall}$,$\overline{\chi_\exists}$):
2   if $\overline{\chi_\forall} = \overline{\chi_\exists} = \epsilon$:
3    return ($\Phi$,$\top$) //(Done)
4   else if $\forall \mathbb{P} \in \overline{\chi_\forall} \cup \overline{\chi_\exists}. \mathbb{P} = \texttt{while}(\_,\_) \fatsemi \_$:
5    return genppLoops($\Phi$, $\overline{\chi_\forall}$, $\overline{\chi_\exists}$)
6   else if $\exists \mathbb{P} \in \overline{\chi_\forall}. \mathbb{P} \neq \texttt{while}(\_,\_) \fatsemi \_$:
7    // Take a step in $\overline{\chi_\forall}$
8    match $\overline{\chi_\forall}$:
9     | $\texttt{skip} \circledast \overline{\chi_\forall}'$: //($\forall$-Skip-E)
10      return genpp($\Phi$,$\overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
11    | $\texttt{skip} \fatsemi \mathbb{P} \circledast \overline{\chi_\forall}'$: //($\forall$-Step)
12      return genpp($\Phi$,$\mathbb{P} \circledast \overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
13    | $(\mathbb{P}_1 \fatsemi \mathbb{P}_2) \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\forall}'$:
14      return genpp($\Phi$,$\mathbb{P}_1 \fatsemi (\mathbb{P}_2 \fatsemi \mathbb{P}_3) \circledast \overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
15    | $\mathbb{P} \circledast \overline{\chi_\forall}'$ when $\mathbb{P} \neq \_ \fatsemi \_$: //($\forall$-Skip-I)
16      return genpp($\Phi$,$\mathbb{P} \fatsemi \texttt{skip} \circledast \overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
17    | $x = e \fatsemi \mathbb{P} \circledast \overline{\chi_\forall}'$: //($\forall$-Step)
18      $\Phi'$ := $\exists x'.\Phi[x'/x] \land x = e[x'/x]$
19      return genpp ($\Phi'$,$\mathbb{P} \circledast \overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
20    | $\texttt{if}(b, \mathbb{P}_1, \mathbb{P}_2) \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\forall}'$:
21      //($\forall$-If)
22      ($\Xi_1$,$\mathcal{C}_1$) :=
23       genpp($\Phi \land b$,$\mathbb{P}_1 \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
24      ($\Xi_2$,$\mathcal{C}_2$) :=
25       genpp($\Phi \land \neg b$,$\mathbb{P}_2 \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
26      return ($\Xi_1 \lor \Xi_2$,$\mathcal{C}_1 \land \mathcal{C}_2$)
27    | $\texttt{assume}(b) \fatsemi \mathbb{P} \circledast \overline{\chi_\forall}'$: //($\forall$-Assume)
28      return genpp($\Phi \land b$,$\mathbb{P} \circledast \overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
29    | $x = \star \fatsemi \mathbb{P} \circledast \overline{\chi_\forall}'$: //($\forall$-Choice)
30      $\Phi'$ := $\exists x. \Phi$
31      return genpp($\Phi'$,$\mathbb{P} \circledast \overline{\chi_\forall}'$,$\overline{\chi_\exists}$)
32    | $\mathbb{P} \circledast \overline{\chi_\forall}'$: //($\forall$-Reorder)
33      return genpp($\Phi$,$\overline{\chi_\forall}' \circledast \mathbb{P}$,$\overline{\chi_\exists}$)
34 else:
35  // Take a step in $\overline{\chi_\exists}$
36  match $\overline{\chi_\exists}$:
37   | $\texttt{skip} \circledast \overline{\chi_\exists}'$ | $\texttt{skip} \fatsemi \mathbb{P} \circledast \overline{\chi_\exists}'$
38   | $(\mathbb{P}_1 \fatsemi \mathbb{P}_2) \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\exists}'$
39   | $\mathbb{P} \circledast \overline{\chi_\exists}'$ when $\mathbb{P} \neq \_ \fatsemi \_$
40   | $x = e \fatsemi \mathbb{P} \circledast \overline{\chi_\exists}'$
41   | $\texttt{if}(b, \mathbb{P}_1, \mathbb{P}_2) \fatsemi \mathbb{P}_3 \circledast \overline{\chi_\exists}'$:
42     //As in lines 9, 11, 17
43     //20, 13, and 15
44   | $\texttt{assume}(b) \fatsemi \mathbb{P} \circledast \overline{\chi_\exists}'$:
45     //($\exists$-Assume)
46     $\mathcal{C}_{assume}$ :=
47       $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}} x. (\Phi \Rightarrow b)$
48     ($\Xi$,$\mathcal{C}$) :=
49       genpp($\Phi \land b$,$\overline{\chi_\forall}$,$\mathbb{P} \circledast \overline{\chi_\exists}'$)
50     return ($\Xi$,$\mathcal{C} \land \mathcal{C}_{assume}$)
51   | $x = \star \fatsemi \mathbb{P} \circledast \overline{\chi_\exists}'$: //($\exists$-Choice)
52     $\mu$ := freshParameter()
53     $\Phi'$ := $(\exists x.\Phi) \land x = \mu$
54     return genpp($\Phi'$,$\overline{\chi_\forall}$,$\mathbb{P} \circledast \overline{\chi_\exists}'$)
55   | $\mathbb{P} \circledast \overline{\chi_\exists}'$:
56     return genpp($\Phi$,$\overline{\chi_\forall}$,$\overline{\chi_\exists}' \circledast \mathbb{P}$)
```

Remark 2. For intuition, it is oftentimes helpful to consider Φ as a parameter-free formula over $\bigcup_{i=1}^{k+l} \mathcal{V}_i$. In this case, most of our steps correspond to the computation of the strongest postcondition [27,56,19] in a purely universal (k-safety) setting. △

Our algorithm analyses the structure of each program and applies the insights from FEHL: If $\overline{\chi_\forall}$ and $\overline{\chi_\exists}$ are empty, we return $(\Phi, \top)$ (line 3), i.e., we do not place any restrictions on the parameters. In case all programs are loops (line 5), we invoke a subroutine `genppLoops` (discussed in Section 4.3). Otherwise, some program has a non-loop statement at the top level, allowing further symbolic analysis. We consider possible steps in $\overline{\chi_\forall}$ (lines 7-33) and in $\overline{\chi_\exists}$ (lines 35-56).

We first consider the case where a universally quantified program has a non-loop statement at its top level (lines 7-33). In lines 9, 11, 13, and 15, we bring the first program into the form $\mathbb{P}_1 \fatsemi \mathbb{P}_2$ where $\mathbb{P}_1 \neq \_ \fatsemi \_$ by potentially inserting `skip` statements in line 15. For a program $x = e \fatsemi \mathbb{P}$ (line 17), we use `(∀-Step)` to handle the assignment. Here, we can compute the strongest postcondition of the assignment as $\exists x'.\Phi[x'/x] \land x = e[x'/x]$ (using Floyd's forward running rule [35]). For conditionals (line 20), we analyze both branches under the strengthened precondition. As our analysis operates on parametric assertions, some of the parameters found in the precondition Φ can be restricted in *both branches*. After we have computed a parametric postcondition for each branch, we therefore combine them into a parametric postcondition for the entire program by constructing the disjunction of the function-formulas Ξ_1 and Ξ_2 (describing the set of states reachable in either of the branches), and conjoining the restriction-formulas $\mathcal{C}_1$ and $\mathcal{C}_2$. For assume statements (line 27), we strengthen the precondition. For nondeterministic assignments $x = \star$ (line 29), we invalidate all knowledge about x. If a program matches none of the previous cases (line 33), it must be of the form `while`$(\_, \_) ⨾ \_$, and we move it to the end of $\overline{\chi_\forall}$, continuing the analysis of the renaming programs in the next recursive iteration. If no universally quantified program can be analyzed further, we continue the investigation with existentially quantified ones (lines 35-56). Many cases are analogous to the treatment in universally quantified programs (lines 37-43), but some cases are handled fundamentally differently: If we encounter an assume statement `assume`(b) (line 45), we need to certify that b holds in all states in Φ (cf. (`∃-Assume`)). As we already hinted in Example 2, we accomplish this by restricting the viable set of parameters in Φ, i.e., we restrict the domain of the function formula Φ. Concretely, we consider the formula $\mathcal{C}_{assume} := \forall_{x \in \mathcal{V}_1 \cup \dots \cup \mathcal{V}_{k+l}} x. (\Phi \Rightarrow b)$ (which is a formula over $\mathfrak{P}$) that characterizes exactly those parameters that ensure that all states in Φ satisfy b. After analyzing the remaining programs, we then conjoin $\mathcal{C}_{assume}$ with the remaining restrictions.

Remark 3. As in Remark 2, we can consider the case where Φ contains no parameter. In this case, $\mathcal{C}_{assume}$ is a variable-free formula that is equivalent to $\top$ iff all states in Φ satisfy b. If Φ does *not* imply b (so $\mathcal{C}_{assume} \equiv \bot$), the resulting parametric postcondition thus cannot prove any FEHT via Theorem 3. △

For nondeterministic assignments $x = \star$ (line 51), we create a fresh parameter μ and continue the analysis under the precondition that $x = \mu$, effectively postponing the choice of a concrete value for x (cf. Example 2).

Example 4. Our algorithm will automatically compute the parametric postcondition from Example 2. In particular, for the `assume`$(y \geq 2)$ statement, we match line 45 with $\Phi = x \geq 9 \wedge y = \mu$ for $\mu \in \mathfrak{P}$ and compute $\mathcal{C}_{assume} := \forall x, y. \Phi \Rightarrow y \geq 2$, which is logically equivalent to $\mu \geq 2$. △

4.3 Generating Parametric Postconditions for Loops

We sketch the postcondition generation for loops in Algorithm 2. As input, `genppLoops` expects a precondition Φ over $\bigcup_{i=1}^{k+l} \mathcal{V}_i \cup \mathfrak{P}$ and universally and existentially quantified loop programs. In the first step, we guess a loop invariant $\mathbb{I}$ and counter values $c_1, \dots, c_{k+l} \in [1, B]$ (cf. (`Loop-Counting`)). In lines 4 and 5, we ensure that $\mathbb{I}$ is initial and guarantees simultaneous termination by computing restrictions $\mathcal{C}_{init}$ and $\mathcal{C}_{sim}$ on the parameters present in Φ (similar to `assume` statements in line 45 of Algorithm 1). Again, in the special case where Φ contains no parameter (as is, e.g., the case when applying our algorithm to k-safety properties), $\mathcal{C}_{init}$ (resp. $\mathcal{C}_{sim}$) is equivalent to $\top$ iff the invariant is initial

Algorithm 2 Parametric postcondition generation for loops

```
1  def genppLoops($\Phi$,$\circledast_{i=1}^{k}\left(\texttt{while}(b_i, \mathbb{P}_i) \fatsemi \mathbb{Q}_i\right)$,$\circledast_{i=k+1}^{k+l}\left(\texttt{while}(b_i, \mathbb{P}_i) \fatsemi \mathbb{Q}_i\right)$):
2      $\mathbb{I}, c_1, \ldots, c_{k+l}$ := guessInvariantAndCounts()
3      $B$ := max($c_1$,...,$c_{k+l}$)
4      $\mathcal{C}_{init}$ := $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}} x. (\Phi \Rightarrow \mathbb{I})$
5      $\mathcal{C}_{sim}$ := $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}} x. (\mathbb{I} \Rightarrow \bigwedge_{i=2}^{k+l} b_1 \leftrightarrow b_i)$
6      $\Xi_1$ := $\mathbb{I}$
7      for $j$ from $1$ to $B$:
8          ($\Xi_{j+1}, \mathcal{C}_{j+1}$) := genpp($\Xi_j \wedge \bigwedge_{i=1 | c_i \geq j}^{k+l} b_i$,$\circledast_{i=1 | c_i \geq j}^{k} \mathbb{P}_i$,$\circledast_{i=k+1 | c_i \geq j}^{k+l} \mathbb{P}_i$)
9          $\mathcal{C}_{j+1}^{cont}$ := $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}} x. (\Xi_{j+1} \Rightarrow \bigwedge_{i=1 | c_i > j}^{k+l} b_i)$
10     $\mathcal{C}_{ind}$ := $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}} x. (\Xi_{B+1} \Rightarrow \mathbb{I})$
11     $(\Xi_{rem}, \mathcal{C}_{rem})$ := genpp($\mathbb{I} \wedge \bigwedge_{i=1}^{k+l} \neg b_i$,$\circledast_{i=1}^{k} \mathbb{Q}_i$, $\circledast_{i=k+1}^{k+l} \mathbb{Q}_i$)
12     return ($\Xi_{rem}$,$\mathcal{C}_{init} \wedge \mathcal{C}_{sim} \wedge \bigwedge_{j=2}^{B+1} \mathcal{C}_j \wedge \bigwedge_{j=2}^{B+1} \mathcal{C}_j^{cont} \wedge \mathcal{C}_{ind} \wedge \mathcal{C}_{rem}$)
```

(resp. guarantees simultaneous termination). Afterward, we check the validity of the guessed counter values $c_1, \ldots, c_{k+l}$. For each j from 1 to B, we compute a parametric postcondition $(\Xi_{j+1}, \mathcal{C}_{j+1})$ for the bodies of all loops that should be executed at least j times (i.e., $c_i \geq j$) starting from precondition Ξ_j via a (mutually recursive) call to `genpp` (line 8). To ensure valid derivation using **`(Loop-Counting)`** we need to ensure that – in Ξ_{j+1} – the guard of all loops that we want to execute *more* than j times still evaluates to true. We ensure this by computing the restriction-formula $\mathcal{C}_{j+1}^{cont}$, which restricts the parameters (both those already present in the precondition Φ and those added during the analysis of the loop bodies) such that all states in Ξ_{j+1} fulfill the guards of all loops with $c_i > j$ (line 9). After we have symbolically executed all loops the desired number of times, we construct a parameter restriction $\mathcal{C}_{ind}$ that ensures that we end within the invariant, i.e., $\Xi_{B+1} \Rightarrow \mathbb{I}$ (line 10). In the last step, we compute a parametric postcondition $(\Xi_{rem}, \mathcal{C}_{rem})$ for the program suffix executed after the loops. We return the parametric postcondition that consists of the function-formula Ξ_{rem} and the conjunction of all restriction-formulas.

4.4 The Main Verification

From the soundness of FEHL (Theorem 1) we directly get:

Proposition 1. *`genpp`(Φ,$\overline{\chi_\forall}$,$\overline{\chi_\exists}$) computes some parametric postcondition for* $(\Phi, \overline{\chi_\forall}, \overline{\chi_\exists})$.

Given an FEHT $\langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$, we can thus invoke `genpp`(Φ,$\overline{\chi_\forall}$,$\overline{\chi_\exists}$) to compute a parametric postcondition, which (if strong enough) allows us to prove that $\langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$ is valid via Theorem 3. If the postcondition is too weak, we can re-run `genpp` using updated invariant guesses (cf. Section 5). For loop-free programs, it is easy to see that `genpp` computes the "strongest possible" parametric postcondition (it effectively executes the programs symbolically without incurring the imprecision inserted by loop invariants). In this case, the query from

Theorem 3 holds if and only if the FEHT is valid; our algorithm thus constitutes a *complete* verification method.

Invalid FEHTs. We stress that the goal of our algorithm is the verification of FEHTs and not proving that an FEHT is *invalid.* For k-safety properties, a refutation (counterexample) consists of a k-tuple of concrete executions that violate the property [56,19]. In contrast, refuting an FEHT corresponds to *proving* a $\exists^*\forall^*$ property, an orthogonal problem that requires independent proof ideas.

5 Implementation and Experiments

We have implemented our verification algorithm in a tool called `ForEx` [9] (short for **For**all **Ex**ists Verification), supporting programs in a minimalistic `C`-like language that features basic control structures (cf. Section 2), arrays, and bitvectors. `ForEx` uses `Z3` [48] to discharge SMT queries and supports the theory of linear integer arithmetic, the theory of arrays, and the theory of finite bitvectors. Compared to the presentation in Section 4, we check satisfiability of restriction-formulas *eagerly*: For example, in Algorithm 2, we compute multiple restriction-formulas and return their conjunction. In `ForEx`, we immediately check these intermediate restrictions for satisfiability; if any restriction is unsatisfiable on its own, any conjunction involving it will be as well, so we can abort the analysis early and re-start parts of the analysis using, e.g., updated invariants and counter values.

5.1 Loop Invariant Generation

Our loop invariant generation and counter value inference follows a standard guess-and-check procedure [34,54,56,19,53], i.e., we generate promising candidates by combining expressions found in the programs and equalities between variables in the loop guards. In most loops, there exist “anchor” variables that effectively couple executions of multiple loops together [56,19]; even in asynchronous cases like Example 1. Exploring more advanced invariant generation techniques is interesting future work. However – even in the simpler setting of k-safety properties – many tools currently rely on a guess-and-check approach [56,19]. We maintain a lattice of possible candidates ordered by implication, which allows us for efficient pruning. For example, if the current candidate is not initial (i.e., $\mathcal{C}_{init}$ computed in line 4 of Algorithm 2 is unsatisfiable), we do not need to consider stronger candidates. Likewise, if the candidate does not ensure simultaneous termination ($\mathcal{C}_{sim}$) we can prune all weaker invariants.

5.2 Experiments

We evaluate `ForEx` in various settings where FEHT-like specifications arise. We compare with `HyPA` (a predicate-abstraction-based solver) [12], `PCSat` (a constraint-based solver that relies on predicate templates) [57], and `HyPro` (a model-checker for $\forall^*\exists^*$ properties in *finite-state* systems) [11]. Our results were obtained on a M1 Pro CPU with 32GB of memory.

Instance	$t_{\texttt{HyPA}}$	$t_{\texttt{ForEx}}$
DoubleSquareNI†	67.12	**0.71**
Exp1x3	3.79	**0.30**
Fig3	8.78	**0.39**
DoubleSquareNIff	4.91	**0.37**
Fig2†	17.7	**0.73**
ColIitemSymm	15.51	**0.20**
CounterDet	5.28	**0.55**
MultEquiv	13.13	**0.60**
HalfSquareNI	**68.04**	-
SquaresSum	**17.03**	-
ArrayInsert	**16.17**	-

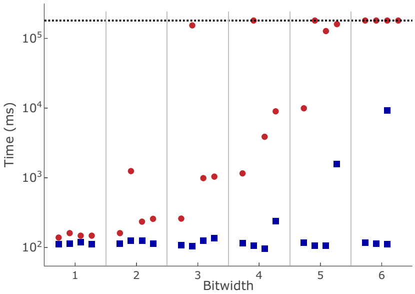
(a)

Instance	$t_{\texttt{HyPA}}$	$t_{\texttt{ForEx}}$
NonDetAdd	3.63	**0.76**
CounterSum	5.05	**1.95**
AsynchGNI	5.20	**0.69**
CompilerOpt1	1.79	**0.59**
CompilerOpt2	2.71	**1.02**
Refine	10.1	**0.57**
Refine2	9.87	**0.64**
Smaller	2.21	**0.69**
CounterDiff	8.05	**0.63**
Fig. 3	8.92	**0.57**

(b)

Instance	$t_{\texttt{PCSat}}$	$t_{\texttt{ForEx}}$
TI_GNI_hFF	26.2	**0.58**
TI_GNI_hTT	32.5	**0.10**
TI_GNI_hFT†,‡	36.2	**0.70**
TS_GNI_hFF	36.6	**0.58**
TS_GNI_hTT‡	96.2	**0.16**
TS_GNI_hFT†,‡	123.3	**2.88**
TI_GNI_hTF	**26.1**	-
TS_GNI_hTF	**44.1**	-

(c)



(d)

Fig. 5: In Figures 5a and 5b, we compare `ForEx` with `HyPA` [12] on k-safety and $\forall^*\exists^*$ properties, respectively. For instances marked with †, `ForEx` required additional user-provided invariant hints. In Figure 5c, we compare `ForEx` with `PCSat` [57]. For instances marked with ‡, `PCSat` required additional invariant hints. In Figure 5d, we compare the running time of `ForEx` (■) and `HyPro` [11] (●). We check each of the 4 GNI instances from [11] with varying bitwidth. The timeout is set to 3 min (marked by the horizontal dotted line).

Limitations of `ForEx`*'s Loop Alignment.* Before we evaluate `ForEx` on $\forall^*\exists^*$ properties, we investigate the counting-based loop alignment principle underlying `ForEx`. We collect the k-safety benchmarks from `HyPA` [12] (which themself were collected from multiple sources [32,31,55,57]) and depict the verification results in Figure 5a. We observe that `ForEx` can verify many of these instances. As it explores a restricted class of loop alignments (guided by `(Loop-Counting)`), it is more efficient on the instances it can solve. However, for some of the instances, `ForEx`'s counting-based alignment is insufficient. Instead, these instances require a loop alignment that is context-dependent, i.e., the alignment is chosen based on the current state of the programs [12,55,32,57].

ForEx and HyPA. HyPA [12] explores a liberal program alignment by exploring a user-provided predicate abstraction. The verification instances considered in [12] include a range of $\forall^*\exists^*$ properties on very small programs, including, e.g., GNI and refinement properties. In Figure 5b, we compare the running time of ForEx with that of HyPA (using the user-defined predicates for its abstraction).[1] We observe that ForEx can verify the instances significantly quicker. Moreover, we stress that ForEx solves a much more challenging problem as it analyzes the program *fully automatically* without any user intervention.

ForEx and PCSat. Unno et al. [57] present an extension of constraint Horn clauses, called pfwCSP, that is able to express a range of relational properties (including $\forall^*\exists^*$ properties). Their custom pfwCSP solver (called PCSat) instantiates predicates with user-provided templates. We compare PCSat and ForEx in Figure 5c. ForEx can verify 6 out of the 8 $\forall^*\exists^*$ instances. ForEx currently does not support termination proofs for loops in existentially quantified programs (which are needed for TI_GNI_hTF and TS_GNI_hTF), whereas PCSat features loop variant templates and can thus reason about the termination of existentially quantified loops in isolation. In the instances that ForEx can solve, it is much faster. We conjecture that this is due to the fact that the constraints generated by ForEx can be solved directly by SMT solvers, whereas PCSat's pfwCSP constraints first require a custom template instantiation.

ForEx and HyPro. Programs whose variables have a finite domain (e.g., boolean) can be checked using explicit-state techniques developed for logics such as HyperLTL [20]. We verify GNI on variants of the four boolean programs from [11] with a varying number of bits. We compare ForEx with the HyperLTL verifier HyPro [11], which converts a program into an explicit-state transition system. We depict the results in Figure 5d. We observe that, with increasing bitwidth, the running time of explicit-state model-checking increases exponentially (note that the scale is logarithmic). In contrast, ForEx can employ symbolic bitvector reasoning, resulting in orders of magnitude faster verification.

6 Related Work

Most methods for k-safety verification are centered around the self-composition of a program [6] and often improve upon a naïve self-composition by, e.g., exploiting the commutativity of statements [55,31,32,29]. Relational program logics

[1] The properties checked by HyPA [12] are temporal, i.e., properties about the infinite execution of programs of the form while$(\top, \mathbb{P})$. To make such programs analyzable in ForEx (which reasons about finite executions), we replaced the *infinite* loop with a loop that executes $\mathbb{P}$ some fixed (but arbitrary) number of times.

for k-safety offer a rich set of rules to *over*-approximate the program behavior [7,60,56,49,28,3,8]. Recently, much effort has been made to employ under-approximate methods that find bugs instead of proving their absence; so far, mostly for unary (non-hyper) properties [50,58,52,47,42,17,62,24].

Dardinier et al. [25] propose *Hyper Hoare Logic* – a logic that can express *arbitrary* hyperproperties, but requires manual deductive reasoning. Dickerson et al. [26] introduce RHLE, a program logic for the verification of $\forall^*\exists^*$ properties, focusing on the composition (and under-approximation) of function calls. They present a weakest-precondition-based verification algorithm that aligns loops in lock-step via user-provided loop invariants. Unno et al. [57] present an extension of constraint Horn-clauses (called pfwCSP). They show that pfwCSP can encode many relational verification conditions, including many hyperliveness properties like GNI (see Section 5). Compared to the pfwCSP encoding, we explore a less liberal program alignment (guided by `(Loop-Counting)`). However, we gain the important advantage of generating standard (first-order) SMT constraints that can be handled using existing SMT solvers (which shows significant performance improvement, cf. Section 5).

Most work on the verification of hyperliveness has focused on more general *temporal* properties, i.e., properties that reason about infinite executions, based on logics such as HyperLTL [20,33,13]. Coenen et al. [22] study a method for verifying hyperliveness in *finite*-state transition systems using strategies to resolve existential quantification. This approach is also applicable to infinite-state systems by means of an abstraction [12,39] (see `HyPA` in Section 5). Bounded model-checking (BMC) for hyperproperties [38] unrolls the system to a fixed bound and can, e.g., find violations to GNI. Existing BMC tools target finite-state (boolean) systems and construct QBF formulas; lifting this to support infinite-state systems by constructing SMT constraints is an interesting future work and could, e.g., complement `ForEx` in the refutation of FEHTs.

7 Conclusion

We have studied the automated program verification of relational $\forall^*\exists^*$ properties. We developed a constraint-based verification algorithm that is rooted in a sound-and-complete program logic and uses a (parametric) postcondition computation. Our experiments show that – while our logic-guided tool explores a restricted class of possible loop alignments – it succeeds in many of the instances we tested. Moreover, the use of off-the-shelf SMT solvers results in faster verification, paving the way toward a future of fully automated tools that can check important hyperliveness properties such as GNI and opacity.

Acknowledgments. This work was supported by the European Research Council (ERC) Grant HYPER (101055412), and by the German Research Foundation (DFG) as part of TRR 248 (389792660).

Data Availability Statement. `ForEx` is available at [9].

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. (2005). https://doi.org/10.1007/s10270-004-0058-x
2. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. (1985). https://doi.org/10.1016/0020-0190(85)90056-0
3. Antonopoulos, T., Koskinen, E., Le, T.C., Nagasamudram, R., Naumann, D.A., Ngo, M.: An algebra of alignment for relational verification. Proc. ACM Program. Lang. (POPL) (2023). https://doi.org/10.1145/3571213
4. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: Symposium on Principles of Programming Languages, POPL 2017 (2017). https://doi.org/10.1145/3009837.3009889
5. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: International Symposium on Logical Foundations of Computer Science, LFCS 2013 (2013). https://doi.org/10.1007/978-3-642-35722-0_3
6. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. (2011). https://doi.org/10.1017/S0960129511000193
7. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Symposium on Principles of Programming Languages, POPL 2004 (2004). https://doi.org/10.1145/964001.964003
8. Beringer, L.: Relational decomposition. In: International Conference on Interactive Theorem Proving, ITP 2011 (2011). https://doi.org/10.1007/978-3-642-22863-6_6
9. Beutner, R.: ForEx: Automated Software Verification of Hyperliveness (2023). https://doi.org/10.5281/zenodo.10436583
10. Beutner, R.: Automated software verification of hyperliveness. CoRR (2024)
11. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: Computer Security Foundations Symposium, CSF 2022 (2022). https://doi.org/10.1109/CSF54842.2022.9919658
12. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13185-1_17
13. Beutner, R., Finkbeiner, B.: AutoHyper: Explicit-state model checking for HyperLTL. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023 (2023). https://doi.org/10.1007/978-3-031-30823-9_8
14. Beutner, R., Finkbeiner, B.: Model checking omega-regular hyperproperties with AutoHyperQ. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2023 (2023). https://doi.org/10.29007/1XJT
15. Beutner, R., Finkbeiner, B., Frenkel, H., Metzger, N.: Second-order hyperproperties. In: International Conference on Computer Aided Verification, CAV 2023 (2023). https://doi.org/10.1007/978-3-031-37703-7_15
16. Biewer, S., Dimitrova, R., Fries, M., Gazda, M., Heinze, T., Hermanns, H., Mousavi, M.R.: Conformance relations and hyperproperties for doping detection in time and space. Log. Methods Comput. Sci. (2022). https://doi.org/10.46298/lmcs-18(1:14)2022
17. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A correctness and incorrectness program logic. J. ACM (2023). https://doi.org/10.1145/3582267

18. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM (2012). https://doi.org/10.1145/2240236.2240262
19. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In: Conference on Computer and Communications Security, CCS 2017 (2017). https://doi.org/10.1145/3133956.3134058
20. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference om Principles of Security and Trust, POST 2014 (2014). https://doi.org/10.1007/978-3-642-54792-8_15
21. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. (2010). https://doi.org/10.3233/JCS-2009-0393
22. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: International Conference on Computer Aided Verification, CAV 2019 (2019). https://doi.org/10.1007/978-3-030-25540-4_7
23. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. (1978). https://doi.org/10.1137/0207005
24. Cousot, P.: Calculational design of [in]correctness transformational program logics by abstract interpretation. Proc. ACM Program. Lang. (POPL) (2024)
25. Dardinier, T., Müller, P.: Hyper hoare logic: (dis-)proving program hyperproperties. CoRR (2023). https://doi.org/10.48550/arXiv.2301.10037
26. Dickerson, R., Ye, Q., Zhang, M.K., Delaware, B.: RHLE: modular deductive verification of relational ∀∃ properties. In: Asian Symposium on Programming Languages and Systems, APLAS 2022 (2022). https://doi.org/10.1007/978-3-031-21037-2_4
27. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science, Springer (1990). https://doi.org/10.1007/978-1-4612-3228-5
28. D'Osualdo, E., Farzan, A., Dreyer, D.: Proving hypersafety compositionally. Proc. ACM Program. Lang. (OOPSLA) (2022). https://doi.org/10.1145/3563298
29. Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Trans. Program. Lang. Syst. (2020). https://doi.org/10.1145/3324783
30. Farina, G.P., Chong, S., Gaboardi, M.: Relational symbolic execution. In: International Symposium on Principles and Practice of Programming Languages, PPDP 2019 (2019). https://doi.org/10.1145/3354166.3354175
31. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: International Conference on Computer Aided Verification, CAV 2019 (2019). https://doi.org/10.1007/978-3-030-25540-4_11
32. Farzan, A., Vandikas, A.: Reductions for safety proofs. Proc. ACM Program. Lang. (POPL) (2020). https://doi.org/10.1145/3371081
33. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: International Conference on Computer Aided Verification, CAV 2015 (2015). https://doi.org/10.1007/978-3-319-21690-4_3
34. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: International Symposium of Formal Methods Europe, FME 2001 (2001). https://doi.org/10.1007/3-540-45251-6_29
35. Floyd, R.W.: Assigning meanings to programs. Program Verification: Fundamental Issues in Computer Science (1993)
36. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Symposium on Principles of Programming Languages, POPL 2004 (2004). https://doi.org/10.1145/964001.964021

37. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM (1969). https://doi.org/10.1145/363235.363259
38. Hsu, T., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021 (2021). https://doi.org/10.1007/978-3-030-72016-2_6
39. Itzhaky, S., Shoham, S., Vizel, Y.: Hyperproperty verification as CHC satisfiability. CoRR (2023). https://doi.org/10.48550/arXiv.2304.12588
40. King, J.C.: Symbolic execution and program testing. Commun. ACM (1976). https://doi.org/10.1145/360248.360252
41. Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: Conference on Computer and Communications Security, CCS 2013 (2013). https://doi.org/10.1145/2508859.2516721
42. Maksimovic, P., Cronjäger, C., Lööw, A., Sutherland, J., Gardner, P.: Exact separation logic: Towards bridging the gap between verification and bug-finding. In: European Conference on Object-Oriented Programming, ECOOP 2023 (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.19
43. Mastroeni, I., Pasqua, M.: Verifying bounded subset-closed hyperproperties. In: International Symposium on Static Analysis, SAS 2018 (2018). https://doi.org/10.1007/978-3-319-99725-4_17
44. Mastroeni, I., Pasqua, M.: Statically analyzing information flows: an abstract interpretation-based hyperanalysis for non-interference. In: Symposium on Applied Computing, SAC 2019 (2019). https://doi.org/10.1145/3297280.3297498
45. McCullough, D.: Noninterference and the composability of security properties. In: Symposium on Security and Privacy, SP 1988. IEEE Computer Society (1988). https://doi.org/10.1109/SECPRI.1988.8110
46. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: Symposium on Research in Security and Privacy, SP 1994 (1994). https://doi.org/10.1109/RISP.1994.296590
47. Möller, B., O'Hearn, P.W., Hoare, T.: On algebra of program correctness and incorrectness. In: International Conference on Relational and Algebraic Methods in Computer Science, RAMiCS 2021 (2021). https://doi.org/10.1007/978-3-030-88701-8_20
48. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
49. Nagasamudram, R., Naumann, D.A.: Alignment completeness for relational hoare logics. In: Symposium on Logic in Computer Science, LICS 2021 (2021). https://doi.org/10.1109/LICS52264.2021.9470690
50. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. (POPL) (2020). https://doi.org/10.1145/3371078
51. Pasareanu, C.S., Visser, W.: Verification of Java programs using symbolic execution and invariant generation. In: International Workshop on Model Checking Software, SPIN 2004 (2004). https://doi.org/10.1007/978-3-540-24732-6_13
52. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: International Conference on Computer Aided Verification, CAV 2020 (2020). https://doi.org/10.1007/978-3-030-53291-8_14
53. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: International Conference on Computer Aided Verification, CAV 2014 (2014). https://doi.org/10.1007/978-3-319-08867-9_6

54. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: European Symposium on Programming Languages and Systems, ESOP 2013 (2013). https://doi.org/10.1007/978-3-642-37036-6_31
55. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: International Conference on Computer Aided Verification, CAV 2019 (2019). https://doi.org/10.1007/978-3-030-25540-4_9
56. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Conference on Programming Language Design and Implementation, PLDI 2016 (2016). https://doi.org/10.1145/2908080.2908092
57. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: International Conference on Computer Aided Verification, CAV 2021 (2021). https://doi.org/10.1007/978-3-030-81685-8_35
58. de Vries, E., Koutavas, V.: Reverse hoare logic. In: International Conference on Software Engineering and Formal Methods, SEFM 2011. LNCS (2011). https://doi.org/10.1007/978-3-642-24690-6_12
59. Wirth, N.: Program development by stepwise refinement. Commun. ACM (1971). https://doi.org/10.1145/362575.362577
60. Yang, H.: Relational separation logic. Theor. Comput. Sci. (2007). https://doi.org/10.1016/j.tcs.2006.12.036
61. Zhang, K., Yin, X., Zamani, M.: Opacity of nondeterministic transition systems: A (bi)simulation relation approach. IEEE Trans. Autom. Control. (2019). https://doi.org/10.1109/TAC.2019.2908726
62. Zilberstein, N., Dreyer, D., Silva, A.: Outcome logic: A unifying foundation for correctness and incorrectness reasoning. Proc. ACM Program. Lang. (OOPSLA) (2023). https://doi.org/10.1145/3586045

A Comprehensive Specification and Verification of the L4 Microkernel API

Leping Zhang[1], Yongwang Zhao[2,3](✉), and Jianxin Li[1]

[1] School of Computer Science and Engineering, Beihang University, Beijing, China
[2] School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, Hangzhou, China
zhaoyw@zju.edu.cn
[3] State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China

Abstract. The L4 API (Application Programming Interface) is a core component of the operating system, which serves as the interface between user-level processes and the microkernel, facilitating communication and interaction. It is crucial to ensure the correctness and reliability of the API. This paper proposes a comprehensive formal specification and verification for the L4 microkernel API. The specification is reusable for all implementations on architectures supported by the microkernel. To further improve reusability (e.g., for the L4 family), a parameterized model is abstracted, which mainly includes variables related to L4 components and safety properties built on them. The desired properties are composed of 350 functional correctness and 39 safety properties, where the safety properties cover existing invariants of the microkernel. Several rewriting rules and reasoning steps are proposed for verification to improve proof efficiency. The proofs of the specification w.r.t these properties are accomplished in the theorem prover Isabelle/HOL, and the results show that all definitions, lemmas, and proofs pass the prover's check. During modeling and verification, 10 bugs in the source code are found, all of which are fixed in this paper.

Keywords: Formal Specification · Theorem Proving · L4 API · Correctness · Safety · Refinement · Isabelle/HOL.

1 Introduction

The L4 API is a fundamental part of an operating system (OS) that allows applications and user-level programs to interact with the kernel, provided by the L4 microkernel family of operating systems. Ensuring the correctness and reliability of the L4 API is paramount, as it forms the foundation for system stability, security, and the seamless operation of L4 microkernel-based operating systems in diverse and demanding environments. Although Klein et al. [3] formally verified the microkernel seL4, a specific implementation of the L4 microkernel, and achieved great results, L4 microkernels are diverse, such as Pistachio, Fiasco, OKL4, and each microkernel has its own API. Based on a standard L4 reference manual [14], this paper starts out from a functional model and is committed to

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 217–234, 2024.
https://doi.org/10.1007/978-3-031-57249-4_11

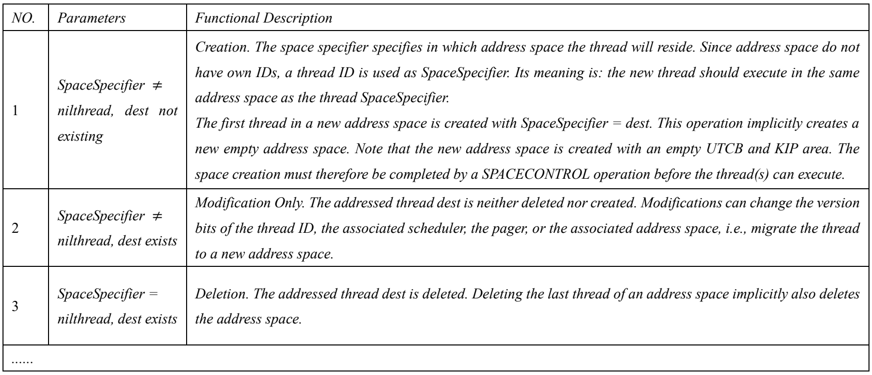
improving the correctness and reliability of the microkernel by formal verification.

So far, there has been substantial research on the formal verification of the microkernel [15,4,5,6]. Nevertheless, these studies predominantly mainly have the following problems. 1)The existing formal specifications [15,4,5,6] are incomplete. For example, Klein et al. built an abstract model for address spaces of the microkernel in Isabelle/HOL, then they formalized parts of the API using the B method, where the latter mainly supplemented threads and Inter-Process Communication (IPC) mechanism but did not include the key scheduling. 2)There are quite a few properties to formalize and verify for the kernel. Klein et al. verified three invariants about address spaces in [15,4], and no property on threads and IPC in [5,6]. 3)Klein's model [15,4] for address spaces is too one-dimensional, which is reflected in the lack of flexible page processing and access permission modeling. Flexibility is one of the design principles of L4, and permissions are an indispensable component in a practical kernel. It is necessary to model flexible pages and access permissions. 4)Following our modeling and verification experience, there are several errors in both specifications [15,5].

This paper aims to model and verify the comprehensive L4 API and make up for all the above shortcomings. We prioritize modeling based on the L4 reference manual to build a formal specification that involves all modules of the L4 microkernel. To verify enough properties (e.g., covering all invariants in Klein's model), we choose a concrete implementation built on the manual and involve more fine-grained modeling referring to the source code. By verification, we try to exclude these errors including the incomplete or unreasonable informal description in the manual, the inconsistency between the source code and the manual, the bugs in the source code, and so on.

During specification and verification for the API, we encountered three challenges. Firstly, the L4 API is quite complex, especially the address space with both tree structures and flexible pages. For this, the sel4 microkernel omits these two features of address spaces for simplicity. Moreover, the coupling of kernel modules is strong, and they form a hierarchical relationship. For example, all functions of address spaces are called by threads or IPC. Secondly, since the implementation of the API is slightly different under different CPU architectures, on the basis of fine-grained modeling, it is a challenge to build a specification that can be reused for all implementations on architectures supported by the microkernel. Thirdly, when facing non-trivial models and considerable properties, verification efficiency is often an important goal. It is necessary to present some reusable proof methods or frameworks.

On the premise of solving the above challenges, we conduct a formal verification for the L4 API in Isabelle/HOL [11], where the concrete implementation is based on the release version of the L4Ka::Pistachio [13]. To our knowledge, this work is the first effort at building the comprehensive specification and verification for the L4 microkernel. The contributions are described in detail as follows.



NO.	*Parameters*	*Functional Description*
1	*SpaceSpecifier ≠ nilthread, dest not existing*	*Creation. The space specifier specifies in which address space the thread will reside. Since address space do not have own IDs, a thread ID is used as SpaceSpecifier. Its meaning is: the new thread should execute in the same address space as the thread SpaceSpecifier.* *The first thread in a new address space is created with SpaceSpecifier = dest. This operation implicitly creates a new empty address space. Note that the new address space is created with an empty UTCB and KIP area. The space creation must therefore be completed by a SPACECONTROL operation before the thread(s) can execute.*
2	*SpaceSpecifier ≠ nilthread, dest exists*	*Modification Only. The addressed thread dest is neither deleted nor created. Modifications can change the version bits of the thread ID, the associated scheduler, the pager, or the associated address space, i.e., migrate the thread to a new address space.*
3	*SpaceSpecifier = nilthread, dest exists*	*Deletion. The addressed thread dest is deleted. Deleting the last thread of an address space implicitly also deletes the address space.*
……		

Fig. 1. Informal Functional Description of *ThreadControl* in the L4 Reference Manual

1) We propose a comprehensive formal specification of the L4 API. The specification can be reused for all implementations on architectures supported by the microkernel.
2) We formalize 350 functional correctness and 39 safety properties. The safety properties on address spaces cover all invariants in Klein's model, simultaneously, a series of new key invariants are proposed in this model.
3) We use Isabelle/HOL to prove that the specification satisfies these properties, which improves the correctness and reliability of the L4 API. In addition, in this stage, we propose several rewrite rules and reasoning steps to improve proof efficiency.
4) We found that there are in total of 10 bugs in the manual and the source code of the microkernel. All of them are fixed in this paper.

2 Preliminaries

2.1 L4 Overview

The L4 microkernel mainly includes four core modules, i.e., thread, address space, IPC, and scheduling, where the thread is the execution unit of the L4 microkernel, the address space provides isolated execution environments, the IPC mechanism enables threads in different address spaces to communicate, and the scheduling mechanism is used to switch contexts.

Thread. L4 used the thread identifier *threadid* to identify a thread, almost all of whose information is recorded in TCB (Thread Control Block). The status of each thread is defined by the status field, and the transition relationship is shown in Fig.2(a). A special phenomenon is that L4 abstracts each interrupt as a thread. When an interrupt is triggered, the kernel notifies the corresponding thread to deliver the interrupt to its handler thread through IPC, ensuring that interrupt processing in a microkernel is completed in user mode.

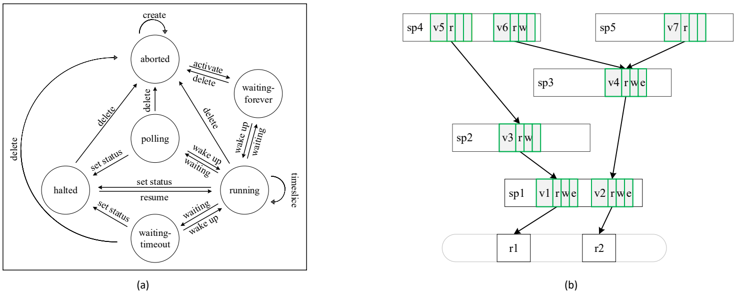


Fig. 2. The graph (a) shows the status transitions of the thread, and the graph (b) shows the tree address space structure.

Address Space. Address space is a logical concept that represents the range of virtual addresses that a thread can access. Each address space contains a page table, which realizes the conversion of the virtual address to a physical address and is a way to achieve memory isolation. The mapping mechanism is one of the features of the L4 microkernel. In addition to maintaining page table data, this mechanism also maintains the relationship between pages in different address spaces, which causes the address space to form a hierarchical structure (tree-shape) as shown in Fig.2(b), and adds difficulty to our proof.

IPC. The IPC mechanism is synchronous which means that a sender blocks until the receiver processes the message and responds. The communication of the sender and receiver can be specified as one or two phases through a special thread identifier called *nilthread*. If there is one phase, the sub-function is either sending or receiving, otherwise, sending then receiving. For the sub-function of receiving messages, the receiver can not only receive from a specific thread but from any thread that is specified by another thread identifier *anythread*.

Scheduling. L4 introduces a unique 256-level, fixed-priority scheduling system, combining time-sharing and round-robin (RR) principles. This scheduler prioritizes threads and executes them in order of their priority until certain conditions are met: a thread blocks in the kernel, gets preempted by a higher-priority thread, or consumes its allocated time quantum.

API. The L4 microkernel provides the API implemented in 10 system calls, shown in Table 1. The concrete sub-function is chosen by specifying the parameters of the system call. For example, Fig. 1 is a fragment of the L4 kernel reference manual, which informally describes three sub-functions of the system called $ThreadControl$. By controlling whether the values of $SpaceSpecifier$ are equal to *nilthread* and whether the target thread *dest* exists, the manual specifies $ThreadControl$ to complete the creation, modification, or deletion operation.

Table 1. L4 API Description

No.	Name	Functional Description
1	ThreadControl	Create, activate, modify, or delete a thread by privileged threads only.
2	ExchangeRegister	Exchange or read thread information such as IP, SP, etc.
3	Schedule	Set scheduling attributes of a thread.
4	SpaceControl	Initialize an address space by privileged threads only.
5	Unmap	Revoke pages that have been mapped or granted.
6	MemoryControl	Set the attributes of pages.
7	IPC	Transfer data from one thread to another.
8	ThreadSwitch	Switch to the specified thread or perform a normal thread scheduling.
9	SystemClock	Return the current clock.
10	ProcessorControl	Set the attributes of CPU by privileged threads only.

2.2 Related Work

Klein et al. modeled the virtual memory subsystem of an L4 kernel and verified three invariants [15,4]. Then they used the B method to model Application Programming Interface (API) as functional specifications without verification efforts [6]. Later, they conducted the refinement verification for the seL4 kernel w.r.t functional correctness [3]. Furthermore, they verified information-flow security properties for the kernel [8]. All scripts for modeling and proofs are implemented in Isabelle/HOL.

Costanzo et al. proposed the CertiKOS architecture for verifying the correctness of concurrent operating system kernels [1]. They implemented the verification by defining a series of logical abstraction layers and context refinement relations.

Nelson et al. [10] proposed the push-button verification for OS. They built three models for Hyperkernel in Python, a kernel with finite interfaces. They used the Z3 solver [7] to prove functional correctness and consistency between the abstract model and the implementation model. Later, they extended the verified properties to information flow security, in which they proposed a framework namely Nickel for verifying noninterference and used it to verify NiStar, NiKOS, and ARINC 653 standard [12]. In addition, they presented the Serval framework for developing automated verifiers [9].

3 Formal Specification of the L4 API

This section details the formal specification of the L4 API. We first define the constants and types, then give the definitions of state and initial state, next show the state transitions according to modules, and finally, provide the parameterized abstract model that improves reusability.

3.1 Constants and Types

In the L4 kernel, constants mainly include several threads and address spaces. When the kernel starts up, two privileged threads (σ_0 and $rootserver$) and all interrupt threads ($IntThreads$) are created, where $IntThreads$ is a set of threads.

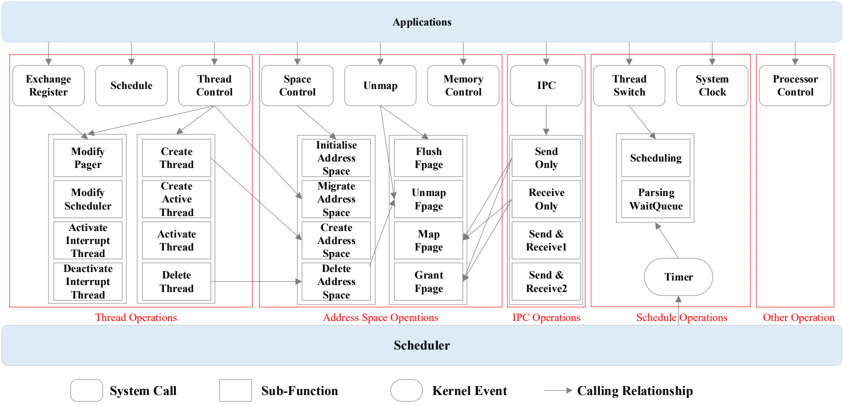


Fig. 3. L4 API, Sub-functions, and Their Relationship Graph.

These threads have their own address space. The σ_0's space is $Sigma0Space$, and $rootserver$'s space is $RootserverSpace$. We define the space of $IntThreads$ as $KernelSpace$ because they are running in kernel mode. There is a special thread namely $idle$, which starts running when the CPU is idle, and its space is $KernelSpace$.

Most types are defined according to the data structure in the source code. The following shows some special selections.

$$threadid\_t \;=\; Global\ globalid\_t \mid nilthread \mid anythread$$

The thread identifiers contain global identifiers, $nilthread$, and $anythread$, where a global identifier may represent a user thread, a kernel thread, or an interrupt thread, and the last two are special identifiers, usually used in the IPC mechanism.

$$fpage\_t \;=\; base \times size \times perms\_t\ set$$

A flexible page can be specified by type $fpage\_t$ that includes three fields of the base address, size, and permissions, where the size field makes the page size variable, and the permissions include read, write, and execute. In Klein's address space model, $fpage\_t$ is not taken into account resulting in all pages having the same size.

$$Space \;=\; spaceName\_t \rightharpoonup v\_page\_t \rightharpoonup page\_t \times perms\_t\ set$$

Where $spaceName\_t$ identifies address spaces, v_page_t identifies pages in address spaces (virtual pages for short), the type $page\_t$ is defined as:

$$page\_t \;=\; Virtual\ spaceName\_t\ v\_page\_t \mid Real\ r\_page\_t$$

Here, virtual pages and physical pages (identified by r_page_t) are unified into pages. Since $spaceName\_t$ appears in both parameters and results, $Space$ is a recursively constructed type with a tree structure. For a given virtual page, it can obtain the mapping page and permissions to access that page through $Space$. Actually, the type models the functionality of both page tables and Mapping DataBase (MDB, a core data structure used for maintaining relationships be-

Table 2. State Fields

Address Space: initialised_spaces, space_threads, space_mapping
Thread: current_thread, threads, active_threads, thread_space, thread_scheduler, thread_state, thread_pager, thread_handler, thread_message, thread_rcvWindow, thread_error, thread_priority, thread_total_quantum, thread_timeslice_length, thread_current_timeslice
IPC: thread_ipc_partner, thread_ipc_timeout, thread_recv_for, thread_recv_timeout, thread_incoming
Scheduling: current_time, wait_queuing, ready_queuing, current_timeslice
Other: heap, tlb

tween virtual pages), because 1) a valid virtual page can translate to a physical page; 2) a virtual page knows who mapped the physical page to it. The symbol $\rightharpoonup$ is the abbreviation of $\Rightarrow$ *option*, i.e., the return value is wrapped with the type *option*. In *Space*, the first $\rightharpoonup$ indicates that it can be known whether the given address space is created, and the second one indicates whether the given virtual page has a mapping.

3.2 State and Initialization

The fields of the state are shown in Table 2, which are built on the data structures in the source code, such as TCB, page tables, and so on. Some special phenomena include: each thread has a scheduler recorded by *thread_scheduler*. The scheduler is a special thread that modifies scheduling-related information for the specified thread. This may easily be confusing because there is a global scheduler used for managing scheduling modules in the source code. In addition, the fields related to the IPC module are defined in TCB.

The initialization operation mainly serves the threads created when the system starts up, including the privileged threads, interrupt threads, etc. For instance, the field *threads* is initialised as $\{\sigma_0,\ rootserver\} \cup IntThreads$.

3.3 State Transitions

In a state machine, state transitions are driven by events, where events refer to system calls shown in Table 1. The transition functions are built on both the Kernel Reference Manual and the source code. In order to reduce the complexity of each module API and the coupling between modules, we analyze and decouple the system calls into a series of sub-functions, shown in Fig. 3. The following shows the formal specification according to modules, including threads, address spaces, IPC, scheduling, and others.

Threads. The operations of thread modules include *ThreadControl*, *Schedule*, and *ExchangeRegister*. Note that *Schedule* is not a traditional schedule function (like switching context), but a function for modifying the scheduling-related fields in TCB of a thread by its *scheduler* thread. Taking *ThreadControl* as an example, the formal specification is shown in Fig. 4. For the sub-functions of

```
1  definition ThreadControl :: "Sys_Config ⇒ State ⇒ threadid_t ⇒ threadid_t ⇒ threadid_t ⇒
       threadid_t ⇒ State" where
2  "ThreadControl SysConf S destNo spaceSpec schedNo pagerNo =
3  (if ThreadControl_Cond SysConf S destNo spaceSpec schedNo pagerNo # basic check
4   then
5    (if ¬ dIsPrivilegedSpace (GetCurrentSpace S) # check the current space
6     then SetError S (current_thread S) eNoPrivilege
7     else
8      (if spaceSpec = nilthread # try to delete a thread
9       then
10       (if ThreadControl_Delete_Cond S destNo
11        then WeakDeleteThread SysConf S destNo
12        else SetError S (current_thread S) (SOME e. e ∈ {eUnavailableThread,eNoPrivilege}))
13      else
14       (if (spaceSpec ≠ nilthread) ∧ (destNo ∉ GetThreadsTids S) # try to create a thread
15        then
16         (if ThreadControl_Create_Cond SysConf S destNo spaceSpec schedNo pagerNo
17          then WeakCreateThread SysConf S destNo spaceSpec schedNo pagerNo
18          else SetError S (current_thread S) (SOME e. e ∈ {eInvalidSpace,eUnavailableThread,
                eInvalidScheduler, eUnavailableThread,eOutOfMemory}))
19        else
20         (if (spaceSpec ≠ nilthread) ∧ (destNo ∈ GetThreadsTids S) ∧ (TidToGno destNo ∉
               kIntThreads) # try to modify a thread
21          then
22           (if ThreadControl_Modify_Cond SysConf S destNo spaceSpec schedNo pagerNo
23            then WeakModifyThread SysConf S destNo spaceSpec schedNo pagerNo
24            else SetError S (current_thread S) (SOME e. e ∈ {eInvalidSpace,eUnavailableThread,
                  eInvalidScheduler,eOutOfMemory}))
25          else
26           (if (spaceSpec ≠ nilthread) ∧ (destNo ∈ GetThreadsTids S) ∧ (TidToGno destNo ∈
                 kIntThreads) # try to handle an interrupt thread.
27            then
28             (if pagerNo ∈ GetThreadsTids S
29              then IntThreadControl SysConf S destNo pagerNo
30              else SetError S (current_thread S) eUnavailableThread)
31            else S)))))
32  else S)"
```

Fig. 4. Formal Definition of the System Call *ThreadControl*

creation, modification, and deletion, the specification corresponds exactly to the informal manual shown in Fig. 1. Since there is no clear description for handling interrupt threads, we refer to the source code and add a conditional branch (shown in Lines 26-30) to supplement information from the manual.

Address Spaces. The operations of address spaces include *SpaceControl*, *Unmap*, and *MemoryControl*. The complex sub-functions focus on the mapping mechanism including unmap, flush, map, and grant for pages. Before formalizing these functions, we introduce several key definitions. We define $s \vdash x \leadsto^1 y$ to represent that both pages are in one path, and the page x can reach the page y by one step. The terms $s \vdash x \leadsto^+ y$ and $s \vdash x \leadsto^* y$ represent transitive, and reflexive and transitive paths, respectively. If a page x is a physical page or x can reach another page in a given state s, then x is a valid page denoted as $s \vdash x$. Leveraging these definitions, we give the specification for the function *map*, shown in Fig. 5. In the above definition, we add the formalization of access permissions in terms of Klein's model. Lines 3 and 5 show the conditions of successfully executing the operation. According to our experience for proving subsequent invariants, even

```
definition map :: "State ⇒ spaceName_t ⇒ v_page_t ⇒ spaceName_t ⇒ v_page_t ⇒ perms_t set
       ⇒ State" where
"map s sp_from v_from sp_to v_to perms =
(if (sp_to ≠ Sigma0Space)
 then
  (if s ⊢ (Virtual sp_from v_from) ∧ perms ≠ {} ∧ perms ⊆ get_perms s sp_from v_from ∧
        sp_from ≠ sp_to ∧ (∀v. ¬s ⊢ (Virtual sp_from v_from) ⇝+ (Virtual sp_to v)) ∧
        space_mapping s sp_to ≠ None ∧ v_to < page_maxnum
   then s⦇space_mapping:= λsp'.
    (if space_mapping s sp' = None
     then None
     else Some (λv_page1.
      (if the (space_mapping s sp') v_page1 = None ∧ Virtual sp' v_page1 ≠ Virtual sp_to v_to
       then None
       else
        (if s ⊢ (Virtual sp' v_page1) ⇝* (Virtual sp_to v_to)
         then
          (if (Virtual sp' v_page1) = (Virtual sp_to v_to)
           then Some ((Virtual sp_from v_from), perms)
           else None)
         else the (space_mapping s sp') v_page1))))⦈
     else s)
 else s)"
```

Fig. 5. Formal Definition of the Sub-function *map*

if conditions related to permissions are not considered, there is a lack of these conditions in Klein's model, causing their first invariant (corresponding to our Invariant 1) to not hold. Based on the above definition, we follow the method of iteratively processing pages of the same size in the source code and define a recursive function to handle flexible pages.

IPC. The IPC mechanism is implemented by the system call *IPC*. By specifying parameter values, the call may involve both the sending phase and the receiving phase. Only after the sending phase is completed, the receiving phase can be executed. According to two phases, we decouple the operation into four subsections, i.e., *send_only*, *receive_only*, *send_receive*1, and *send_receive*2, where the difference between the last two definitions is whether the sending operation is performed successfully. If not, the data of the receiving phase must be saved in a stack. For instance, the parameters used for specifying the receiver's information are saved in state fields *thread_recv_for* and *thread_recv_timeout* in our model.

Scheduling. The operations of the scheduling module include *ThreadSwitch* and *SystemClock*. For the former, the user thread can use it to actively switch current context. To make our specification more complete, we model the unique operation guided by the kernel, i.e., timer interrupt. The operation is used to complete thread scheduling and its strategy is either preemption or a common scheduling.

Others. In addition to the above operations, the behaviors of the Memory Management Unit (MMU) and Translation-Lookaside Buffer (TLB) also are involved in our specification. For these, we just provide a model, and the proofs of the functional correctness and safety properties are not in the scope of this work.

3.4 Parameterized Abstract Model

In addition to the above concrete model, we build a parameterized abstract model to improve reusability using the locale system provided by Isabelle/HOL. The system generally includes variables and assumptions, in which variables are parameters of the system, and assumptions describe the relationships between variables. In our model, the variables involve the initial state s_0, the state transition function *step*, and some key L4 components such as σ_0, *rootserver*, and so on, denoted as:

$$M_{abs}\ \{s_0,\ step,\ \sigma_0,\ rootserver,\ \cdots\}$$

where M_{abs} is the system name, and the types of its parameters are abstract to improve reusability, because data structures in different implementations of L4 microkernels vary slightly. The assumptions contain only some general safety properties. For example, an active thread must be a created thread, i.e., $active\ s \subseteq threads\ s$, which applies to almost all L4 microkernels. If a property relies too much on the implementation of the API or sub-functions, then it will be difficult to reuse. This is also why we do not consider adding functional correctness to the assumptions. In subsequent refinement proofs, these assumptions must be proven to be true.

4 Formalizing Functional Correctness and Safety Properties

The section depicts the formalization of the functional correctness and safety properties. In general, functional correctness means that a program has a correct output for a given input, which can be easily described by the Hoare Triple [2], i.e., $\{P\}\ c\ \{Q\}$, where P is the pre-condition, Q is the post-condition, and c represents the program. Safety means that there is no unsafe state in the whole state space, which can be represented as a series of invariants. The form of expressing an invariant lemma is similar to the Hoare triple, which can be defined as $\{I\}\ c\ \{I\}$ or $\{I_1 \wedge I_2 \wedge \cdots\}\ c\ \{I\}$. In total, we formalized 350 functional correctness and 39 safety properties.

Functional Correctness. Almost of functional correctness lemmas are built on sub-functions and auxiliary definitions (the sum of the two quantities is 50). These lemmas describe the changes in all state fields which are divided into 7 parts (current, UTCB, TCB, address space, mapping, IPC, scheduling). For a given sub-function f, the correctness lemmas include two cases: 1) After executing f, the changed fields are set to correct values. 2) The values of unchanged fields in the original state and the new state are equal. The advantage of constructing lemmas in this way is that the functional correctness is relatively complete. In addition, these lemmas built on sub-functions provide convenience for subsequent proofs, e.g., we can exploit these lemmas and eliminate the function through a substitution strategy instead of directly unfolding its definition causing the structure to be destroyed.

The following shows one of the functional correctness lemmas of the sub-function *map* whose function is to add mapping to a page.

Lemma 1. $\neg s \vdash (Virtual\ sp\ v) \rightsquigarrow^* (Virtual\ sp\_to\ v\_to) \Longrightarrow$
$s \vdash (Virtual\ sp\ v) \rightsquigarrow^+ page \Longrightarrow$
$(map\ s\ sp\_from\ v\_from\ sp\_to\ v\_to\ perms) \vdash (Virtual\ sp\ v) \rightsquigarrow^+ page$

The lemma means that if a virtual page $Virtual\ sp\ v$ is not on the path to $Virtual\ sp\_to\ v\_to$, then the operation has no effect on $Virtual\ sp\ v$, and its accessibility to other pages remains unchanged.

Safety Properties. Safety properties are represented as invariants. Parts of invariants with cumbersome proofs are mainly related to the address space module, and they are shown as follows.

Invariant 1 *Pages do not form rings in the address space structure.*

$$\forall\, s\ sp\ v1.\ (\nexists v2.\ s \vdash (Virtual\ sp\ v1) \rightsquigarrow^+ (Virtual\ sp\ v2))$$

In Klein's model, the invariant is defined as $\forall\, s.\ (\nexists x.\ s \vdash x \rightsquigarrow^+ x)$, which only ensures that for a given virtual page there is no loops on this page, and is a corollary of Invariant 1. In fact, it is naturally unreasonable if the page can reach another page in its address space because they will eventually be translated to the physical page. Our definition solves this problem by allowing that $v1$ is not equal to $v2$.

Invariant 2 *A page is valid if and only if there is a physical page translated from the valid page.*

$$\forall\, s\ x.\ (s \vdash x \longleftrightarrow (\exists\, r.\ s \vdash x \rightsquigarrow^* (Real\ r)))$$

Invariant 2 improves the corresponding invariant in Klein's model from implication to equivalence.

Invariant 3 *A page has a subset of the permissions of its direct parent page.*

$$\forall\, s\ sp1\ sp2\ v1\ v2.\ s \vdash (Virtual\ sp1\ v1) \rightsquigarrow^1 (Virtual\ sp2\ v2) \longrightarrow$$
$$get\_perms\ s\ sp1\ v1 \subseteq get\_perms\ s\ sp2\ v2)$$

Invariant 4 *The permissions of valid pages are not empty.*

$$\forall\, s\ sp\ v.\ s \vdash (Virtual\ sp\ v) \longrightarrow get\_perms\ s\ sp\ v \neq \{\}$$

Invariants 3 and 4 are proposed to ensure properties on the additional permission fields.

Invariant 5 *A created thread must have an address space, and the space has been created.*

$$\forall\, s\ t.\ t \in threads\ s \longrightarrow$$
$$(\exists\, sp.\ sp \in spaces\ s \wedge thread\_space\ s\ t\ =\ Some\ sp)$$

Invariant 6 *For an arbitrary created address space sp, the set of threads in sp is equal to that of threads whose space is sp.*

$$\forall\, s\ sp.\ sp \in spaces\ s \longrightarrow$$
$$the\ (space\_threads\ s\ sp)\ =\ \{t.\ thread\_space\ s\ t\ =\ Some\ sp\}$$

Invariants 5 and 6 are used to associate threads with address spaces.

Invariant 7 *The identifier of the thread that has not been created must be within the configured range.*

$$\forall s\ x.\ x \notin threads\ s \longrightarrow x \in Threads\_Gno\ SysConf$$

Following the source code, the global identifier does not exceed $2^{18} - 1$. We use *SysConf* to define the system environment, and *Threads_Gno* obtains the set of global identifiers from *SysConf*.

Invariant 8 *A created thread must have a scheduler, and the scheduler has been created.*

$$\forall s\ t.\ t \in threads\ s \longrightarrow (\exists sche.\ sche \in threads\ s \land thread\_scheduler\ s\ t\ =\ Some\ sche)$$

When invariants 7 and 8 are proved, some bugs in the source code are discovered, and they are discussed in detail in Section 6.

5 Formal Verification

The section illustrates the formal verification for the L4 API. The proof task consists of three parts: functional correctness, safety properties, and refinement between the abstract model and the concrete model. The following first introduces the rewrite rules and reasoning steps that improve verification efficiency, then separately shows proofs of the three parts.

5.1 Rewrite Rules and Reasoning Steps

Since different strategies can be used and the order of proof can also be different, the properties can be proven by various proof methods. However, whether the proof method is excellent can greatly affect the verification efficiency. A good method generally wishes proof steps to be concise and reusable. A typical counterexample is the abuse of automatic tactics provided by Isabelle/HOL. For example, the tactic *auto* tries its best to make the proof goal as simple as possible, but the extent of simplification is unclear, in other words, the subgoal obtained must be re-analyzed every time, as long as the original goal has not been proven completely. Even worse, some of the structure in the original goals has been destroyed.

To avoid these problems, we first propose 21 rewrite rules to simplify the goal and obtain the desired subgoals. These rules are constructed for the three expressions of if, let, and case, which are common in our specification. Some typical rules are shown as follows:

- $if.\ (Q \Longrightarrow P\ x) \Longrightarrow (\neg Q \Longrightarrow P\ y) \Longrightarrow P\ (if\ Q\ then\ x\ else\ y)$
- $let.\ P\ s\ t \Longrightarrow (\bigwedge s\ t.\ P\ s\ t \Longrightarrow P\ (f\ s)\ (f\ t)) \Longrightarrow P\ (Let\ s\ f)\ (Let\ t\ f)$
- $case.\ (opt\ =\ None \Longrightarrow P\ f1) \Longrightarrow ((opt \neq None) \Longrightarrow P\ (f2\ (the\ opt))) \Longrightarrow P\ (case\ opt\ of\ None \Rightarrow f1 \mid Some\ x \Rightarrow f2\ x)$

```
lemma "¬s⊢(Virtual sp v)⇝*(Virtual sp_to v_to) ⟹ s⊢(Virtual sp v)⇝+y ⟹ (map s
      sp_from v_from sp_to v_to perms)⊢(Virtual sp v)⇝+y"
proof-
  assume a1:"s⊢(Virtual sp v)⇝+y" and a2:"¬s⊢(Virtual sp v)⇝*(Virtual sp_to v_to)"
  then show ?thesis
  proof(induction rule:tran_path.induct)
    case (one_path x y) # the case of direct path
    then have "(map s sp_from v_from sp_to v_to perms)⊢x⇝1y"
      using map_not_path_direct FatherIsVirtual by metis
    then show ?case using tran_path.intros by blast
  next
    case (tran_path x y z) # the case of transitive path
    then have "¬s⊢y⇝*(Virtual sp_to v_to)"
      using refl_tran by blast
    then have h1:"(map s sp_from v_from sp_to v_to perms)⊢y⇝+z"
      using tran_path by simp
    have "(map s sp_from v_from sp_to v_to perms)⊢x⇝1y"
      using tran_path map_not_path_direct FatherIsVirtual by metis
    then show ?case using h1 tran_path.intros by simp
  qed
qed
```

Fig. 6. Formal Proof for Correctness of the Sub-function *map*

To our knowledge, the theory library provided by Isabelle/HOL does not include the rules we proposed, although many of them look very similar, especially for the *if* rule above. But a tiny difference such as replacing $\Longrightarrow$ with $\longrightarrow$ will produce different results.

Second, we construct some general reasoning steps to improve verification efficiency. Our construction follows two principles: 1)If the goal is not fully proven in the current proof step, then this step must be deterministic (i.e., producing specific subgoals); 2)Allows the use of automated tactics that only work on the current goal if the current goal can be directly proven; 3)Allows the use of any automated proof tactic in the last step. Since the reasoning procedure is quite similar for a given invariant or function, these steps are mainly used for proving invariants and they are especially effective when the state fields involved in the invariant do not appear in the functions being proved. Leveraging these steps, we often only need to replace auxiliary lemmas without modifying proof tactics. Indeed, in our experience, most lemma proofs are written through a copy-replace-paste procedure. In Section 5.3, we take proving a concrete safety property as an example to show the use of reasoning steps.

5.2 Functional Correctness Proofs

Sophisticated proofs of functional correctness focus on mapping operations. On the one hand, these operations involve the reflexive and transitive path, causing that for almost every correctness lemma, we must use the **induction** tactic introduced by hand; On the other hand, it is not easy to clarify the relationship between virtual pages in the tree address space structure. Fig. 6 shows the proof of Lemma 1 related to mapping operations. Except for mapping operations,

```
 1  lemma DeleteThread_Inv_Space_Perms_IsNot_Empty:
 2    assumes p1:"Inv_Space_Perms_IsNot_Empty s"
 3    shows "Inv_Space_Perms_IsNot_Empty (DeleteThread s gno)"
 4    apply(subst DeleteThread_eq)
 5    apply(rule elim_if)
 6    subgoal
 7      apply(rule SetError_Inv_Space_Perms_IsNot_Empty)
 8      apply(rule delete3_Inv_Space_Perms_IsNot_Empty)
 9      apply(rule delete2_Inv_Space_Perms_IsNot_Empty)
10      apply(rule delete1_Inv_Space_Perms_IsNot_Empty)
11      using assms by simp
12    using assms by simp
```

Fig. 7. Formal Proof for Invariant 4 on *DeletingThread*

other operations are proven to be functionally correct mainly by unfolding their definitions.

5.3 Safety Proofs

To verify safety properties, we prove that the specification satisfies all of the invariants. An invariant usually is proved by induction, i.e., both the initial state and the transition step are established on the invariant. The former is proved by unfolding the definition of the initial state s_0_def, while the latter is demonstrated by an example, shown in Fig. 7. The lemma describes that the transition of deleting a thread satisfies Invariant 4. Here, due to the complexity of the function *DeleteThread*, we equivalently replace this function with an execution sequence [detele1, delete2, delete3, SetError] to simplify the proof. Line 5 shows the application of the rewrite rule *elim_if* for if expressions, which decouples the goal to two subgoals. The proof task is concentrated on the first that was brought out by the keyword **subgoal** in Line 6. We leverage the lemma that every element in the sequence satisfies the invariant to reduce our conclusion to the hypothesis in reverse order. The second subgoal is the case when the execution condition is not met (the state is unchanged), which is proved by *simp*. The whole proof process forms the general reasoning steps, which is firstly applicable to almost all proofs of *DeleteThread* on invariants, and secondly can be reused for other functions but only requires modification of the auxiliary definitions or lemmas used to assist the proof.

5.4 Refinement Proofs

The refinement proofs are used to ensure consistency between the abstract level and the concrete level. Recalling the abstract model M_{abs} using the locale system, we must instantiate it into the concrete model, which can be organized by the keyword **interpretation** as follows.

$$\textbf{interpretation}\ M_{abs}\ \{s_0',\ \mathit{step}',\ \sigma_0',\ \mathit{rootserver}',\ \cdots\}$$

The elements within the brackets are defined in the concrete model, and they replace corresponding parameters of the system M_{abs}. Thus, an instantiation

Table 3. Efforts for Specification and Proofs

Item	**Specification**		**Correctness**		**Invariants**	
	LOC	PM	LOP	PM	LOP	PM
Threads	∼600	1	∼2500	1	∼9000	2
Address Space	∼300	1	∼4500	2	∼1300	1.5
IPC	∼200	0.5	∼1000	0.5	∼3000	1
Scheduling	∼150	0.5	∼2000	1	∼5000	2
Others	∼300	0.5	—	—	∼500	0.5
Total	∼1550	3.5	∼10000	4.5	∼18800	7

theorem is defined if there is no type conflict. Next, our task is to prove that the assumptions on these concrete variables hold. Since these assumptions are invariants proved in Section 5.3, the theorem can be easily derived through the tactic *auto*.

6 Discussion

Result. We leverage Isabelle/HOL to build a comprehensive formal specification for the L4 API. The specification completely covers all modules in the kernel, including address spaces, threads, IPC, scheduling, and others. We prove that the formal specification satisfies all functional correctness and invariants we proposed. To improve the readability of formal proofs, most formal proofs are written in a structured language Isabelle/Isar [16], especially some complex lemmas. In total, this work produces about 1.5K lines of code(LOC) for specifications and about 29K lines of proofs(LOP) and takes about 15 person-months(PM). Details of this work are described in Table 3.

Verified Issues. During specification and verification, we found 10 bugs that violate functional correctness and safety properties. They are classified into 6 categories and reported as follows.

- **Out-of-Bounds Access.** When we prove the invariant 7, we found that in the system calls *ThreadControl*, *IPC*, and *ExchangeRegister*, there is no policy to limit the range of the destination thread's identifier *dest_tid*. This bug allows threads to access non-TCB areas. We recommend adding a condition expression into the source code, ensuring that these system calls work only if *dest_tid* does not exceed the maximum.
- **Illegal Deletion.** The source code allows the privileged threads to delete any unprivileged thread, which may cause exceptions to occur. We know that once a thread is created, it is assigned a scheduler used for managing its scheduling-related fields. Thus, invariant 8 needs to be guaranteed in the whole state space. However, the invariant on the sub-function of deleting thread cannot be proven to be true, this is because there will be no scheduler to serve the thread if the deleted object is the scheduler. Our recommendation is to only allow privileged threads such as *rootserver* to serve

as the scheduler for a thread. The advantage is that it can be implemented by modifying very little code because there is no need to check the dependencies between unprivileged threads.

- **Lack of Validity Checks.** The system call *ThreadControl* does not check whether the identifier of the parameter *scheduler_tid* is valid, where validity means the thread represented by *scheduler_tid* has been created. The lack of these checks still violates the invariant 8 when reasoning about the subfunction of creating threads. Following our specification of *Create_Thread*, the bug can be fixed by determining whether *scheduler_tid* exists in the created thread collection (*threads*) that is defined as a ring queue called *present_list* in the source code.
- **Unfinished Definitions.** When modeling the functionality of the activating thread, we found there is no handling of the case when the activation operation fails. We recommend that before activating a thread, make a backup of the fields that need to be changed during activation, and restore the values of these fields if the activation fails. Some similar bugs include a lack of handling failure to allocate address space; and a lack of implementation for the system call *ProcessorControl*.
- **Incomplete Initialization.** In the header file *schedule.h* of the source code of the Pistachio0.4 version, the initialization function *init* does not initialize the last priority queue. In detail, in the code snippet of $for(int\ i\ = 0;\ i < MAX\_PRIO;\ i++)\{\cdots\}$, the loop condition should be set as $i <= MAX\_PRIO$. It was discovered when we compared the initial state of our model with that of the source code. Fortunately, this bug is fixed in the latest version.
- **Inconsistent Implementation.** In the source code, the handler of each interrupt thread is recorded by the field *scheduler* in TCB, while the advice given in the manual is to take the field *pager* in UTCB. In our experience, it is reasonable to record the handler by *scheduler*, because the interrupt threads are executed in the kernel mode, and can be viewed as kernel threads, thus, there is no need to assign them UTCB areas.

7 Conclusion and Future Work

This paper proposes a comprehensive formal specification and verification for an L4 microkernel API. The formal specification makes up for the missing core components of the existing models and fixes the errors in these models. To improve the correctness and reliability, 350 functional correctness and 39 safety properties are formalized. After machine-checking proof in Isabelle/HOL, the formal specification strictly satisfies the proposed 350 functional correctness and 39 safety properties. Through decoupling functionalities into some sub-functions, abstracting the parameterized model, rewriting proof rules, and building reason patterns, we solve the challenges in this work on complexity, reusability, and efficiency. During specification and verification, we found 10 bugs in the kernel reference manual and source code of the L4 microkernel, and we provided solutions to fix them. In the future, we will expand the verification for the API to that for the entire source code and may pay more attention to the automation technology in refined verification.

Acknowledgment

This work has been supported in part by the Natural Science Foundation of China under Grants No. 62132014 and No. U2341212, and Zhejiang Science and Technology Plan Project under Grant No.2022C01045.

References

1. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent os kernels. In: Keeton, K., Roscoe, T. (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 653–669. USENIX Association (2016), `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu`
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
3. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D.A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an os kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009. pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596, `https://doi.org/10.1145/1629575.1629596`
4. Klein, G., Tuch, H.: Towards verified virtual memory in l4. TPHOLs Emerging Trends **4**, 16 (2004)
5. Kolanski, R.: A formal model of the μ-kernel api using the b method. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney **2052** (2004)
6. Kolanski, R., Klein, G.: Formalising the l4 microkernel api. In: Proceedings of the Twelfth Computing: The Australasian Theory Symposium-Volume 51. pp. 53–68 (2006)
7. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, `https://doi.org/10.1007/978-3-540-78800-3_24`
8. Murray, T.C., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: sel4: From general purpose to a proof of information flow enforcement. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013. pp. 415–429. IEEE Computer Society (2013). https://doi.org/10.1109/SP.2013.35, `https://doi.org/10.1109/SP.2013.35`

9. Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., Wang, X.: Scaling symbolic evaluation for automated verification of systems code with serval. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. pp. 225–242. ACM (2019). https://doi.org/10.1145/3341301.3359641, https://doi.org/10.1145/3341301.3359641
10. Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., Wang, X.: Hyperkernel: Push-button verification of an OS kernel. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017. pp. 252–269. ACM (2017). https://doi.org/10.1145/3132747.3132748, https://doi.org/10.1145/3132747.3132748
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A proof assistant for higher-order logic. Springer-Verlag (2002)
12. Sigurbjarnarson, H., Nelson, L., Castro-Karney, B., Bornholt, J., Torlak, E., Wang, X.: Nickel: A framework for design and verification of information flow control systems. In: Arpaci-Dusseau, A.C., Voelker, G. (eds.) 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018. pp. 287–305. USENIX Association (2018), https://www.usenix.org/conference/osdi18/presentation/sigurbjarnarson
13. Team, L.: Pistachio microkernel. https://www.l4ka.org/65.php (2010)
14. Team, L.: L4 experimental kernel reference manual version x.2 (2011), https://www.l4ka.org/l4ka/l4-x2-r7.pdf
15. Tuch, H., Klein, G.: Verifying the l4 virtual memory subsystem. In: Proc. NICTA FM Workshop on OS Verification. pp. 73–97 (2004)
16. Wenzel, M., et al.: The isabelle/isar reference manual (2004)

Probabilistic Systems

Accurately Computing Expected Visiting Times and Stationary Distributions in Markov Chains

Hannah Mertens(✉), Joost-Pieter Katoen, Tim Quatmann, and Tobias Winkler

RWTH Aachen University, Aachen, Germany
{hannah.mertens,katoen,tim.quatmann,tobias.winkler}@cs.rwth-aachen.de

Abstract. We study the accurate and efficient computation of the expected number of times each state is visited in discrete- and continuous-time Markov chains. To obtain sound accuracy guarantees efficiently, we lift interval iteration and topological approaches known from the computation of reachability probabilities and expected rewards. We further study applications of expected visiting times, including the sound computation of the stationary distribution and expected rewards conditioned on reaching multiple goal states. The implementation of our methods in the probabilistic model checker Storm scales to large systems with millions of states. Our experiments on the quantitative verification benchmark set show that the computation of stationary distributions via expected visiting times consistently outperforms existing approaches — sometimes by several orders of magnitude.

1 Introduction

Expected visiting times. Common questions for the quantitative analysis of Markov chains include reachability probabilities, stationary distributions, and expected rewards [34]. Many authors [36,23,55,24,44,48,19] have recognized the importance of another quantity called *expected visiting times (EVTs)*, which describe the expected time a system spends in each state. EVTs are characterized as the unique solution of a linear equation system [36]. They are not only relevant in their own right, but also useful to obtain various other quantities, including the ones mentioned above. This applies particularly to *forward analyses* which aim at computing, e.g., the distribution over terminal states given an initial distribution.

Sound approximation of EVTs. In the context of (probabilistic) model checking, the two main requirements for any numeric procedure are *scalability* and *soundness*, i.e., the error in the reported result has to be bounded by a predefined threshold. Scalability is typically achieved via numerically robust iterative methods [52,57,59] such as the *Jacobi* or *Gauss-Seidel method* [57]. In general, these methods do not converge to the exact solution after a finite number of iterations. Thus, the procedure is usually stopped as soon as a termination criterion is satisfied [52]. However, standard stopping criteria such as small difference of consecutive iterations are not sound in the above sense: They do not actually

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 237–257, 2024.
https://doi.org/10.1007/978-3-031-57249-4_12

indicate how close the approximation is to the true solution. Since the correctness of results in model checking, especially for safety-critical systems, is crucial, several authors have proposed sound iterative algorithms [26,6,50,30]. While these works focus on computing quantities such as reachability probabilities and expected rewards, the sound computation of EVTs has not yet been studied.

Motivating example: Verifying sampling algorithms. To illustrate the use of EVTs in probabilistic verification tasks, consider the Markov chain in Figure 1. It is a finite-state model of a program — the *Fast Dice Roller* [42] — which takes as input an integer $N \geq 1$ and produces a uniformly distributed output in $\{1, \dots, N\}$ using unbiased coin flips only. The Fast Dice Roller thus solves a generalized *Bernoulli Factory* problem [35]. Our model in Figure 1 is for the case where $N = 6$ is fixed. How can we establish that each of the terminal states ⚀, . . . , ⚅ is indeed reached with probability sufficiently close to or exactly $\frac{1}{6}$? The standard approach for answering this question is to solve N linear equation systems, one for each terminal state [5, Ch. 10]. An alternative (and seemingly less well-known) method is to compute the EVTs of each state in the Markov chain, which requires solving just a single linear equation system. All N desired probabilities can then easily be derived from the EVTs [36]: For instance, the states s_3, s_4, s_6 can all be shown[1] to have EVT $\frac{1}{3}$, and thus the reachability probabilities of the terminal states are all $\frac{1}{6} = \frac{1}{2} \cdot \frac{1}{3}$. Similarly, EVTs are useful for computing conditional expected rewards. For Bernoulli Factories, this allows us to examine if some outcomes take longer to compute on expectation than others, which is important to analyze possible side channel attacks in a security context. Furthermore, we show in this paper how computing stationary distributions reduces to EVTs. Such distributions provide insight into a system's long-term behaviour; applications include the *mean-payoff* of a given policy in a Markov decision process (MDP) [49, Pr. 8.1.1], the distribution computed by a Chemical Reaction Network [12], and the semantics of a probabilistic NetKAT network [56].

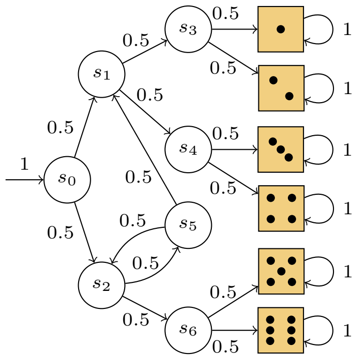


Fig. 1: Fast Dice Roller

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

- We describe, analyze, and implement the *first sound numerical approximation algorithm for EVTs* in finite discrete- and continuous-time Markov chains. Our algorithm is an adaption of the known *Interval Iteration (II)* [43,27,6].
- We show that computing (sound bounds on) a Markov chain's *stationary distribution* reduces to EVT computations. The resulting algorithm significantly outperforms preexisting techniques [41,45] for stationary distributions.

[1] See Section 4 for how to compute these numbers. An EVT of $\frac{1}{3}$ means that *on average*, the state is visited $\frac{1}{3}$ times. In general, EVTs of reachable recurrent states are ∞, and the EVT of reachable transient states can take any value in $(0, \infty)$.

- Similarly, we show how the *conditional expected rewards* until reaching each of the, say, M absorbing states of a Markov chain can be obtained by computing the EVTs and solving a second linear equation system — this is in contrast to the standard approach which requires solving M equation systems [5, Ch. 10].
- We implement our algorithm in the probabilistic model checker Storm [32] and demonstrate its scalability on various benchmarks.

Outline. We define general notation and EVTs in Sections 2 and 3, respectively. In Section 4, we present our sound iterative algorithms for computing EVTs approximately. Sections 5 and 6 present the reductions of stationary distributions and conditional expected rewards to EVTs. We report on the experimental evaluation of our algorithms in Section 7 and summarize related work in Section 8.

2 Background

Let $\mathbb{N}$ denote the set of non-negative integers and $\overline{\mathbb{R}} = \mathbb{R} \cup \{\infty, -\infty\}$ the set of extended real numbers. We equip finite sets $S \neq \emptyset$ with an arbitrary indexing $S = \{s_1, \ldots, s_n\}$ and identify functions of type $\mathbf{v} \colon S \to \overline{\mathbb{R}}$ and $\mathbf{A} \colon S \times S' \to \overline{\mathbb{R}}$ with (column) vectors $\mathbf{v} \in \overline{\mathbb{R}}^{|S|}$ and matrices $\mathbf{A} \in \overline{\mathbb{R}}^{|S|\times|S'|}$, respectively. $\mathbf{I}$ denotes the identity matrix. Vectors are compared component-wise, i.e., $\mathbf{v} \leq \mathbf{v}'$ iff for all $s \in S$, $\mathbf{v}(s) \leq \mathbf{v}'(s)$. *Iverson brackets* $[\![B]\!]$ cast the truth value of a Boolean expression B to a numerical value 1 or 0, such that $[\![B]\!] = 1$ iff B is true.

Definition 1. *A* discrete-time Markov chain (DTMC) *is defined as a triple* $\mathcal{D} = (S^{\mathcal{D}}, \mathbf{P}^{\mathcal{D}}, \iota^{\mathcal{D}}_{\text{init}})$*, where* $S^{\mathcal{D}}$ *is a finite set of states,* $\mathbf{P}^{\mathcal{D}} \colon S^{\mathcal{D}} \times S^{\mathcal{D}} \to [0,1]$ *is the transition probability function satisfying* $\sum_{t\in S} \mathbf{P}^{\mathcal{D}}(s,t) = 1$ *for all* $s \in S$*, and* $\iota^{\mathcal{D}}_{\text{init}} \colon S \to [0,1]$ *is the initial distribution with* $\sum_{s\in S} \iota^{\mathcal{D}}_{\text{init}}(s) = 1$*.*

We often omit the superscript from objects associated with a DTMC $\mathcal{D}$ whenever this is clear from context, e.g., we write $\mathbf{P}$ rather than $\mathbf{P}^{\mathcal{D}}$. An *infinite path* $\pi = s_0 s_1 \cdots \in S^{\omega}$ in a DTMC $\mathcal{D} = (S, \mathbf{P}, \iota_{\text{init}})$ is a sequence of states such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \in \mathbb{N}$. We use $\pi[i] = s_i$ to refer to the i-th state. $\mathsf{Paths}^{\mathcal{D}}$ denotes the set of all infinite paths in $\mathcal{D}$. The probability measure $\Pr^{\mathcal{D}}$ over measurable subsets of $\mathsf{Paths}^{\mathcal{D}}$ is obtained by a standard construction: For finite path $\widehat{\pi}$ we set $\Pr^{\mathcal{D}}(\mathsf{Cyl}(\widehat{\pi})) = \iota_{\text{init}}(\widehat{\pi}[0]) \cdot \prod_{k=0}^{|\widehat{\pi}|-1} \mathbf{P}(\widehat{\pi}[k], \widehat{\pi}[k+1])$, where the cylinder set $\mathsf{Cyl}(\widehat{\pi}) = \{\pi \in \mathsf{Paths}^{\mathcal{D}} \mid \forall i \in \{0, \ldots, |\widehat{\pi}|\} \colon \pi[i] = \widehat{\pi}[i]\}$ contains all possible infinite continuations of $\widehat{\pi}$. We write $\Pr^{\mathcal{D}}_{s}$ for the probability measure induced by $\mathcal{D}$ with the initial distribution assigning probability 1 to $s \in S$. We use LTL-style notation for measurable sets of infinite paths. For $R, T \subseteq S$ and $k \in \mathbb{N}$, let $R \,\mathsf{U}^{=k}\, T = \{\pi \in \mathsf{Paths}^{\mathcal{D}} \mid \pi[k] \in T \wedge \forall i < k \colon \pi[i] \in R\}$ be the set of infinite paths that visit a state $s \in T$ in the k-th step while only visiting states in R before. We also define $R \,\mathsf{U}\, T = \bigcup_{k\geq 0} R \,\mathsf{U}^{=k}\, T$, $\Diamond T = S \,\mathsf{U}\, T$ and $\Diamond^{=k} T = S \,\mathsf{U}^{=k}\, T$.

Expected Rewards. A (non-negative) *random variable* over the probability space induced by $\mathcal{D}$ is a measurable function $\mathsf{v} \colon \mathsf{Paths}^{\mathcal{D}} \to \overline{\mathbb{R}}_{\geq 0}$. Its *expected value* is given by the Lebesgue integral $\mathbb{E}^{\mathcal{D}}[\mathsf{v}] = \int_{\mathsf{Paths}^{\mathcal{D}}} \mathsf{v} \, d\Pr^{\mathcal{D}}$. We write $\mathbb{E}^{\mathcal{D}}_{s}$ for the expectation obtained under $\Pr^{\mathcal{D}}_{s}$. The *total reward* w.r.t. a reward structure

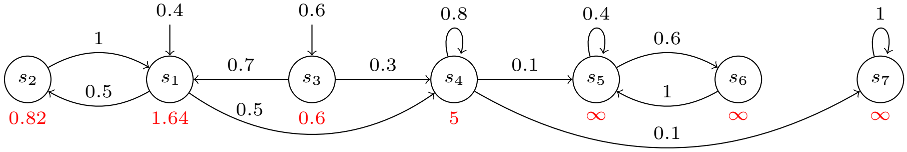


Fig. 2: Running example DTMC. The individual EVTs are below the states.

$\mathsf{rew}\colon S \to \mathbb{R}_{\geq 0}$ is defined by the random variable $\mathsf{tr}_{\mathsf{rew}}\colon \mathsf{Paths}^{\mathcal{D}} \to \overline{\mathbb{R}}_{\geq 0}$ with $\mathsf{tr}_{\mathsf{rew}}(\pi) = \sum_{k=0}^{\infty} \mathsf{rew}(\pi[k])$; the *expected total reward* is $\mathbb{E}^{\mathcal{D}}[\mathsf{tr}_{\mathsf{rew}}]$.

Connectivity in DTMCs. A *strongly connected component (SCC)* of a DTMC $\mathcal{D}$ is a set of states $C \subseteq S$ such that any $s, t \in C$ are mutually reachable, i.e., $\Pr_s(\lozenge\{t\}) > 0$ and $\Pr_t(\lozenge\{s\}) > 0$, and there is no proper subset of C that satisfies this property. An SCC C is called *bottom SCC (BSCC)* if no state outside C is reachable from C. In the following, $\mathsf{SCC}^{\mathcal{D}}$ denotes the set of SCCs of $\mathcal{D}$. We call a DTMC *absorbing* if all its BSCCs are singleton sets, and *irreducible* if $\mathsf{SCC}^{\mathcal{D}} = \{S\}$. The SCCs are ordered by a strict partial order $\hookrightarrow$ based on the topology of the DTMC, where $C' \hookrightarrow C$ if and only if $\Pr_{s'}(\lozenge C) > 0$ for some $s' \in C'$ and $C \neq C'$. An *SCC chain* of $\mathcal{D}$ is a sequence $\kappa = C_0 \hookrightarrow C_1 \hookrightarrow \cdots \hookrightarrow C_k$ of SCCs $C_0, C_1, \ldots, C_k \in \mathsf{SCC}^{\mathcal{D}}$, where $k \geq 0$. The set of all SCC chains in $\mathcal{D}$ is denoted by $\mathsf{Chains}^{\mathcal{D}}$, and $\mathsf{Chains}^{\mathcal{D}}_{\mathrm{tr}}$ denotes the set of SCC chains that do not contain a BSCC. The *length* of SCC chain $\kappa = C_0 \hookrightarrow C_1 \hookrightarrow \cdots \hookrightarrow C_k$ is $|\kappa| = k$.

A state s is called *transient* if the probability that the DTMC, starting from s, will ever return to s is strictly less than one, otherwise, s is a *recurrent* state. Thus, in a finite MC, recurrent states are precisely the states contained in the BSCCs whereas the transient states coincide with non-BSCC states. The sets of recurrent and transient states in a DTMC are denoted by S_{re} and S_{tr}, respectively.

Example 1. In the DTMC $\mathcal{D}$ depicted in Figure 2, states s_1, s_2, s_3, s_4 are transient, and s_5, s_6, s_7 are recurrent. Also, $\mathsf{SCC}^{\mathcal{D}} = \{\{s_1, s_2\}, \{s_3\}, \{s_4\}, \{s_5, s_6\}, \{s_7\}\}$, where only $\{s_5, s_6\}$ and $\{s_7\}$ are BSCCs. An example SCC chain is $\{s_3\} \hookrightarrow \{s_1, s_2\} \hookrightarrow \{s_4\} \hookrightarrow \{s_7\}$; its length is 3.

Stationary distributions. The stationary distribution (also referred to as steady-state or long-run distribution) is a probability distribution that specifies the fraction of time spent in each state in the long run (see, e.g., [5, Def. 10.79]).

Definition 2. *The* stationary *distribution of DTMC $\mathcal{D}$ is given by $\theta^{\mathcal{D}} \in [0,1]^{|S|}$ with $\theta^{\mathcal{D}}(s) = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \Pr(\lozenge^{=k}\{s\})$.*

If $\mathcal{D}$ is irreducible, the stationary distribution is given by the unique eigenvector $\theta^{\mathcal{D}}$ satisfying $\theta^{\mathcal{D}} = \theta^{\mathcal{D}} \cdot \mathbf{P}$ and $\sum_{s\in S} \theta^{\mathcal{D}}(s) = 1$, see, e.g., [39, Thm. 4.18]. If $\mathcal{D}$ is reducible, $\theta^{\mathcal{D}}$ can be obtained by combining a reachability analysis and the eigenvector computation for each BSCC individually, see Section 5.

Definition 3. *A* continuous-time Markov chain (CTMC) *is a quadruple $\mathcal{C} = (S, \mathbf{P}, \iota_{\mathrm{init}}, \boldsymbol{r})$, where $(S, \mathbf{P}, \iota_{\mathrm{init}})$ is a DTMC and $\boldsymbol{r}\colon S \to \mathbb{R}_{>0}$ defines exit rates.*

CTMCs extend DTMCs by assigning the rate of an exponentially distributed residence time to each state $s \in S$. We denote the *embedded DTMC* of a CTMC $\mathcal{C}$ by $\mathsf{emb}(\mathcal{C}) = (S, \mathbf{P}, \iota_{\text{init}})$. The semantics are defined in the usual way (see, e.g., [4,40]). An *infinite timed path* in a CTMC $\mathcal{C} = (S, \mathbf{P}, \iota_{\text{init}}, \boldsymbol{r})$ is a sequence $\pi = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} \cdots$ consisting of states $s_0, s_1, \ldots \in S$ and time instances $\tau_0, \tau_1 \ldots \in \mathbb{R}_{\geq 0}$, such that $\mathbf{P}\big(s_i, s_{i+1}\big) > 0$ for all $i \in \mathbb{N}$. We denote the time τ_i spent in state s_i by $time_i(\pi)$. Notations $\pi[i]$ and $\mathsf{Paths}^{\mathcal{C}}$ are as for DTMCs. The probability measure of $\mathcal{C}$ over infinite timed paths [4] is denoted by $\Pr^{\mathcal{C}}$. In a CTMC, the *total reward* w.r.t. a reward structure $\mathsf{rew}\colon S \to \mathbb{R}_{\geq 0}$ is the random variable $\mathsf{tr}_{\mathsf{rew}}\colon \mathsf{Paths}^{\mathcal{C}} \to \overline{\mathbb{R}}_{\geq 0}$ with $\mathsf{tr}_{\mathsf{rew}}(\pi) = \sum_{i=0}^{\infty} \mathsf{rew}(\pi[i]) \cdot time_i(\pi)$.

3 Expected Visiting Times

We provide characterizations of expected visiting times for a fixed DTMC $\mathcal{D} = (S, \mathbf{P}, \iota_{\text{init}})$. Omitted proofs are in the extended version of this paper [47].

Definition 4. *The* expected visiting time (EVT) *of a state $s \in S$ is the expected value $\mathbb{E}^{\mathcal{D}}[\mathsf{vt}_s]$ of the random variable vt_s with $\mathsf{vt}_s(\pi) = \sum_{k=0}^{\infty} [\![\pi[k] = s]\!]$.*

Example 2. The EVTs of the DTMC from Figure 2 are depicted below its states.

Intuitively, random variable vt_s *counts* the number of times state s occurs on an infinite path. Consequently, the EVTs of unreachable states and reachable recurrent states in a DTMC are always 0 and ∞, respectively. For this reason we focus on the EVTs of the transient states S_{tr}. The following lemma provides an alternative characterization of EVTs in terms of expected total rewards.

Lemma 1. *For a fixed $s \in S$ and $x \in \mathbb{R}_{>0}$ the reward structure $\mathsf{rew}\colon S \to \mathbb{R}_{\geq 0}$ given by $\mathsf{rew}(t) = x \cdot [\![t = s]\!]$ satisfies $\mathbb{E}[\mathsf{vt}_s] = \frac{1}{x} \cdot \mathbb{E}[\mathsf{tr}_{\mathsf{rew}}]$.*

By Lemma 1, EVTs can be obtained using existing algorithms for expected total rewards. This approach is, however, inefficient for computing the EVTs of multiple states since it requires solving an equation system for each single state.

Next, we elaborate on EVTs for multiple states as a solution of a single linear equation system. In [36, Def. 3.2.2], EVTs are defined using the so-called *fundamental matrix* for absorbing DTMCs. The fundamental matrix contains as its coefficients for each possible start and target state s and t the EVT $\mathbb{E}_s[\mathsf{vt}_t]$.Computing the fundamental matrix explicitly becomes infeasible for large models as it requires determining the inverse of a $|S_{\text{tr}}| \times |S_{\text{tr}}|$ matrix. To obtain the vector $(\mathbb{E}^{\mathcal{D}}[\mathsf{vt}_s])_{s \in S_{\text{tr}}}$ of EVTs that take the initial distribution of $\mathcal{D}$ into account, it suffices to solve an equation system which is linear in the size of the DTMC. The same equation system arises by applying the dual linear program for expected rewards in MDPs [49, Ch. 7.2.7] to the special case of DTMCs.

Theorem 1 ([36, Cor. 3.3.6]). *$(\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\text{tr}}}$ is the* unique *solution $(\mathbf{x}(s))_{s \in S_{\text{tr}}}$ of the following equation system: $\forall s \in S_{\text{tr}}\colon \mathbf{x}(s) = \iota_{\text{init}}(s) + \sum_{t \in S_{\text{tr}}} \mathbf{P}(t, s) \cdot \mathbf{x}(t)$.*

Intuitively, this equation system shows that a state s can be visited initially and that it can receive visits from its predecessor states, i.e., the EVT is computed by considering the *incoming* transitions to a state. As a consequence, we obtain that the EVTs of the transient states are always finite. In particular, if $s \in S_{\text{tr}}$ is reachable, then $\mathbb{E}[\mathsf{vt}_s] \in \mathbb{R}_{>0}$, and otherwise $\mathbb{E}[\mathsf{vt}_s] = 0$.

Example 3. Reconsider the DTMC from Figure 2 with transient states $S_{\text{tr}} = \{s_1, s_2, s_3, s_4\}$. The EVTs of S_{tr} are the unique solution $(\mathbf{x}(s))_{s \in S_{\text{tr}}}$ of

$$\mathbf{x}(s_1) = 0.4 + 1.0 \cdot \mathbf{x}(s_2) + 0.7 \cdot \mathbf{x}(s_3) \qquad \mathbf{x}(s_2) = 0.5 \cdot \mathbf{x}(s_1)$$
$$\mathbf{x}(s_4) = 0.5 \cdot \mathbf{x}(s_1) + 0.3 \cdot \mathbf{x}(s_3) + 0.8 \cdot \mathbf{x}(s_4) \qquad \mathbf{x}(s_3) = 0.6$$

Expected visiting times in CTMCs. Following [37], we define the EVT of a state $s \in S^{\mathcal{C}}$ of a CTMC $\mathcal{C}$ as the expected value $\mathbb{E}^{\mathcal{C}}[\mathsf{vt}_s]$ of the random variable $\mathsf{vt}_s \colon \mathsf{Paths}^{\mathcal{C}} \to \overline{\mathbb{R}}_{\geq 0}$ with $\mathsf{vt}_s(\pi) = \sum_{k=0}^{\infty} [\![\pi[k] = s]\!] \cdot \mathit{time}_k(\pi)$. Intuitively, vt_s considers the total time the system spends in state s. Computing EVTs in CTMCs reduces to the discrete-time case: The EVT of state s coincides with the EVT in the embedded DTMC weighted by the expected residence time $\frac{1}{\boldsymbol{r}(s)}$ in s:

Theorem 2. *For all states $s \in S$, it holds that* $\mathbb{E}^{\mathcal{C}}[\mathsf{vt}_s] = \frac{1}{\boldsymbol{r}(s)} \cdot \mathbb{E}^{\mathsf{emb}(\mathcal{C})}[\mathsf{vt}_s]$.

Theorem 2 implies that all results and algorithms to compute ETVs in DTMCs are readily applicable to CTMCs, too. We thus focus on DTMCs in the remainder.

4 Accurately Computing EVTs

In this section, we discuss algorithms to compute EVTs approximately: An unsound value iteration algorithm (Section 4.1), its sound interval iteration extension (Section 4.2), and finally a topological, i.e., SCC-wise algorithm (Section 4.3). Since the EVTs for recurrent states are always either 0 or ∞, we focus on the EVTs of the transient states. Omitted proofs are in the extended version [47].

4.1 Value Iteration

Value Iteration (VI) was originally introduced to approximate expected rewards in MDPs [7]. In a broader sense, VI simply refers to iterating a function $f \colon \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ (called *Bellman operator* in the MDP setting) from some given initial vector $\mathbf{x}^{(0)}$, i.e., to compute the sequence $\mathbf{x}^{(1)} = f(\mathbf{x}^{(0)}), \mathbf{x}^{(2)} = f(\mathbf{x}^{(1)})$, etc. Instances of VI are usually set up such that the sequence converges to a (generally non-unique) fixed point $\mathbf{x} = f(\mathbf{x})$. In this paper, we only consider VI for the case where f is a linear function $f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$, where $\mathbf{A}$ and $\mathbf{b}$ are a matrix and a vector, respectively. A fixed point $\mathbf{x}$ of f is then a solution of the linear equation system $(\mathbf{I} - \mathbf{A})\mathbf{x} = \mathbf{b}$. Other iterative methods for solving linear equation systems such as the Jacobi or Gauss-Seidel method can be considered optimized variants of VI, and are applicable in our setting as well, see [47].

Value iteration for EVTs. For EVTs, the function iterated during VI is as follows:

Algorithm 1: Value iteration for EVTs without precision guarantee.

```
Input: DTMC $\mathcal{D}$, $\mathbf{x}^{(0)} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$, $crit \in \{abs, rel\}$, $\epsilon > 0$
Output: $\mathbf{x} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$
for $k = 1, 2, 3, \dots$ do
    $\mathbf{x}^{(k)} \leftarrow \Phi(\mathbf{x}^{(k-1)})$ // $\mathbf{x}^{(k)}(s) = \iota_{\mathrm{init}}(s) + \sum_{t \in S_{\mathrm{tr}}} \mathbf{x}^{(k-1)}(t) \cdot \mathbf{P}(t, s)$, $s \in S_{\mathrm{tr}}$
    if $\mathsf{diff}^{crit}(\mathbf{x}^{(k-1)}, \mathbf{x}^{(k)}) \leq \epsilon$ then return $\mathbf{x}^{(k)}$
```

Definition 5. *The* EVTs-operator $\Phi\colon \mathbb{R}^{|S_{\mathrm{tr}}|} \to \mathbb{R}^{|S_{\mathrm{tr}}|}$ *for DTMC* $\mathcal{D}$ *is defined as*

$$\Phi(\mathbf{x}) = \Big(\iota_{\mathrm{init}}(s) + \sum_{t \in S_{\mathrm{tr}}} \mathbf{x}(t) \cdot \mathbf{P}(t, s)\Big)_{s \in S_{\mathrm{tr}}} .$$

The above definition is motivated by Theorem 1. The following result, which is analogous to [49, Thm. 6.3.1], means that VI for EVTs (stated explicitly as Algorithm 1 for the sake of concreteness) works for arbitrary initial vectors.

Theorem 3. *The EVTs-operator from Definition 5 has the following properties:*
(i) $(\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}$ *is the unique fixed point of* Φ.
(ii) *For all* $\mathbf{x}^{(0)} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$ *we have* $\lim_{k \to \infty} \Phi^{(k)}(\mathbf{x}^{(0)}) = (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}$.

When to stop VI? A general issue with value iteration is that even if the generated sequence converges to the desired fixed point in the limit, it is not easy to determine how many iterations are necessary to obtain an ϵ-precise result. An ad hoc solution, which is implemented in probabilistic model checkers such as Storm [32], prism [41], and mcsta [29], is to stop the iteration once the difference between two consecutive approximations is small, i.e., the number of iterations is the smallest $k > 0$ such that $\mathsf{diff}(\mathbf{x}^{(k)}, \mathbf{x}^{(k-1)}) < \epsilon$ for some predefined fixed $\epsilon > 0$. Common choices for the distance diff between vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{|S|}$ are the *absolute difference* $\mathsf{diff}^{abs}(\mathbf{x}, \mathbf{y}) = \mathsf{max}_{s \in S} |\mathbf{x}(s) - \mathbf{y}(s)|$, and the *relative difference*

$$\mathsf{diff}^{rel}(\mathbf{x}, \mathbf{y}) = \mathsf{max}_{s \in S} \left| \frac{\mathbf{x}(s) - \mathbf{y}(s)}{\mathbf{y}(s)} \right| ,$$

where by convention $0/0 = 0$ and $a/0 = \infty$ for $a \neq 0$. As pointed out by various authors [59,27,6,43,50], there exist instances where the iteration terminates with a result which vastly differs from the true fixed point, even if ϵ is small (e.g. $\epsilon = 10^{-6}$). An example of this for the EVT variant of VI is given in [47].

4.2 Interval Iteration

Interval iteration (II) [43,27,6] is an extension of VI that formally guarantees ϵ-close results for all possible inputs. The general idea of II is to construct *two* sequences of vectors $(\mathbf{l}^{(k)})_{k \in \mathbb{N}}$ and $(\mathbf{u}^{(k)})_{k \in \mathbb{N}}$ such that for all $k \in \mathbb{N}$ we have $\mathbf{l}^{(k)} \leq \mathbf{x} \leq \mathbf{u}^{(k)}$, where $\mathbf{x}$ is the desired fixed point solution. II can be stopped with precision guarantee ϵ once it detects that $\mathsf{diff}(\mathbf{l}^{(k)}, \mathbf{u}^{(k)}) \leq \epsilon$.

Initial bounds for II. In general, II requires initial vectors $\mathbf{l}^{(0)}$ and $\mathbf{u}^{(0)}$ which are already sound (but perhaps very crude) lower and upper bounds on the solution. In the case of EVTs, we can use $\mathbf{l}^0 = \mathbf{0}$. Finding an upper bound $\mathbf{u}^{(0)} \geq (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}$ is more involved since EVTs may be unboundedly large in general. We solve this issue using a technique from [6]. *II for EVTs.* In Lemma 2 below we show that once we have found initial bounds $\mathbf{l}^{(0)}$ and $\mathbf{u}^{(0)}$, we can readily perform a sound II for EVTs by simply iterating the operator Φ from Definition 5 on $\mathbf{l}^{(0)}$ and $\mathbf{u}^{(0)}$ in parallel. Inspired by [6], we propose the following optimization to speed up convergence: Whenever Φ *decreases* the current lower bound in some entries, we retain the old values for these entries (and similar for upper bounds). The next definition formalizes this.

Definition 6. *The* Max and Min EVTs-operators $\Phi_{max}, \Phi_{min} \colon \mathbb{R}^{|S_{\mathrm{tr}}|} \to \mathbb{R}^{|S_{\mathrm{tr}}|}$ *are defined by* $\Phi_{max}(\mathbf{x}) = \mathsf{max}\{\mathbf{x}, \Phi(\mathbf{x})\} = (\mathsf{max}\{\mathbf{x}(s), (\Phi(\mathbf{x}))(s)\})_{s \in S_{\mathrm{tr}}}$ *and* $\Phi_{min}(\mathbf{x}) = \mathsf{min}\{\mathbf{x}, \Phi(\mathbf{x})\} = (\mathsf{min}\{\mathbf{x}(s), (\Phi(\mathbf{x}))(s)\})_{s \in S_{\mathrm{tr}}}$.

The following result is analogous to [6, Lem. 3.3]:

Lemma 2. *Let* $\mathbf{u}, \mathbf{l} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$ *with* $\mathbf{l} \leq (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}} \leq \mathbf{u}$. *Then,*

(i) $\Phi^{(k)}_{max}(\mathbf{l}) \leq \Phi^{(k+1)}_{max}(\mathbf{l})$ *and* $\Phi^{(k)}_{min}(\mathbf{u}) \geq \Phi^{(k+1)}_{min}(\mathbf{u})$ *for all* $k \in \mathbb{N}$.

(ii) $\Phi^{(k)}(\mathbf{l}) \leq \Phi^{(k)}_{max}(\mathbf{l}) \leq (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}} \leq \Phi^{(k)}_{min}(\mathbf{u}) \leq \Phi^{(k)}(\mathbf{u})$ *for all* $k \in \mathbb{N}$.

(iii) $\lim_{k \to \infty} \Phi^{(k)}_{max}(\mathbf{l}) = \lim_{k \to \infty} \Phi^{(k)}_{min}(\mathbf{u}) = (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}$.

The resulting II algorithm for EVTs is presented as Algorithm 2. Note the following additional optimization: The algorithm stops as soon as $\mathsf{diff}^{crit}(\mathbf{u}^{(k)}, \mathbf{l}^{(k)}) \leq 2\epsilon$ and returns the mean of $\mathbf{u}^{(k)}$ and $\mathbf{l}^{(k)}$, ensuring that the absolute or relative difference between $(\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}$ and the returned result is at most ϵ.

Example 4. We illustrate a run of Algorithm 2 on the DTMC from Figure 2, with *crit* = *abs* and $\epsilon = 0.05$ (the following numbers are rounded to 4 decimal digits):

k	$\mathbf{l}^{(k)}$	$\mathbf{u}^{(k)}$	$\mathsf{diff}^{abs}(\mathbf{l}^{(k)}, \mathbf{u}^{(k)})$
0	(0.000, 0.000, 0.000, 0.000)	(2.000, 2.000, 1.000, 5.000)	5.000
1	(0.400, 0.000, 0.600, 0.000)	(2.000, 1.000, 0.600, 5.000)	5.000
2	(0.820, 0.200, 0.600, 0.380)	(1.820, 1.000, 0.600, 5.000)	4.602
	...	...	...
22	(1.639, 0.819, 0.600, 4.899)	(1.640, 0.820, 0.600, 5.000)	0.101
23	(1.639, 0.819, 0.600, 4.919)	(1.640, 0.820, 0.600, 5.000)	**0.081**

After $k = 23$ iterations, the algorithm stops as $\mathsf{diff}^{abs}(\mathbf{l}^{(k)}, \mathbf{u}^{(k)}) = 0.081 \leq 2 \cdot \epsilon$ and outputs the mean $\frac{1}{2}(\mathbf{l}^{(23)} + \mathbf{u}^{(23)})$.

Theorem 4 (Correctness of Algorithm 2). *Given an input DTMC* $\mathcal{D}$, *initial vectors* $\mathbf{l}^{(0)}$, $\mathbf{u}^{(0)}$ *with* $\mathbf{l}^{(0)} \leq (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}} \leq \mathbf{u}^{(0)}$, *crit* $\in \{abs, rel\}$, *and a threshold* $\epsilon > 0$, *Algorithm 2 terminates and returns a vector* $\mathbf{x}^{res} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$ *satisfying* $\mathsf{diff}^{crit}(\mathbf{x}^{res}, (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}) \leq \epsilon$.

Remark 1. The monotonicity of the sequences $(\Phi^{(k)}_{max}(\mathbf{l}))_{k \in \mathbb{N}}$ and $(\Phi^{(k)}_{min}(\mathbf{u}))_{k \in \mathbb{N}}$ (see Lemma 2 (i)) is not used in the proof of Theorem 4. By Lemma 2 (ii), we can replace Φ_{max} and Φ_{min} with Φ in Algorithm 2 and still obtain ϵ-sound results. However, using Φ_{max} and Φ_{min} instead of Φ can lead to faster convergence.

Algorithm 2: Interval iteration for EVTs with precision guarantee.

```
Input: DTMC $\mathcal{D}$, $\mathbf{l}^{(0)} \le (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}$, $\mathbf{u}^{(0)} \ge (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}$, $crit \in \{abs, rel\}, \epsilon > 0$
Output: $\mathbf{x} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$ with $\mathsf{diff}^{crit}(\mathbf{x}, (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}) \le \epsilon$
for $k = 1, 2, \ldots$ do
    $\mathbf{l}^{(k)} \leftarrow \Phi_{max}(\mathbf{l}^{(k-1)})$ ;   $\mathbf{u}^{(k)} \leftarrow \Phi_{min}(\mathbf{u}^{(k-1)})$
    if $\mathsf{diff}^{crit}(\mathbf{u}^{(k)}, \mathbf{l}^{(k)}) \le 2 \cdot \epsilon$ then return $\frac{1}{2}(\mathbf{l}^{(k)} + \mathbf{u}^{(k)})$
```

4.3 Topological Algorithm

To increase the efficiency of VI for the analysis of rewards and probabilities in MDPs, several authors have proposed topological VI [15,16], which is also known as blockwise VI [14]. The idea is to avoid the analysis of the complete model at once and instead consider the strongly connected components (SCCs) sequentially based on the order relation $\hookrightarrow$. We lift this approach to EVTs and in particular consider error propagations when approximative methods are used.

SCC Restrictions. To formalize the topological approach, we introduce the *SCC restriction* $\mathcal{D}|_C[\mathbf{x}]$ of DTMC $\mathcal{D}$ to $C \in \mathsf{SCC}^{\mathcal{D}}$ with parameters $\mathbf{x} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$. Intuitively, $\mathcal{D}|_C[\mathbf{x}]$ is a DTMC-like model obtained by restricting $\mathcal{D}$ to the states C and assigning each $s \in C$ the "initial value" $\iota_{\mathrm{init}}^{\mathcal{D}}(s) + \sum_{s' \in S \setminus C} \mathbf{P}^{\mathcal{D}}(s', s) \cdot \mathbf{x}(s')$. The idea is that $\mathbf{x}$ is an approximation of the EVTs of the predecessor SCCs $C' \hookrightarrow C$. We also define $\mathcal{D}|_C = \mathcal{D}|_C[\mathbf{x}]$ with $\mathbf{x}(s) = \mathbb{E}^{\mathcal{D}}[\mathsf{vt}_s]$ (i.e., the *exact* EVT) for each state $s \in S$ that can reach C with positive probability in one step. See [47, Definition 7] for more formal details.

Example 5. The SCC restriction of the DTMC $\mathcal{D}$ from Figure 2 to the SCC $C = \{s_1, s_2\}$ with parameters $\mathbf{x} = (\mathbb{E}^{\mathcal{D}}[\mathsf{vt}_s])_{s \in S}$ is depicted in Figure 3. Note that the initial values depend only on the initial distribution $\iota_{\mathrm{init}}^{\mathcal{D}}$ and the $\mathbf{x}$-values of the states that can reach the SCC C in one step.

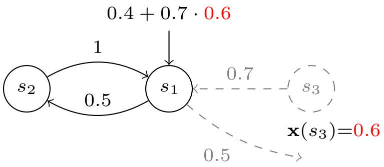


Fig. 3: SCC restriction.

Lemma 3. *For non-bottom SCC C and $s \in C$ we have $\mathbb{E}^{\mathcal{D}}[\mathsf{vt}_s] = \mathbb{E}^{\mathcal{D}|_C}[\mathsf{vt}_s]$.*

Remark 2. Since a parametric SCC restriction $\mathcal{D}|_C[\mathbf{x}]$ is defined for arbitrary vectors $\mathbf{x} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$, the initial values do not necessarily form a probability distribution. Strictly speaking, this means that $\mathcal{D}|_C[\mathbf{x}]$ is not a DTMC in general. Thus, by abuse of notation, we define the "EVTs" in the parametric SCC restriction $\mathcal{D}|_C[\mathbf{x}]$ as the unique solution of the following linear equation system: For all $s \in C$: $\mathbf{x}(s) = \iota_{\mathrm{init}}^{\mathcal{D}|_C[\mathbf{x}]}(s) + \sum_{t \in S_{\mathrm{tr}}} \mathbf{P}^{\mathcal{D}|_C}(t, s) \cdot \mathbf{x}(t)$. To solve this system, we can still apply the methods described in Sections 4.1 and 4.2 without further ado.

Algorithm 3: Topological EVT algorithm with relative precision.

```
Input: DTMC $\mathcal{D}$, $\epsilon \geq 0$
Output: $\mathbf{x} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$ with $\mathsf{diff}^{rel}(\mathbf{x}, (\mathbb{E}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}) \leq \epsilon$
1 $\{C_1, \ldots, C_n\} \leftarrow \mathsf{SCC}^{\mathcal{D}}_{\mathrm{tr}}$  // obtain the non-bottom SCCs, $C_i \hookrightarrow C_j$ implies $i < j$
2 $\mathbf{x} \leftarrow \mathbf{0}$
3 $\delta \leftarrow \sqrt[L+1]{1+\epsilon} - 1$ // $L$ is the length of a longest non-bottom SCC chain
4 for $j = 1$ to $n$ do
5     compute $\widehat{\mathbf{x}}$ such that $\mathsf{diff}^{rel}((\mathbb{E}^{\mathcal{D}|_{C_j}[\mathbf{x}]}[\mathsf{vt}_s])_{s \in C_j}, \widehat{\mathbf{x}}) \leq \delta$ // use e.g. Alg. 2
6     for $s \in C_j$ do $\mathbf{x}(s) \leftarrow \widehat{\mathbf{x}}(s)$
7 return $\mathbf{x}$
```

We now describe an algorithm for computing the EVTs SCC-wise in topological order. The desired precision $\epsilon \geq 0$ is an input parameter ($\epsilon = 0$ is possible). Due to space limitations we only discuss relative precision, see [47] for an algorithm with absolute precision. The idea is to solve the linear equation systems tailored to the parametric SCC restrictions, each of which is constructed based on the analysis of the preceding SCCs. Algorithm 3 outlines the procedure.

The algorithm first decomposes the input DTMC $\mathcal{D}$ into its non-bottom SCCs: The function $\mathsf{SCC}^{\mathcal{D}}_{\mathrm{tr}}$ called in Line 1 returns the set $\{C_1, \ldots, C_n\}$ of SCCs of $\mathcal{D}$ which consist of transient states, i.e., the non-bottom SCCs. Furthermore, we assume that the SCCs are indexed such that $C_i \hookrightarrow C_j$ implies $i < j$. The algorithm considers each SCC in topological order. We assume that a "black box" can obtain the (approximate) EVTs in Line 5. Then, the vector $\mathbf{x} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$ is updated such that it contains the approximations of the EVTs in $\mathcal{D}$ upon termination. For the analysis of an SCC C_j for some $j \in \{1, \ldots, n\}$, the algorithm considers the parametric SCC restriction $\mathcal{D}|_{C_j}[\mathbf{x}]$, which is based on the result $\mathbf{x}(t)$ for states t in SCCs C_i that are topologically before C_j. For each parametric SCC restriction $\mathcal{D}|_{C_j}[\mathbf{x}]$, the algorithm computes the vector $(\mathbb{E}^{\mathcal{D}|_{C_j}[\mathbf{x}]}[\mathsf{vt}_s])_{s \in C}$ in Line 5. Then, the algorithm updates the corresponding entries in $\mathbf{x}$ in Line 6. After each non-bottom SCC has been considered, the algorithm terminates and returns the vector $\mathbf{x}$ that contains the updated value for each transient state. The following lemma provides an upper bound on the error that accumulates during the topological computation.

Lemma 4. *Let $\epsilon \in [0, 1)$ and let $\mathbf{x} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$ such that for every non-bottom SCC C, $(\mathbf{x}(s))_{s \in C}$ satisfies $\mathsf{diff}^{rel}\left((\mathbf{x}(s))_{s \in C}, (\mathbb{E}^{\mathcal{D}|_C[\mathbf{x}]}[\mathsf{vt}_s])_{s \in C}\right) \leq \epsilon$. Then*

$$\mathsf{diff}^{rel}\left(\mathbf{x}, (\mathbb{E}^{\mathcal{D}}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}\right) \leq (1+\epsilon)^{L+1} - 1,$$

where $L = \mathsf{max}_{\kappa \in \mathsf{Chains}^{\mathcal{D}}_{\mathrm{tr}}} |\kappa|$ is the largest length of a chain of non-bottom SCCs.

Theorem 5 (Correctness of Algorithm 3). *Algorithm 3 returns a vector $\mathbf{x}^{res} \in \mathbb{R}^{|S_{\mathrm{tr}}|}$ such that $\mathsf{diff}^{rel}(\mathbf{x}^{res}, (\mathbb{E}^{\mathcal{D}}[\mathsf{vt}_s])_{s \in S_{\mathrm{tr}}}) \leq \epsilon$.*

5 Stationary Distributions via EVTs

We show that EVTs can be used to determine *sound* approximations of the stationary distribution $\theta^{\mathcal{D}}$ of a (reducible) DTMC $\mathcal{D}$. It is known that $\theta^{\mathcal{D}}$ can be computed as follows (see, e.g., [39, Thm. 4.23]). For each BSCC B:

- Compute the reachability probability $\mathrm{Pr}^{\mathcal{D}}(\lozenge B)$ to B.
- Determine the stationary distribution $\theta^{\mathcal{D}|_B}$ of the DTMC restricted to B.

Then, we obtain the stationary distribution for the recurrent states s in the BSCC B: $\theta^{\mathcal{D}}(s) = \mathrm{Pr}^{\mathcal{D}}(\lozenge B) \cdot \theta^{\mathcal{D}|_B}(s)$. For the remaining transient states, we have $\theta^{\mathcal{D}}(s) = 0$. We show how both, $\mathrm{Pr}^{\mathcal{D}}(\lozenge B)$ and $\theta^{\mathcal{D}|_B}$, can be computed efficiently for every BSCC B using EVTs. We also elaborate on how relative errors propagate through the computation, allowing us to derive sound lower- and upper bounds for the stationary distribution $\theta^{\mathcal{D}}$. Omitted proofs are in the extended report [47].

Computing BSCC reachability probabilities. The *absorption probabilities*, i.e., the probability of reaching a singleton BSCC can be computed using EVTs [36]. By collapsing the BSCCs into a single state, a slightly generalised result is obtained:

Theorem 6 ([36, Thm. 3.3.7]). *For any BSCC B of a DTMC $\mathcal{D}$ it holds that* $\mathrm{Pr}(\lozenge B) = \sum_{s \in B} \left(\iota_{\mathrm{init}}(s) + \sum_{s' \in S_{\mathrm{tr}}} \mathbf{P}(s', s) \cdot \mathbb{E}[\mathsf{vt}_{s'}]\right)$.

Applying Theorem 6, we can compute the EVTs *once* to derive the reachability probabilities for *every* BSCC. Further, when using interval iteration from Section 4.2 to obtain $\mathbf{x} \in \mathbb{R}^{|S|}$ with $\mathsf{diff}^{rel}\left(\mathbf{x}, (\mathbb{E}[\mathsf{vt}_s])_{s \in S}\right) \leq \epsilon$ for some $\epsilon \in (0,1)$, the relative error does not increase when deriving the reachability probabilities, i.e., $\mathsf{diff}^{rel}\left(\sum_{s \in B}\left(\iota_{\mathrm{init}}(s) + \sum_{s' \in S_{\mathrm{tr}}} \mathbf{P}(s', s) \cdot \mathbf{x}(s)\right), \mathrm{Pr}(\lozenge B)\right) \leq \epsilon$.

Computing the stationary distribution within a BSCC. Next, we leverage EVTs to compute the stationary distribution $\theta^{\mathcal{B}}$ of an irreducible DTMC $\mathcal{B} = (S, \mathbf{P}, \iota_{\mathrm{init}})$. This method can be applied to derive the stationary distribution $\theta^{\mathcal{D}|_B}$ of $\mathcal{D}|_B$ since the latter is an irreducible DTMC for each BSCC B. Let $v \in S$ be an arbitrary state. We construct the DTMC $\mathcal{B}{\Rsh}^{\hat{v}}$ in which all incoming transitions of state v are redirected to a fresh absorbing state $\hat{v}$. Thus, its only BSCC is $\{\hat{v}\}$, all other states are transient. Formally, $\mathcal{B}{\Rsh}^{\hat{v}} = (S \uplus \{\hat{v}\}, \hat{\mathbf{P}}, \hat{\iota}_{\mathrm{init}})$, where $\hat{\mathbf{P}}(\hat{v}, \hat{v}) = 1$, $\hat{\mathbf{P}}(s, \hat{v}) = \mathbf{P}(s, v)$ and $\hat{\mathbf{P}}(s, v) = 0$ for all $s \in S$, $\hat{\mathbf{P}}(s, t) = \mathbf{P}(s, t)$ for all $s \in S, t \in S \setminus \{v\}$, and $\hat{\iota}_{\mathrm{init}}(v) = 1$.

Theorem 7. *The stationary distribution $\theta^{\mathcal{B}}$ of an irreducible DTMC $\mathcal{B}$ is given by* $\theta^{\mathcal{B}}(s) = \frac{\mathbb{E}^{\mathcal{B}{\Rsh}^{\hat{v}}}[\mathsf{vt}_s]}{\sum_{t \in S} \mathbb{E}^{\mathcal{B}{\Rsh}^{\hat{v}}}[\mathsf{vt}_t]}$. *Further, if* $\mathsf{diff}^{rel}\left(\mathbf{x}, (\mathbb{E}^{\mathcal{B}{\Rsh}^{\hat{v}}}[\mathsf{vt}_s])_{s \in S}\right) \leq \epsilon$ *for* $\mathbf{x} \in \mathbb{R}^{|S|}$ *and* $\epsilon \in (0,1)$, *then* $\mathsf{diff}^{rel}\left(\frac{\mathbf{x}}{\sum_{s \in S} \mathbf{x}(s)}, \theta^{\mathcal{B}}\right) \leq \frac{2\epsilon}{1-\epsilon}$.

The first part of Theorem 7 can also be established by considering the renewal processes embedded in $\mathcal{B}$ (see, e.g., [58, Theorem 2.2.3]).

Combining both steps. Theorems 6 and 7 and the interval iteration method yield approximations p_B and d_s for $\mathrm{Pr}^{\mathcal{D}}(\lozenge B)$ and $\theta^{\mathcal{D}|_B}(s)$, respectively, where B is a BSCC $s \in B$, and $\epsilon_1, \epsilon_2 \in (0,1)$ such that $\mathsf{diff}^{rel}\left(p_B, \mathrm{Pr}^{\mathcal{D}}(\lozenge B)\right) \leq \epsilon_1$ and $\mathsf{diff}^{rel}\left(d_s, \theta^{\mathcal{D}|_B}(s)\right) \leq \epsilon_2$. The product $p_B \cdot d_s$ approximates $\theta^{\mathcal{D}}(s) = \mathrm{Pr}^{\mathcal{D}}(\lozenge B) \cdot$

$\theta^{\mathcal{D}|_B}(s)$ such that $\mathsf{diff}^{rel}\big(p_B \cdot d_s, \theta^{\mathcal{D}}(s)\big) \le \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2$.

Example 6. We compute the stationary distribution of the running example DTMC $\mathcal{D}$ from Figure 2. Its only non-trivial BSCC is $\mathcal{B} = \mathcal{D}|_{\{s_5,s_6\}}$. The DTMC $\mathcal{B}{\upharpoonright}^{\hat{s_6}}$ along with its (exact) EVTs is depicted in Figure 4. We conclude that $\theta^{\mathcal{B}}$ is proportional to $(\frac{3}{5}, 1)$, i.e., $\theta^{\mathcal{B}} = (\frac{5}{8}, \frac{3}{8})$. Since the two BSCCs $\{s_5, s_6\}$ and $\{s_7\}$ are both reached with probability $\frac{1}{2}$, it follows that the stationary probabilities of the three recurrent states s_5, s_6, s_7 are $\frac{5}{16}$, $\frac{3}{16}$, and $\frac{1}{2}$, respectively.

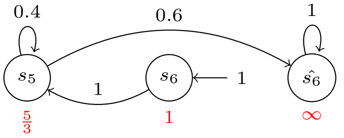


Fig. 4: The DTMC $\mathcal{B}{\upharpoonright}^{\hat{s_6}}$.

6 Conditional Expected Rewards

Theorem 6 states that the EVTs of the transient states of a DTMC $\mathcal{D}$ can be used to compute the probability to reach each individual BSCC of $\mathcal{D}$. We now generalize this result and show that the EVTs can also be used to compute the *total expected rewards conditioned on reaching each BSCC.*

The total expected reward conditioned on reaching a set $T \subseteq S$ of states with $\Pr^{\mathcal{D}}(\lozenge T) > 0$ is defined as:

$$\mathbb{E}^{\mathcal{D}}[\mathsf{tr}_{\mathsf{rew}} | \lozenge T] = \frac{1}{\Pr^{\mathcal{D}}(\lozenge T)} \int_{\lozenge T} \mathsf{tr}_{\mathsf{rew}} \, d\Pr^{\mathcal{D}}$$

Our next result asserts that given the EVTs of $\mathcal{D}$, all the values $\{\mathbb{E}^{\mathcal{D}}[\mathsf{tr}_{\mathsf{rew}} | \lozenge B] \mid B$ a BSCC of $\mathcal{D}\}$ can be computed by solving a *single* linear equation system (the standard approach is to solve one linear equation system *per BSCC* [5, Ch. 10]). For simplicity, we state the result only for BSCCs $\{r\}$ with a single absorbing state r, and for reward functions that assign zero reward to all recurrent (BSCC) states; this is w.l.o.g. as larger BSCCs can be collapsed, and positive reward in a (reachable) BSCC causes the conditional expected reward w.r.t. this BSCC to be ∞, rendering numeric computations unnecessary.

Theorem 8. *Let* $\mathsf{rew}\colon S \to \mathbb{Q}_{\ge 0}$ *with* $\mathsf{rew}(S_{\mathrm{re}}) = \{0\}$. *Then the equation system*

$$\forall s \in S_{\mathrm{tr}}: \qquad \mathbf{y}(s) = \mathsf{rew}(s) \cdot \mathbb{E}^{\mathcal{D}}[\mathsf{vt}_s] + \sum_{t \in S_{\mathrm{tr}}} \mathbf{P}(t,s) \cdot \mathbf{y}(t)$$

has a unique solution $(\mathbf{y}(s))_{s \in S_{\mathrm{tr}}}$ *and for all absorbing* $r \in S$, $\Pr^{\mathcal{D}}(\lozenge\{r\}) > 0$,

$$\mathbb{E}^{\mathcal{D}}[\mathsf{tr}_{\mathsf{rew}} | \lozenge\{r\}] = \frac{\sum_{t \in S_{\mathrm{tr}}} \mathbf{P}(t,r) \cdot \mathbf{y}(t)}{\iota_{\mathrm{init}}(r) + \sum_{t \in S_{\mathrm{tr}}} \mathbf{P}(t,r) \cdot \mathbb{E}^{\mathcal{D}}[\mathsf{vt}_t]} \,.$$

Theorem 8 assumes rational rewards as required in our proof in [47].

7 Experimental Evaluation

Implementation details. We integrated the presented algorithms for EVTs and stationary distributions in the model checker Storm [32]. The implementation is part of Storm's main release available at https://stormchecker.org. It uses explicit data structures such as sparse matrices and vectors. When computing EVTs, we can use value iteration (`VI`) or interval iteration (`II`) as presented in Algorithms 1 and 2. Alternatively, the corresponding linear equation systems can be solved using `LU` factorization (a direct method implemented in the Eigen library [25]) or `gmres` (a numerical method implemented in gmm++ [51]). Each EVT approach can be used in combination with the topological algorithm (`topo`) from Section 4.3. We use double precision floating point numbers. For `II`, the propagation of (relative) errors is respected in a way that the error of the end result does not exceed a user-defined threshold (here: $\epsilon = 10^{-3}$). Implementing `II` with safe rounding modes as in [28] is left for future work. The methods `gmres` and `VI` are configured with a fixed relative precision parameter (here: $\epsilon = 10^{-6}$). For `LU`, floating point errors are the only source of inaccuracies. We also consider an exact configuration $\texttt{LU}^X$ that uses rational arithmetic instead of floats.

Stationary distributions can be computed in Storm using the approaches `Classic`, `EVTreach`, and `EVTfull`. The `Classic` approach computes each BSCC reachability probability separately and the stationary distributions within the BSCCs are computed using the standard equation system [39, Thm. 4.18]. `EVTreach` and `EVTfull` implement our approaches from Section 5, where `EVTreach` only considers EVTs for BSCC reachability and `EVTfull` also derives the BSCC distributions from EVTs. As for EVTs, we use $\texttt{LU}^{(X)}$ or `gmres` to solve linear equation systems. For the BSCC reachability probabilities, a topological algorithm can be enabled as well. Using `EVTfull` with `II` yields sound approximations.

Experimental setup. The experiments ran on an Intel® Xeon® Platinum 8160 Processor limited to 4 cores and 12 GB of memory. The time timeout was set to 30 minutes. Our implementation does not use multi-threading.

7.1 Verifying the Fast Dice Roller

Recall Lumbroso's Fast Dice Roller [42] from Section 1. For a given parameter $N \geq 1$, we verify that the resulting distribution is indeed uniform by computing the stationary distribution of the corresponding DTMC which, for this model family, coincides with the individual BSCC reachability probabilities as each BSCC consists of a single state. We conducted our experiments with an equivalent state-reduced variant of the Fast Dice Roller which we obtained automatically using the technique from [60], i.e., for every given N, our variant has fewer states than the original algorithm from [42]. The plot in Figure 5 shows for

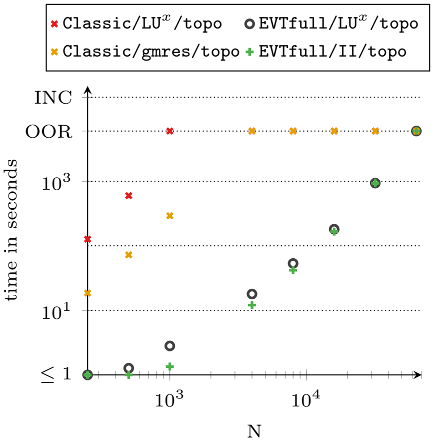


Fig. 5: Fast Dice Roller Results.

different values of N (x-axis) the runtime (y-axis) of the approaches `Classic` (using `gmres` or `LU`^{x}) and `EVTfull` (using `II` or `LU`^{x}), all using topological algorithms. Our novel `EVTfull` approach is significantly faster than the `Classic` method, enabling us to verify large instances with up to 4 800 255 states ($N = 32\,000$) within the time limit. In particular, we can compute the values using exact arithmetic as `LU`^{x} has runtimes similar to those of `II` in the `EVTfull` approach.

7.2 Performance Comparison

To evaluate the various approaches, we computed EVTs and stationary distributions for all applicable finite models of type DTMC or CTMC of the Quantitative Verification Benchmark Set (QVBS) [31]: We excluded 2 model families for which none of the tested parameter valuations allowed any algorithm to complete using the available resources and for the EVT computation we excluded 8 models that do not contain any transient states. In addition to QVBS, we included Lumbroso's Fast Dice Roller [42] and the handcrafted models (`branch` and `loop`) introduced in [45]. We considered multiple parameter valuations yielding a total of 62 instances (including 12 CTMCs) for computing EVTs and 79 instances (including 28 CTMCs) for computing the stationary distribution.

Our experiments for stationary distributions also include the implementations of the naive and guided sampling approaches (`ap-naive` and `ap-sample`) from [45] as well as the implementation of the `classic` approach in prism [41][2] as external baselines. These tools do not support the Jani models and can not compute EVTs.

We measured the runtime of the respective computation including the time for model construction. In cases where the exact results are known (using exact computations via `LU`^{X}), we consider results as *incorrect* if the relative difference to the exact value is greater than 10^{-3}. Results provided by the tool from [45] and by prism are not checked for correctness. We set the relative termination threshold of `gmres` and `VI` to $\epsilon = 10^{-6}$ to compensate for inaccuracies of the unsound methods. When using `II` or prism, a relative precision of $\epsilon = 10^{-3}$ was requested. For the implementation of [45] — which exclusively support absolute precision — we set the threshold to $\epsilon = 10^{-3}$. See [47] for more experiments.

Computing EVTs. The quantile plot at the top of Figure 6 indicates the time that is required for computing the EVTs in 62 models for different approaches. A point at position (x, y) indicates that the corresponding method solved the x^{th} fastest instance in y seconds, where only correctly solved instances are considered. The unsound methods `VI` and `gmres` produced 7 and 2 incorrect results, respectively. Furthermore, as errors accumulated, the topological variations of `VI` and `gmres` more frequently exceeded the threshold of 10^{-3}. The variants of `II` always produced correct results.

The plot indicates that (topological) `LU` and `gmres` outperform `VI` and `II` for easier instances. However, `II` catches up for the more intricate instances as it is more scalable than `LU` and always yields correct results. The exact method `LU`^{X} is significantly slower compared to the other methods. We also observe that

[2] We consider the `explicit` engine of prism v4.8 with the `Jacobi` method (default).

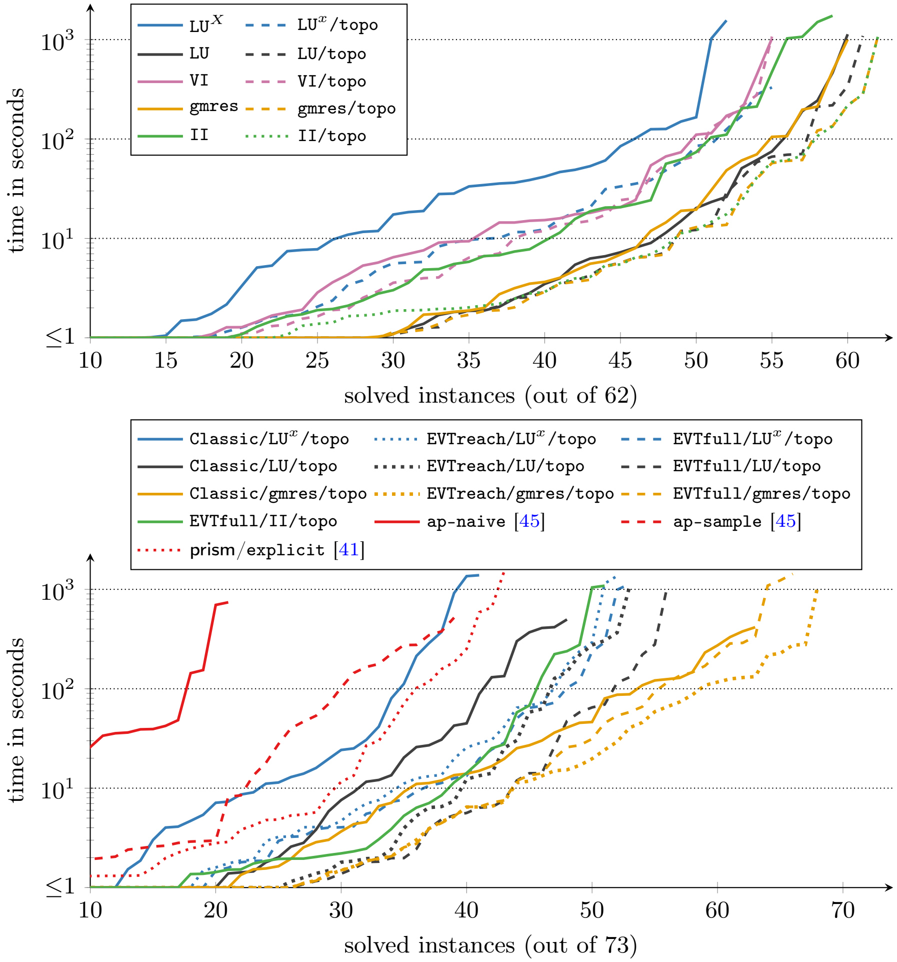


Fig. 6: Runtime comparison for EVTs (top) and stationary distributions (bottom).

the topological algorithms are superior to the non-topological variants. This is confirmed by the leftmost scatter plot in Figure 7 which compares the topological and non-topological variant of `II`. Here, each point (x, y) indicates an instance for which the methods on the x-axis and the y-axis respectively required x and y seconds to compute the EVTs. Instances that contain only singleton SCCs are depicted as triangles (▲), whereas cycles (●) represent the remaining instances. No incorrect results (INC) were obtained. The scatter plot in the middle in Figure 7 indicates that the iterative methods are more scalable than exact $\texttt{LU}^X$. We also see that models with millions of states can be solved in reasonable time.

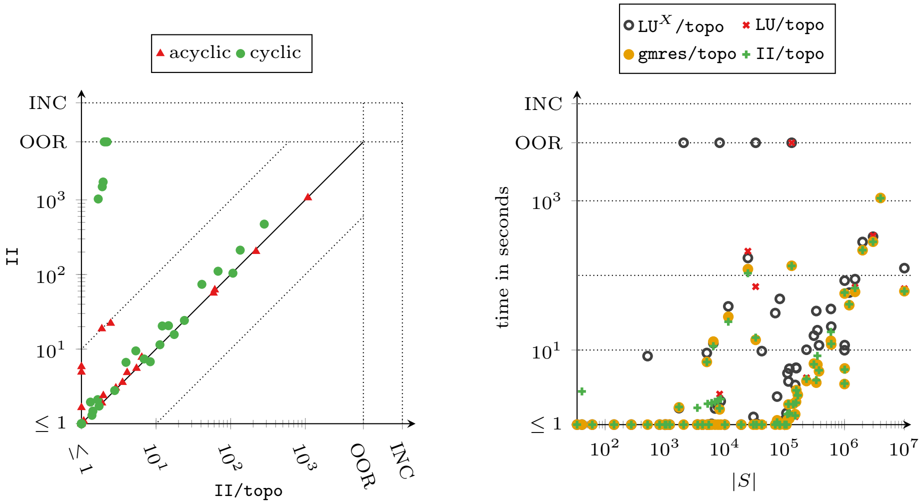


Fig. 7: Scatter plots showing the EVT computation runtime for standard and topological II (left) as well as the runtime of the EVT (right) approaches for different state space sizes $|S|$.

Computing stationary distributions. The quantile plot at the bottom of Figure 6 summarizes the runtimes for the different stationary distribution approaches[3]. We only consider the topological variants of the approaches of Storm as they were consistently faster. No incorrect results were observed in this experiment.

The plot indicates that the guided sampling method (ap-sample) from [45] and prism perform significantly better than ap-naive. However, all algorithms provided by Storm — except for Classic/LU^x — outperform the other implementations. For LU and GMRES, we observe that the EVTreach and EVTfull variants are significantly faster than the Classic approach. The EVTreach approach using gmres provides the fastest configuration, but is also the least reliable one in terms of accuracy. For the sound methods, we observe that the EVTfull approach with II is outperformed by LU combined with either EVTreach or EVTfull, where the latter shows the better performance.

8 Related Work

Computing stationary distributions. Other methods for the sound computation of the stationary distribution have been proposed in, e.g., [21,11,9]. In contrast to our work, they consider only subclasses of Markov chains: [21,11] introduce an (iterative) algorithm applicable to Markov chains with positive rows while [9] presents a technique limited to time-reversible Markov chains. The recent approach from [45] can also handle general Markov chains. Our technique ensures

[3] Six Jani [10] models that ap-naive, ap-sample, and prism do not support are omitted.

soundness with respect to both absolute and relative differences, whereas the approaches of [45] only consider absolute precision.

Other applications of EVTs. The authors of [36] have suggested to use EVTs for the *expected time to absorption*, i.e., the expected number of steps until the chain reaches an absorbing state. Indeed, this quantity is given by the sum of the vector $(\mathbb{E}[\mathsf{vt}_s])_{s\in S_{\mathrm{tr}}}$ [36, Thm. 3.3.5]. For acyclic DTMCs the EVT of a state coincides with the probability of reaching this state. This is relevant in the context of Bayesian networks [53,54] since inference queries in the network can be reduced to reachability queries by translating Bayesian networks into tree-like DTMCs. Existing procedures for multi-objective model checking of MDPs employ linear programming methods relying on the EVTs of state-action pairs [20,18,13,17]. EVTs are also employed in an algorithm proposed in [8] for LTL model checking of *interval Markov chains*. Moreover, EVTs have been leveraged for minimizing and learning DTMCs [1,2]. Further recent applications of EVTs to MDPs include verifying *cause-effect dependencies* [3], as well as an abstraction-refinement procedure that measures the importance of states based on the EVTs under a fixed policy [33]. [22] employs EVTs in the context of policy iteration in reward-robust MDPs.

9 Conclusion

We elaborated on the computation of EVTs in DTMCs and CTMCs: The EVTs in DTMCs can be determined by solving a linear equation system, while computing EVTs in CTMCs reduces to the discrete-time setting. We developed an iterative algorithm based on the value iteration [49] algorithm lacking assurance of precision. Building on interval iteration [27,6] — an algorithm for the sound computation of reachability probabilities and expected rewards — we developed an algorithm for approximating EVTs with accuracy guarantees. To enhance efficiency, we adapted a topological algorithm [15,16,14] to compute EVTs SCC-wise in topological order. We showed that EVTs enable the sound approximation of the stationary distribution and the efficient computation of conditional expected rewards. For future work, we want to extend our implementation provided in the model checker Storm [32] by symbolic computations. Another direction is to combine EVT-based computations with approximate verification approaches based on partially exploring relevant parts of the system [38,45]. We conjecture that EVTs serve as a good heuristic to identify significant sub-regions within the state space.

Data availability statement. The datasets generated and analyzed in this study and code to regenerate them are available in the accompanying artifact [46].

References

1. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: On the metric-based approximate minimization of Markov chains. In: ICALP. LIPIcs, vol. 80, pp. 104:1–104:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
2. Bacci, G., Ingólfsdóttir, A., Larsen, K.G., Reynouard, R.: Active learning of Markov decision processes using Baum-Welch algorithm. In: ICMLA. pp. 1203–1208. IEEE (2021)
3. Baier, C., Funke, F., Piribauer, J., Ziemek, R.: On probability-raising causality in Markov decision processes. In: FoSSaCS. Lecture Notes in Computer Science, vol. 13242, pp. 40–60. Springer (2022)
4. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Software Eng. **29**(6), 524–541 (2003)
5. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
6. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: CAV (1). Lecture Notes in Computer Science, vol. 10426, pp. 160–180. Springer (2017)
7. Bellman, R.: A Markovian Decision Process. Journal of Mathematics and Mechanics **6**(5), 679–684 (1957)
8. Benedikt, M., Lenhardt, R., Worrell, J.: LTL model checking of interval Markov chains. In: TACAS. Lecture Notes in Computer Science, vol. 7795, pp. 32–46. Springer (2013)
9. Bressan, M., Peserico, E., Pretto, L.: On approximating the stationary distribution of time-reversible Markov chains. Theory Comput. Syst. **64**(3), 444–466 (2020)
10. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: TACAS (2). Lecture Notes in Computer Science, vol. 10206, pp. 151–168 (2017)
11. Busic, A., Fourneau, J.: Iterative component-wise bounds for the steady-state distribution of a Markov chain. Numer. Linear Algebra Appl. **18**(6), 1031–1049 (2011)
12. Cardelli, L., Kwiatkowska, M., Laurenti, L.: Programming discrete distributions with chemical reaction networks. Nat. Comput. **17**(1), 131–145 (2018)
13. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Markov decision processes with multiple objectives. In: STACS. Lecture Notes in Computer Science, vol. 3884, pp. 325–336. Springer (2006)
14. Ciesinski, F., Baier, C., Größer, M., Klein, J.: Reduction techniques for model checking Markov decision processes. In: QEST. pp. 45–54. IEEE Computer Society (2008)
15. Dai, P., Goldsmith, J.: Topological value iteration algorithm for Markov decision processes. In: IJCAI. pp. 1860–1865 (2007)
16. Dai, P., Mausam, Weld, D.S., Goldsmith, J.: Topological value iteration algorithms. J. Artif. Intell. Res. **42**, 181–209 (2011)
17. Delgrange, F., Katoen, J., Quatmann, T., Randour, M.: Simple strategies in multi-objective MDPs. In: TACAS (1). Lecture Notes in Computer Science, vol. 12078, pp. 346–364. Springer (2020)
18. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of Markov decision processes. Log. Methods Comput. Sci. **4**(4) (2008)

19. Fiondella, L., Rajasekaran, S., Gokhale, S.S.: Efficient software reliability analysis with correlated component failures. IEEE Trans. Reliab. **62**(1), 244–255 (2013)
20. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: TACAS. Lecture Notes in Computer Science, vol. 6605, pp. 112–127. Springer (2011)
21. Fourneau, J., Quessette, F.: Some improvements for the computation of the steady-state distribution of a Markov chain by monotone sequences of vectors. In: ASMTA. Lecture Notes in Computer Science, vol. 7314, pp. 178–192. Springer (2012)
22. Gadot, U., Derman, E., Kumar, N., Elfatihi, M.M., Levy, K., Mannor, S.: Solving non-rectangular reward-robust MDPs via frequency regularization. CoRR **abs/2309.01107** (2023)
23. Gokhale, S.S., Trivedi, K.S.: Reliability prediction and sensitivity analysis based on software architecture. In: ISSRE. pp. 64–78. IEEE Computer Society (2002)
24. Gokhale, S.S., Wong, W.E., Horgan, J.R., Trivedi, K.S.: An analytical approach to architecture-based software performance and reliability prediction. Perform. Evaluation **58**(4), 391–412 (2004)
25. Guennebaud, G., Jacob, B.: Eigen v3. http://eigen.tuxfamily.org (2010), [Accessed 10-Nov-2022]
26. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)
27. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)
28. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic. In: TACAS (2). Lecture Notes in Computer Science, vol. 13244, pp. 41–59. Springer (2022)
29. Hartmanns, A., Hermanns, H.: The modest toolset: An integrated environment for quantitative modelling and verification. In: TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 593–598. Springer (2014)
30. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: CAV (2). Lecture Notes in Computer Science, vol. 12225, pp. 488–511. Springer (2020)
31. 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)
32. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022)
33. Junges, S., Spaan, M.T.J.: Abstraction-refinement for hierarchical probabilistic models. In: CAV (1). Lecture Notes in Computer Science, vol. 13371, pp. 102–123. Springer (2022)
34. Katoen, J.: The probabilistic model checking landscape. In: LICS. pp. 31–45. ACM (2016)
35. Keane, M.S., O'Brien, G.L.: A bernoulli factory. ACM Trans. Model. Comput. Simul. **4**(2), 213–219 (1994)
36. Kemeny, J., Snell, J.: Finite Markov Chains. Undergraduate texts in mathematics, Springer (1976)
37. Kemeny, J.G., Snell, J.L.: Finite continuous time Markov chains. Theory of Probability & Its Applications **6**(1), 101–105 (1961)
38. Křetínský, J., Meggendorfer, T.: Of cores: A partial-exploration framework for Markov decision processes. Log. Methods Comput. Sci. **16**(4) (2020)
39. Kulkarni, V.: Modeling and Analysis of Stochastic Systems. Chapman & Hall/CRC Texts in Statistical Science, CRC Press (2020)

40. Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic model checking. In: SFM. Lecture Notes in Computer Science, vol. 4486, pp. 220–270. Springer (2007)
41. 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)
42. Lumbroso, J.O.: Optimal discrete uniform generation from coin flips, and applications. CoRR **abs/1304.1916** (2013)
43. McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: ICML. ACM International Conference Proceeding Series, vol. 119, pp. 569–576. ACM (2005)
44. Meedeniya, I., Moser, I., Aleti, A., Grunske, L.: Architecture-based reliability evaluation under uncertainty. In: QoSA/ISARCS. pp. 85–94. ACM (2011)
45. Meggendorfer, T.: Correct approximation of stationary distributions. In: TACAS (1). Lecture Notes in Computer Science, vol. 13993, pp. 489–507. Springer (2023)
46. Mertens, H., Katoen, J., Quatmann, T., Winkler, T.: Accurately Computing Expected Visiting Times and Stationary Distributions in Markov Chains (Artifact). Zenodo (Dec 2023). https://doi.org/10.5281/zenodo.10438916
47. Mertens, H., Katoen, J.P., Quatmann, T., Winkler, T.: Accurately computing expected visiting times and stationary distributions in Markov chains (2024). https://doi.org/10.48550/arXiv.2401.10638
48. Pietrantuono, R., Russo, S., Trivedi, K.S.: Online monitoring of software system reliability. In: EDCC. pp. 209–218. IEEE Computer Society (2010)
49. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994)
50. Quatmann, T., Katoen, J.: Sound value iteration. In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 643–661. Springer (2018)
51. Renard, Y.: Gmm++. https://getfem.org/gmm.html (2004), [Accessed 10-Nov-2022]
52. Saad, Y.: Iterative methods for sparse linear systems. SIAM (2003)
53. Salmani, B., Katoen, J.: Bayesian inference by symbolic model checking. In: QEST. Lecture Notes in Computer Science, vol. 12289, pp. 115–133. Springer (2020)
54. Salmani, B., Katoen, J.: Fine-tuning the odds in Bayesian networks. In: ECSQARU. Lecture Notes in Computer Science, vol. 12897, pp. 268–283. Springer (2021)
55. Sharma, V.S., Trivedi, K.S.: Reliability and performance of component based software systems with restarts, retries, reboots and repairs. In: ISSRE. pp. 299–310. IEEE Computer Society (2006)
56. Smolka, S., Kumar, P., Kahn, D.M., Foster, N., Hsu, J., Kozen, D., Silva, A.: Scalable verification of probabilistic networks. In: PLDI. pp. 190–203. ACM (2019)
57. Varga, R.S.: Matrix Iterative Analysis. Springer Series in Computational Mathematics, Springer Berlin Heidelberg (1999)
58. Volino, C.A.: A first course in stochastic models. Technometrics **47**(3), 375 (2005)
59. Wimmer, R., Kortus, A., Herbstritt, M., Becker, B.: Probabilistic model checking and reliability of results. In: DDECS. pp. 207–212. IEEE Computer Society (2008)
60. Winkler, T., Lehmann, J., Katoen, J.: Out of control: Reducing probabilistic models by control-state elimination. In: VMCAI. Lecture Notes in Computer Science, vol. 13182, pp. 450–472. Springer (2022)

CTMCs with Imprecisely Timed Observations *

Thom Badings[1](✉), Matthias Volk[2], Sebastian Junges[1],
Marielle Stoelinga[1,3], and Nils Jansen[1,4]

[1] Radboud University, Nijmegen, the Netherlands
thom.badings@ru.nl
[2] Eindhoven University of Technology, Eindhoven, the Netherlands
[3] University of Twente, Enschede, the Netherlands
[4] Ruhr-University Bochum, Bochum, Germany

Abstract. Labeled continuous-time Markov chains (CTMCs) describe processes subject to random timing and partial observability. In applications such as runtime monitoring, we must incorporate past observations. The timing of these observations matters but may be uncertain. Thus, we consider a setting in which we are given a sequence of imprecisely timed labels called the evidence. The problem is to compute reachability probabilities, which we condition on this evidence. Our key contribution is a method that solves this problem by unfolding the CTMC states over all possible timings for the evidence. We formalize this unfolding as a Markov decision process (MDP) in which each timing for the evidence is reflected by a scheduler. This MDP has infinitely many states and actions in general, making a direct analysis infeasible. Thus, we abstract the continuous MDP into a finite interval MDP (iMDP) and develop an iterative refinement scheme to upper-bound conditional probabilities in the CTMC. We show the feasibility of our method on several numerical benchmarks and discuss key challenges to further enhance the performance.

1 Introduction

Continuous-time Markov chains (CTMCs) are stochastic processes subject to random timing, which are ubiquitous in reliability engineering [48], network processes [33,35], and systems biology [14,20]. Here, we consider finite-state labeled CTMCs, which exhibit partial observability through a labeling function, such that analysis can only be done based on observations of the state. Specific techniques such as model checking algorithms compute quantitative aspects of CTMC behavior under the assumption of a static and known initial state [4,10].

Conditional probabilities In applications such as runtime monitoring [13,49], we need to analyze an already running system without a static initial state. Instead, we must incorporate past observations, which are given as a sequence of CTMC labels, each of which is observed at a specific time. We call this sequence of timed labels the *evidence*. We want to incorporate this evidence by conditioning the

* This work has been funded by the NWO grant PrimaVera (NWA.1160.18.238), the EU Consolidator Grant 864075 (CAESAR) the EU Horizon 2020 project MISSION (101008233), and the ERC Starting Grant 101077178 (DEUCE).

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 258–278, 2024.
https://doi.org/10.1007/978-3-031-57249-4_13

state of the CTMC on the evidence. For example, "what is the probability of a failure for a production machine (modeled as a CTMC) before time T, given that we have observed particular labels at earlier times $t_1, t_2, \ldots, t_n$?"

Imprecise observation times These conditional probabilities depend on the exact time at which each label was observed. However, in realistic scenarios, the times for the labels in the evidence may not be known precisely. For example, inspections are always done in the first week of a month, but the precise moment of inspection may be unknown. Intuitively, we can interpret such *imprecisely timed evidence* as a potentially infinite set of (precisely timed) *instances* of the evidence that vary only in the observation times. For example, an inspection done on "*January 2 exactly at noon*" is an instance of the imprecise observation time of "*the first week of January.*" This perspective motivates a robust version of the previous question: "Given the imprecisely timed evidence, what is the maximal probability of a failure before time T over all instances of the evidence?"

Problem statement In this paper, we are given a labeled CTMC together with imprecisely timed evidence. For each instance of the evidence, we can define the probability of reaching a set of target states, conditioned on that evidence. The problem is to compute the supremum over these conditional probabilities for all instances of the evidence. We generalize this problem by considering *weighted* conditional reachability probabilities (or simply the *weighted reachability*), where we assign to each state a nonnegative weight. Standard conditional reachability is then a special case with a weight of one for the target states and zero elsewhere.

Contributions Our main contribution is the first method to compute weighted conditional reachability probabilities in CTMCs with imprecisely timed evidence. Our approach consists of the following three steps.

1) Unfolding In Sect. 3, we introduce a method that *unfolds* the CTMC over all possible timings of the imprecisely timed evidence. We formalize this unfolding as a Markov decision process (MDP) [47], in which the timing imprecision is reflected by nondeterminism. We show that the weighted reachability can be computed via (unconditional) reachability probabilities on a transformed version of this MDP [12,39]. For the special case of evidence with precise observation times, we obtain a precise solution to the problem that we can directly compute.

2) Abstraction In general, imprecisely timed evidence yields an unfolded MDP with infinitely many states and actions. In Sect. 4, we propose an abstraction of this continuous MDP as a finite interval MDP (iMDP) [27], similar to game-based abstractions [41]. A robust analysis of the iMDP yields upper and lower bounds on the weighted reachability for the CTMC. Moreover, we propose an iterative refinement scheme that converges to the weighted reachability in the limit.

3) Computing bounds in practice In Sect. 5, we use the iMDP abstraction and refinement to obtain sound upper and lower bounds on the weighted reachability in practice. In Sect. 6, we show the feasibility of our method across several numerical benchmarks. Concretely, we show that we obtain reasonably tight bounds on the weighted reachability within a reasonable time. Finally, we discuss the key challenges in further enhancing the performance of our method in Sect. 8.

Related work Closest to our problem are works on model checking CTMCs against deterministic timed automata (DTA) [2,22,23]. Evidence can be expressed as a single-clock DTA, and tools such as MC4CSL [1] can calculate the weighted reachability for precise timings. However, for imprecisely timed evidence, checking CTMCs against DTAs yields the *sum of probabilities* over all instances of the evidence, whereas we are interested in the *maximal probability* over all instances.

Our setting is also similar to synthesizing timeouts in CTMCs with fixed-delay transitions [9,15,42]. Finding optimal timeouts is similar to our objective of finding an instance of the imprecisely timed evidence such that the weighted reachability is maximized. While timeouts can model the time *between* observations, we consider *global* observation times, i.e., the time between observations depends on the previous time of observation—which cannot be modeled with timeouts.

We discuss other related work in more detail in Sect. 7.

2 Problem Statement

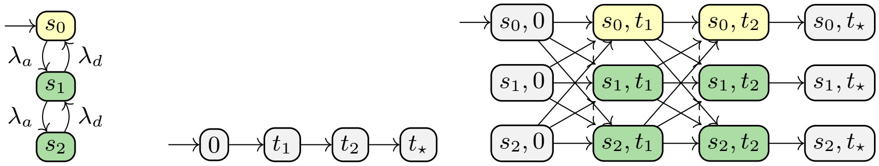
We recap continuous-time Markov chains (CTMCs) [4,10] and formalize the problem statement. The set of all probability distributions over a finite set X is denoted as $\mathit{Dist}(X)$. We write tuples $\langle a, b \rangle$ with square brackets, and $\mathbb{1}_x$ is the indicator function over x, i.e., $\mathbb{1}_{(y=z)}$ is one if $y = z$ and zero otherwise. We use the standard temporal operators $\Diamond$ and $\Box$ to denote *eventually* reaching or *always* being in a state [11].

Definition 1 (CTMC). *A (labeled)* continuous-time Markov chain $\mathcal{C}$ *is a tuple* $\langle S, s_I, \Delta, E, C, L \rangle$ *with a finite set* S *of* states*, an* initial state $s_I \in S$*, a* transition matrix $\Delta \colon S \to \mathit{Dist}(S)$*,* exit-rates $E \colon S \to \mathbb{Q}_{\geq 0}$*, a finite set of* colors C*, and a labeling function* $L \colon S \to C$*.*

A *(timed) CTMC path* $\pi = s_0 t_0 s_1 t_2 s_3 \cdots \in \Pi = S \times (\mathbb{R}_{\geq 0} \times S)^*$ is an alternating sequence of states and residence times, where $\Delta(s_i)(s_{i+1}) > 0 \, \forall i \in \mathbb{N}$. The path $s_0 3 s_1 4 s_2$ means we stayed exactly 3 time units in s_0, then transition to s_1, where we stayed 4 time units before moving to s_2. The CTMC state at time $t \in \mathbb{R}_{\geq 0}$ is denoted by $\pi(t) \in S$, e.g., $\pi(6.2) = s_1$ for the example path above.

An alternative (and equivalent) view of CTMCs is to combine the transition matrix Δ and exit-rates E in a transition rate matrix $R \colon S \times S \to \mathbb{Q}_{\geq 0}$, where $R(s, s') = \Delta(s, s') \cdot E(s) \; \forall s, s' \in S$ [40]. From state $s \in S$, the *transient probability distribution* $\Pr_s(t) \in \mathit{Dist}(S)$ after time $t \geq 0$ is $\Pr_s(t) = \delta_s \cdot e^{(R - \mathrm{diag}(E))t}$, where $\delta_s \in \{0, 1\}^{|S|}$ is the Dirac distribution for state s, and $\mathrm{diag}(E)$ is the diagonal matrix with the exit rates E on the diagonal. Thus, the probability of starting in state s and being in state $s' \in S$ after time t is $\Pr_s(t)(s') \in [0, 1]$.

Example 1. Consider a simple, single-product inventory where the number of items in stock ranges from 0 to 2, but we can only observe if the inventory is empty or not. This system is modeled by the CTMC shown in Fig. 1a with states $S = \{s_0, s_1, s_2\}$ (modeling the stock) and labels shown by the two colors (🟡 for empty and 🟢 for nonempty). The rates at which items arrive and deplete are $R(s_0, s_1) = R(s_1, s_2) = 3$ and $R(s_1, s_0) = R(s_2, s_1) = 2$, respectively.



(a) CTMC. (b) Evidence graph. (c) Unfolded MDP.

Fig. 1: The CTMC (a) for Example 1, (b) the graph for the precise evidence $\rho = \langle t_1, o_1 \rangle, \langle t_2, o_2 \rangle$, and (c) the states of the MDP unfolding defined by Def. 4.

2.1 Problem statement

The key problem we want to solve is to compute reachability probabilities for the CTMC conditioned on a timed sequence of labels, which we call the *evidence*.

Evidence The *evidence* $\rho = \langle t_1, o_1 \rangle, \dots, \langle t_d, o_d \rangle \in (\mathbb{R}_{>0} \times C)^d$ is a sequence of d times and labels such that $t_i < t_{i+1}$ for all $i \in \{1, \dots, d-1\}$. A timed label $\langle t_i, o_i \rangle$ means that at time t_i, the CTMC was in a state $s \in S$, that is, $L(s) = o_i$. Since each time $t \in \mathbb{R}_{>0}$ can only occur once in ρ, we overload ρ and denote the evidence at time $t \in \{t_1, \dots, t_d\}$ by $\rho(t) = o \in C$, such that $\langle t, o \rangle \in \rho$. While a timed path of a CTMC describes the state at every continuous point in time, the evidence only contains the observations at d points in time. We say that a path π is *consistent* with evidence ρ, written as $\pi \models \rho$, if each timed label in ρ matches the label of path π at time t, i.e., if $L(\pi(t)) = \rho(t) \, \forall t \in \{t_1, \dots, t_d\}$.

Conditional probabilities We want to compute the conditional probability $\mathbb{P}_{\mathcal{C}}(\pi(t_d) = s) \mid [\pi \models \rho])$ that the CTMC $\mathcal{C}$ with initial state s_I generates a path being in state s at time t_d, conditioned on the evidence ρ. Using Bayes' rule, we can characterize this conditional probability as follows (assuming $\frac{0}{0} = 0$, for brevity):

$$\mathbb{P}_{\mathcal{C}}(\pi(t_d) = s \mid [\pi \models \rho]) = \frac{\mathbb{P}_{\mathcal{C}}([\pi(t_d) = s] \cap [\pi \models \rho])}{\mathbb{P}_{\mathcal{C}}(\pi \models \rho)}. \tag{1}$$

Imprecise timings We extend evidence with uncertainty in the timing of each label. The *imprecisely timed evidence* (or *imprecise evidence*) $\varOmega = \langle \mathcal{T}_1, o_1 \rangle, \dots, \langle \mathcal{T}_d, o_d \rangle$ is a sequence of d labels and uncertain timings $\mathcal{T}_i = \cup_{j=1}^{q} [\underline{t}_j, \overline{t}_j]$, with $\underline{t}_j \leq \overline{t}_j$ and $q \in \mathbb{N}$. Observe that $\mathcal{T}$ can model both singletons ($\mathcal{T}_i = \{1, 2, 3\}$) and unions of intervals ($\mathcal{T}_i = [1, 1.5] \cup [2, 2.5]$). We require that $\max_{t \in \mathcal{T}_i}(t) < \min_{t' \in \mathcal{T}_{i+1}}(t')$ for all $i \in \{1, \dots, d-1\}$, i.e., the order of the labels is known, despite the uncertainty in the observation times. Again, we overload notation and denote the evidence at time t by $\varOmega(t) = o$, such that $\exists \langle \mathcal{T}, o \rangle \in \varOmega$ with $t \in \mathcal{T}$. Imprecise evidence induces a set of *instances* of the evidence that only differ in the label times. This set of instances is uncountably infinite whenever one of the imprecise timings $\mathcal{T}$ is a continuous set. Formally, the evidence $\rho = \langle t_1, o_1 \rangle, \dots, \langle t_d, o_d \rangle$ is an instance of the imprecise evidence $\varOmega$, written as $\rho \in \varOmega$, if $t_i \in \mathcal{T}_i$ for all $i = 1, \dots, d$.

Example 2. An example of imprecise evidence for the CTMC in Example 1 is $\varOmega = \langle [0.2, 0.8], 🟡 \rangle, \langle [1.4, 2.1], 🟢 \rangle$. The precise evidence $\rho = \langle 0.4, 🟡 \rangle, \langle 1.9, 🟢 \rangle$ is an

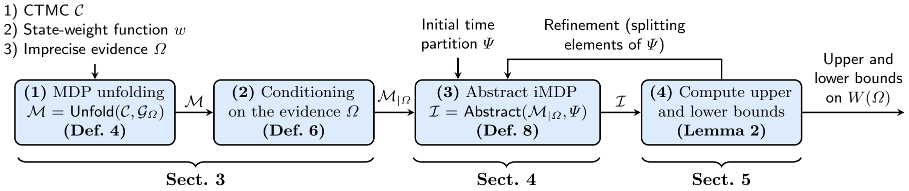

1) CTMC $\mathcal{C}$
2) State-weight function w
3) Imprecise evidence Ω
(1) MDP unfolding $\mathcal{M} = \mathsf{Unfold}(\mathcal{C}, \mathcal{G}_\Omega)$ (Def. 4)
$\mathcal{M}$
(2) Conditioning on the evidence Ω (Def. 6)
$\mathcal{M}_{|\Omega}$
Initial time partition Ψ
Refinement (splitting elements of Ψ)
(3) Abstract iMDP $\mathcal{I} = \mathsf{Abstract}(\mathcal{M}_{|\Omega}, \Psi)$ (Def. 8)
$\mathcal{I}$
(4) Compute upper and lower bounds (Lemma 2)
Upper and lower bounds on $W(\Omega)$
Sect. 3
Sect. 4
Sect. 5

Fig. 2: Conceptual workflow of our approach for solving Problem 1.

instance of Ω, i.e., $\rho \in \Omega$. However, $\rho' = \langle 0.1,$ 🟡$\rangle, \langle 1.9,$ 🟢$\rangle$ and $\rho'' = \langle 0.4,$ 🟢$\rangle, \langle 1.9,$ 🟢$\rangle$ are not, i.e., $\rho' \notin \Omega$, $\rho'' \notin \Omega$, as the timings and labels do not match, respectively.

State-weights Let $w \colon S \to \mathbb{R}_{\geq 0}$ be a *state-weight function*, which assigns to each CTMC state $s \in S$ a non-negative weight. The weight $w(s)$ represents a general measure of risk associated with each state $s \in S$, as used in [39]. For example, $w(s)$ may represent the probability of reaching a set of target states S_T from s within some time horizon $h \geq 0$. We then consider the following problem.

Problem 1 (Weighted conditional reachability probability). Given a CTMC $\mathcal{C}$, a state-weight function w, and the imprecisely timed evidence Ω, compute the (maximal) weighted conditional reachability probability $W(\Omega)$:

$$W(\Omega) = \sup_{\rho \in \Omega} \sum_{s \in S} \mathbb{P}_{\mathcal{C}}(\pi(t_d) = s \mid [\pi \models \rho]) \cdot w(s). \tag{2}$$

Example 3. For the CTMC in Example 1, consider the state-weight function that assigns to each state the probability of reaching state s_0 within time $t = 0.1$. Then, the problem above is interpreted as: *Given the imprecisely timed evidence Ω, compute the probability (conditioned on Ω) of reaching state s_0 within time $t = 0.1$ (after the end of the evidence).*

Our overall workflow to solve Problem 1 is summarized in Fig. 2 and consists of four blocks, which we discuss in Sects. 3 to 5, respectively.

Variations To instead minimize Eq. (2), we would swap every inf and sup (and max and min) in the paper, but our general approach remains the same. Furthermore, by setting $w(s) = 1$ for all $s \in S_T$ and zero otherwise, we can also compute the probability of being in a state in S_T *immediately* after the evidence. Finally, we remark that Problem 1 only considers events *after* the end of the evidence. This setting is motivated by applications where the exact system state is not observable, but actual system failures can be observed. Thus, one can typically assume that the system has not failed yet and the problem as formalized in Problem 1 is to predict the conditional probability of a future system failure.

2.2 Interval Markov decision processes

We recap interval MDPs (iMDPs) [27] and define standard MDPs as special case. We denote (i)MDP states by $q \in Q$, whereas CTMC states are denoted $s \in S$.

Definition 2 (iMDP). *An* interval MDP $\mathcal{I}$ *is a tuple* $\langle Q, q, A, \mathcal{P}\rangle$*, with* Q *a set of states,* $q \in Q$ *the initial state,* A *a set of actions, and where the uncertain transition function* $\mathcal{P}\colon Q \times A \times Q \rightharpoonup \mathbb{I} \cup \{[0,0]\}$ *is defined over intervals* $\mathbb{I} = \{[a,b] \mid a, b \in (0,1]$ *and* $a \leq b\}$*. The actions enabled in state* $q \in Q$ *are* $A(q) \subseteq A$*.*

The assumption that an interval cannot have a lower bound of 0 except the $[0,0]$ interval is standard, see, e.g., [46,52]. An MDP is a special case of iMDP, where the upper and lower bounds coincide, i.e., $\mathcal{P}(q,a,q') = [b,b]$, $b \in [0,1]$ for all intervals, and each $\mathcal{P}(q,a,\cdot) \in \mathit{Dist}(Q)$ is a distribution over states. We denote an MDP as $\mathcal{M} = \langle Q, q, A, P\rangle$, with transition function $P\colon Q \times A \times Q \rightharpoonup [0,1]$. For an MDP $\mathcal{M}$ with transition function P, we write $P \in \mathcal{P}$ if for all $q, q' \in Q$ and $a \in A$ we have $P(q,a,q') \in \mathcal{P}(q,a,q')$ and each $P(q,a,\cdot) \in \mathit{Dist}(Q)$. Fixing a transition function $P \in \mathcal{P}$ for iMDP $\mathcal{I}$ yields an induced MDP $\mathcal{I}[P]$.

The nondeterminism in an iMDP $\mathcal{I}$ is resolved by a memoryless scheduler $\sigma\colon Q \to A$, with $\sigma \in \mathrm{Sched}_{\mathcal{I}}$ the set of all schedulers. We denote a finite (i)MDP path by $\xi = q_0, \dots, q_n \in \varXi^{\sigma}_{\mathcal{I}}$, where $\varXi^{\sigma}_{\mathcal{I}}$ is the set of all paths under scheduler σ. For the Markov chain induced by scheduler σ in $\mathcal{I}[P]$, we use the standard probability measure $\mathbb{P}^{\sigma}_{\mathcal{I}[P]}$ over the smallest sigma-algebra containing the cylinder sets of all finite paths $\xi \in \varXi^{\sigma}_{\mathcal{I}}$; see, e.g., [11]. If $\mathrm{Sched}_{\mathcal{I}}$ is a singleton (i.e., $\mathcal{I}$ has only one scheduler), we omit the script σ and simply write $\mathbb{P}_{\mathcal{I}[P]}$ and $\varXi_{\mathcal{I}}$. For MDPs $\mathcal{M}$, we use the analogous notation with subscripts $\mathcal{M}$.

3 Conditional Reachability with Imprecise Evidence

In this section, we treat the first two blocks of Fig. 2. In Sect. 3.1, we *unfold* the CTMC over the times in the imprecise evidence into an MDP. The main result of this section, Theorem 1, states that the conditional reachability on the CTMC in Problem 1 is equal to the *maximal* conditional reachability probabilities in the MDP over a *subset of schedulers* (those that we call *consistent*; see Def. 5). In Sect. 3.2, we use results from [12] to determine these conditional probabilities via unconditional reachability probabilities on a transformed version of the MDP.

3.1 Unfolding the CTMC into an MDP

We interpret the (precisely timed) evidence $\rho = \langle t_1, o_1\rangle, \dots, \langle t_d, o_d\rangle$ as a directed graph that encodes the trivial progression over the time steps $t_1, \dots, t_d$.

Definition 3 (Evidence graph). *An* evidence graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E}\rangle$ *is a directed graph where each node* $t \in \mathcal{N} \subseteq \mathbb{R}_{>0}$ *is a point in time, and with directed edges* $\mathcal{E} \subset \{t \to t' : t, t' \in \mathcal{N}\}$*, such that* $t' > t$ *for all* $t \to t' \in \mathcal{E}$*.*

The graph $\mathcal{G}_\rho = \langle \mathcal{N}_\rho, \mathcal{E}_\rho\rangle$ for the precise evidence ρ has nodes $\mathcal{N}_\rho = \{0, t_1, \dots, t_d, t_\star\}$ and edges $\mathcal{E}_\rho = \{t_{i-1} \to t_i : i = 2, \dots, d\} \cup \{0 \to t_1, t_d \to t_\star\}$. As illustrated in Fig. 1b, the graph $\mathcal{G}_\rho$ has exactly one path, which follows the time points $t_1, \dots, t_d$ of the evidence ρ itself. Likewise, we model the imprecise evidence $\varOmega$ as a graph $\mathcal{G}_\varOmega$ which is the union of all graphs $\mathcal{G}_\rho$ for all instances $\rho \in \varOmega$, i.e.,

$$\mathcal{G}_\varOmega = \langle \mathcal{N}_\varOmega, \mathcal{E}_\varOmega\rangle = \cup_{\rho\in\varOmega}(\mathcal{G}_\rho) = \langle \cup_{\rho\in\varOmega}(\mathcal{N}_\rho), \cup_{\rho\in\varOmega}(\mathcal{E}_\rho)\rangle. \tag{3}$$

If Ω has infinitely many instances, then $\mathcal{G}_\Omega$ has infinite branching. Every path $t_0 t_1 \dots t_d t_\star$ through graph $\mathcal{G}_\Omega$ corresponds to the time points of the precise evidence $\rho = \langle t_1, o_1 \rangle, \dots, \langle t_d, o_d \rangle \in \Omega$ (and vice versa).

We denote the successor nodes of $t \in \mathcal{N}$ by $\mathsf{post}(t) = \{t' \in \mathcal{N} : t \to t' \in \mathcal{E}\}$. For example, the graph in Fig. 1b has $\mathsf{post}(0) = t_1$, $\mathsf{post}(t_1) = t_2$ and $\mathsf{post}(t_2) = t_\star$. We introduce the *unfolding operator* $\mathcal{M} = \mathsf{Unfold}(\mathcal{C}, \mathcal{G})$, which takes a CTMC $\mathcal{C}$ and a graph $\mathcal{G}$, and returns the *unfolded MDP* $\mathcal{M}$ defined as follows.

Definition 4 (Unfolded MDP). *For a CTMC $\mathcal{C} = \langle S, s_I, \Delta, E, C, L \rangle$ and a graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$, the* unfolded MDP $\mathsf{Unfold}(\mathcal{C}, \mathcal{G}) = \langle Q, q_I, A, P \rangle$ *has states states $Q = S \times \mathcal{N}$, initial state $q_I = \langle s_I, 0 \rangle$, actions $A = \mathcal{N}$, and transition function P, which is defined for all $\langle s, t \rangle \in Q$, $t' \in \mathsf{post}(t)$, $s' \in S$ as*

$$P\big(\langle s, t \rangle, t', \langle s', t' \rangle\big) = \begin{cases} Pr_s(t' - t)(s') & \text{if } t' \neq t_\star, \\ \mathbb{1}_{(s = s')} & \text{if } t' = t_\star, \end{cases} \tag{4}$$

The unfolding of the CTMC in Fig. 1a over the graph in Fig. 1b is shown in Fig. 1c. A state $\langle s, t \rangle \in Q$ in the unfolded MDP is interpreted as being in CTMC state $s \in S$ at time t. In state $\langle s, t \rangle$, the set of enabled actions is $A(\langle s, t \rangle) = \mathsf{post}(t) \subset \mathcal{N}$, and taking an action $t' \in \mathsf{post}(t)$ corresponds to *deterministically* jumping to time t'. The effect of this action is *stochastic* and determines the next CTMC state. The transition probability $P(\langle s, t \rangle, t', \langle s', t' \rangle)$ for $t' \neq t_\star$ models the probability of starting in CTMC state $s \in S$ and being in state $s' \in S$ after time $t' - t$ has elapsed, which is precisely the transient probability $\mathrm{Pr}_s(t' - t)(s')$ defined in Sect. 2. Finally, the (terminal) states $\langle s, t_\star \rangle$ for all $s \in S$ are absorbing.

Interpretation of schedulers Every instance $\rho \in \Omega$ of the imprecise evidence $\Omega = \langle \mathcal{T}_1, o_1 \rangle, \dots, \langle \mathcal{T}_d, o_d \rangle$ corresponds to fixing a precise time $t_i \in \mathcal{T}_i$ for all $i = 1, \dots, d$. For each such $\rho \in \Omega$, there exists a scheduler $\sigma \in \mathrm{Sched}_\mathcal{M}$ for MDP $\mathcal{M} = \mathsf{Unfold}(\mathcal{C}, \mathcal{G}_\Omega)$ that induces a Markov chain which only visits those time points $t_1, \dots, t_d$. We call such a scheduler σ *consistent* with the evidence ρ.

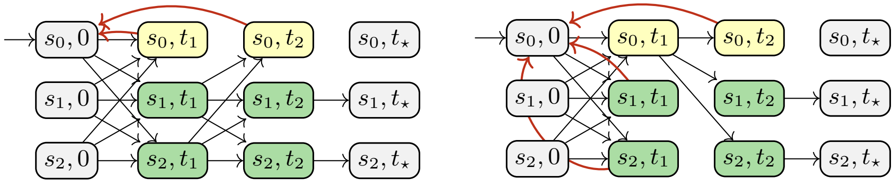
Definition 5 (Consistent scheduler). *A scheduler $\sigma \in \mathrm{Sched}_\mathcal{M}$ is consistent with $\rho = \langle t_1, o_1 \rangle, \dots, \langle t_d, o_d \rangle \in \Omega$, written as $\sigma \sim \rho$, if for all CTMC states $s \in S$:*

$$\sigma(\langle s, 0 \rangle) = t_1, \quad \sigma(\langle s, t_i \rangle) = t_{i+1} \, \forall i \in \{0, \dots, d-1\}, \quad \sigma(\langle s, t_d \rangle) = t_\star. \tag{5}$$

We denote the set of all consistent schedulers by $\mathrm{Sched}^{\mathsf{con}}_\mathcal{M} \subseteq \mathrm{Sched}_\mathcal{M}$.

A consistent scheduler chooses the same action $\sigma(\langle s, t \rangle) = \sigma(\langle s', t' \rangle)$ in any two MDP states $\langle s, t \rangle, \langle s', t' \rangle \in Q$ for which $t = t'$. There is a one-to-one correspondence between choices $\rho \in \Omega$ and consistent schedulers: for every $\rho \in \Omega$, there exists a scheduler $\sigma \in \mathrm{Sched}^{\mathsf{con}}_\mathcal{M}$ such that $\sigma \sim \rho$, and vice versa.

Example 4. Consider imprecise evidence $\Omega = \langle [0.2, 0.8],$ 🟡$\rangle, \langle [1.4, 2.1],$ 🟢$\rangle$ for the CTMC in Example 1. A scheduler with $\sigma(\langle s_0, 0.4 \rangle) = 1.5$, $\sigma(\langle s_1, 0.4 \rangle) = 1.8$ is inconsistent as it chooses different actions in MDP states with the same time.



(a) For $\rho = \langle t_1,$ 🟢 $\rangle, \langle t_2,$ 🟢 $\rangle$.

(b) For $\rho = \langle t_1,$ 🟡 $\rangle, \langle t_2,$ 🟢 $\rangle$.

Fig. 3: The unfolded MDP from Fig. 1c conditioned on different precise evidences. States that do not agree with the evidence are looped back to the initial state.

Remark 1. The unfolded MDP $\mathcal{M}' = \mathsf{Unfold}(\mathcal{C}, \mathcal{G}_\rho)$ for the precise evidence ρ has only a single action enabled in every state (i.e., $\mathcal{M}'$ directly reduces to a discrete-time Markov chain). Hence, $\mathcal{M}'$ has only one scheduler, and $\mathrm{Sched}^{\mathsf{con}}_{\mathcal{M}'} = \mathrm{Sched}_{\mathcal{M}'}$.

Conditional reachability on unfolded MDP As a main result, we show that $W(\Omega)$ in Problem 1 can be expressed as maximizing conditional reachability probabilities in the unfolded MDP $\mathcal{M}$ over the consistent schedulers $\mathrm{Sched}^{\mathsf{con}}_{\mathcal{M}} \subset \mathrm{Sched}_{\mathcal{M}}$.

Theorem 1. *For a CTMC $\mathcal{C}$ and the imprecise evidence Ω with graph $\mathcal{G}_\Omega$, let $\mathcal{M} = \mathsf{Unfold}(\mathcal{C}, \mathcal{G}_\Omega)$ be the unfolded MDP. Then, using the notation from Sect. 2.2 (for the probability measure $\mathbb{P}^\sigma_{\mathcal{M}}$ over paths $\xi \in \Xi^\sigma_{\mathcal{M}}$), Eq. (2) is rewritten as*

$$W(\Omega) = \sup_{\sigma \in \mathrm{Sched}^{\mathsf{con}}_{\mathcal{M}}} \sum_{s \in S} \mathbb{P}^\sigma_{\mathcal{M}}(\Diamond \langle s, t_\star \rangle \mid [\xi \models \rho,\ \sigma \sim \rho]) \cdot w(s). \tag{6}$$

Proof. The proof is in [8, Appendix A] and shows that for every instance $\rho \in \Omega$, the conditional transient probabilities in the CTMC are equivalent to conditional reachability probabilities in the unfolded MDP under a $\sigma \sim \rho$ consistent to ρ. □

3.2 Computing conditional probabilities in MDPs

We describe a transformation of the unfolded MDP to compute the conditional reachability probabilities in Eq. (6). Intuitively, we *refute* all paths through the MDP that do not agree with the labels in the evidence. Specifically, we find the subset of MDP states $Q_{\mathsf{reset}}(\Omega) \subset Q$ that disagree with the evidence, defined as

$$Q_{\mathsf{reset}}(\Omega) = \big\{ \langle s, t \rangle \in Q : L(s) \neq \Omega(t) \big\} \subset Q. \tag{7}$$

We *reset* all states in $Q_{\mathsf{reset}}(\Omega)$ by adding transitions back to the initial state with probability one. Formally, we define the *conditioned MDP* $\mathcal{M}_{|\Omega}$ as follows.

Definition 6 (Conditioned MDP). *For $\mathcal{M} = \mathsf{Unfold}(\mathcal{C}, \mathcal{G}_\Omega) = \langle Q, q_I, A, P \rangle$, the conditioned MDP $\mathcal{M}_{|\Omega} = \langle Q, q_I, A, P_{|\Omega} \rangle$ has the same states and actions, but the transition function is defined for all $\langle s, t \rangle \in Q$, $t' \in \mathsf{post}(t)$, $s' \in S$ as*

$$P_{|\Omega}\big(\langle s, t \rangle, t', \langle s', t' \rangle\big) = \begin{cases} P\big(\langle s, t \rangle, t', \langle s', t' \rangle\big) & \text{if } \langle s, t \rangle \notin Q_{\mathsf{reset}}(\Omega), \\ \mathbb{1}_{(s' = s_I)} & \text{if } \langle s, t \rangle \in Q_{\mathsf{reset}}(\Omega). \end{cases} \tag{8}$$

Two examples of conditioning on precise evidence are shown in Fig. 3. Compared to Fig. 1c, we removed all probability mass over paths that are not consistent with the evidence and normalized the probabilities for all other paths. The following result from [12] shows that conditional reachabilities in the unfolded MDP are equal to *unconditional* reachabilities in the conditioned MDP.

Lemma 1 (Thm. 1 in [12]). *For the imprecise evidence Ω, unfolded MDP $\mathcal{M} = \mathsf{Unfold}(\mathcal{C}, \mathcal{G}_\Omega)$, and conditioned MDP $\mathcal{M}_{|\Omega}$ defined by Def. 6, it holds that*

$$\mathbb{P}^{\sigma}_{\mathcal{M}}(\Diamond \langle s, t_\star \rangle \mid [\xi \models \rho,\ \sigma \sim \rho]) = \mathbb{P}^{\sigma}_{\mathcal{M}_{|\Omega}}(\Diamond \langle s, t_\star \rangle) \quad \forall \sigma \in \mathrm{Sched}_{\mathcal{M}}\ \forall s \in S. \tag{9}$$

Finally, combining Lemma 1 with Theorem 1 directly expresses the conditional reachability $W(\Omega)$ in terms of reachability probabilities on the conditioned MDP.

Theorem 2. *Given a CTMC $\mathcal{C}$, a state-weight function w, and the imprecisely timed evidence Ω, let $\mathcal{M} = \mathsf{Unfold}(\mathcal{C}, \mathcal{G}_\Omega)$. Then, it holds that*

$$W(\Omega) = \sup_{\sigma \in \mathrm{Sched}^{\mathsf{con}}_{\mathcal{M}}} \sum_{s \in S} \mathbb{P}^{\sigma}_{\mathcal{M}_{|\Omega}}(\Diamond \langle s, t_\star \rangle) \cdot w(s). \tag{10}$$

Solving Problem 1 with precisely timed evidence is now straightforward by solving a finite DTMC, see Remark 1. Furthermore, if the imprecise evidence has finitely many instances, then the MDP is finite. A naive approach to optimize over the consistent schedulers is enumeration, which we discuss in details Sect. 5.

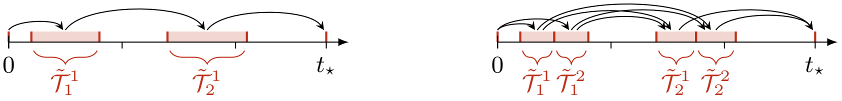
Remark 2 (Variations on Problem 1). With minor modifications to our approach, we can compute, e.g., the likelihood that a CTMC generates precise evidence ρ. Concretely, we define a transformed version $\mathcal{M}_\rho$ of the unfolded MDP in which all states in Q_{reset} are absorbing. We discuss this variation in [8, Appendix C]

4 Abstraction of Conditioned MDPs

For imprecisely timed evidence with *infinitely many instances* (e.g., imprecise timings over intervals), the conditioned MDP from Sect. 3 has infinitely many states and actions. In this section, we treat block (3) of Fig. 2 and propose an abstraction of this continuous MDP into a finite interval MDP (iMDP). Similar to game-based abstractions [29,30,41], we capture abstraction errors as nondeterminism in the transition function of the iMDP. Robust reachability probabilities in the iMDP yield sound bounds on the conditional reachability $W(\Omega)$. The crux of our abstraction is to create a finite *partition* of the (infinite) sets of uncertain timings in the evidence, as illustrated by Fig. 4.

Definition 7 (Time partition). *A* time partition Ψ *of the imprecise evidence $\Omega = \langle \mathcal{T}_1, o_1 \rangle, \ldots, \langle \mathcal{T}_d, o_d \rangle$ is a set $\Psi = \cup_{i=1}^{d} \mathsf{partition}(\mathcal{T}_i) \cup \{0, t_\star\}$, where each $\mathsf{partition}(\mathcal{T}_i) = \{\mathcal{T}_i^1, \ldots, \mathcal{T}_i^{n_i}\}$ is a finite partition*[5] *of $\mathcal{T}_i$ into $n_i \in \mathbb{N}$ elements.*

[5] A partition $\mathsf{partition}(X) = (X_1, \ldots, X_n)$ covers X (i.e., $X = \cup_{i=1}^{n} X_i$) and the interior of each element is disjoint (i.e., $\mathrm{int}(X_i) \cap \mathrm{int}(X_j) = \varnothing,\ i, j \in \{1, \ldots, n\}, i \neq j$).


0
$\tilde{\mathcal{T}}_1^1$
$\tilde{\mathcal{T}}_2^1$
$t_\star$
0
$\tilde{\mathcal{T}}_1^1$ $\tilde{\mathcal{T}}_1^2$
$\tilde{\mathcal{T}}_2^1$ $\tilde{\mathcal{T}}_2^2$
$t_\star$

(a) Coarsest time partition.

(b) Refined time partition.

Fig. 4: Two partitions of imprecise evidence $\Omega = \langle [0.2, 0.8], o_1\rangle, \langle [1.4, 2.1], o_2\rangle$. The partition in (a) consists of two elements, such that $\tilde{\mathcal{T}}_1^1 = [0.2, 0.8]$ and $\tilde{\mathcal{T}}_2^1 = [1.4, 2.1]$, where (b) refines this to $\tilde{\mathcal{T}}_1^1 \cup \tilde{\mathcal{T}}_1^2 = [0.2, 0.8]$ and $\tilde{\mathcal{T}}_2^1 \cup \tilde{\mathcal{T}}_2^2 = [1.4, 2.1]$.

With abuse of notation, the element of Ψ containing time t is $\Psi(t) \in \Psi$, and $\Psi^{-1}(\psi) = \{t : \Psi(t) = \psi\}$ is the set of times mapping to $\psi \in \Psi$. As shown by Fig. 4, for each $i \in \{1, \dots, d\}$, the sets $\tilde{\mathcal{T}}_i^1, \dots, \tilde{\mathcal{T}}_i^{n_i}$ are a partition of the set $\mathcal{T}_i$.

To illustrate the abstraction, let $\langle s, t\rangle \xrightarrow{t':P'} \langle s', t'\rangle$ denote the MDP transition from state $\langle s, t\rangle \in Q$, under action $t' \in A(\langle s, t\rangle)$ to state $\langle s', t'\rangle \in Q$, which has probability P'. With this notation, we can express any MDP path as

$$\langle s_I, 0\rangle \xrightarrow{t:P} \langle s, t\rangle \xrightarrow{t':P'} \langle s', t'\rangle \xrightarrow{t'':P''} \cdots \xrightarrow{t''':P'''} \langle s, t_\star\rangle. \tag{11}$$

For every element $\psi \in \Psi$ of partition Ψ, the abstraction merges all MDP states $\langle s, t\rangle \in Q$ for which the time t belongs to the element ψ, that is, for which $t \in \Psi^{-1}(\psi)$. Thus, we merge infinitely many MDP states into finitely many abstract states. The MDP path in Eq. (11) matches the next path in the abstraction:

$$\langle s_I, 0\rangle \xrightarrow{\mathcal{T}:\mathcal{P}} \langle s, \mathcal{T}\rangle \xrightarrow{\mathcal{T}':\mathcal{P}'} \langle s', \mathcal{T}'\rangle \xrightarrow{\mathcal{T}'':\mathcal{P}''} \cdots \xrightarrow{\mathcal{T}''':\mathcal{P}'''} \langle s, t_\star\rangle, \tag{12}$$

where each $t \in \mathcal{T}$, and each $\mathcal{P}$ is a *set of probabilities*. The abstraction contains the behavior of the continuous MDP if $P \in \mathcal{P}$ at every step in Eqs. (11) and (12), see, e.g., [38]. The following iMDP abstraction satisfies these requirements.

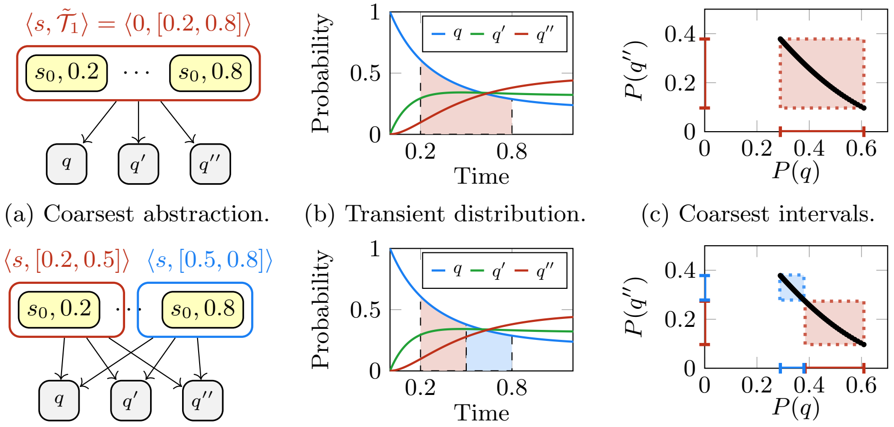
Definition 8 (iMDP abstraction). *For a conditioned MDP $\mathcal{M}_{|\Omega} = \langle Q, q_I, A, P\rangle$ and a time partition Ψ of Ω, the iMDP abstraction $\mathcal{I} = \mathsf{Abstract}(\mathcal{M}_{|\Omega}, \Psi) = \langle \tilde{Q}, q_I, \tilde{A}, \mathcal{P}\rangle$, with states $\tilde{Q} = \big\{\langle s, \Psi(t)\rangle : \langle s, t\rangle \in Q\big\}$, actions $\tilde{A} = \big\{\Psi(t) : t \in A\big\}$, and uncertain transition function $\mathcal{P}$ defined for all $\langle s, \mathcal{T}\rangle, \langle s', \mathcal{T}'\rangle \in \tilde{Q}$ as*

$$\mathcal{P}\big(\langle s, \mathcal{T}\rangle, \mathcal{T}', \langle s', \mathcal{T}'\rangle\big) \;=\; \mathrm{cl}\Big(\bigcup_{t \in \Psi^{-1}(\mathcal{T}), t' \in \Psi^{-1}(\mathcal{T}')} P\big(\langle s, t\rangle, t', \langle s', t'\rangle\big) \Big), \tag{13}$$

where $\mathrm{cl}(x) = [\min(x), \max(x)]$ is the interval closure of x.

An abstraction under the coarse time partition from Fig. 4 is shown in Fig. 5a. The transition probabilities for each MDP state are defined by transient probabilities for the CTMC. Thus, the uncertain transition function $\mathcal{P}$ of the iMDP overapproximates these transient probabilities over a *range of times* (as shown in Fig. 5b), yielding probability intervals as in Fig. 5c.

Conditional reachability on iMDP We show that the iMDP abstraction can be used to obtain sound upper and lower bounds on the conditional reachability



(a) Coarsest abstraction. (b) Transient distribution. (c) Coarsest intervals.

(d) Refined abstraction. (e) Transient distribution. (f) Refined intervals.

Fig. 5: Abstraction of an infinite set of MDP states for all times $t \in [0.2, 0.8]$ into (a) a single iMDP state $\langle s, [0.2, 0.8] \rangle$ with probability intervals that overapproximate the transient distribution (b) as the rectangular set in (c), where the line shows the MDP transition probabilities for all $t \in [0.2, 0.8]$. The refinement (d) into two iMDP states $\langle s, [0.2, 0.5] \rangle$ and $\langle s, [0.5, 0.8] \rangle$ splits the approximation of the transient (e) into the two (less conservative) rectangular sets in (f).

$W(\Omega)$. Let $W_{\mathcal{I}}(\tilde{P}, \sigma) \geq 0$ denote the value for the MDP $\mathcal{I}[\tilde{P}]$ induced by iMDP $\mathcal{I}$ under transition function $\tilde{P}$, and with scheduler $\sigma \in \mathrm{Sched}_{\mathcal{I}}$:

$$W_{\mathcal{I}}(\tilde{P}, \sigma) \coloneqq \sum_{s \in S} \mathbb{P}^{\sigma}_{\mathcal{I}[\tilde{P}]}(\Diamond \langle s, t_\star \rangle) \cdot w(s). \tag{14}$$

The next theorem, proven in [8, Appendix B], is the main result of this section.

Theorem 3. *Let $\mathcal{I} = \mathsf{Abstract}(\mathcal{M}_{|\Omega}, \Psi)$ be the iMDP abstraction for a conditioned MDP $\mathcal{M}_{|\Omega}$ and a time partition Ψ of Ω. Then, it holds that*

$$\max_{\sigma \in \mathrm{Sched}^{\mathrm{con}}_{\mathcal{I}}} \min_{\tilde{P} \in \mathcal{P}} W_{\mathcal{I}}(\tilde{P}, \sigma) \leq W(\Omega) \leq \max_{\sigma \in \mathrm{Sched}^{\mathrm{con}}_{\mathcal{I}}} \max_{\tilde{P} \in \mathcal{P}} W_{\mathcal{I}}(\tilde{P}, \sigma). \tag{15}$$

Construction of the iMDP We want to construct the abstract iMDP directly from the CTMC without first constructing the continuous MDP $\mathcal{M}_{|\Omega}$. Consider computing the probability interval $\mathcal{P}(\langle s, \mathcal{T} \rangle, \mathcal{T}', \langle s', \mathcal{T}' \rangle)$ for the iMDP transition from state $\langle s, \mathcal{T} \rangle$ to $\langle s', \mathcal{T}' \rangle$. This interval is given by the minimum and maximum transient probabilities $\mathrm{Pr}_s(t' - t)(s')$ over all $t \in \mathcal{T}$ and $t' \in \mathcal{T}'$. However, the problem is that the transient probabilities are not monotonic over time in general (see Fig. 5b), so it is unclear how to compute this interval.

Instead, we compute upper and lower bounds for the transient probabilities. Let $\underline{t} = \min(\mathcal{T})$ and $\bar{t} = \max(\mathcal{T})$. An upper bound on the transient probability is given by the probability to reach s' from s at *some* time $t' - t$, $t \in \mathcal{T}$, $t' \in \mathcal{T}'$:

$$\sup_{t \in \mathcal{T}, t' \in \mathcal{T}'} \mathrm{Pr}_s(t' - t)(s') \leq \sup_{t \in \mathcal{T}, t' \in \mathcal{T}'} \mathbb{P}_{\mathcal{C},s}(\Diamond^{[t,t']} s') = \mathbb{P}_{\mathcal{C},s}(\Diamond^{[\underline{t},\bar{t}']} s'), \qquad (16)$$

where $\mathbb{P}_{\mathcal{C},s}$ is the probability measure for the CTMC starting in initial state s, and $\bar{t}' - \underline{t}$ is the maximal time difference. A lower bound is given symmetrically by the transient probability to reach s' in the CTMC at the *earliest* possible time $\underline{t}' - \bar{t}$ and staying there for the *full* remaining time $(\bar{t}' - \underline{t}) - (\underline{t}' - \bar{t})$:

$$\inf_{t \in \mathcal{T}, t' \in \mathcal{T}'} \mathrm{Pr}_s(t' - t)(s') \geq \mathrm{Pr}_s(\underline{t}' - \bar{t})(s') \cdot \mathbb{P}_{\mathcal{C},s'}(\Box^{[0,(\bar{t}' - \underline{t}) - (\underline{t}' - \bar{t})]} s'). \qquad (17)$$

Abstraction refinement

To improve the tightness of the bounds in Theorem 3, we propose a refinement step that splits elements of the time partition $\varPsi$. For example, we may split the single abstract state in Fig. 5a into the two states in Fig. 5d.

Definition 9 (Refinement of time partition). *Let $\varPsi$ and $\varPsi'$ be partitions as per Def. 7, for which $|\varPsi'| > |\varPsi|$. We call $\varPsi'$ a refinement of $\varPsi$ if for all $\psi' \in \varPsi'$, there exists a $\psi \in \varPsi$ such that $\psi' \subseteq \psi$.*

Any refinement $\varPsi$' of partition $\varPsi$ can be constructed by finitely many splits. We lift the refinement to the iMDP, see also Figs. 5c and 5f. The refined iMDP $\mathcal{I}' = \mathsf{Abstract}(\mathcal{M}_{|\varOmega}, \varPsi')$ has more states and actions, but each union in Eq. (13) is over a smaller set than in iMDP $\mathsf{Abstract}(\mathcal{M}_{|\varOmega}, \varPsi)$. Thus, the refinement leads to smaller probability intervals and, in general, to tighter bounds in Theorem 3. Repeatedly refining every element of the partition yields an iMDP with arbitrarily many states and actions and with arbitrarily small probability intervals. Hence, in the limit, we may recover the original continuous MDP by refinements, which also implies that the bounds in Theorem 3 on the refined iMDP converge.

Refinement strategy By splitting every element of the partition $\varPsi$, the number of iMDP states and actions double per iteration, and the number of transitions grows exponentially. Thus, we employ the following *guided refinement strategy*. At each iteration, we extract the scheduler $\sigma^\star$ that attains the upper bound in Theorem 3 and determine the set $\tilde{Q}^{\sigma^\star}_{\mathsf{reach}} \subset \tilde{Q}$ of reachable iMDP states. We only refine the reachable elements $\psi \in \varPsi$, that is, for which there exists a $t \in \psi$ and $s \in S$ such that $\langle s, t \rangle \in \tilde{Q}^{\sigma^\star}_{\mathsf{reach}}$. Using this guided strategy, we iteratively shrink only the relevant probability intervals, resulting in the same convergence behavior as the naive strategy but without the severe increase in abstraction size.

5 Computing Bounds on the Conditional Reachability

Theorem 3 provides bounds on the conditional reachability $W(\varOmega)$ in Problem 1, but computing these bounds involves optimizing over the subset of consistent schedulers. Recall from Def. 5 that a consistent scheduler chooses the same actions in different states.[6] As we are not aware of any efficient algorithm to optimize over the consistent schedulers, we compute the following straightforward bounds:

Lemma 2 (Bounds on Problem 1). *Let* $\mathcal{I} = \mathsf{Abstract}(\mathcal{M}_{|\Omega}, \Psi)$ *be the iMDP abstraction for the unfolded MDP* $\mathcal{M}_{|\Omega}$ *and a time partition* Ψ*. It holds that*

$$W(\Omega) \leq \max_{\sigma \in \mathrm{Sched}_{\mathcal{I}}^{\mathsf{con}}} \max_{\tilde{P} \in \mathcal{P}} W_{\mathcal{I}}(\tilde{P}, \sigma) \leq \max_{\sigma \in \mathrm{Sched}_{\mathcal{I}}} \max_{\tilde{P} \in \mathcal{P}} W_{\mathcal{I}}(\tilde{P}, \sigma). \tag{18}$$

Moreover, any consistent scheduler $\hat{\sigma} \in \mathrm{Sched}_{\mathcal{I}}^{\mathsf{cons}}$ *results in a lower bound.*

Obtaining lower bounds While we can use *any* consistent scheduler in Lemma 2 to compute a lower bound on $W(\Omega)$, we obtain better bounds by modifying a (potentially non-consistent) optimal scheduler σ^- under the worst-case choice of probabilities, i.e., $\sigma^- = \arg\max_{\sigma \in \mathrm{Sched}_{\mathcal{I}}} \min_{\tilde{P} \in \mathcal{P}} W_{\mathcal{I}}(\tilde{P}, \sigma)$. We check for inconsistency of scheduler σ^- by evaluating the following condition in all pairs of states $\langle s, t \rangle, \langle s', t' \rangle \in \tilde{Q}^{\sigma^-}_{\mathsf{reach}} \subset \tilde{Q}$ reachable under σ^-:

$$t = t' \implies \sigma(\langle s, t \rangle) = \sigma(\langle s', t \rangle) \quad \forall \langle s, t \rangle, \langle s', t' \rangle \in \tilde{Q}^{\sigma^-}_{\mathsf{reach}}. \tag{19}$$

We remove inconsistencies by changing the action in one of the states to match the others. We take a greedy approach and always adapt to the action chosen most often across all iMDP states $\langle s, t \rangle \in \tilde{Q}$ for the same time t. For example, if $\sigma(\langle s, t \rangle) = \sigma(\langle s', t \rangle) \neq \sigma(\langle s'', t \rangle)$, then we only modify $\sigma(\langle s'', t \rangle)$ to match the other actions. Because the set $\tilde{Q}^{\sigma^-}_{\mathsf{reach}}$ is finite by construction, a finite number of modifications suffices to render any scheduler consistent. The experiments in Sect. 6 show that modifying an inconsistent scheduler yields tighter lower bounds than taking the maximum over many sampled consistent schedulers.

Obtaining upper bounds The set of consistent schedulers is finite but prohibitively large, so enumerating over all consistent schedulers is infeasible. For a sound upper bound, we instead optimize over all schedulers. The experiments in Sect. 6 show that we obtain (relatively) tight bounds. To further refine these upper bounds, the literature suggests another abstraction refinement loop, which can be formulated either directly on the imprecise evidence [21] or on the consistent schedulers [51]. The latter approach leverages the fact that consistent schedulers can also be modeled as searching for (memoryless) schedulers in partially observable MDPs, where the schedulers would only observe the time but not the state. Finally, the hardness of optimizing over consistent schedulers in the iMDP remains open: Classical NP-hardness results for the problems above do not carry over.

6 Numerical Experiments

We implemented our approach in a prototypical Python tool, which is available at `https://doi.org/10.5281/zenodo.10438984`. The tool builds on top

[6] Consistent schedulers are similar to (memoryless) schedulers in partially observable MDPs that choose the same action in states with the same observation label.

Table 1: Overview of considered benchmarks.

Example		CTMC size		State-weight function
Name	Evid. len. ($\|\Omega\|$)	States	Transit.	Property
Invent	3-14	3	4	“Prob. empty inventory within time 0.1”
Ahrs	4	74	196	“Prob. system failure within time 50”
Phil	4	34	89	“Prob. deadlock within time 1”
Tandem	2	120	363	“Prob. both queues full within time 10”
Polling	3	576	2208	“Prob. all stations empty within time 10”

of Storm [34] for the analysis of CTMCs and iMDPs. It takes as input a CTMC $\mathcal{C}$, a property defining the state-weight function w, and imprecisely timed evidence Ω. The tool constructs the abstract iMDP for the coarsest time partition, computing the probability intervals as per Eqs. (16) and (17). The bounds on the conditional reachability in Lemma 2 are computed using robust value iteration. Then, the tool applies guided refinements, as in Sect. 4, and starts a new iteration with the refined partition. After a predefined time limit, the tool returns the lower bound $\underline{W(\Omega)}$ and upper bound $\overline{W(\Omega)}$ on the conditional reachability $W(\Omega)$:

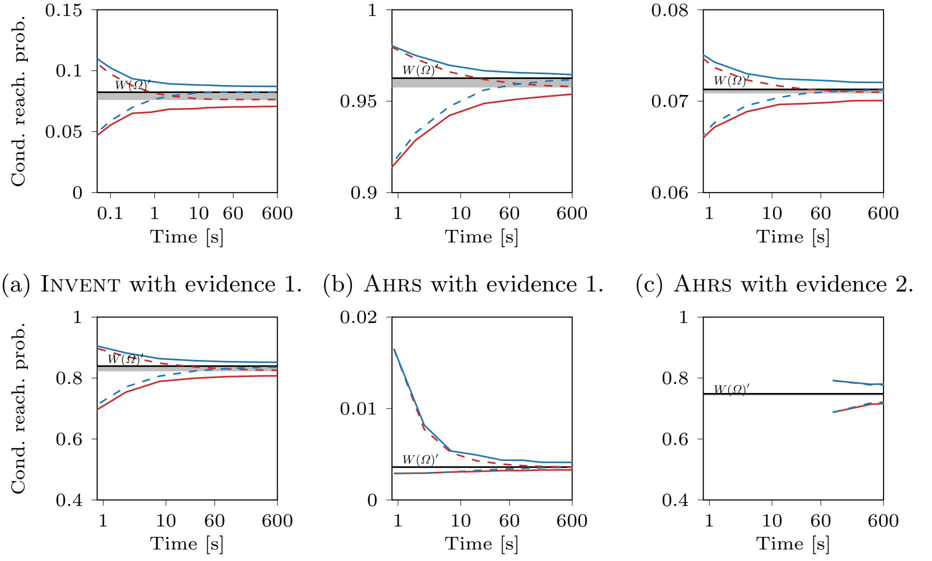
$$\underline{W(\Omega)} = \min_{\tilde{P} \in \mathcal{P}} W_{\mathcal{I}}(\tilde{P}, \hat{\sigma}) \leq W(\Omega) \leq \max_{\sigma \in \text{Sched}_{\mathcal{I}}} \max_{\tilde{P} \in \mathcal{P}} W_{\mathcal{I}}(\tilde{P}, \sigma) = \overline{W(\Omega)}, \qquad (20)$$

where the consistent scheduler $\hat{\sigma}$ for the lower bound is obtained by fixing all inconsistencies in the scheduler σ^- defined in Sect. 5. The tool can also compute minimal conditional reachabilities (by swapping all min and max operators).

Benchmarks We evaluate our approach on several CTMCs from the literature, creating multiple imprecisely timed evidence for each CTMC. Table 1 lists the evidence length (i.e., the number of observed times and labels), the number of CTMC states and transitions, and the property specifying the state-weight function. More details on the benchmarks are in [8, Appendix D.1], All experiments run on an Intel Core i5 with 8GB RAM, using a time limit of 10 minutes.

Feasibility of our approach We investigate if our approach yields tight bounds on the weighted reachability. Fig. 6 shows the results for each example with different imprecise evidences. The gray area shows the weighted reachabilities (as per Theorem 2) for 500 precisely timed instances $\rho \in \Omega$ sampled from the imprecise evidence. Recall that the weighted reachability $W(\Omega)$ is an upper bound to the weighted reachability for each precisely timed evidence $\rho \in \Omega$. Thus, the upper bound of the gray areas in Fig. 6, indicated as $W(\Omega)'$, is a lower bound of the actual (but unknown) value $W(\Omega)$. The blue lines are the upper bound $\overline{W(\Omega)}$ (solid) and lower bound $\underline{W(\Omega)}$ (dashed) on $W(\Omega)$ returned by our approach over the runtime (note the log-scale). Similarly, the red lines are the bounds obtained for *minimizing* the minimal weighted reachability.

Tightness of bounds Fig. 6 shows that we obtain reasonably tight bounds within a minute. In all examples, the lower bound converges close to the maximum



(a) Invent with evidence 1. (b) Ahrs with evidence 1. (c) Ahrs with evidence 2.

(d) Phil with evidence 1. (e) Tandem with evidence 1. (f) Polling with evidence 1.

Fig. 6: Results for different CTMCs and different imprecisely timed evidence. The blue lines are the upper bound $\overline{W(\Omega)}$ (solid) and lower bound $\underline{W(\Omega)}$ (dashed) on $W(\Omega)$; red lines show the analogous lower bounds.

of the samples. The improvement is steepest at the start, indicating that the bounds can be quickly improved by only a few refinement steps. In the long run, the improvement of the bounds diminishes, both because each refinement takes longer, and the improvement in each iteration gets smaller.

While not clearly visible in Fig. 6a, the lower bound $\underline{W(\Omega)}$ (dashed blue line) slightly exceeds the maximal sampled value $W(\Omega)'$ (gray area) in the end. Thus, the lower bound $\underline{W(\Omega)}$ is closer to the actual weighted reachability $W(\Omega)$ than the maximal lower bound obtained by sampling. We observed the same results when increasing the number of samples used to compute $W(\Omega)'$ to 10 000.

Figs. 6b and 6c show the general benefit of conditioning on evidence. While evidence 1 for AHRS results in a state in which a system failure within the next 50 time units is very likely, a failure conditioned on evidence 2 is very unlikely.

Scalability We investigate the scalability of our approach. Table 2 provides the refinement statistics, bounds, model sizes, and runtimes for all benchmarks. The refinement statistics show the number of iterations (Iter.) and the total number of splits made in the partition. The bounds on $W(\Omega)$ (which are the solid and dashed blue lines in Fig. 6) and the iMDP sizes are both given for the final iteration. For the timings, we provide the total time (over all iterations) and distinguish between the time spent on unfolding the model, i.e., constructing the iMDP, and analyzing it. Our approach terminates if after an iteration, the

Table 2: Results for all benchmarks (evidence length $|\Omega|$ is given after the name).

Example	Refine		Results	iMDP size			Timings [s]		
Name ($|\Omega|$)	Iter.	#split	Bounds on $W(\Omega)$	States	Actions	Transit.	Unfold	Analysis	Total
Invent-1 (4)	25	555	[0.082536, 0.087138]	898	128307	278163	537.51	100.28	637.81
Invent-2 (4)	27	585	[0.071768, 0.078328]	1180	167917	503537	606.91	43.85	650.74
Invent-3 (9)	14	1176	[0.071757, 0.078577]	2372	369329	1107877	658.77	127.83	786.57
Invent-4 (15)	7	528	[0.070924, 0.080409]	1016	39927	115119	42.63	974.89	1017.50
Ahrs-1 (4)	6	177	[0.962041, 0.964306]	6283	282538	1415346	620.75	179.65	800.39
Ahrs-2 (4)	8	154	[0.071239, 0.072057]	727	20626	81362	577.64	69.19	646.85
Ahrs-3 (4)	6	176	[0.964936, 0.969535]	6112	280954	1334231	749.38	152.61	902.00
Ahrs-4 (4)	7	300	[0.209591, 0.213820]	7179	535763	3618439	1801.81	111.39	1913.18
Phil-1 (5)	7	339	[0.836695, 0.851548]	4122	370091	3887339	851.92	60.32	912.23
Phil-2 (5)	6	209	[0.236734, 0.246067]	4050	203549	3669721	419.97	376.73	796.70
Tandem-1 (2)	9	77	[0.003577, 0.004009]	1203	24561	362657	917.29	3.11	920.42
Tandem-2 (2)	7	80	[0.130187, 0.162762]	587	25096	75548	549.03	327.93	876.96
Polling-1 (3)	2	9	[0.731410, 0.781912]	3267	9798	2379462	348.83	2603.08	2951.89

total run time so far exceeds the time limit of 10 minutes. The total runtime can, therefore, be significantly longer than 10 minutes.

CTMC size The size of the CTMC has a large impact on the total runtime. For example, for evidence with 4 labels, we can perform up to 27 iterations for Invent (3 CTMC states) but only 6-8 for Ahrs (74 CTMC states). For Polling (576 states) with evidence of length 2, performing 2 iterations takes nearly 50 minutes. The CTMC size affects the unfolding, which requires computing the transient probabilities from all states in one layer to all states in the next one. A clear example is Tandem-1 (120 CTMC states), where nearly all of the runtime is spent on the unfolding. A larger CTMC also leads to more transitions in the iMDP and thus, can increase the analysis time. An example is Polling-1 (576 CTMC states), where most of the runtime is spent in the analysis.

Length of evidence The time per refinement step increases with the length of the evidence. For example, for Invent-4 (with 15 labels), only 7 iterations are performed because the resulting iMDP has 15 layers, so the value iteration becomes the bottleneck (nearly 96% of the runtime for this example is spent on analyzing the iMDP). This is consistent with experiments on unfolded MDPs in [32,39], where policy iteration-based methods lead to better results.

Caching improves performance To reduce runtimes, we implemented caching in our tool, which allows reusing transient probability computations. For example, if all labels in the evidence have a time interval of the same width (which is the case for Ahrs-1), transient probabilities are the same between layers of the unfolding. Table 1 shows that the unfolding times for Ahrs-1 are indeed lower than for, e.g., Ahrs-3, which has time intervals of different widths.

Likelihood of evidence The size of the iMDP is influenced by the number of CTMC states corresponding to the observed labels. Less likely observations can, therefore, mean that fewer CTMC states need to be considered in each layer. For example, the evidence in Ahrs-2 is 17 times less likely (probability of 0.01, with

569 states) than Ahrs-4 (probability of 0.17, with 4007 states), and as a result the total runtime of Ahrs-2 is less than for Ahrs-4.

7 Related work

Beyond the related work discussed in Sect. 1 on DTAs [2,22,23] and synthesis of timeouts [9,15,42], the following work is related to ours.

Imprecisely timed evidence can also be expressed via multiphase timed until formulas in continuous-time linear logic [28]. However, similar to DTA, conditioning and computing the maximal weighted reachability are not supported.

Conditional probabilities naturally appear in runtime monitoring [13,49] and speech recognition [24], and is, e.g., studied for hidden Markov models [50] and MDPs [12,39]. Approximate model checking of conditional continuous stochastic logic for CTMCs is studied in [25,26] by means of a product construction formalized as CTMC, but their algorithm is incompatible with imprecise observation times. Conditional sampling in CTMCs is studied by [36], and maximum likelihood inference of paths in CTMCs by [45].

The abstraction of continuous stochastic models into iMDPs is well-studied [43]. Various papers develop abstractions of stochastic hybrid and dynamical systems into iMDPs [6,7,19] and relate to early work in [38]. Our abstraction in Sect. 4 is similar to a game-based abstraction, in which the (possibly infinite-state) model is abstracted into a two-player stochastic game [29,30,41]. In particular, iMDPs are a special case of a stochastic game in which the actions of the second player in each state only differ in transition probabilities [37,44]. An interesting extension of our approach is to consider CTMCs with uncertain *transition rates*, which have recently also been studied extensively, e.g., in [5,16–18,20,31].

8 Conclusion

We have presented the first method for computing reachability probabilities in CTMCs that are conditioned on evidence with imprecise observation times. The method combines an unfolding of the problem into an infinite MDP with an iterative abstraction into a finite iMDP. Our experiments have shown the applicability of our method across several benchmarks.

A natural next step is to embed our method in a predictive runtime monitoring framework, which introduces the challenge of running our algorithm in realtime. Another interesting extension is to consider uncertainty in the observed labels. Furthermore, this paper gives rise to four concrete challenges. First, finding better methods to overapproximate the union over MDP probabilities in Eq. (13) may lead to tighter bounds on the weighted reachability. Second, we want to optimize over the consistent schedulers only, potentially via techniques used in [3]. Third, we wish to explore better refinement strategies for the iMDP. The final challenge is to improve the computational performance of our implementation. One promising option to improve performance is to adapt symbolic policy iteration [9], which only considers small sets of candidate actions instead of all actions.

References

1. Amparore, E.G., Donatelli, S.: MC4CSLTA: an efficient model checking tool for CSLTA. In: QEST. pp. 153–154. IEEE Computer Society (2010). https://doi.org/10.1109/QEST.2010.26
2. Amparore, E.G., Donatelli, S.: Efficient model checking of the stochastic logic CSL$^{\text{TA}}$. Perform. Evaluation **123-124**, 1–34 (2018). https://doi.org/10.1016/j.peva.2018.03.002
3. Andriushchenko, R., Ceska, M., Junges, S., Katoen, J.P., Stupinský, S.: PAYNT: A tool for inductive synthesis of probabilistic programs. In: CAV (1). LNCS, vol. 12759, pp. 856–869. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_40
4. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. ACM Transactions on Computational Logic **1**(1), 162–170 (2000)
5. Badings, T.S., Jansen, N., Junges, S., Stoelinga, M., Volk, M.: Sampling-based verification of CTMCs with uncertain rates. In: CAV (2). LNCS, vol. 13372, pp. 26–47. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_2
6. Badings, T.S., Romao, L., Abate, A., Jansen, N.: Probabilities are not enough: Formal controller synthesis for stochastic dynamical models with epistemic uncertainty. In: AAAI. pp. 14701–14710. AAAI Press (2023). https://doi.org/10.1609/aaai.v37i12.26718
7. Badings, T.S., Romao, L., Abate, A., Parker, D., Poonawala, H.A., Stoelinga, M., Jansen, N.: Robust control for dynamical systems with non-Gaussian noise via formal abstractions. J. Artif. Intell. Res. **76**, 341–391 (2023). https://doi.org/10.1613/jair.1.14253
8. Badings, T.S., Volk, M., Junges, S., Stoelinga, M., Jansen, N.: CTMCs with imprecisely timed observations. Tech. rep., CoRR, abs/2401.06574 (2024). https://doi.org/10.48550/arXiv.2401.06574
9. Baier, C., Dubslaff, C., Korenciak, L., Kucera, A., Rehák, V.: Mean-payoff optimization in continuous-time Markov chains with parametric alarms. ACM Trans. Model. Comput. Simul. **29**(4), 28:1–28:26 (2019). https://doi.org/10.1145/3310225
10. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Software Eng. **29**(6), 524–541 (2003). https://doi.org/10.1109/TSE.2003.1205180
11. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
12. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: TACAS. LNCS, vol. 8413, pp. 515–530. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_43
13. Bartocci, E., Deshmukh, J.V., Donzé, A., Fainekos, G., Maler, O., Nickovic, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In: Lectures on Runtime Verification, LNCS, vol. 10457, pp. 135–175. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_5
14. Bortolussi, L., Silvetti, S.: Bayesian statistical parameter synthesis for linear temporal properties of stochastic models. In: TACAS (2). LNCS, vol. 10806, pp. 396–413. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_23
15. Brázdil, T., Korenciak, L., Krcál, J., Novotný, P., Rehák, V.: Optimizing performance of continuous-time stochastic systems using timeout synthesis. In: QEST. LNCS, vol. 9259, pp. 141–159. Springer (2015). https://doi.org/10.1007/978-3-319-22264-6_10

16. Calinescu, R., Ceska, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N.: Efficient synthesis of robust models for stochastic systems. J. Syst. Softw. **143**, 140–158 (2018). https://doi.org/10.1016/j.jss.2018.05.013
17. Cardelli, L., Grosu, R., Larsen, K.G., Tribastone, M., Tschaikowski, M., Vandin, A.: Lumpability for uncertain continuous-time Markov chains. In: QEST. LNCS, vol. 12846, pp. 391–409. Springer (2021). https://doi.org/10.1007/978-3-030-85172-9_21
18. Cardelli, L., Grosu, R., Larsen, K.G., Tribastone, M., Tschaikowski, M., Vandin, A.: Algorithmic minimization of uncertain continuous-time Markov chains. IEEE Transactions on Automatic Control pp. 1–16 (2023). https://doi.org/10.1109/TAC.2023.3244093
19. Cauchi, N., Abate, A.: StocHy: Automated verification and synthesis of stochastic processes. In: TACAS (2). LNCS, vol. 11428, pp. 247–264. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_14
20. Ceska, M., Dannenberg, F., Paoletti, N., Kwiatkowska, M., Brim, L.: Precise parameter synthesis for stochastic biochemical systems. Acta Informatica **54**(6), 589–623 (2017). https://doi.org/10.1007/s00236-016-0265-2
21. Ceska, M., Jansen, N., Junges, S., Katoen, J.P.: Shepherding hordes of Markov chains. In: TACAS (2). LNCS, vol. 11428, pp. 172–190. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_10
22. Chen, T., Han, T., Katoen, J.P., Mereacre, A.: Model checking of continuous-time Markov chains against timed automata specifications. Log. Methods Comput. Sci. **7**(1) (2011). https://doi.org/10.2168/LMCS-7(1:12)2011
23. Feng, Y., Katoen, J.P., Li, H., Xia, B., Zhan, N.: Monitoring CTMCs by multi-clock timed automata. In: CAV (1). LNCS, vol. 10981, pp. 507–526. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_27
24. Gales, M.J.F., Young, S.J.: The application of hidden Markov models in speech recognition. Found. Trends Signal Process. **1**(3), 195–304 (2007). https://doi.org/10.1561/2000000004
25. Gao, Y., Hahn, E.M., Zhan, N., Zhang, L.: CCMC: A conditional CSL model checker for continuous-time Markov chains. In: ATVA. LNCS, vol. 8172, pp. 464–468. Springer (2013). https://doi.org/10.1007/978-3-319-02444-8_36
26. Gao, Y., Xu, M., Zhan, N., Zhang, L.: Model checking conditional CSL for continuous-time Markov chains. Inf. Process. Lett. **113**(1-2), 44–50 (2013). https://doi.org/10.1016/j.ipl.2012.09.009
27. Givan, R., Leach, S.M., Dean, T.L.: Bounded-parameter Markov decision processes. Artif. Intell. **122**(1-2), 71–109 (2000). https://doi.org/10.1016/S0004-3702(00)00047-3
28. Guan, J., Yu, N.: A probabilistic logic for verifying continuous-time Markov chains. In: TACAS (2). LNCS, vol. 13244, pp. 3–21. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_1
29. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: abstraction refinement for infinite probabilistic models. In: TACAS. LNCS, vol. 6015, pp. 353–357. Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_30
30. Hahn, E.M., Norman, G., Parker, D., Wachter, B., Zhang, L.: Game-based abstraction and controller synthesis for probabilistic hybrid systems. In: QEST. pp. 69–78. IEEE Computer Society (2011). https://doi.org/10.1109/QEST.2011.17
31. Han, T., Katoen, J.P., Mereacre, A.: Approximate parameter synthesis for probabilistic time-bounded reachability. In: RTSS. pp. 173–182. IEEE Computer Society (2008). https://doi.org/10.1109/RTSS.2008.19

32. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: TACAS (1). LNCS, vol. 13993, pp. 469–488. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_24
33. Haverkort, B.R., Hermanns, H., Katoen, J.P.: On the use of model checking techniques for dependability evaluation. In: SRDS. pp. 228–237. IEEE Computer Society (2000). https://doi.org/10.1109/RELDI.2000.885410
34. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022). https://doi.org/10.1007/s10009-021-00633-z
35. Hermanns, H., Meyer-Kayser, J., Siegle, M.: Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In: 3rd Int. Workshop on the Numerical Solution of Markov Chains. pp. 188–207. Citeseer (1999)
36. Hobolth, A., Stone, E.A.: Simulation from endpoint-conditioned, continuous-time Markov chains on a finite state space, with applications to molecular evolution. The annals of applied statistics **3**(3), 1204 (2009)
37. Iyengar, G.N.: Robust dynamic programming. Math. Oper. Res. **30**(2), 257–280 (2005). https://doi.org/10.1287/moor.1040.0129
38. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: LICS. pp. 266–277. IEEE Computer Society (1991). https://doi.org/10.1109/LICS.1991.151651
39. Junges, S., Torfah, H., Seshia, S.A.: Runtime monitors for Markov decision processes. In: CAV (2). LNCS, vol. 12760, pp. 553–576. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_26
40. Katoen, J.P.: The probabilistic model checking landscape. In: LICS. pp. 31–45. ACM (2016). https://doi.org/10.1145/2933575.2934574
41. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Formal Methods Syst. Des. **36**(3), 246–280 (2010). https://doi.org/10.1007/s10703-010-0097-6
42. Korenciak, L., Kucera, A., Rehák, V.: Efficient timeout synthesis in fixed-delay CTMC using policy iteration. In: MASCOTS. pp. 367–372. IEEE Computer Society (2016). https://doi.org/10.1109/MASCOTS.2016.34
43. Lavaei, A., Soudjani, S., Abate, A., Zamani, M.: Automated verification and synthesis of stochastic hybrid systems: A survey. Autom. **146**, 110617 (2022). https://doi.org/10.1016/j.automatica.2022.110617
44. Nilim, A., Ghaoui, L.E.: Robust control of Markov decision processes with uncertain transition matrices. Oper. Res. **53**(5), 780–798 (2005). https://doi.org/10.1287/opre.1050.0216
45. Perkins, T.J.: Maximum likelihood trajectories for continuous-time Markov chains. In: NIPS. pp. 1437–1445. Curran Associates, Inc. (2009)
46. Puggelli, A., Li, W., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Polynomial-time verification of PCTL properties of MDPs with convex uncertainties. In: CAV. LNCS, vol. 8044, pp. 527–542. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_35
47. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887
48. Ruijters, E., Stoelinga, M.: Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15**, 29–62 (2015). https://doi.org/10.1016/j.cosrev.2015.03.001

49. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). https://doi.org/10.1007/s10703-019-00337-w
50. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: RV. LNCS, vol. 7186, pp. 193–207. Springer (2011). https://doi.org/10.1007/978-3-642-29860-8_15
51. Winterer, L., Junges, S., Wimmer, R., Jansen, N., Topcu, U., Katoen, J.P., Becker, B.: Strategy synthesis for POMDPs in robot planning via game-based abstractions. IEEE Trans. Autom. Control. **66**(3), 1040–1054 (2021). https://doi.org/10.1109/TAC.2020.2990140
52. Wolff, E.M., Topcu, U., Murray, R.M.: Robust control of uncertain Markov decision processes with temporal logic specifications. In: CDC. pp. 3372–3379. IEEE (2012). https://doi.org/10.1109/CDC.2012.6426174

Pareto Curves for Compositionally Model Checking String Diagrams of MDPs[*]

Kazuki Watanabe[1,2(✉)**], Marck van der Vegt[3,**], Ichiro Hasuo[1,2], Jurriaan Rot[3], and Sebastian Junges[3]

[1] National Institute of Informatics, Tokyo, Japan
{kazukiwatanabe,hasuo}@nii.ac.jp
[2] The Graduate University for Advanced Studies (SOKENDAI), Hayama, Japan
[3] Radboud University, Nijmegen, the Netherlands
{marck.vandervegt,sebastian.junges}@ru.nl, jrot@cs.ru.nl

Abstract. Computing schedulers that optimize reachability probabilities in MDPs is a standard verification task. To address scalability concerns, we focus on MDPs that are compositionally described in a high-level description formalism. In particular, this paper considers *string diagrams*, which specify an algebraic, sequential composition of subMDPs. Towards their compositional verification, the key challenge is to locally optimize schedulers on subMDPs without considering their context in the string diagram. This paper proposes to consider the schedulers in a subMDP which form a *Pareto curve* on a combination of local objectives. While considering all such schedulers is intractable, it gives rise to a highly efficient sound approximation algorithm. The prototype on top of the model checker Storm demonstrates the scalability of this approach.

Keywords: Markov decision process· compositional verification · probabilistic model checking · multi-objective optimization

1 Introduction

Markov decision processes (MDPs) are a ubiquitous model for describing systems with both nondeterministic and probabilistic uncertainty. A key problem is to compute the best-case probability of reaching a goal state in a given MDP, i.e., to compute maximal reachability probabilities. Reachability probabilities can efficiently be computed for MDPs with $\approx 10^7$ states [4, 15], using mature model checkers such as Storm [17], PRISM [22] or Modest [13]. However, scalability beyond state space sizes suffers from the memory limitations inherent to explicitly

[*] I.H. and K.W. are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603) and the ASPIRE grant No. JPMJAP2301, JST. K.W. is supported by the JST grants No. JPMJFS2136 and JPMJAX23CU. J.R. is supported by the NWO grant No. OCENW.M20.053.

[**] Equal contribution.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 279–298, 2024.
https://doi.org/10.1007/978-3-031-57249-4_14

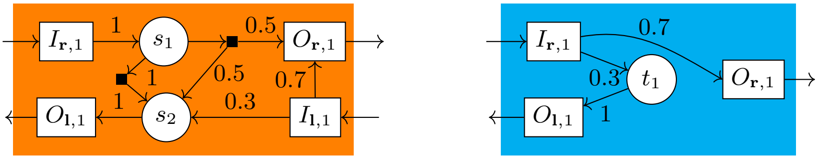


Fig. 1: Open Markov decision processes $\mathcal{A}$ and $\mathcal{B}$.

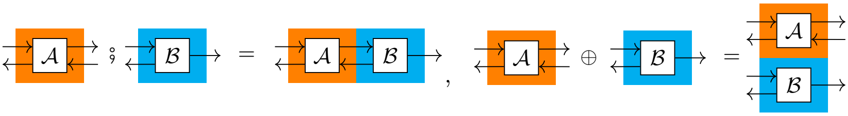


Fig. 2: Sequential composition $\mathcal{A} \fatsemi \mathcal{B}$ and sum $\mathcal{A} \oplus \mathcal{B}$

storing the transition matrix. While decision diagrams [2, 3, 19] are powerful, compact representations, they fail to concisely represent many MDPs [7].

Sequential composition. Compositional techniques attempt to avoid reasoning on the complete state space. We distinguish *parallel* and *sequential* compositions. This paper considers the compositional analysis of sequentially composed models [6]. This type of compositionality allows to reduce the peak memory consumption by reasoning about the individual parts and allows to exploit the typical existence of isomorphic parts of the state space. Sequentially composed MDPs have seen a surge in interest recently [20, 21, 26, 32, 33].

String diagrams. We focus on *string diagrams of MDPs* [33], which are MDPs composed by two algebraic operations: the *sequential composition* $\fatsemi$ and the *sum* $\oplus$. More precisely, we use *open MDPs*, extending MDPs with entrance and exit states. Fig. 1 shows open MDPs $\mathcal{A}$ (left) and $\mathcal{B}$ (right). The open MDP $\mathcal{A}$, for instance, has two entrances $i_{\mathbf{r},1}, i_{\mathbf{l},1}$ and two exits $o_{\mathbf{r},1}, o_{\mathbf{l},1}$. The algebraic operations define how the open MDPs are *subMDPs of a larger, monolithic MDP*, cf. Fig. 2. We highlight that, in our *bi-directional* framework, the sequential composition of acyclic open MDPs may lead to a cyclic monolithic MDP.

Optimal local schedulers. The idea of compositional reasoning is to analyze the open MDPs individually and combine these results to answer reachability queries on the monolithic MDP. The key challenge is that during the analysis of an individual open MDP, it is unclear which exits are (un)desirable to be reached. Equivalently, a priori, we do not know the objective that a scheduler should (locally) optimize for, to resolve the nondeterminism in the open MDP.

State-of-the-art. So far, this problem has been circumvented in the literature on compositional MDP verification. In [21], the notion of locally optimal policies is used. In essence, the technique relies on syntactic restrictions, such as open MDPs without nondeterminism or with a dedicated *desirable* exit. In [26], it is assumed that agents must optimize to reach one of the exits and that reaching another exit is equivalent to reaching an error. The results in [33] are the first to support general string diagrams of MDPs. Algorithmically, they enumerate over all (deterministic memoryless) schedulers in every open MDP, reducing the resulting set of schedulers only a posteriori using so-called meagre semantics. Finally, work on compositional *planning* aims to find a good scheduler compositionally, but without any guarantees on the optimality [6].

A multi-objective perspective. Towards a compositional analysis, we reformulate and generalize our problem slightly. Rather than considering *Given an open MDP, what is the maximal probability to reach a dedicated exit?*, we maximize the probability towards each exit individually, i.e., *What is the maximal probability to reach the first and second exit, respectively?* For this question, trade-offs are possible: In $\mathcal{A}$ when starting in $I_{r,1}$ we either reach $O_{l,1}$ with probability 1 (with one scheduler) or $O_{r,1}$ with probability 0.5 (with another scheduler). However, we search for *one scheduler* that makes this trade-off somehow. The unknown objective that a scheduler in an open MDP optimizes for is not arbitrary, but it is given by this trade-off between reaching the different exits. A key insight is that it suffices to only consider schedulers that refer to an optimal trade-off between the different exits. As the context of the open MDP determines the preferred trade-off, we compute all schedulers that are optimal for a specific trade-off. These schedulers are *Pareto-optimal* and their computation is well-studied [9,12,30].

Our approach. Towards a compositional algorithm, we suggest to compute Pareto curves recursively on the structure of the given string diagram of MDPs: given the Pareto curves for the open MDPs, we can compute the Pareto curve of their composition. As the set of Pareto-optimal schedulers remains exponential, we exploit efficient but approximative approaches to compute sound over-and-under approximations [12]. In practice, this means that tight approximations can be achieved that are concisely represented using only a few schedulers. Given these sound approximations for each open MDP, we compute sound approximations[4] for their composition, and ultimately for the whole string diagram.

Contributions. Our technical contribution is as follows. We provide a novel framework for analysing sequentially composed MDPs. In particular, it takes off-the-shelf analysis of *multi-objective monolithic MDPs* to provide an *compositional MDP model checking algorithm.* The approximative version of this algorithm computes guaranteed over- and under-approximations of reachability probabilities and scales to models with both billions of states and schedulers, while generating tight bounds. We implement the algorithm on top of the probabilistic model checker Storm to demonstrate its performance.

2 Overview

Illustrative example. We consider the small multi-room grid world with three rooms $\mathcal{A}, \mathcal{B}, \mathcal{C}$ and probabilistic outcomes to actions, as illustrated on the left in Fig. 3. The doors can be travelled through in both directions. The grid world can be modelled as a monolithic MDP. In Fig. 3(right), we show how to express the MDP compositionally as a string diagram $\mathcal{A} \fatsemi (\mathcal{B} \oplus \mathrm{id}) \fatsemi \mathcal{C}$, where id is an

[4] Sound approximations are standard in probabilistic model checking, where the standard and highly scalable algorithms [16] provide sound approximations [15].

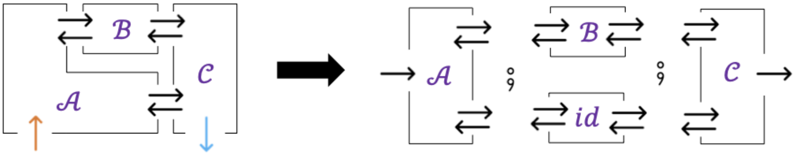


Fig. 3: From a multi-room MDP to a string diagram $\mathcal{A} \fatsemi (\mathcal{B} \oplus \mathrm{id}) \fatsemi \mathcal{C}$.

MDP where the unique entrance reaches the exit with probability one. Now, if we are interested (how to) reach the (main, rightmost) exit in $\mathcal{C}$ from the (main, leftmost) entrance in $\mathcal{A}$, we maximize the reachability probability in the monolithic MDP. However, to determine the optimal scheduler compositionally, we must know the optimal reachabilities in the each room individually. This is hard: In particular, it may be true that in order to reach the main exit from the door between $\mathcal{A}$ and $\mathcal{C}$, it is still optimal to go via room $\mathcal{B}$.

The multi-objective perspective. We consider room $\mathcal{A}$ from the perspective of the main entrance. There are two doors, which means that the underlying open MDP has two exit states. Fig. 6a shows a Pareto-plot that clarifies the possible trade-offs, e.g., for the main entrance[5]. In particular, $\mathbf{p}_1$ reflects a scheduler reaching the first door with probability 0.3, while the second door is then reached with probability 0.1. The other points reflect other schedulers that reach these doors with different probabilities. The other entrances in $\mathcal{A}$ induce other Pareto curves. In room $\mathcal{C}$, there are three exits (two doors and the main exit), making the Pareto curves three-dimensional. For MDP id, the curve is a (trivial) point.

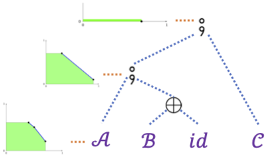


Fig. 4: Our approach

The approach illustrated. We approach the syntax tree of the string diagram recursively (see Fig. 4), i.e., we consider the string diagram as a syntax tree and (conceptually) annotate the MDPs $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and id with Pareto curves. Instead of computing these Pareto curves precisely, we create sound approximations as in Fig. 6b where the vertices of the green area L underapproximate the Pareto curve and the vertices of the green and white area $\mathbb{R}^2 \setminus U$ overapproximate the Pareto-curve. For the algebraic operations, we then combine these approximations. This is straightforward for the $\oplus$ as the two MDPs are independent. For the sequential composition, the cyclic dependencies are more involved. However, our algorithm is straightforward: we take the individual approximations of the Pareto curves, translate them to small so-called *shortcut MDPs*, which we then compose. On these small MDPs, we then compute (approximations of) the Pareto curves that are sound approximations to the composition at hand.

[5] An actual MDP corresponding to this Pareto curve is given in Fig. 5a

3 Formal Problem Statement

We recap MDPs, and discuss string diagrams of MDPs. We then give a multi-objective version of the problem statement, leading to a compositional algorithm.

For a finite set X, we write $\mathcal{D}(X)$ for the set of distributions on X and $\mathcal{D}_{\leq 1}(X)$ for the subdistributions. The support of $\mu \in \mathcal{D}_{\leq 1}(X)$ is denoted by $\mathrm{supp}(\mu)$.

3.1 Markov Decision Processes

Definition 1 (MDP). *An MDP $M = (S, A, P)$ is a tuple with a finite set S of* states*, finite set A of* actions*, and a partial* transition function $P \colon S \times A \rightharpoonup \mathcal{D}(S)$.

We use S_M, A_M and P_M to refer to the states, actions and transition function of an MDP M. For a state s, the *enabled actions* are $A(s) = \{a \in A \mid P(s, a) \neq \bot\}$. A state s is a *terminal* if $A(s) = \emptyset$. We write $P(s, a, s') := P(s, a)(s')$ if $a \in A(s)$ and $P(s, a, s') := 0$ otherwise. A state s is called *absorbing* if $P(s, a, s) = 1$ for all $a \in A(s)$. Terminals can be made absorbing by adding a self-loop. A *path* π is an (in)finite alternating sequence of states and actions, i.e., $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots$ such that $s_i \in S$ and $P(s_i, a_i, s_{i+1}) \neq 0$. For a finite path π, $\mathsf{last}(\pi)$ denotes the last state of π. The set of all finite paths is denoted FPath_M, the set of all infinite paths is denoted by IPath_M. We drop the subscript, whenever M is clear from the context. Finally, a *Markov chain* (MC) is an MDP with $|A(s)| \leq 1$. We write MCs as a tuple (S, P), formally representing the MDP $(S, \{\bot\}, P)$.

Schedulers. Schedulers (a.k.a. *policies* or *strategies*) resolve the nondeterminism in an MDP. In general, a (history-dependent) scheduler σ for MDP M is a (measurable) function $\mathsf{FPath}_M \to \mathcal{D}(A_M)$ with $\mathrm{supp}(\sigma(\pi)) \subseteq A(\mathsf{last}(\pi))$. The set of all history-dependent schedulers is denoted $\varSigma^M_{\mathrm{h}}$. A scheduler σ is *memoryless* (a.k.a. *positional*) if for every $\pi, \pi' \in \mathsf{FPath}$, $\mathsf{last}(\pi) = \mathsf{last}(\pi')$ implies $\sigma(\pi) = \sigma(\pi')$. The set of all (stochastic) memoryless schedulers is denoted $\varSigma^M_{\mathrm{sm}}$. We often write such schedulers as $S_M \to \mathcal{D}(A_M)$, i.e., as a map from the last state to a distribution over actions. Finally, deterministic (memoryless) schedulers map paths (or states, respectively) to Dirac distributions. We write such schedulers as maps from paths to actions, $\mathsf{FPath}_M \to A_M$. The set of all deterministic memoryless schedulers is denoted $\varSigma^M_{\mathrm{dm}}$. We also call these schedulers DM schedulers.

Reachability probabilities. Let M be an MDP without terminals. For an initial state s_ι and a scheduler σ, we obtain probability measure $\mathsf{Pr}^{s_\iota,\sigma}_M \colon \mathsf{IPath}_M \to \mathbb{R}_{\geq 0}$ on infinite paths via the standard cylinder set construction [5]. For a set of target states $T \subseteq S_M$, we define the set of paths that visit T, $\lozenge T = \{\pi \in \mathsf{IPath}_M \mid \exists i. \pi_i \in T\}$. The reachability probability $\mathrm{RPr}^{M,\sigma}(s_\iota, T)$ is the integral over the reachability measure, $\mathrm{RPr}^{M,\sigma}(s_\iota, T) := \int_{\pi \in \lozenge T} \mathsf{Pr}^{s_\iota,\sigma}_M(\pi)$. We relax notation and write $\mathrm{RPr}^{M,\sigma}(s_\iota, t) := \mathrm{RPr}^{M,\sigma}(s_\iota, \{t\})$. We write $\mathrm{RPr}^M_{\max}(s_\iota, t) := \max_\sigma \mathrm{RPr}^{M,\sigma}(s_\iota, t)$.

3.2 String Diagrams of MDPs

MDPs are given as a string diagram, i.e., as algebraically composed *open MDPs.*

Definition 2 (oMDP). *An* open MDP *(oMDP)* $\mathcal{A} = (M, \mathsf{IO})$ *is a pair consisting of an MDP* M *with states* S *and* open ends $\mathsf{IO} = (I_{\mathbf{r}}, I_{\mathbf{l}}, O_{\mathbf{r}}, O_{\mathbf{l}})$*, where* $I_{\mathbf{r}}, I_{\mathbf{l}}, O_{\mathbf{r}}, O_{\mathbf{l}} \subseteq S$ *are pairwise disjoint and totally ordered sets. The states* $I := I_{\mathbf{r}} \cup I_{\mathbf{l}}$ *is the* entrances*, and the states* $O := O_{\mathbf{r}} \cup O_{\mathbf{l}}$ *is the* exits*, respectively.*

Fig. 1 shows two oMDPs as examples. We assume that exactly the exits are terminals. We lift the notions of policies and reachability probabilities straightforwardly from MDPs[6]. As the open ends are ordered, we may enumerate, e.g., its entrances I using $I_{\mathbf{r},1}, \dots, I_{\mathbf{r},|I_{\mathbf{r}}|}, I_{\mathbf{l},1}, \dots, I_{\mathbf{l},|I_{\mathbf{l}}|}$ from *rightward* to *leftward*. We specialise the notation for reachability probabilities $\mathrm{RPr}^*(i, j)$ to denote $\mathrm{RPr}^*(I_i, O_j)$. We explicitly write $\mathrm{tp}(\mathcal{A})\colon (m_{\mathbf{r}}, m_{\mathbf{l}}) \to (n_{\mathbf{r}}, n_{\mathbf{l}})$ for the *arities* of $\mathcal{A}$, where $m_{\mathbf{r}} := |I_{\mathbf{r}}|$, $m_{\mathbf{l}} := |O_{\mathbf{l}}|$, $n_{\mathbf{r}} := |O_{\mathbf{r}}|$, and $n_{\mathbf{l}} := |I_{\mathbf{l}}|$.

String diagrams of MDPs use two algebraic operations on oMDPs: the *sequential composition* $⨾$ and the *sum* $\oplus$, that we illustrated already in Fig. 2.

Definition 3 ($⨾$ operator). *Let* $\mathcal{A}$*,* $\mathcal{B}$ *be oMDPs,* $\mathrm{tp}(\mathcal{A}) = (m_{\mathbf{r}}, m_{\mathbf{l}}) \to (l_{\mathbf{r}}, l_{\mathbf{l}})$, $\mathrm{tp}(\mathcal{B}) = (l_{\mathbf{r}}, l_{\mathbf{l}}) \to (n_{\mathbf{r}}, n_{\mathbf{l}})$*. Their* sequential composition $\mathcal{A} ⨾ \mathcal{B}$ *is an oMDP* (M, IO') *with* $\mathsf{IO}' = (I^{\mathcal{A}}_{\mathbf{r}}, I^{\mathcal{B}}_{\mathbf{l}}, O^{\mathcal{B}}_{\mathbf{r}}, O^{\mathcal{A}}_{\mathbf{l}})$*,* $M := (S^{\mathcal{A}} \uplus S^{\mathcal{B}}, A^{\mathcal{A}} \uplus A^{\mathcal{B}}, P)$ *and* P *s.t.*

$$P(s,a,s') := \begin{cases} P^{\mathcal{D}}(s,a,s') & \textit{if } \mathcal{D} \in \{\mathcal{A}, \mathcal{B}\},\ s \in S^{\mathcal{D}},\ a \in A^{\mathcal{D}},\ s' \in S^{\mathcal{D}}, \\ 1 & \textit{if } s = O^{\mathcal{A}}_{\mathbf{r},i},\ s' = I^{\mathcal{B}}_{\mathbf{r},i} \textit{ for some } 1 \le i \le l_{\mathbf{r}}, \\ 1 & \textit{if } s = O^{\mathcal{B}}_{\mathbf{l},i},\ s' = I^{\mathcal{A}}_{\mathbf{l},i} \textit{ for some } 1 \le i \le l_{\mathbf{l}}, \\ 0 & \textit{otherwise.} \end{cases}$$

If $\mathcal{A} ⨾ \mathcal{B}$ is well-defined, we say that $\mathcal{A} ⨾ \mathcal{B}$ type-matches.

Definition 4 ($\oplus$ operator). *Let* $\mathcal{A}, \mathcal{B}$ *be oMDPs. Their* sum $\mathcal{A} \oplus \mathcal{B}$ *is an oMDP* (M, IO') *with* $\mathsf{IO}' = (I^{\mathcal{A}}_{\mathbf{r}} \uplus I^{\mathcal{B}}_{\mathbf{r}}, I^{\mathcal{A}}_{\mathbf{l}} \uplus I^{\mathcal{B}}_{\mathbf{l}}, O^{\mathcal{A}}_{\mathbf{r}} \uplus O^{\mathcal{B}}_{\mathbf{r}}, O^{\mathcal{A}}_{\mathbf{l}} \uplus O^{\mathcal{B}}_{\mathbf{l}})$ *and* $M = (S^{\mathcal{A}} \uplus S^{\mathcal{B}}, A^{\mathcal{A}} \uplus A^{\mathcal{B}}, P)$ *where* P *is given by*

$$P(s,a,s') := \begin{cases} P^{\mathcal{D}}(s,a,s') & \textit{if } \mathcal{D} \in \{\mathcal{A}, \mathcal{B}\},\ s \in S^{\mathcal{D}},\ a \in A^{\mathcal{D}},\ \textit{and } s' \in S^{\mathcal{D}}, \\ 0 & \textit{otherwise.} \end{cases}$$

Here, the total order in $I^{\mathcal{A}}_{\mathbf{r}} \uplus I^{\mathcal{B}}_{\mathbf{r}}$ *is given by* $I^{\mathcal{A}}_{\mathbf{r},1}, \dots, I^{\mathcal{A}}_{\mathbf{r},m_{\mathbf{r}}}, I^{\mathcal{B}}_{\mathbf{r},1}, \dots, I^{\mathcal{B}}_{\mathbf{r},k_{\mathbf{r}}}$*, and the total orders in other open ends are defined similarly.*

Using the algebraic operators, we define *string diagrams of MDPs.*

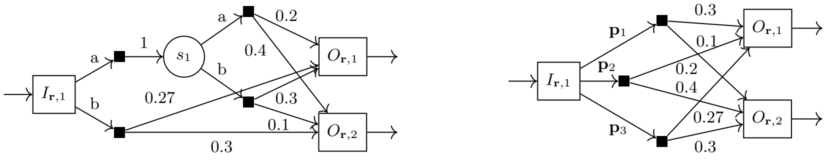
Definition 5. *A* string diagram $\mathbb{D}$ of MDPs *is a term adhering to the grammar*

$$\mathbb{D} ::= \mathcal{A} \mid \mathbb{D} ⨾ \mathbb{D} \mid \mathbb{D} \oplus \mathbb{D}$$

where $\mathcal{A}$ *ranges over oMDPs. The* operational semantics $[\![\mathbb{D}]\!]$ *is the oMDP which is inductively defined by Definitions 3 and 4.*

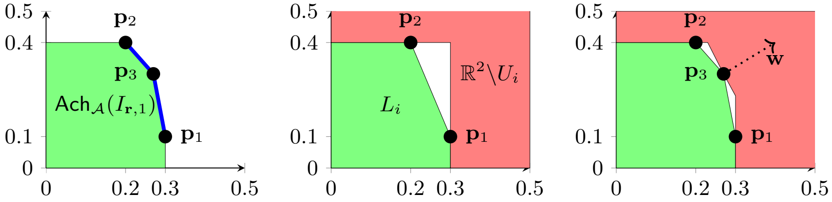
Throughout this paper, we assume that $[\![\mathbb{D}]\!]$ is well-defined, i.e., that all operations type-match. We omit syntactic sugar operations, such as probabilistic or nondeterministic branching, as these can be modelled inside oMDPs. In the literature, the $[\![\mathbb{D}]\!]$ are also referred to as the *monolithic MDP* for $\mathbb{D}$.

[6] Where terminals have to be made absorbing by adding a self-loop.



(a) oMDP $\mathcal{A}$, $\mathrm{tp}(\mathcal{A}) = (1, 0) \to (2, 0)$ (b) Shortcut oMDP $\mathcal{C}(\mathsf{Ach}_{\mathcal{A}})$.

Fig. 5: Two oMDPs. We omit transitions to a sink for readability.



(a) $\mathsf{Ach}_{\mathcal{A}}(I_{\mathbf{r},1})$. (b) Sound approx. (L_i, U_i) (c) Approx. (L_i, U_i).

Fig. 6: Pareto curves (blue) and achievable regions and their underapproximation (green) as well as their overapproximation (white+green).

Single-Exit Problem Statement: Given a string diagram $\mathbb{D}$, an entrance i, an exit j, and an error bound $\epsilon \in [0, 1]$, compute a scheduler σ such that

$$\mathrm{RPr}^{[\![\mathbb{D}]\!]}_{\max}(i, j) - \mathrm{RPr}^{[\![\mathbb{D}]\!], \sigma}(i, j) \leq \epsilon.$$

We remark that DM schedulers suffice for this problem statement.

Remark 1. String diagrams are traditionally graphical languages based on category theory, which involve not only terms but also equations; see [18, 25, 31]. The definition of string diagrams of MDPs in [33] follows in this tradition and satisfies certain equational axioms. In this paper, the equations do not play a role explicitly; our algorithms assume a syntactic presentation. Solely for the purpose of exposition, we use a more concise definition where some axioms do not hold (although they "essentially" hold, modulo isomorphisms and removing redundant open ends). All definitions, results, and even proofs in this paper are concretely described and self-contained, without any use of category theory.

3.3 A Multi-Objective Generalization

Towards a recursive, compositional formulation of the problem statement, we generalize the single-exit problem to allow for multiple exits. Concretely, the generalized problem statement is to compute *Pareto curves* [12, 27] that represent combinations of reachability probabilities towards a set of exits.

Example 1. Consider the oMDP $\mathcal{A}$ in Fig. 5a. From the entrance, the scheduler σ_1 that always chooses the a yields reachability probabilities $(0.2, 0.4)$ to reach the first and second exit, respectively, while σ_2 that chooses b in s_1 yields $(0.3, 0.1)$.

Geometry. For two points $\mathbf{p}, \mathbf{p}' \in \mathbb{R}^n$, we write $\mathbf{p} \preceq \mathbf{p}'$ for pointwise inequality, i.e., $p_j \leq p'_j$ for every $j \leq n$. The set of (normalized) weight vectors of dimension $n \in \mathbb{N}$ is given by $\mathbf{W}^n = \{\mathbf{w} = \langle w_1, \dots, w_n \rangle \in \mathbb{R}^n_{\geq 0} \mid \sum_i w_i = 1\}$. The *convex closure* of a set X is the set $\mathsf{ConvCl}(X) := \{\sum w_i \cdot \mathbf{p}_i \mid w \in \mathbf{W}^n, \mathbf{p}_1, \dots, \mathbf{p}_n \in X\}$. The *downward closure* of a set X is the set $\mathsf{DwCl}(X) := \{\mathbf{p} \mid \exists \mathbf{p}' \in X. \mathbf{p} \preceq \mathbf{p}'\}$. The downward convex closure is $\mathsf{DwConvCl}(X) := \mathsf{DwCl}(\mathsf{ConvCl}(X))$. A set X is convex or downward-closed, if X is equal to its convex- or downward-closure, respectively. A convex, downward-closed set is *finitely generated*, if it is the downward convex closure of a finite set of points that we call *vertices*.
We write $\mathrm{RPr}^{\mathcal{A},\sigma}(i, O) \in \mathbb{R}^O$ for a point with $\mathrm{RPr}^{\mathcal{A},\sigma}(i, O)(j) = \mathrm{RPr}^{\mathcal{A},\sigma}(i, j)$.

Definition 6 (Achievable). *For an oMDP $\mathcal{A}$, $i \in I^{\mathcal{A}}$ and a scheduler σ*

$$\mathsf{Ach}^{\sigma}_{\mathcal{A}}(i) := \{\mathbf{p} \in \mathbb{R}^O \mid \mathbf{p} \preceq \mathrm{RPr}^{\mathcal{A},\sigma}(i, O)\}$$

is the points achieved by σ. For a set of schedulers Σ, $\mathsf{Ach}^{\Sigma}_{\mathcal{A}}(i) := \bigcup_{\sigma \in \Sigma} \mathsf{Ach}^{\sigma}_{\mathcal{A}}(i)$. The set of achievable points *is $\mathsf{Ach}_{\mathcal{A}}(i) := \mathsf{Ach}^{\Sigma^{\mathcal{A}}_h}_{\mathcal{A}}(i)$. The* Pareto curve *is the set*

$$\mathcal{S}(\mathcal{A}, i) := \{\mathbf{p} \in \mathsf{Ach}_{\mathcal{A}}(i) \mid \textit{for all } \mathbf{p}' \succ \mathbf{p} : \mathbf{p}' \notin \mathsf{Ach}_{\mathcal{A}}(i)\}.$$

A scheduler σ is Pareto-optimal (w.r.t. i), if $\mathrm{RPr}^{\mathcal{A},\sigma}(i, O) \in \mathcal{S}(\mathcal{A}, i)$.

Points in the Pareto curve are also called *Pareto-optimal*. The set of achievable points is convex and downward-closed, the downward closure of the Pareto curve is the set of achievable points [9,11]. Fig. 6a illustrates the achievable points (the green region) and the Pareto curve (the blue line) of $\mathcal{A}$ from the entrance $I_{\mathbf{r},1}$. From facts on the necessary schedulers in multi-objective model-checking [9] with the fact that exits are absorbing, we only need to consider memoryless schedulers and the vertices of the achievable points all correspond to DM schedulers:

Lemma 1. *For oMDP $\mathcal{A}$ and entrance $i \in I$:*

$$\mathsf{DwCl}(\mathcal{S}(\mathcal{A}, i)) = \mathsf{Ach}_{\mathcal{A}}(i) = \mathsf{Ach}^{\Sigma^{\mathcal{A}}_{sm}}_{\mathcal{A}}(i) = \mathsf{DwConvCl}(\mathsf{Ach}^{\Sigma^{\mathcal{A}}_{dm}}_{\mathcal{A}}(i)).$$

We define the *error* $\mathsf{E}(X, Y)$ between downward-closed convex sets $X \subseteq Y \subseteq \mathbb{R}^n$ using the L^2 norm $\|\_\|_2$ as in [29]: $\mathsf{E}(X, Y) := \sup_{\mathbf{p} \in Y} \inf_{\mathbf{p}' \in X} \|\mathbf{p} - \mathbf{p}'\|_2$.

To simplify notation, we write $\mathsf{Ach}^{\Sigma}_{\mathcal{A}}$ for the indexed family $\left(\mathsf{Ach}^{\Sigma}_{\mathcal{A}}(i)\right)_{i \in I}$ of the achievable points. Given families $X = \{X_i\}_{i \leq k}$ and $Y = \{Y_i\}_{i \leq k}$ with $X(i) \subseteq Y(i)$, we define $\mathsf{E}(X, Y) := \sup_{i \leq k} \mathsf{E}(X_i, Y_i)$.

We conservatively extend the single-exit problem statement. We now want to find a *set* of schedulers such that the error between the achieved points corresponding to this set and the (unknown) Pareto curve is small.

Algorithm 1 `approxMultiObjMDP`: Approximation of the Pareto curve

```
Input: an oMDP $\mathcal{A}$ and an imprecision $\eta \in [0,1]$.
Output: a sound approximation $(L,U)$ of $\mathcal{A}$ s.t. $\mathsf{E}(L,U) \leq \eta$.
initialize $L_i := \emptyset$ and $U_i := \mathcal{D}_{\leq 1}(O)$, for $i \in I$.
while $\mathsf{E}(L,U) > \eta$ do
    select an entrance $i \in I$, a weight vector $\mathbf{w} := (w_o)_{o \in O}$, and $\delta \geq 0$.
    find $l_{\mathbf{w}}, u_{\mathbf{w}}$ s.t. $l_{\mathbf{w}} \leq \sup_\sigma \mathrm{WRPr}^\sigma(\mathbf{w}, i) \leq u_{\mathbf{w}}$, $|l_{\mathbf{w}} - u_{\mathbf{w}}| \leq \delta$.
    compute $\mathbf{p}^l$ such that $l_{\mathbf{w}} = \mathbf{w} \cdot \mathbf{p}^l$.
    update $L_i := L_i \cup \{\mathbf{p}^l\}$ and $U_i := U_i \cap \{\mathbf{p} \mid \mathbf{w} \cdot \mathbf{p} \leq u_{\mathbf{w}}\}$.
return $(L,U)$.
```

Multi-Exit Problem Statement: Given a string diagram $\mathbb{D}$, an error bound $\epsilon \in [0,1]$, compute a set of schedulers Σ s.t. $\mathsf{E}(\mathsf{Ach}_{\mathcal{A}}^{\Sigma}, \mathsf{Ach}_{\mathcal{A}}) \leq \epsilon$.

4 Compositional Algorithm

We present a compositional algorithm by soundly approximating Pareto curves.

4.1 Approximating Pareto Curves on oMDPs

For an oMDP $\mathcal{A}$, we can efficiently approximate $\mathsf{Ach}_{\mathcal{A}}$ via an off-the-shelf multi-objective model checking algorithm [12,30]. We outline this algorithm, tailored to oMDPs. The key idea is that the algorithm iteratively refines a sound approximation of the Pareto curve.

Definition 7. *Let $\mathcal{A}$ be an oMDP. An* under-approximation *L is a family $L := (L_i)_{i \in I}$ such that L_i is a convex, downward-closed, and finitely generated with $L_i \subseteq \mathsf{Ach}_{\mathcal{A}}(i)$ for $i \in I$. An* over-approximation *U is analogously defined, with $\mathsf{Ach}_{\mathcal{A}}(i) \subseteq U_i$ for $i \in I$. We call (L,U) a* sound approximation *for $\mathcal{A}$.*

Algorithm 1 summarizes the approach. L and U are initialized as a trivial approximation. The algorithm iteratively refines them by computing *weighted reachability* probabilities for some weight vector $\mathbf{w} \in \mathbf{W}^O$, which is adequately chosen [30][7]. For that $\mathbf{w}$, we denote the weighted reachability using $\mathrm{WRPr}^{\mathcal{A},\sigma}(\mathbf{w}, i) := \sum_{j \in O} \mathbf{w}_j \cdot \mathrm{RPr}^{\mathcal{A},\sigma}(i,j)$. A scheduler τ is *weighted optimal* w.r.t. $\mathbf{w}$ and i if $\mathrm{WRPr}^{\mathcal{A},\tau}(\mathbf{w}, i) = \sup_{\sigma \in \mathrm{Sched}(\mathcal{A})} \mathrm{WRPr}^{\mathcal{A},\sigma}(\mathbf{w}, i)$. Weighted reachability can be computed via standard reachability query on a mildly modified MDP that has a fresh unique target and sink state. This implies that DM schedulers suffice for optimality. Furthermore, every Pareto optimal scheduler optimizes some weighted reachability. Finally, from a lower bound $l_{\mathbf{w}}$ we can compute an achievable $\mathbf{p}^l$ and use this point to update L. As the weighted optimal scheduler is optimal, we

[7] While our algorithm is indeed correct for $\delta > 0$, we only discuss $\delta = 0$ here.

obtain a linear inequality on the reachability probabilities that can be achieved in the direction of $\mathbf{w}$[8]. Fig. 6c illustrates how Algorithm 1 works.

Proposition 1 ([30]). *Algorithm 1 is correct.*

4.2 From Pareto Curves to Shortcut MDPs

In our algorithm, it will be convenient to construct a (small) MDP with a particular Pareto curve. In particular, we construct the oMDP in Fig. 5b from the Pareto curve in Fig. 6a. This construction is rather straightforward and we give it for both finite sets of (Pareto-optimal) points and for finitely generated convex sets. Due to the exits being terminals, $\mathrm{RPr}^{\mathcal{A},\sigma}(i, O) \in \mathcal{D}_{\leq 1}(O)$, where we liberally interpret distributions as points in $\mathbb{R}^{O}$.

Definition 8 (shortcut oMDP). *Let $\mathcal{A}$ be an oMDP, B be an indexed family $B := (B_i)_{i \in I^{\mathcal{A}}}$ of finite sets $B_i \subseteq \mathcal{D}_{\leq 1}(O^{\mathcal{A}})$. The* shortcut oMDP *for B is $\mathcal{C}(B) := (M, \mathsf{IO}^{\mathcal{A}})$, with $M := (S, A, P)$, $S := I^{\mathcal{A}} \cup O^{\mathcal{A}} \cup \{\star\}$, $A := \bigcup_{i \in I^{\mathcal{A}}} B_i$,*

$$P(s, a, s') := \begin{cases} a(s') & \text{if } s \in I^{\mathcal{A}},\ a \in B_s, \text{ and } s' \in O^{\mathcal{A}}, \\ 1 - \sum_{o \in O^{\mathcal{A}}} a(o) & \text{if } s \in I^{\mathcal{A}},\ a \in B_s, \text{ and } s' = \star, \\ 0 & \text{otherwise.} \end{cases}$$

Additionally, if B is an indexed family $B := (B_i)_{i \in I}$ of convex, downward-closed and finitely generated $B_i \subseteq \mathcal{D}_{\leq 1}(O)$ with vertices B_i^V, then $\mathcal{C}(B) := \mathcal{C}((B_i^V)_{i \in I})$.

For each entrance s, exit s' and point $\mathbf{p} \in B_s$, the probability transition $P(s, \mathbf{p}, s')$ is the reachability probability from s to s' in $\mathcal{A}$ induced by schedulers that yielded reachability probabilities in point $\mathbf{p}$. The sink $\star$ is introduced to ensure that we obtain proper distributions.

Proposition 2. *For oMDP $\mathcal{A}$, $i \in I^{\mathcal{A}}$, and sound approximation (L, U):*

$$\mathsf{Ach}_{\mathcal{C}(L)}(i) \quad \subseteq \quad \mathsf{Ach}_{\mathcal{A}}(i) \quad \subseteq \quad \mathsf{Ach}_{\mathcal{C}(U)}(i).$$

For the inclusions, a key element in the correctness of this statement that, intuitively, we can add points that are not vertices of B without changing the Pareto curve $\mathcal{C}(B)$. Based on the idea, we establish the following implication for $X, Y \subseteq \mathcal{D}(O)^{I}$ where $X(i), Y(i)$ are finitely generated downward-closed convex sets: $X(i) \subseteq Y(i)$ implies $\mathsf{Ach}_{\mathcal{C}(X)}(i) \subseteq \mathsf{Ach}_{\mathcal{C}(Y)}(i)$. Finally, we apply this implication twice. Details are given in [34, Appendix B].

Prop. 2 also implies the following corollary, which claims that Pareto-optimal schedulers suffice for optimality:

Corollary 1. *For oMDP $\mathcal{A}$,*

$$\mathsf{Ach}_{\mathcal{A}} = \mathsf{Ach}_{\mathcal{C}(\mathsf{Ach}_{\mathcal{A}})}.$$

[8] By construction, U_i is convex, downward-closed and finitely generated.

Algorithm 2 approxMultiObjSD: Approximation of the Pareto curve

```
Input: a string diagram $\mathbb{D}$ and a local imprecision $\eta \in [0, 1]$.
Output: A sound approximation $(L, U)$ for $[\![\mathbb{D}]\!]$. If $\eta = 0$, then $L = U$.
if $\mathbb{D} = \mathcal{A}$ then
    return approxMultiObjMDP($\mathcal{A}, \eta$)                          ▷ Invoke Algorithm 1
else                                                   ▷ $\mathbb{D} = \mathbb{D}_1 * \mathbb{D}_2$, $* \in \{\fatsemi, \oplus\}$
    $(L_1, U_1) \leftarrow$ approxMultiObjSD($\mathbb{D}_1, \eta$)                      ▷ Recursion
    $(L_2, U_2) \leftarrow$ approxMultiObjSD($\mathbb{D}_2, \eta$)                      ▷ Recursion
    $\mathcal{A}_L \leftarrow \mathcal{C}(L_1) * \mathcal{C}(L_2)$                                      ▷ See Def. 8
    $\mathcal{A}_U \leftarrow \mathcal{C}(U_1) * \mathcal{C}(U_2)$                                      ▷ See Def. 8
    return $\Big($approxMultiObjMDP($\mathcal{A}_L, \eta$)$\Big)_1$, $\Big($approxMultiObjMDP($\mathcal{A}_U, \eta$)$\Big)_2$
```

It is often convenient to obtain schedulers on either $\mathcal{C}(\mathsf{Ach}_\mathcal{A})$ and obtain a scheduler on $\mathcal{A}$ or vice versa. Furthermore, while Prop. 2 considers every entrance separately, we aim to have schedulers that match on *every* entrance *simultaneously* by remembering which entrance was taken (proof in [34, Appendix B]).

Proposition 3. *Let $\mathcal{A}$ be an oMDP and $\sigma \in \Sigma_{dm}^{\mathcal{C}(\mathsf{Ach}_\mathcal{A})}$. There is a scheduler τ on $\mathcal{A}$ s.t. $\mathrm{RPr}^{\mathcal{A},\tau}(i, O^\mathcal{A}) = \mathrm{RPr}^{\mathcal{C}(\mathsf{Ach}_\mathcal{A}),\sigma}(i, O^\mathcal{A})$ for every $i \in I^\mathcal{A}$. Let $\tau' \in \Sigma_{dm}^\mathcal{A}$, there is $\sigma' \in \Sigma_{dm}^{\mathcal{C}(\mathsf{Ach}_\mathcal{A})}$ s.t. $\mathrm{RPr}^{\mathcal{A},\tau'}(i, O^\mathcal{A}) \preceq \mathrm{RPr}^{\mathcal{C}(\mathsf{Ach}_\mathcal{A}),\sigma'}(i, O^\mathcal{A})$ for every $i \in I^\mathcal{A}$.*

4.3 Approximating Pareto Curves for String Diagrams

We now provide a recursive algorithm in Algorithm 2. In the base case, when is a single oMDP, we analyze the oMDP using Algorithm 1. Otherwise, we have an string diagram $\mathbb{D}_1 * \mathbb{D}_2$ for $* \in \{\fatsemi, \oplus\}$. We recursively compute sound approximations for $\mathbb{D}_1$ and $\mathbb{D}_2$. Next, we compose the under- and over-approximations, respectively. We discuss the under-approximation, the over-approximation is handled analogously. Given both under-approximations L_1, L_2, we create the corresponding shortcut MDPs $\mathcal{C}(L_1)$ and $\mathcal{C}(L_2)$ and then take their sequential composition $\mathcal{A}_L$. This oMDP can be analyzed using Algorithm 1. Any under-approximation for $\mathcal{A}_L$ is an underapproximation for $\mathbb{D}(:= \mathbb{D}_1 * \mathbb{D}_2)$. This algorithm easily supports additional operations such as n-ary compositions that significantly reduce overhead. We remark that contrary to Algorithm 1, we do not guarantee any error on the approximation that we return, unless all computations are precise ($\eta = 0$). We discuss error bounds in §5.

Correctness We first state that under-approximations and over-approximations are preserved by the algebraic operations:

Proposition 4 (Case $\fatsemi$). *For oMDPs $\mathcal{A}, \mathcal{B}$, and sound approximations $(L^\mathcal{A}, U^\mathcal{A})$ and $(L^\mathcal{B}, U^\mathcal{B})$, the tuple $\Big(\mathsf{Ach}_{\mathcal{C}(L^\mathcal{A})\fatsemi\mathcal{C}(L^\mathcal{B})}, \mathsf{Ach}_{\mathcal{C}(U^\mathcal{A})\fatsemi\mathcal{C}(U^\mathcal{B})}\Big)$ is a sound approximation for $\mathcal{A} \fatsemi \mathcal{B}$, i.e., for any $i \in I^{\mathcal{A}\fatsemi\mathcal{B}}$, the following conditions hold:*

$$\mathsf{Ach}_{\mathcal{C}(L^\mathcal{A})\fatsemi\mathcal{C}(L^\mathcal{B})}(i) \quad \subseteq \quad \mathsf{Ach}_{\mathcal{A}\fatsemi\mathcal{B}}(i) \quad \subseteq \quad \mathsf{Ach}_{\mathcal{C}(U^\mathcal{A})\fatsemi\mathcal{C}(U^\mathcal{B})}(i).$$

The proof (§4.4) is not trivial due to the cyclic dependency between $\mathcal{A}$ and $\mathcal{B}$. Similar to Corollary 2, Pareto-optimal schedulers suffice for compositionally solving Pareto curves w.r.t. the sequential composition.

Corollary 2. *For oMDPs $\mathcal{A}, \mathcal{B}$,*

$$\mathsf{Ach}_{\mathcal{A}\fatsemi\mathcal{B}} = \mathsf{Ach}_{\mathcal{C}(\mathsf{Ach}_{\mathcal{A}})\fatsemi\mathcal{C}(\mathsf{Ach}_{\mathcal{B}})}.$$

The similar compositionality result also holds for the sum:

Proposition 5 (Case $\oplus$). *For oMDPs $\mathcal{A}, \mathcal{B}$, and sound approximations $(L^{\mathcal{A}}, U^{\mathcal{A}})$ and $(L^{\mathcal{B}}, U^{\mathcal{B}})$, the tuple $\left(\mathsf{Ach}_{\mathcal{C}(L^{\mathcal{A}})\oplus\mathcal{C}(L^{\mathcal{B}})}, \mathsf{Ach}_{\mathcal{C}(U^{\mathcal{A}})\oplus\mathcal{C}(U^{\mathcal{B}})}\right)$ is a sound approximation for $\mathcal{A}\oplus\mathcal{B}$, i.e., for any $i \in I^{\mathcal{A}\oplus\mathcal{B}}$, the following conditions hold:*

$$\mathsf{Ach}_{\mathcal{C}(L^{\mathcal{A}})\oplus\mathcal{C}(L^{\mathcal{B}})}(i) \quad\subseteq\quad \mathsf{Ach}_{\mathcal{A}\oplus\mathcal{B}}(i) \quad\subseteq\quad \mathsf{Ach}_{\mathcal{C}(U^{\mathcal{A}})\oplus\mathcal{C}(U^{\mathcal{B}})}(i).$$

The statement is straightforward due to the lack of interaction between $\mathcal{A}$ and $\mathcal{B}$.

Theorem 1. *Algorithm 2 is correct. Additionally, if $\eta = 0$ then $\mathsf{E}(L, U) = 0$.*

Proof. We prove that output (L, U) is a sound approximation of $[\![\mathbb{D}]\!]$. We prove this recursively over the structure of $\mathbb{D}$. If $\mathbb{D} = \mathcal{A}$, Prop. 1 applies. Otherwise, we focus on L being a lower bound and with $* = \fatsemi$ the upper bound and $\oplus$ are analogous. Indeed $L(i) \subseteq \mathsf{Ach}_{\mathcal{C}(L_1)\fatsemi\mathcal{C}(L_2)}(i)$ for any entry i by Algorithm 1. Likewise, L_1, L_2 are lower bounds to $[\![\mathbb{D}_1]\!]$ and $[\![\mathbb{D}_2]\!]$ and thus the theorem follows Prop. 4 and Prop. 5. If $\eta = 0$, then L and U recursively coincide. □

Obtaining schedulers To obtain a scheduler, first observe that in Algorithm 1, every point in L_i can be annotated with a memoryless scheduler (using standard model checking). Now, when we obtain a memoryless scheduler for some $\mathcal{A} * \mathcal{B}$, then we can translate this straightforwardly to memoryless schedulers for $\mathcal{A}$ and $\mathcal{B}$. Finally, if we obtain a scheduler for $\mathcal{C}(L')$ and L' is an underapproximation for $\mathcal{A}$, then Prop. 3 states that we can recover a scheduler for $\mathcal{A}$.

4.4 Proof outline for Prop. 4

We give the main ingredients for Prop. 4, the key ingredient for our approach. We discuss the crux for showing $\mathsf{Ach}_{\mathcal{C}(\mathsf{Ach}_{\mathcal{A}})\fatsemi\mathcal{C}(\mathsf{Ach}_{\mathcal{B}})}(i) = \mathsf{Ach}_{\mathcal{A}\fatsemi\mathcal{B}}(i)$: We can map (memoryless) schedulers in $\mathcal{A}\fatsemi\mathcal{B}$ and $\mathcal{C}(\mathsf{Ach}_{\mathcal{A}}) \fatsemi \mathcal{C}(\mathsf{Ach}_{\mathcal{B}})$ to each other while matching reachability probabilities. More precisely, we lift Prop. 3 to the sequential composition, while using that Prop. 3 already established the mapping for $\mathcal{A}$ and $\mathcal{C}(\mathsf{Ach}_{\mathcal{A}})$. Therefore, we note that for any σ, $i \in I^{\mathcal{A}} \cup I^{\mathcal{B}}$ and $j \in O^{\mathcal{A}\fatsemi\mathcal{B}}$, the following equations hold directly from the definition of the sequential composition and from the definition of reachability probabilities by adequately partitioning the paths from entrance to exit, i.e. $\mathrm{RPr}^{\mathcal{A}\fatsemi\mathcal{B},\tau}(i,j) =$

$$\begin{cases} \mathrm{RPr}^{\mathcal{A},\tau}(i,j) + \sum_{k\in O^{\mathcal{A}}_{\mathrm{r}}} \mathrm{RPr}^{\mathcal{A},\tau}(i,k)\cdot \mathrm{RPr}^{\mathcal{A}\fatsemi\mathcal{B},\tau}(\mathsf{Nx}(k),j) & \text{if } i \in I^{\mathcal{A}}, \\ \mathrm{RPr}^{\mathcal{B},\tau}(i,j) + \sum_{k\in O^{\mathcal{B}}_{\mathrm{l}}} \mathrm{RPr}^{\mathcal{B},\tau}(i,k)\cdot \mathrm{RPr}^{\mathcal{A}\fatsemi\mathcal{B},\tau}(\mathsf{Nx}(k),j) & \text{if } i \in I^{\mathcal{B}}, \end{cases}$$

where $\mathsf{Nx}((O^{\mathcal{A}}_{\mathbf{r}})_i) = (I^{\mathcal{B}}_{\mathbf{l}})_i$ and $\mathsf{Nx}((O^{\mathcal{B}}_{\mathbf{l}})_i) = (I^{\mathcal{A}}_{\mathbf{r}})_i$, are the next states that are visited from any exits which are *not* in $O^{\mathcal{A} \fatsemi \mathcal{B}}$. Naturally, by substitution we obtain the equations for $\mathcal{C}(\mathsf{Ach}_{\mathcal{A}}) \fatsemi \mathcal{C}(\mathsf{Ach}_{\mathcal{B}})$. We observe that various parts of these equations are independent of the sequential composition. Thus, for these, Prop. 3 applies. Once we apply these, we obtain two times the *same* linear equation system with variables for $\mathrm{RPr}^{\mathcal{A} \fatsemi \mathcal{B}, \tau}(i, j)$ and $\mathrm{RPr}^{\mathcal{C}(\mathsf{Ach}_{\mathcal{A}}) \fatsemi \mathcal{C}(\mathsf{Ach}_{\mathcal{B}}), \tau}(i, j)$, respectively, which shows that the probabilities coincide. In [34, Appendix C], we derive the inclusions formally and show that it indeed preserves reachability probabilities. We also establish the inclusions in Prop. 4 analogously to Prop. 2.

5 Compositional Estimation of Error Bounds

As we discussed above, Algorithm 2 provides a way to obtain the Pareto curve *precisely*, for $\eta = 0$. However, setting the imprecision $\eta = 0$ is often infeasible. When setting an $\eta > 0$, we only have soundness of the bounds, but no guarantee on the tightness. A naive extension to Algorithm 2 would be to *a posteriori* determine the error and tighten the sound approximation if the error is not matched. This process terminates with the required error bound as there are only finitely many schedulers. In this section, we discuss the (im)possibility of a one-shot approach, where we would recursively compute sound approximations with an approximate error bound given *a priori*. To that end, we study how the error propagates through the composition. We show positive results by restricting the compositional structure: we maintain errors on string diagrams that are only constructed by the sum and the rightward sequential composition, as we show in Prop. 6 and Thm. 2. After showing negative results that explode the error on the (general) sequential composition in Ex. 3, we end with a positive note, showing that the final result can have an error that is (significantly) *smaller* than the individual errors in Ex. 4.

The L^∞-Error. For conciseness, this section uses the L^∞-error between sound approximations. Let (L, U) be a sound approximation. The L^∞*-error* is $\mathsf{E}_\infty(L, U) := \sup_{i \in I} \sup_{\mathbf{p} \in U_i} \inf_{\mathbf{p}' \in L_i} \|\mathbf{p} - \mathbf{p}'\|_\infty$, where $\|\_\|_\infty$ is the L^∞ norm. The L^∞-error $\mathsf{E}_\infty(L, U)$ and the error $\mathsf{E}(L, U)$ are equivalent in the sense that a sequence $\big(\mathsf{E}_\infty(L_n, U_n)\big)_{n \in \mathbb{N}}$ converges to 0 iff $\big(\mathsf{E}(L_n, U_n)\big)_{n \in \mathbb{N}}$ converges to 0.

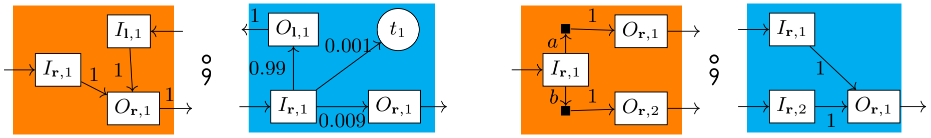
The sum. For the sum $\mathcal{A} \oplus \mathcal{B}$, we can easily obtain an error bound compositionally, since there are no interactions between $\mathcal{A}$ and $\mathcal{B}$.

Proposition 6. *Let $\mathcal{A}, \mathcal{B}$ be oMDPs, and $(L^{\mathcal{A}}, U^{\mathcal{A}}), (L^{\mathcal{B}}, U^{\mathcal{B}})$ be sound approximations. Then: $\mathsf{E}_\infty(L^{\mathcal{A} \oplus \mathcal{B}}, U^{\mathcal{A} \oplus \mathcal{B}}) \leq \max\big(\mathsf{E}_\infty(L^{\mathcal{A}}, U^{\mathcal{A}}), \mathsf{E}_\infty(L^{\mathcal{B}}, U^{\mathcal{B}})\big)$, where $L^{\mathcal{A} \oplus \mathcal{B}} := \mathsf{Ach}_{\mathcal{C}(L^{\mathcal{A}}) \oplus \mathcal{C}(L^{\mathcal{B}})}$, and $U^{\mathcal{A} \oplus \mathcal{B}} := \mathsf{Ach}_{\mathcal{C}(U^{\mathcal{A}}) \oplus \mathcal{C}(U^{\mathcal{B}})}$.*

Rightward composition. An open MDP $\mathcal{A}$ is *rightward* if $\mathrm{tp}(\mathcal{A}) = (m, 0) \to (l, 0)$.

Example 2. Let $\mathcal{A}, \mathcal{B}$ be rightward oMDPs with $\mathrm{tp}(\mathcal{A}) = (1, 0) \to (2, 0)$ and $\mathrm{tp}(\mathcal{B}) = (2, 0) \to (1, 0)$, and $(L^{\mathcal{A}}, U^{\mathcal{A}}), (L^{\mathcal{B}}, U^{\mathcal{B}})$ be sound approximations all generated by a singleton, such that we can write:

$$L^{\mathcal{A}}_1 := (0.3, 0.2), U^{\mathcal{A}}_1 := (0.4, 0.3), L^{\mathcal{B}}_1 := 0.7, L^{\mathcal{B}}_2 := 0.6, U^{\mathcal{B}}_1 := 0.75, U^{\mathcal{B}}_2 := 0.65.$$



(a) Exploding errors.

(b) Collapsing errors.

Fig. 7: Bidirectional sequential compositions $\mathcal{A} \fatsemi \mathcal{B}$.

Then the lower bound $L^{\mathcal{A} \fatsemi \mathcal{B}}$ on their composition consists of one point $0.3 \cdot 0.7 + 0.2 \cdot 0.6 = 0.33$, and the upper bound consists of one point $0.4 \cdot 0.75 + 0.3 \cdot 0.65 = 0.495$. These values can be easily calculated from the shortcut MDPs. While the error was 0.1 and 0.05 respectively on $\mathcal{A}$ and $\mathcal{B}$, the composition has an error of 0.165.

We can estimate sufficient error bounds for rightward $\mathcal{A}, \mathcal{B}$ in order to ensure a certain error bound for the sequential composition $\mathcal{A} \fatsemi \mathcal{B}$:

Theorem 2. *Let $\mathcal{A}, \mathcal{B}$ be rightward oMDPs, $(L^{\mathcal{A}}, U^{\mathcal{A}}), (L^{\mathcal{B}}, U^{\mathcal{B}})$ sound approximations, and $(L^{\mathcal{A} \fatsemi \mathcal{B}}, U_X), (L_X, U^{\mathcal{A} \fatsemi \mathcal{B}})$ be sound approximations of $\mathcal{C}(L^{\mathcal{A}}) \fatsemi \mathcal{C}(L^{\mathcal{B}})$ and $\mathcal{C}(U^{\mathcal{A}}) \fatsemi \mathcal{C}(U^{\mathcal{B}})$, respectively. Then $\mathsf{E}_\infty(L^{\mathcal{A} \fatsemi \mathcal{B}}, U^{\mathcal{A} \fatsemi \mathcal{B}})$ is bounded from above by*

$$|O^{\mathcal{A}}| \cdot \mathsf{E}_\infty(L^{\mathcal{A}}, U^{\mathcal{A}}) + \mathsf{E}_\infty(L^{\mathcal{B}}, U^{\mathcal{B}}) + \mathsf{E}_\infty(L^{\mathcal{A} \fatsemi \mathcal{B}}, U_X) + \mathsf{E}_\infty(L_X, U^{\mathcal{A} \fatsemi \mathcal{B}})$$

See [34, Appendix D] for the proof. Whereas the first two summands are inherent to the approximation of $\mathcal{A}$ and $\mathcal{B}$, the latter two terms originate from the approximations when computing a Pareto curve of the composed shortcut MDPs. Thm. 2 thus provides reasonably tight error bounds for the sequential composition on rightward oMDPs (only) when the number of exits $O^{\mathcal{A}}$ is small.

(General, bidirectional) sequential composition. In general, we cannot obtain tight error bounds for $\mathcal{A} \fatsemi \mathcal{B}$, even if their error bounds of $\mathcal{A}$ and $\mathcal{B}$ are small.

Example 3. For oMDPs $\mathcal{A}, \mathcal{B}$ in Fig. 7a, and $(L^{\mathcal{A}}, U^{\mathcal{A}}), (L^{\mathcal{B}}, U^{\mathcal{B}})$ be sound approximations with $L_1^{\mathcal{A}}, L_2^{\mathcal{A}}, U_1^{\mathcal{A}}, U_2^{\mathcal{A}}$ are all singleton sets with 1, and $L_1^{\mathcal{B}} := (0.001, 0.99)$, $U_1^{\mathcal{B}} := (0.009, 0.99)$. Then, $\mathsf{E}_\infty(L^{\mathcal{A}}, U^{\mathcal{A}}) = 0$, $\mathsf{E}_\infty(L^{\mathcal{B}}, U^{\mathcal{B}}) = 0.008$. In the composition $\mathcal{C}(L^{\mathcal{A}}) \fatsemi \mathcal{C}(L^{\mathcal{B}})$, we obtain a Pareto point on $\frac{0.001}{1-0.99} = 0.1$, while for $\mathcal{C}(U^{\mathcal{A}}) \fatsemi \mathcal{C}(U^{\mathcal{B}})$, we obtain $\frac{0.009}{1-0.99} = 0.9$. Then, $\mathsf{E}_\infty(L^{\mathcal{A} \fatsemi \mathcal{B}}, U^{\mathcal{A} \fatsemi \mathcal{B}}) = 0.8$.

The example demonstrates that with highly-likely loops, errors measured in the infinity norm may be amplified. This motivates looking at different distance measures, e.g., based on ratios [8]. These bounds may be tight for various shortcut MDPs, but require additional assumptions, such as inducing the same graph in the shortcut MDPs. We briefly discuss these bounds in [34, Appendix D].

Finally, the error may also disappear when composing oMDPs. For the motivating example in Fig. 3, a large error in room B is irrelevant if the best scheduler never visits this room. We provide a concrete example:

Example 4. Consider oMDPs $\mathcal{A}$, $\mathcal{B}$ in Fig. 7b. A lower bound L for $\mathcal{A}$ is a Pareto curve that only contains $(1, 0)$, i.e., the point induced by taking action a. The error for this lower bound is 1, as we may reach exit $O_{\mathbf{r},2}$ with probability 1. However, the error for $\mathcal{C}(\mathsf{Ach}_L) \fatsemi \mathcal{B}$ is 0 as we already recover an optimal scheduler.

6 Implementation and Experiments

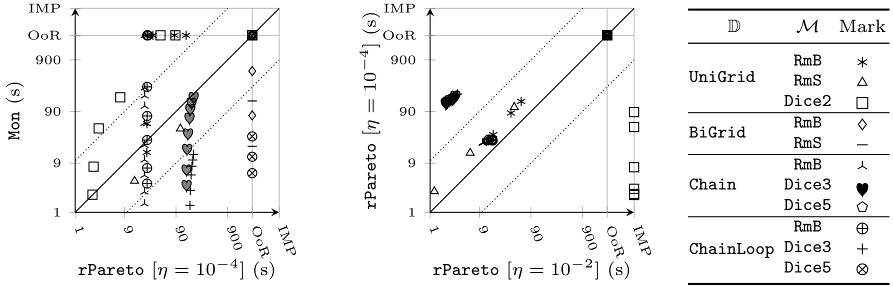
Implementation We have implemented a prototypical `C++` extension of Storm [17] that takes a string diagram in JSON-format as input. The syntax allows to name terms for simple reuse and oMDPs are defined as PRISM models with a list of entrances and exits defined via expressions over the variables. The tool supports caching of Pareto curves and shortcut MDPs, which is hugely beneficial if the same string diagram occurs in multiple contexts. We provide dedicated support for some syntactic sugar, most notably n-ary operations and the *trace* operator [33]. We have implemented two approaches. The *monolithic* (`Mon`) approach takes a string diagram $\mathbb{D}$ and inductively constructs the monolithic MDP $[\![\mathbb{D}]\!]$. The *recursive Pareto* computation (`rPareto`) follows the explanation in §4.

Setup We run the algorithms using a time out of 15 minutes and a memory limit of 16GB. All experiments run on a single core of an Intel i9-10980XE processor.

Benchmarks. Our benchmarks exhibit fundamental topological structures such as chains, which seems common structures for discretized (grid) worlds, and protocols that work in rounds or keep track of the number of rounds won/lost. Specifically, we create 8 benchmarks families with 50 different instances. We use three simple types of string diagrams: a two-dimensional grid of rooms with bidirectional doors (`BiGrid`), a grid of rooms that can only be passed in one direction (`UniGrid`), a big chain (`Chain`), and a chain with a loop (`ChainLoop`). Each string diagram is initialized at the leaves with 6 to 16 different simple open MDPs of similar shape that occur multiple times. The shapes are a small room `RmS`, a big room `RmB`, and a selection of biased dice `Dice`. Details are given in [34, Appendix A].

Baselines. The only comparable compositional algorithm in the literature executes scheduler enumeration [33]. We approximate this using `Prec` = `rPareto`$(\eta = 0)$, i.e., with precise Pareto curve computations. While pure scheduler enumeration produces less overhead, it requires analyzing all schedulers, whereas `Prec` only computes the Pareto-optimal DM schedulers. Scheduler enumeration over more than 10^{12} schedulers is completely infeasible. All benchmarks in our benchmark set have over 10^{32} schedulers. The monolithic algorithm is not optimised but uses mature data structures for sparse model construction, in particular for building the oMDPs. All algorithms use standard settings for MDP solving in Storm, in particular OVI with precision 10^{-4} and double arithmetic.

Results In Fig. 8a (log-log scale) we compare the run time of `rPareto` and `Mon`. Every point matches a benchmark, the point (x, y) indicates that `Mon` required x seconds, while `rPareto` took y seconds. A point above the diagonal means that `rPareto` was faster. A point above the dotted diagonal means that `rPareto` was 10x faster. A point is on the IMP line, if the error is bigger then $8 \cdot 10^{-3}$, i.e., significantly above 10^{-4}. Fig. 8b similarly compares `rPareto`$(\eta = 10^{-4})$ vs `rPareto`$(\eta = 10^{-2})$ to visualise the performance benefit when reducing η. Table 1 gives details: The columns give the string diagram and class of open MDPs, the



(a) Performance vs baseline (b) Performance for imprecision (c) Legend

Fig. 8: Performance (time in s, OoR=time-out or memory-out, IMP=imprecise)

Table 1: Benchmark details (time in s, MO=memory out, TO=time out)

$\mathbb{D}$	$\mathcal{M}$	$\lvert S_{[\![\mathbb{D}]\!]}\rvert$	$\#\mathcal{A}$	$\lvert S_{\mathcal{A}}\rvert$	Mon		Prec	rPareto [$\eta = 10^{-2}$]				rPareto [$\eta = 10^{-4}$]			
					t	t_m	t	t	t_m	E	p	t	t_m	E	p
UniGrid100	RmB	1.1e+08	14	148582	MO	MO	MO	17	2	7e-11	58	32	2	4e-12	155
UniGrid200	RmB	4.2e+08	14	148582	MO	MO	MO	37	2	2e-19	58	84	2	1e-20	155
UniGrid250	RmB	6.6e+08	14	148582	MO	MO	MO	58	2	5e-23	58	141	2	3e-24	155
UniGrid100	Dice	5.4e+07	14	74424	170	41	MO	5	<1	3e-01	56	7	<1	2e-04	94
UniGrid200	Dice	2.1e+08	14	65424	MO	MO	MO	25	<1	8e-01	56	45	<1	5e-04	94
UniGrid250	Dice	3.3e+08	14	74424	MO	MO	MO	46	<1	1e+00	56	88	<1	7e-04	94
Chain1000	RmB	1.6e+07	6	42463	181	16	MO	11	<1	2e-50	44	22	<1	6e-52	82
Chain2000	RmB	2.1e+07	6	42463	258	24	MO	11	<1	3e-68	44	22	<1	7e-70	82
BiGrid100	RmS	8.5e+05	16	1353	19	1	TO	TO	TO	TO	TO	TO	TO	TO	TO
BiGrid200	RmS	3.4e+06	16	1352	145	5	TO	TO	TO	TO	TO	TO	TO	TO	TO

number of monolithic states $|S_{[\![\mathbb{D}]\!]}|$, the unique $\#\mathcal{A}$ oMDPs in $\mathbb{D}$, and their state count $|S_{\mathcal{A}}|$. We then consider the Mon and rPareto with $\eta \in \{0, 10^{-2}, 10^{-4}\}$ respectively: We list the total time t, the model building time t_m, the error E, and the total number of vertices p in all sound approximations (L, U) combined. For Mon, the error is guaranteed to be below 10^{-4}. A complete table is in [34, Appendix A].

Discussion We discuss performance and error bounds.

Performance. Our Monolithic baseline is reasonably fast, can construct a monolithic MDP with millions of states in a matter of seconds and analyze it in minutes. However, the approach reaches memory limitations when handling $> 10^8$ states. In contrast, rPareto reduces the amount of time for model building, may speed up model checking by orders of magnitude, and is applicable even when the (monolithic) MDP has $> 10^8$ states. The 'vertical lines' in Fig. 8a are due to the effect of caching intermediate results. However, rPareto is not a silver bullet: In particular, handling open MDPs with more than 3 exits, in particular as in

BiGrid, is challenging to the multi-objective engine. This effect is amplified when using open MDPs that are challenging for value iteration, due to the iterative nature of Algorithm 1. Precisely computing Pareto curves is not competitive at all for any of these benchmarks due to the large number of schedulers.

Error bounds. In all experiments, when $\eta = 10^{-4}$, the approximations are sufficiently tight such that $E < 8 \cdot 10^{-3}$. The error actually mostly collapses: The lower bound typically includes a scheduler 'close' to the optimal scheduler, similar to Ex. 4. Consider Fig. 8b: Increasing η from 10^{-4} to 10^{-2} decreases the number of Pareto points roughly by a factor 2-3 and similarly the model checking time: However, the error sometimes explodes and for UniGrid250/Dice, the error is 1.

7 Related Work and Conclusion

Related Work Compositional verification methods for sequential MDPs [6,21, 26,33] have been discussed in §1. For hierarchical MCs [1], there are no schedulers. Hierarchical methods for reinforcement learning (RL) have been surveyed in [28]. Particularly interesting is [20], which applies compositional RL w.r.t. to some sequentially composed MDP, derived from the task specification.

Compositional probabilistic verification w.r.t. the parallel composition is investigated in [10, 23, 35]. Most relevant is an multi-objective optimization framework [24] that reasons compositionally in an assume-guarantee fashion.

The computation of multi-objective reward-bounded properties [14] generalizes topological value iteration. Their notion of episodes resembles rightward sequential composition. We support bidirectional composition where topological value iteration may not apply. For rightward MDPs, our approach caches the Pareto curves and thus supports exponentially many episodes in linear time.

Conclusion In this work, we employ multi-objective model checking of monolithic MDPs to obtain a novel compositional algorithm for MDPs compositionally defined by string diagrams. Future work includes support for reward properties and a complete temporal logic, in particular also including support for finite horizon properties. Furthermore, it is interesting to develop property-driven algorithms that compute Pareto-curves up to a context-dependent precision and to improve scalability for oMDPs with many exits. Finally, we think it is important to extract the compositional structure in form of string diagrams from popular formalisms like Prism programs.

References

1. Ábrahám, E., Jansen, N., Wimmer, R., Katoen, J., Becker, B.: DTMC model checking by SCC reduction. In: QEST. pp. 37–46. IEEE Computer Society (2010)
2. de Alfaro, L., Kwiatkowska, M.Z., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: TACAS. LNCS, vol. 1785, pp. 395–410. Springer (2000)
3. Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: ICALP. LNCS, vol. 1256, pp. 430–440. Springer (1997)
4. Baier, C., Hermanns, H., Katoen, J.: The 10, 000 facets of MDP model checking. In: Computing and Software Science, LNCS, vol. 10000, pp. 420–451. Springer (2019)
5. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
6. Barry, J.L., Kaelbling, L.P., Lozano-Pérez, T.: Deth*: Approximate hierarchical solution of large Markov decision processes. In: IJCAI. pp. 1928–1935. IJCAI/AAAI (2011)
7. Budde, C.E., Hartmanns, A., Klauck, M., Kretínský, J., Parker, D., Quatmann, T., Turrini, A., Zhang, Z.: On correctness, precision, and performance in quantitative verification - qcomp 2020 competition report. In: ISoLA (4). LNCS, vol. 12479, pp. 216–241. Springer (2020)
8. Chatterjee, K.: Robustness of structurally equivalent concurrent parity games. In: FOSSACS. LNCS, vol. 7213, pp. 270–285. Springer (2012). https://doi.org/10.1007/978-3-642-28729-9_18, https://doi.org/10.1007/978-3-642-28729-9_18
9. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of Markov decision processes. Log. Methods Comput. Sci. **4**(4) (2008). https://doi.org/10.2168/LMCS-4(4:8)2008, https://doi.org/10.2168/LMCS-4(4:8)2008
10. Feng, L., Han, T., Kwiatkowska, M.Z., Parker, D.: Learning-based compositional verification for synchronous probabilistic systems. In: ATVA. LNCS, vol. 6996, pp. 511–521. Springer (2011)
11. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: TACAS. LNCS, vol. 6605, pp. 112–127. Springer (2011). https://doi.org/10.1007/978-3-642-19835-9_11, https://doi.org/10.1007/978-3-642-19835-9_11
12. Forejt, V., Kwiatkowska, M.Z., Parker, D.: Pareto curves for probabilistic model checking. In: ATVA. LNCS, vol. 7561, pp. 317–332. Springer (2012), https://doi.org/10.1007/978-3-642-33386-6_25
13. Hartmanns, A., Hermanns, H.: The Modest toolset: An integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS. LNCS, vol. 8413, pp. 593–598. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_51, https://doi.org/10.1007/978-3-642-54862-8_51
14. Hartmanns, A., Junges, S., Katoen, J., Quatmann, T.: Multi-cost bounded tradeoff analysis in MDP. J. Autom. Reason. **64**(7), 1483–1522 (2020)
15. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: TACAS (1). LNCS, vol. 13993, pp. 469–488. Springer (2023)
16. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: CAV (2). LNCS, vol. 12225, pp. 488–511. Springer (2020)

17. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022). https://doi.org/10.1007/s10009-021-00633-z, https://doi.org/10.1007/s10009-021-00633-z
18. Hinze, R., Marsden, D.: Introducing String Diagrams: The Art of Category Theory. Cambridge University Press (2023). https://doi.org/10.1017/9781009317825
19. Holtzen, S., Junges, S., Vazquez-Chanlatte, M., Millstein, T.D., Seshia, S.A., den Broeck, G.V.: Model checking finite-horizon Markov chains with probabilistic inference. In: CAV (2). LNCS, vol. 12760, pp. 577–601. Springer (2021)
20. Jothimurugan, K., Bansal, S., Bastani, O., Alur, R.: Compositional reinforcement learning from logical specifications. In: NeurIPS. pp. 10026–10039 (2021)
21. Junges, S., Spaan, M.T.J.: Abstraction-refinement for hierarchical probabilistic models. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. LNCS, vol. 13371, pp. 102–123. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_6, https://doi.org/10.1007/978-3-031-13185-1_6
22. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. LNCS, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47, https://doi.org/10.1007/978-3-642-22110-1_47
23. Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: TACAS. LNCS, vol. 6015, pp. 23–37. Springer (2010)
24. Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Compositional probabilistic verification through multi-objective model checking. Inf. Comput. **232**, 38–65 (2013). https://doi.org/10.1016/j.ic.2013.10.001, https://doi.org/10.1016/j.ic.2013.10.001
25. Mac Lane, S.: Categories for the working mathematician, Graduate Texts in Mathematics, vol. 5. Springer-Verlag, New York, second edn. (1978)
26. Neary, C., Verginis, C.K., Cubuktepe, M., Topcu, U.: Verifiable and compositional reinforcement learning systems. In: ICAPS. pp. 615–623. AAAI Press (2022)
27. Papadimitriou, C.H., Yannakakis, M.: On the approximability of trade-offs and optimal access of web sources. In: FOCS. pp. 86–92. IEEE Computer Society (2000). https://doi.org/10.1109/SFCS.2000.892068, https://doi.org/10.1109/SFCS.2000.892068
28. Pateria, S., Subagdja, B., Tan, A., Quek, C.: Hierarchical reinforcement learning: A comprehensive survey. ACM Comput. Surv. **54**(5), 109:1–109:35 (2021)
29. Quatmann, T., Junges, S., Katoen, J.: Markov automata with multiple objectives. Formal Methods Syst. Des. **60**(1), 33–86 (2022). https://doi.org/10.1007/s10703-021-00364-6, https://doi.org/10.1007/s10703-021-00364-6
30. Quatmann, T., Katoen, J.: Multi-objective optimization of long-run average and total rewards. In: TACAS. LNCS, vol. 12651, pp. 230–249. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_13, https://doi.org/10.1007/978-3-030-72016-2_13
31. Selinger, P.: A survey of graphical languages for monoidal categories. New structures for physics pp. 289–355 (2011)
32. Watanabe, K., Eberhart, C., Asada, K., Hasuo, I.: A compositional approach to parity games. In: MFPS. EPTCS, vol. 351, pp. 278–295 (2021). https://doi.org/10.4204/EPTCS.351.17, https://doi.org/10.4204/EPTCS.351.17

33. Watanabe, K., Eberhart, C., Asada, K., Hasuo, I.: Compositional probabilistic model checking with string diagrams of MDPs. In: CAV. LNCS, vol. 13966, pp. 40–61. Springer (2023), https://doi.org/10.1007/978-3-031-37709-9_3
34. Watanabe, K., van der Vegt, M., Hasuo, I., Rot, J., Junges, S.: Pareto curves for compositionally model checking string diagrams of MDPs (2024), https://arxiv.org/abs/2401.08377, a longer version
35. Xu, D.N., Gößler, G., Girault, A.: Probabilistic contracts for component-based design. In: ATVA. LNCS, vol. 6252, pp. 325–340. Springer (2010)

Learning Explainable and Better Performing Representations of POMDP Strategies *

Alexander Bork[1], Debraj Chakraborty[2], Kush Grover[3], Jan Křetínský[2,3], and Stefanie Mohr[3](✉)

[1] RWTH Aachen University, Aachen, Germany
alexander.bork@cs.rwth-aachen.de
[2] Masaryk University, Brno, Czechia
{chakraborty,kretinsky,jan.kretinsky}@fi.muni.cz
[3] Technical University of Munich, Munich, Germany
{kush.grover,stefanie.mohr}@tum.de

Abstract. Strategies for partially observable Markov decision processes (POMDP) typically require memory. One way to represent this memory is via automata. We present a method to learn an automaton representation of a strategy using a modification of the L^*-algorithm. Compared to the tabular representation of a strategy, the resulting automaton is dramatically smaller and thus also more explainable. Moreover, in the learning process, our heuristics may even improve the strategy's performance. We compare our approach to an existing approach that synthesizes an automaton directly from the POMDP, thereby solving it. Our experiments show that our approach can lead to significant improvements in the size and quality of the resulting strategy representations.

1 Introduction

Partially Observable Markov Decision Processes (POMDPs) combine non-determinism, probability and partial observability. Consequently, they have gained popularity in various applications as a model of planning in an unsafe and only partially observable environment. Coming from the machine learning community [30], they also gained interest in the formal methods community [25,11,14,22]. They are a very powerful model, able to faithfully capture real-life scenarios where we cannot assume perfect knowledge, which is often the case. Unfortunately, the great power comes with the hardness of analysis. Typical objectives of interest such as reachability or total reward already result in undecidable problems [25]. Namely, the resolution of the non-determinism (a.k.a. synthesis of a *strategy*, policy, scheduler, or controller) cannot be done algorithmically while guaranteeing optimality w.r.t. the objective. Consequently, *heuristics*

* This research was funded in part by the German Research Foundation (DFG) project 427755713 GOPro, the MUNI Award in Science and Humanities (MUNI/I/1757/2021) of the Grant Agency of Masaryk University, the DFG GRK 2428 (ConVeY) and the DFG RTG 2236/2 (UnRAVeL).

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 299–319, 2024.
https://doi.org/10.1007/978-3-031-57249-4_15

to synthesize practically well-performing strategies became of significant interest. Let us name several aspects playing a key role in applicability of such synthesis procedures:

① *quality* of the synthesized strategies,
② *size* and *explainability* of the representation of the synthesized strategies,
③ *scalability* of the computation method.

Strategy Representation. While ① and ③ are of obvious importance, it is important to note the aspect ②. A strategy is a function mapping the current history (sequence of observations so far) to an action available in the current state. When written as a list of history-action pairs, it results in a large and incomprehensible table. In contrast, when equivalently written as a Mealy machine transducing the stream of observation to a stream of actions, its size may be dramatically lower (making it easier to implement and more efficient to execute) and its representation more explainable (making it easier to certify). Besides, better understandability allows for easier maintenance and modification. To put it in a contrast, explicit (table-like) or, e.g., neural-network representations of the function can hardly be hoped to be understandable by any human (even domain expert). Compact and understandable representations of strategies have recently gained attention, e.g., [12,27], also for POMDP [21,3], and even tool support [7] and [4], respectively. See [6] for detailed aspects of motivation for compact representations.

Current Approaches For POMDP, the state of the art is torn into two streams.

On the one hand, tools such as Storm [11] feature a classic *belief-based* analysis, which essentially blows up the state space, making it easier to analyze. Consequently, it is still reasonably scalable ③, but the size of the resulting strategy is even larger than that of the state space of the POMDP and is simply given as a table, i.e., not doing well w.r.t. the representation ②. Moreover, to achieve the scalability (and in fact even termination), the analysis has to be stopped at some places ("cut-offs"), resulting in poorer performance ①. On the other hand, the exhaustive bounded synthesis as in PAYNT [4] tries to synthesize a small Mealy machine representing a good strategy (while thus solving the POMDP) and if it fails, it tries again with an increased allowed size of the automaton. While this approach typically achieves better quality ① and, by principle, better size and explainability ②, it is extremely expensive and does not scale at all if the strategy requires a larger automaton ③. While symbiotic approaches are emerging [2], the best of both worlds has not been achieved yet.

Our Contribution We design a highly scalable postprocessing step, which improves the quality and the representation of the strategy. It is compatible with any framework producing any strategy representation, requiring only that we can query the strategy function (which action corresponds to a given observation sequence). In particular, Storm, which itself is scalable, can thus profit from

improving the quality and the representation of the produced strategies. Our procedure learns a compact representation of the given strategy as a Mealy machine using *automata-learning* techniques, in two different ways. First, through learning the complete strategy, we get its automaton representation, which is *fully equivalent* and thus achieving also the same value. Second, we provide *heuristics* learning small modifications of the strategy. Indeed, for some inputs (observation sequences), we ignore what the strategy suggests, in particular when the strategy is not defined, but also when it explicitly states that it is unsure about its choice (such as at the cut-off points, where the sequences become too long and the strategy was not optimised well at these later points). Whenever we ignore the strategy, we try to devise with a possibly better solution. For instance, we can adopt the decision that the currently learnt automaton suggests, or we can reflect other decisions in similar situations. This way we produce a *simpler strategy* (thus also comparatively smaller), which can, in principle, *fix the suboptimal decisions* of the strategy stemming from the limitations of the original analysis (such as bounds on the exploration) or any other irregularities. Of course, this only works well if the true optimal strategy is "sensible", i.e., has inner structure allowing for a simple automaton representation. For practical, hence sensible, problems, this is typically the case.
Summary of our contribution:

- We provide a method to take any POMDP strategy and transform it into an equivalent or similar (upon choice) automaton, yielding **small** size and potential for **explainability**.
- Thereby we often improve the **quality** of the strategy.
- The experiments confirm the improvements and frequent proximity to best known values (typically of PAYNT) on the simpler benchmarks.
- The experiments indicate great **scalability** even on harder benchmarks where the comparison tool times out. The auspicious comparison on simpler benchmarks warrants the trust in good absolute quality and size on the harder ones.

Related Work Methods to solve planning problems on POMDPs have been studied extensively in the literature [34,18,32]. Many state-of-the-art solvers use point-based methods like *PBVI* [29], *Perseus* [35] and *SARSOP* [23] to treat bounded and unbounded discounted properties. For these methods, strategies are typically represented using so called α-vectors. Apart from a significant overhead in the analysis, they completely lack of explainability. Notably, while the *SARSOP* implementation provides an export of its computed strategies in an automaton format, we have not been able to find an explanation of how it is generated.

Methods based on the (partial) exploration and solving of the belief MDP underlying the POMDP [28,10,11] have been implemented in the probabilistic model checkers Storm [20] and Prism [24]. The focus of these methods is optimizing infinite-horizon objectives *without* discounting. Recent work [2] describes how strategies are extracted from the results of these belief exploration methods. The resulting strategy representation, however, is rather large and contains potentially redundant information.

Orthogonal to the methods above, there are approaches that *directly* synthesize strategies from a space of candidates [16,26]. The synthesized strategy is then applied to the POMDP to yield a Markov chain. Analyzing this Markov chain yields the objective value achieved by the strategy. Methods used for searching policies include using inductive synthesis [3], gradient decent [19] or convex optimization [1,21,15]. [2] describes an integration of a belief exploration approach [11] with inductive synthesis [3].

Our approach is orthogonal to the solution methods in that it uses an *existing* strategy representation and learns a new, potentially more concise finite-state controller representation. Furthermore, our modifications of learned strategy representations shares similarities with approaches for strategy improvement [36,13,33].

2 Preliminaries

For a countable set S, we denote its power set by 2^S. A *(discrete) probability distribution* on a countable set S is a function $d : S \to [0,1]$ such that $\sum_{s \in S} d(S) = 1$. We denote the set of all probability distributions on the set S as $\mathsf{Dist}(S)$. For $d \in \mathsf{Dist}(S)$, the *support* of d is $\mathsf{supp}(d) = \{s \in S \mid d(s) > 0\}$. We use the Iverson bracket notation where $[x] = 1$ if the expression x is true and 0 otherwise. For two sets S, T, we define the set of concatenations of S with T as $S \cdot T = \{s \cdot t \mid s \in S, t \in T\}$. We analogously define the set of n-times concatenation of S with itself as S^n for $n \geq 1$ and $S^0 = \{\epsilon\}$ is the set containing the empty string. We denote by $S^* = \bigcup_{i=0}^{\infty} S^n$ the set of all *finite strings* over S and by $S^+ = \bigcup_{i=1}^{\infty} S^n$ the set of all *non-empty* finite strings over S. For a finite string $w = w_1 w_2 \dots w_n$, the string $w[0,i]$ with $w[0,0] = \epsilon$ and $w[0,i] = w_1 \dots w_i$ for $0 < i \leq n$ is a *prefix* of w. The string $w[i,n] = w_i \dots w_n$ with $0 < i \leq n$ is a *suffix* of w. A set $W \subseteq S^*$ is *prefix-closed* if for all $w \in S^*$, $w = w_1 \dots w_n \in W$ implies $w[0,i] \in W$ for all $0 \leq i \leq n$. A set $W' \subseteq S^*$ is *suffix-closed* if $\epsilon \notin W$ and for all $w \in S^*$, $w = w_1 \dots w_n \in W$ implies $w[i,n] \in W$ for all $0 < i \leq n$.

Definition 1 (MDP). *A* Markov decision process *(MDP) is a tuple $\mathcal{M} = (S, A, P, s_0)$ where S is a countable set of states, A is a finite set of actions, $P : S \times A \rightharpoonup \mathsf{Dist}(S)$ is a partial transition function, and $s_0 \in S$ is the initial state.*

For an MDP $\mathcal{M} = (S, A, P, s_0)$, $s \in S$ and $a \in A$, let $\mathsf{Post}^{\mathcal{M}}(s,a) = \{s' \mid P(s,a,s') > 0\}$ be the set of successor states of s in $\mathcal{M}$ that can be reached by taking the action a. We also define the set of *enabled actions* in $s \in S$ by $A(s) = \{a \in A \mid P(s,a) \neq \bot\}$. A *Markov chain* (MC) is an MDP with $|A(s)| = 1$ for all $s \in S$. For an MDP $\mathcal{M}$, a *finite path* $\rho = s_0 a_0 s_1 \dots s_i$ of length $i \geq 0$ is a sequence of states and actions such that for all $t \in [0, i-1]$, $a_t \in A(s_t)$ and $s_{t+1} \in \mathsf{Post}^{\mathcal{M}}(s_t, a_t)$. Similarly, an infinite path is an infinite sequence $\rho =$

$s_0 a_0 s_1 a_1 s_2 \ldots$ such that for all $t \in \mathbb{N}$, $a_t \in A(s_t)$ and $s_{t+1} \in \mathsf{Post}^{\mathcal{M}}(s_t, a_t)$. For an MDP $\mathcal{M}$, we denote the set of all finite paths by $\mathsf{FPaths}_{\mathcal{M}}$, and of all infinite paths by $\mathsf{IPaths}_{\mathcal{M}}$.

Definition 2 (POMDP). *A* partially observable MDP (POMDP) *is a tuple* $\mathcal{P} = (\mathcal{M}, Z, \mathcal{O})$ *where* $\mathcal{M} = (S, A, P, s_0)$ *is the underlying MDP with finite number of states, Z is a finite set of observations, and $\mathcal{O} : S \to Z$ is an observation function that maps each state to an observation.*

For POMDPs, we require that states with the same observation have the same set of enabled actions, i.e., $\mathcal{O}(s) = \mathcal{O}(s')$ implies $A(s) = A(s')$ for all $s, s' \in S$. This way, we can lift the notion of enabled actions to an observation $z \in Z$ by setting $A(z) = A(s)$ for some state $s \in S$ with $\mathcal{O}(s) = z$. The notion of observation $\mathcal{O}$ for states can be lifted to paths: for a path $\rho = s_0 a_0 s_1 a_1 \ldots$, we define $\mathcal{O}(\rho) = \mathcal{O}(s_0) a_0 \mathcal{O}(s_1) a_1 \ldots$. Two paths ρ_1 and ρ_2 are called *observation-equivalent* if $\mathcal{O}(\rho_1) = \mathcal{O}(\rho_2)$. We call an element $\bar{o} \in Z^*$ an *observation sequence* and denote the observation sequence of a path $\rho = s_0 a_0 s_1 \ldots$ by $\overline{\mathcal{O}}(\rho) = \mathcal{O}(s_0)\mathcal{O}(s_1) \ldots$.

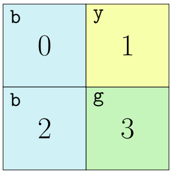


Fig. 1: Running example: POMDP

Example 1. Consider the POMDP graphically depicted in Fig. 1, modeling a basic robot planning task. A robot is dropped uniformly at random in one of four grid cells. Its goal is to reach cell 3. The robot's sensors cannot to distinguish cells 0 and 2, while cells 1 and 3 provide unique information. For the POMDP model, we use states 0, 1, 2, and 3 to indicate the robot's position. We mimic the random initialization by introducing a unique initial state s_0 with a unique observation `i` (init). s_0 has a single action that reaches any of the other four states with equal probability 0.25. Thus, the state space of the POMDP is $S = \{s_0, 0, 1, 2, 3\}$. To represent the observations of the robot, we use three observations `b`, `y` and `g`, so $Z = \{\mathtt{i}, \mathtt{b}, \mathtt{y}, \mathtt{g}\}$. States 0 and 2 have the same observation, while states 1 and 3 are uniquely identifiable, formally $\mathcal{O} = \{(s_0 \to \mathtt{i}), (0 \to \mathtt{b}), (1 \to \mathtt{y}), (2 \to \mathtt{b}), (3 \to \mathtt{g})\}$. The goal is for the robot to reach state 3. In each state, it can choose to move up, down, left, or right, $A = \{s, u, d, l, r\}$. In each step, executing the chosen action may fail with a probability of $p = 0.5$, causing the robot to remain in its current cell without changing states.

Definition 3 (Strategy). *A strategy for an MDP $\mathcal{M}$ is a function $\pi : \mathsf{FPaths}_{\mathcal{M}} \to \mathsf{Dist}(A)$ such that for all paths $\rho \in \mathsf{FPaths}_{\mathcal{M}}$, $\mathsf{supp}(\pi(\rho)) \subseteq A(\mathsf{last}(\rho))$.*

A strategy π is *deterministic* if $|\mathsf{supp}(\pi(\rho))| = 1$ for all paths $\rho \in \mathsf{FPaths}_{\mathcal{M}}$. Otherwise, it is *randomized*. A strategy π is called *memoryless* if it depends only on *last*(ρ) *i.e.* for any two paths $\rho_1, \rho_2 \in \mathsf{FPaths}_{\mathcal{M}}$, if *last*$(\rho_1)$ = *last*(ρ_2) then $\pi(\rho_1) = \pi(\rho_2)$. As general strategies have access to *full* state information, they are unsuitable for partially observable domains. Therefore, POMDPs require a notion of strategies based only on observations. For a POMDP $\mathcal{P}$, we call a strategy *observation-based* if for any $\rho_1, \rho_2 \in \mathsf{FPaths}_{\mathcal{M}}$, $\mathcal{O}(\rho_1) = \mathcal{O}(\rho_2)$ implies $\pi(\rho_1) = \pi(\rho_2)$, i.e. the strategy has same output on observation-equivalent paths.

We are interested in representing observation-based strategies approximating optimal objective values for infinite horizon objectives without discounting, also called *indefinite-horizon objectives*, i.e., maximum/minimum reachability probabilities and expected total reward objectives. We emphasize that our general learning framework also generalizes straightforwardly to strategies for different objectives. In contrast to fully observable MDPs, deciding if a given strategy is optimal for an indefinite-horizon objective on a POMDP is generally undecidable [25]. In fact, optimal behavior requires access to the *full* history of observations, necessitating an arbitrary amount of memory. As such, our goal is to learn a small representation of a strategy using only a finite amount of memory that approximates optimal values as well as possible.

We represent these strategies as *finite-state controllers (FSCs)* – automata that compactly encode strategies with access to memory and randomization in a POMDP.

Definition 4 (Finite-State Controller). *A* finite-state controller *(FSC) is a tuple $\mathcal{F} = (N, \gamma, \delta, n_0)$ where N is a finite set of nodes, $\gamma : N \times Z \rightarrow \mathsf{Dist}(A)$ is an action mapping, $\delta : N \times Z \rightarrow N$ is the transition function, and n_0 is the initial node.*

We denote by $\pi_{\mathcal{F}}$ the strategy represented by the FSC $\mathcal{F}$ and use $\mathfrak{F}$ for the set of all FSCs for a POMDP $\mathcal{P}$. Given an FSC $\mathcal{F} = (N, \gamma, \delta, n_0)$ that is currently in node n, and a POMDP $\mathcal{P}$ with underlying MDP $\mathcal{M} = (S, A, P, s_0)$, in state s, the action to play by an agent following $\mathcal{F}$ is chosen randomly from the distribution $\gamma(n, \mathcal{O}(s))$. $\mathcal{F}$ then updates its current node to $n' = \delta(n, z)$. The state of the POMDP is updated according to P. As such, an FSC induces a Markov chain $\mathcal{M}_{\mathcal{F}} = (S \times N, \{\alpha\}, P^{\mathcal{F}}, (s_0, n_0))$ where $P^{\mathcal{F}}((s, n), \alpha, (s', n'))$ is $[\delta(n, \mathcal{O}(s)) = n'] \cdot \sum_{a \in A(s)} \gamma(n, \mathcal{O}(s))(a) \cdot P(s, a, s')$.

An FSC can be interpreted as a Mealy machine: nodes correspond directly to states of the Mealy machine, which takes observations as input. The set of output symbols is the set of all distributions over actions occurring in the FSC.

3 Learning a Finite-State Controller

We present a framework for learning a concise finite-state controller representation from a given strategy for a POMDP. Our approach mimics an extension of the L* automaton learning approach [5] for learning Mealy machines [31]. The main difference in our approach is that we have a sparse learning space: not all observations of a POMDP are possible to reach from all states. Thus, there are many observation sequences that can never occur in the POMDP. To mark situations where this occurs, i.e. where a learned FSC has complete freedom to decide what to do, we introduce a "don't-care" symbol †.

Furthermore, for some policy computation methods, the strategy we receive as input may be incomplete. Although some observation sequence can appear

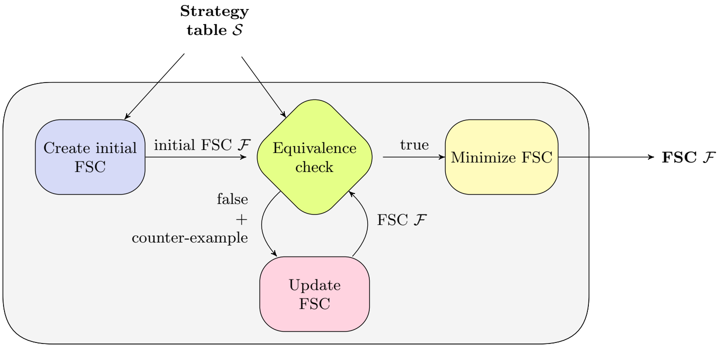


Fig. 2: Depiction of the FSC learning framework

in the POMDP, the strategy does not specify what to do when it occurs. This can for example be caused by reaching the depth limit in an exploration based approach. We use a "don't-know" symbol χ to mark such cases. While the non-occurring sequences do not directly influence the learning process, they cannot be ignored completely. These χ need to be replaced by actual actions using some heuristics for the final FSC to yield a complete strategy (see Section 3.4).

An overview of the learning process is depicted in Fig. 2. We expect as input a (partially defined) strategy in the form of a table that maps observation sequences in the POMDP to a distribution over actions.

Definition 5 (Strategy Table). *A* strategy table $\mathcal{S}$ *for a POMDP* $\mathcal{P}$ *is a relation* $\mathcal{S} \subseteq Z^* \times (\mathsf{Dist}(A) \cup \{\chi\})$. *A* row *of* $\mathcal{S}$ *is an element* $(\bar{o}, d) \in \mathcal{S}$.

For $(\bar{o}, d) \in \mathcal{S}$, if $\mathsf{supp}(d)$ contains only a single action a, we write it as $(\bar{o}, a)$. We say a strategy table $\mathcal{S}$ is *consistent* if and only if for $\bar{o} \in Z^*$, $(\bar{o}, d_1) \in \mathcal{S}$ and $(\bar{o}, d_2) \in \mathcal{S}$ implies $d_1 = d_2$, i.e. each observation sequence has at most one unique output. A consistent strategy table $\mathcal{S}$ (partially) defines an observation-based strategy $\pi_{\mathcal{S}}$ with $\pi_{\mathcal{S}}(\rho) = d$ if and only if $(\overline{\mathcal{O}}(\rho), d) \in \mathcal{S}$ and $d \neq \chi$. For consistent strategy tables, the FSC resulting from our approach correctly represents the partially defined strategy.

Example 2. Table 1 depicts a strategy table for the POMDP described in Example 1. The table does not specify what to do in state 3 as at that point, the robot has already achieved its target. The action chosen at that point is irrelevant. Intuitively, the strategy table describes that the robot should go right as long as it sees `b`, and goes down once it sees `y`. The FSC in Figure 3 fully captures the behaviour described by the strategy table and thus accurately represents it.

In our framework, the input strategy table is used to build an initial FSC which is then compared to the input. If the initial FSC is already equivalent to the given strategy table, we are done and we output the FSC. Otherwise, we get a counterexample and use it to update the FSC. This process of checking for

Observation sequence	Action
i	s
i y	d
i b	r

Table 1: Example strategy table for the POMDP in Example 1. It only contains observation sequences of length at most 2.

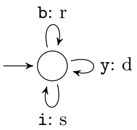


Fig. 3: FSC representing the strategy table of Table 1.

equivalence and updating the FSC is repeated until the FSC is equivalent to the table.

In the sequel, we first explain how our learning approach works on general input of the form described above. Then we show how the learning approach is integrated with an existing POMDP solution method by means of the belief exploration framework from [11]. Lastly, we introduce heuristics for improvement of the learned policies when the information in the table is incomplete.

3.1 Automaton Learning

The regular L* approach is used to learn a DFA for a regular language. It is intuitively described as: a *teacher* has information about an automaton and a *student* wants to learn that automaton. The student asks the teacher whether specific words are part of the language (*membership query*). At some point, the student proposes a solution candidate (in case of L*, a DFA) and asks the teacher whether it is correct, i.e. whether the proposed automaton accepts the language (*equivalence query*). Instead of the membership query of standard L*, the extension to Mealy machines [31] uses an *output query*, since we are not interested in the membership of a word in a language but rather the output of the Mealy machine corresponding to a specific word. As such, our learning approach needs access to an output query, specifying the output of the strategy table for a given observation sequence, and an equivalence query, checking whether an FSC accurately represents the strategy table. We formally define the two types of queries.

Definition 6 (Output Query (OQ)). *The* output query *for a strategy table $\mathcal{S}$ is the function $OQ_{\mathcal{S}} : Z^* \to \mathsf{Dist}(A) \cup \{\chi, \dagger\}$ with $OQ_{\mathcal{S}}(\bar{o}) = d$ if $(\bar{o}, d) \in \mathcal{S}$ and $OQ_{\mathcal{S}}(\bar{o}) = \dagger$ otherwise.*

Definition 7 (Equivalence Query (EQ)). *The* equivalence query *for a strategy table $\mathcal{S}$ is a function $EQ_{\mathcal{S}} : \mathfrak{F} \to Z^*$ defined as follows: $EQ_{\mathcal{S}}(\mathcal{F}) = \epsilon$ if for all $(\bar{o}, d) \in \mathcal{S}$ and for all ρ with $\overline{\mathcal{O}}(\rho) = \bar{o}$, $\pi_{\mathcal{F}}(\rho) = d$. Otherwise, $EQ_{\mathcal{S}}(\mathcal{F}) = c$ where $c \in \{\bar{o} \mid (\bar{o}, d) \in \mathcal{S}, \exists \rho \in \mathsf{FPaths}_{\mathcal{M}}(\mathcal{P}) : \overline{\mathcal{O}}(\rho) = \bar{o} \wedge \pi_{\mathcal{F}}(\rho) \neq d\}$ is a* counterexample *where $\mathcal{S}$ and $\mathcal{F}$ have different output.*

The output query (OQ) takes an observation sequence $\bar{o}$, and outputs the distribution (or the χ symbol) suggested by the strategy table. If the given observation sequence is not present in the strategy table, it returns the † symbol, i.e., a ”don’t care”-symbol. The equivalence query (EQ) takes a hypothesis FSC $\mathcal{F}_{hyp}$ and asks whether it accurately represents $\mathcal{S}$. In case it does not, an observation sequence where $\mathcal{F}_{hyp}$ and $\mathcal{S}$ differ is generated as a counterexample.

Using the definitions of these two queries, we formalise our problem statement as follows:

Problem Statement: Given a POMDP $\mathcal{P}$, a strategy table $\mathcal{S}$, an output query $OQ_{\mathcal{S}}$ and an equivalence query $EQ_{\mathcal{S}}$, compute a small FSC $\mathcal{F}$ such that $EQ_{\mathcal{S}}(\mathcal{F}) = \epsilon$.

Learning Table We aim at solving the problem using a learning framework similar to L^*. We learn an FSC by creating a *learning table* which keeps track of the observation sequences and the outputs the learner assumes they should yield in the strategy. Formally, it is defined as follows:

Definition 8 (Learning Table). *A* learning table *for POMDP $\mathcal{P}$ is a tuple $\mathcal{T} = (R, C, \mathcal{E})$ where $R \subset Z^*$ is a prefix-closed finite set of finite strings over the observations representing the* upper row indices, *the set $R \cdot Z$ are the* lower rows indices *and $C \subset Z^+$ is a suffix-closed finite set of non-empty finite strings over Z – the* columns. *$\mathcal{E} : (R \cup R \cdot Z) \times C \to \mathsf{Dist}(A) \cup \{\chi, \dagger\}$ is a mapping that represents the* entries *of the table.*

	i	b	y
ϵ	s	†	†
i	†	r	d
b	†	†	†
y	†	†	†

Table 2: Running example - initial table

Intuitively speaking, the table is divided into *upper* and *lower* rows. Initially, the columns of the learning table are the observations in the POMDP. Additional columns may be added in the learning process to further refine the behavior of the learned FSC. Upper rows effectively result in nodes of the learned FSC, while lower rows specify destinations of the transitions. For a row in the upper rows, each entry represents the output of the FSC corresponding to their respective observation (column). For an upper row, if a column is labelled only with an observation, the corresponding entry represents the output of the FSC on that observation. As an example, Table 2 contains the initial learning table for our running example. We do not include observation g for the target state as we are not interested in the behavior of the strategy after the target has been reached.

We say that two rows $r_1, r_2 \in R \cup R \cdot Z$ are *equivalent* ($r_1 \equiv r_2$) if they fully agree on their entries, i.e., $r_1 \equiv r_2$ if and only if $\mathcal{E}(r_1, c) = \mathcal{E}(r_2, c)$ for all $c \in C$. The equivalence class of a row $r \in R \cup R \cdot Z$ is $[r] = \{r' \mid r \equiv r'\}$.

From Learning Table to FSC To transform a learning table into an FSC, the table needs to be of a specific form. In particular, it needs to be *closed* and *consistent*. A learning table is *closed* if for each lower row $l \in R \cdot Z$, there is an upper row $u \in R$ such that $l \equiv u$.

	x	y
ϵ	a	b
x	a	b
y	a	b

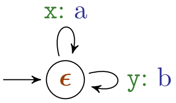


Fig. 4: Transformation of a learning table to an FSC.

A learning table is *consistent* if for each $r_1, r_2 \in R$ such that $r_1 \equiv r_2$, we have $r_1 \cdot e \equiv r_2 \cdot e$ for all $e \in Z$. Closure of a learning table guarantees that each transition – defined in the FSC by a lower row – leads to a valid node, i.e. the node corresponding to the equivalent upper row. Consistency, on the other hand, guarantees that the table unambiguously defines the output of a node in the FSC given an observation.

Using the notions of closure and consistency, we can define the transformation of a learning table into the learned FSC :

Definition 9 (Learned FSC). *Given a closed and consistent learning table $\mathcal{T} = (R, C, \mathcal{E})$, we obtain a* learned FSC $\mathcal{F}_\mathcal{T} = (N_\mathcal{T}, \gamma_\mathcal{T}, \delta_\mathcal{T}, n_{0,\mathcal{T}})$ *where:* $N_\mathcal{T} = \{[r] \mid r \in R\}$*, i.e., the nodes are the upper rows of the table;* $\gamma_\mathcal{T}([r], o) = \mathcal{E}(r, o)$ *for all* $o \in Z$*, i.e. the output of a transition is defined by its entry in the table;* $\delta_\mathcal{T}([r], o) = [r \cdot o]$ *for all* $r \in R, o \in Z$*, i.e., the destination of a transition from node* $[r]$ *with observation* o *is the node corresponding to the upper row equivalent to the lower row* $r \cdot o$*;* $n_{0,\mathcal{T}} = [\epsilon]$*, i.e., the initial state is* $[\epsilon]$*.*

Example 3. We demonstrate how to transform a table to an FSC in Fig. 4. The upper rows become states, the lower rows show the transitions. In this example, on both the observations x,y, we stay in the state and play action a and b, respectively.

3.2 Algorithm

We present our algorithm for learning an FSC from a strategy table. We have already seen the abstract view of the approach in Fig. 2. Algorithm 1 contains the pseudo-code for our learning algorithm. It consists of four main parts, also pictured in Fig. 2: initialization, equivalence check, update of the FSC, minimization.

First, we initialise the learning table. The columns are initially filled with all available observations Z, i.e. we set $C \leftarrow Z$. We start with a single upper row ϵ, representing the empty observation sequence. In the lower rows, we add the observation sequences of length 1. The entries of the table are then filled using output queries. For example, consider the strategy table in Table 1. The learning table after initialisation is shown in Table 2. The strategy table only contains observation sequences starting with `i`. Thus, for any sequence starting with `b` or `y`, all entries are †.

After initialising the table, we check whether it is closed. If the table is not closed, all rows in the lower part of the table that do not occur in the upper part are moved to the upper part. Formally, we set $R \leftarrow R \cup \{l\}$ for all $l \in R \cdot Z$ with $l \not\equiv u$ for all $u \in R$. In our example, this means that we move the rows (`i` | † r d) and (`b` | † † †) to the upper part of the table.

Algorithm 1 Learning an FSC

```
Input: POMDP $\mathcal{P}$, strategy table $\mathcal{S}$
$R \leftarrow \{\epsilon\}, C \leftarrow Z$
for all $r \in R \cup R \cdot Z, e \in C$ do
    $\mathcal{E}(r, e) \leftarrow$OutputQuery$(r \cdot e)$
end for
MakeClosedAndConsistent$(R, C, \mathcal{E})$
$c \leftarrow$EquivalenceQuery$(\mathcal{S}, \mathcal{F}_{(R,C,\mathcal{E})})$
while $c \neq \epsilon$ do
    $C \leftarrow C \cup$ set of all prefixes of $c$
    for all $r \in R \cup R \cdot Z, e \in C$ do
        $\mathcal{E}(r, e) \leftarrow$OutputQuery$(r \cdot e)$
    end for
    MakeClosedAndConsistent$(R, C, \mathcal{E})$
    $c \leftarrow$EquivalenceQuery$(\mathcal{S}, \mathcal{F}_{(R,C,\mathcal{E})})$
end while
$\mathcal{T} \leftarrow (R, C, \mathcal{E})$,$\mathcal{T} \leftarrow$Minimize$(\mathcal{T})$
Output: FSC $\mathcal{F}_{\mathcal{T}}$ generated from $\mathcal{S}$
```

Once the table is closed (and naturally consistent), we check for each row in the given strategy table $\mathcal{S}$ whether it coincides with the action provided for this observation sequence by our hypothesis FSC $\mathcal{F}_{hyp}$. This is done formally using the equivalence query, i.e. we check if $EQ_{\mathcal{S}}(\mathcal{F}_{hyp}) = \epsilon$. If our hypothesis is not correct, we get a counterexample $c \in Z^+$ where the output of $\mathcal{S}$ and $\mathcal{F}_{hyp}$ differ. We add all non-empty prefixes of c to C and fill the table. We repeat this until $\mathcal{F}_{hyp}$ is equivalent to the strategy table $\mathcal{S}$.

After the equivalence has been established, we use the "don't-care" entries † to further minimise the FSC. These entries only appear for observation sequences that do not occur in the strategy table. Thus, changing them to any action does not change the FSC's behaviour with respect to the strategy table. We use this fact to merge nodes of the FSC to obtain a smaller one that still captures the behaviour of the strategy table. It is not trivial to already exploit "don't care" entries during the learning phase. Two upper rows that are compatible in terms of the outputs they suggest, i.e. they either agree or have a † where the other suggests an output, might be split when a new counterexample is added. As such, we postpone minimisation of the FSC until the learning is finished.

3.3 Proof of Concept: Belief Exploration

For integrating our learning approach with an existing POMDP solution framework, we need to consider how the strategy table is constructed. Assume that the solution method outputs *some* representation of a strategy. For strategies that are equivalent to some FSC, one possibility is to pre-compute the strategy table. However, it is not clear how to determine the length of observation sequences that need to be considered. A more reasonable view is considering the strategy representation as a *symbolic* representation of the strategy table as long as it permits computable output and equivalence queries.

We demonstrate how this works by considering the belief exploration framework of [11]. The idea of belief exploration is to explore (a fragment of) the *belief MDP* corresponding to the POMDP. Then, model checking techniques are used on this finite MDP to optimise objectives and find a strategy. States of the belief MDP are *beliefs* – distributions over states of the POMDP that describe the likelihood of being in a state given the observation history. The strategy output of the belief exploration is a memoryless deterministic strategy π_{bel} that maps each belief to the optimal action. It is well-known that there is a direct correspondence between strategies on the belief MDP and its POMDP [34]. A decision in a belief corresponds to a decision in the POMDP for all observation sequences that lead to the belief in the belief MDP. Thus, π_{bel} can also be interpreted as a strategy for the POMDP that we want to learn using our approach.

First, assume that the belief MDP is finite. Defining the computation of the output query is conceptually straightforward. During each output query, we search for the belief b that corresponds to the observation sequence in the belief MDP. If we find it, the output is $\pi_{bel}(b)$, otherwise the query outputs "don't care" ($\dagger$). For the equivalence query, we consider one representative observation sequence for each belief b. We compare whether $\pi_{bel}(b)$ coincides with the output of the hypothesis FSC on the corresponding observation sequence. If not, this sequence is a counterexample. To deal with infinite belief MDPs, [11] employs a partial exploration of the reachable belief space of the POMDP. At the points where the exploration has been stopped (*cut-off states*), they use approximations based on pre-computed, small strategies on the POMDP to yield a finite abstraction of the belief MDP. The strategy π_{bel} computed on this abstraction, however, does not output valid actions for the POMDP in the cut-off states. We modify the output query described above and introduce a set of χ symbols, i.e., $\chi_0, ...\chi_n$. On observation sequences of cut-off states, the output query returns "don't-know" corresponding to that cutoff, i.e., χ_i for "cut-off" strategy i. This allows us to later integrate the strategies used for approximation in our learned FSC or even substitute these strategies by different ones.

3.4 Improving Learned FSCs for Incomplete Information

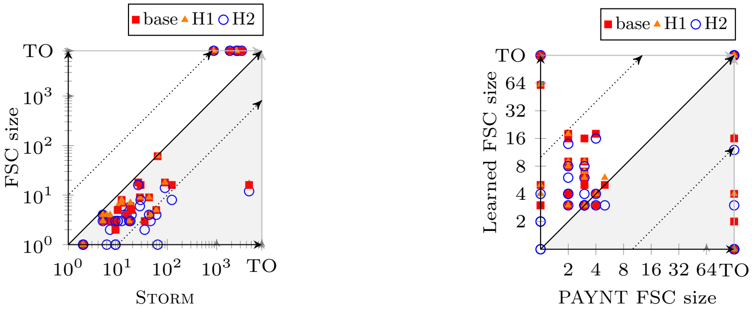
FSCs learned using the learning approach described in Section 3.2 may still contain transitions with output "don't-know" (χ). To make the FSC applicable to a POMDP, these outputs need to be replaced by distributions over actions of the POMDP. For this purpose, we suggest two heuristics. They are designed to be general, i.e. they do not consider any information that the underlying POMDP solution method provides. Furthermore, they use the idea that already learned behavior might offer a basis for generalization. As a result, the information already present in the FSC is used to replace the "don't-know" outputs. We note that additional heuristics can take for example the structure of the POMDP or information available in the POMDP solution method used to generate the strategy table into account. For illustrating the heuristics, we assume that all output distributions are Dirac. We denote the number of transitions in the FSC with observation o with output not equal to $\dagger_i$ or χ_i for some i by $\#(o)$ and the number of transitions with output action a for o by $\#(o, a)$.

- *Heuristic 1 – Distribution:* Intuitively, this heuristic replaces "don't know" by a distribution over all actions that the FSC already chooses for an observation. The resulting FSC therefore represents a randomized strategy, i.e. the strategy may probabilistically choose between actions. This happens only in nodes of the FSC where "don't know" occurs. Furthermore, this does not mean that the FSC itself is randomized; its structure remains deterministic. Only some outputs represent randomization over actions. In this method, we replace the ith "don't know" χ_i by an action distribution where the probability of action a under observation o is given by $\frac{\#(o,a)}{\#(o)}$. If $\#(o) = 0$, we keep χ_i instead which, in the belief exploration approach of Storm, represents a precomputed cutoff strategy. In approaches where the strategy does not provide any information at all it can be replaced by $\dagger$. Intuitively, we try to copy the behavior of the FSC for an observation and since the optimal action is unknown, we use a distribution over all possible actions.
- *Heuristic 2 – Minimizing Using $\dagger$-transitions:* As for ease of implementation and explainability, smaller FSCs are preferable, this heuristic aims at replacing χ_i outputs such that we can minimise the FSC as much as possible. For this purpose, we simply replace all occurrences of χ_i by $\dagger$, i.e. we replace "don't-know" by "don't-care" outputs. This allows the FSC to behave arbitrarily on these transitions. By then applying an additional minimisation step, we can potentially reduce the size of our FSC. Intuitively, this allows for a smaller FSC that might be able to generalize better than specifying all actions directly. Note that this heuristic will transform any deterministic FSC into a smaller representation that is still deterministic, and will not induce any randomization.

4 Experimental Evaluation

We implemented a prototype of the policy automaton learning framework on top of version 1.8.1 of the probabilistic model checker Storm [20]. As input, our implementation takes the belief MC induced by the optimal policy on the belief MDP abstraction computed by Storm's belief exploration for POMDPs [11]. This Markov chain, labeled with observations and actions chosen by the computed strategy, encodes all information necessary for our approach as described in Section 3.3. We apply our learning techniques to obtain a finite-state controller representation of a policy. This FSC can be exported into a human-readable format or analyzed by building the Markov chain induced by the learned policy *directly* on the POMDP. As a baseline comparison for the learned FSC, we use the tool PAYNT [4]. Recall that PAYNT uses a technique called *inductive synthesis* to directly synthesize FSCs with respect to a given objective.

Recent research has shown that PAYNT's performance greatly improves when working in tandem with belief exploration [2]. As such, the comparison



(a) Size of input MC (from STORM) vs. size of learned FSC (number of nodes)

(b) Size of learned FSC in comparison to PAYNT (in number of nodes)

Fig. 5: Comparison of the resulting FSC size

made here does not show the full capabilities of PAYNT. The tandem approach is likely to outperform our approach in many cases. We want to, however, show a comparison of our approach with a more basic, and thus more comparable, method. We emphasize furthermore that integrating our approach in the framework of [2] is a promising prospect for future work.

Setup. The experiments are run on two cores of an Intel® Xeon® Platinum 8160 CPU using 64GB RAM and a time limit of 1 hour. We run STORM's POMDP model checking framework using default parameters. In particular, we use the heuristically determined exploration depth for the belief MDP approximation and apply cut-offs where we choose not to explore further. We refer to [11] for more information. For PAYNT, we use abstraction-refinement with multi-core support [3]. We run experiments for the two heuristics described in Section 3.4. Additionally, we provide another result described as the "base" approach. This is specific to the input given by STORM and encodes the strategy obtained from STORM exactly by keeping the cut-off strategies, represented as χ_i (see the extended version of this paper [8] for more technical details).

Benchmarks. As benchmarks for our evaluation, we consider the models from [2]. The benchmark set contains models taken from the literature [3,10,11,17] meant to illustrate the strengths and weaknesses of the belief exploration and inductive synthesis approaches. As such, they also showcase how our learning approach transforms the output of the belief exploration concerning the size and quality of the computed FSC. An overview of the used benchmarks is available as part of the extended version of this paper [8].

4.1 Results

Our approach is general and meant to be used on top of other algorithms to transform possibly big and hardly explainable strategies into small FSCs. However, we want to explore whether our results are comparable to state-of-the-art work for directly learning FSCs. Therefore, we compare our FSCs to PAYNT.

First, we talk about the size of the FSC generated by our method compared to the MC generated by Storm and the FSCs generated by PAYNT. Secondly, we show the scalability of our approach by comparing the runtime with PAYNT. Lastly, we discuss the quality of the synthesized FSCs compared to PAYNT and also discuss the trade-off between runtime and quality of the FSC.

Small and Explainable FSCs. Given a strategy table, our approach results in the *smallest possible* FSC for the represented strategy. As an overview, in Fig. 5a, we show a comparison of the sizes of the belief MC from Storm to the size of our FSC. The dashed line corresponds to a 10-fold reduction in size, showing our approach's usefulness. We generate FSCs of sizes 1 to 64; however, more than 80% of the FSCs are smaller than ten nodes, and only two are bigger than 60. More than half of the generated FSCs have less than four nodes. In one case, we reduce 4517 states in the MC to an FSC of size 12.

We claim that these concise representations can generally be considered explainable, in particular when compared to huge original strategy representations. When the given strategy is deterministic, our learning approach would construct a deterministic FSC which is easy to explain. While improving the FSC by replacing the "don't know" actions (Section 3.4), heuristic 2 still keeps the FSC deterministic as it only replaces the χ actions with † actions before minimisation. Heuristic 1 often introduces some randomization when it replaces the χ actions with a distribution. But even in that case, they are only in selected sink states which does not impede explainability.

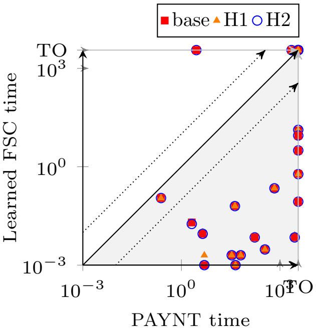


Fig. 6: Runtime comparison: our approach vs. PAYNT

In Fig. 5b, we provide the size comparison of PAYNT's FSCs and ours. Our FSCs are slightly bigger than PAYNT's in general, but our approach also returns smaller FSCs in some cases. This is to be expected since the approach of PAYNT is iteratively searching through the space of FSCs, starting with only one memory node and adding memory only once it is necessary. Therefore, it is meant to find the smallest possible FSC. However, PAYNT times out much more often because of its exhaustive search on small FSC. Additionally, our FSC are bound to be as big as necessary to represent the given strategy. Let us consider the benchmark `grid-avoid-4-0`. In this model, a robot moves in a grid of size four by four with no knowledge about its position. It starts randomly at any place in the grid and has to move towards a goal without falling into a "hole". PAYNT produces a strategy of size 3, which moves right once and then iterates between moving right and down. The nature of Storm's exploration leads to a strategy that moves right three times and then down forever. This can be represented in an FSC of size at least 5.

Scalability. Regarding scalability, Figure 6 shows that our approach outperforms PAYNT on almost all cases. The dotted lines show differences by a factor of 10. There are only two benchmarks, for which our approach times out and

Table 3: Comparison to PAYNT on value, size, and time (in that order) on selected benchmarks. The reported time for our approach includes the time of Storm for producing the strategy table **and** the time for learning the FSC.

			Learning heuristics			
Category	Model	Storm	base	H$_1$	H$_2$	Paynt
A	problem-paynt-storm-combined	8.07	8.07	**7.67**	**7.67**	**7.67**
		18	6	7	**3**	**3**
	Rmin	<1s	**<1s**			349s
	problem-storm-extended	3009.0	3009.0	**98.0**	**98.0**	**98.0**
		64	61	62	**1**	**1**
	Rmin	<1s	**<1s**			**<1 s**
	refuel-20	0.14	0.14	**0.23**	**0.23**	TO
		46	4	4	**3**	
	Pmax	73s	s 75s	**74s**	**74s**	
	grid-avoid-4-01	0.75	0.75	0.9	0.67	**0.93**
		10	5	6	**3**	5
	Pmax	<1s	**<1s**			726s
B	posterior-awareness	12.0	12.0	12.0	12.0	**11.99**
		5	**4**	**4**	**4**	**4**
	Rmin	<1s	**<1s**			**<1 s**
	4x5x2-95	1.29	1.29	1.28	1.26	**2.02**
		26	18	18	16	**4**
	Rmax	<1s	**<1s**			2807s
C	query-s2	395.66	395.66	391.9	343.94	**486.69**
		43	9	9	4	**2**
	Rmax	<1s	**<1s**			5s
	drone-4-1	0.75	TO	TO	TO	**0.87**
		3217				**1**
	Pmax	1s				**2250s**

PAYNT does not. In one of these cases, PAYNT also takes more than 2000s to produce a result.

Runtime and Quality of FSCs. Comparing the quality of results, we need to put into consideration that our approach often runs within a fraction of the available time. We run Storm with its default values to get a strategy. As demonstrated in [3], running Storm using non-default parameters, specifically larger exploration thresholds, results in better strategies at the cost of longer runtimes. Our approach directly profits from such better input strategies.

Since the learning is done in far less than a second for most of the benchmarks, we suggest using a portfolio of the heuristics. This allows us to output the optimal solution among all our heuristics with negligible computational overhead. To simplify the presentation of our results, we categorize the benchmarks into three groups: A, B, and C, based on the overall performance of our method. Due to space constraints, we provide detailed results for only a selection of benchmarks for each category and do not discuss benchmarks for which both approaches experienced timeouts. The complete set of results is given in [8].

Category A. This category represents benchmarks where our approach is arguably favored, assuming the portfolio approach. There are a total of 19 benchmarks in this category, and we observe that we can improve all variants of properties using heuristics. Only one time, PAYNT produces a slightly better probability value (0.93 vs 0.9), but it takes significantly more time (726s vs $<$ 1s).

There are 7 cases where we can generate FSCs while PAYNT times out and on 6 out of these 7 cases, we get the **smallest** FSCs reported in state-of-the-art [2]. In this category, we also include benchmarks on which the heuristics improved on STORM's strategy to achieve the same value as PAYNT while being more efficient, e.g. `problem-paynt-storm-combined`. Also, for the benchmark `problem-storm-extended`, designed to be difficult for STORM, we reduce the approximate total reward from 3009 to 98, resulting in an FSC of size **1** in < 1**s**.

Category B. This category contains benchmarks on which there is no clear frontrunner. There are a total of 7 benchmarks in this category. Three of these benchmarks are similar to `posterior-awareness`, where the results produced and the time taken are quite similar for both approaches. The other 4 benchmarks (similar to `4x5x2-95`) show that the value generated by our approach is significantly worse; however, it takes significantly less time. Depending on the situation, this trade-off between quality and runtime may favor either approach.

Category C. This category shows the weakness of our method compared to PAYNT. In this category, there are a total of 3 benchmarks, out of which our approach times out 2 times. It is notable that the `drone`-benchmarks seem to be generally hard: PAYNT needs 2250s for `drone-4-1`, and both approaches time out for the bigger instances. There is only one benchmark, `query-s2`, where we produce a worse value without any significant time advantage over PAYNT.

5 Conclusion

In this paper, we present an approach to learn an FSC for representing POMDP strategies. Our FSCs are (i) always *smaller* than the given representation, and (ii) the FSC structure is simple, which together increases the strategy's *explainability*. The structure of the FSC is always deterministic. Additionally, one of our heuristics only generates deterministic output actions (without randomization). The other heuristic typically represents a randomized strategy. However, only output actions are randomized, *not* the FSC structure. Besides, this randomization happens in only a very restricted form. Further, our heuristics achieved notable improvements in the *performance* of many strategies produced by STORM and provably perform equal or better than the baseline, while retaining negligible resource consumption. Altogether, our comparison against PAYNT underscores the competitiveness of our method, frequently yielding FSCs of comparable quality with significant improvements in terms of runtime and size.

This attests to the scalability and efficiency of our approach and also highlights its applicability in scenarios challenging for other tools.

Concerning future work, several directions open up. Further heuristics can be designed to solve some of the patterns occurring in the cases where our approach could not match the size achieved by PAYNT. Furthermore, we would like to integrate our approach into other approaches in order to improve them, in particular the tandem synthesis approach from [2] is a suitable candidate.

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

References

1. Amato, C., Bernstein, D.S., Zilberstein, S.: Optimizing fixed-size stochastic controllers for pomdps and decentralized pomdps. Auton. Agents Multi Agent Syst. **21**(3), 293–320 (2010), `https://doi.org/10.1007/s10458-009-9103-z`
2. Andriushchenko, R., Bork, A., Ceska, M., Junges, S., Katoen, J., Macák, F.: Search and explore: Symbiotic policy synthesis in pomdps. In: Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13966, pp. 113–135. Springer (2023), `https://doi.org/10.1007/978-3-031-37709-9_6`
3. Andriushchenko, R., Ceska, M., Junges, S., Katoen, J.: Inductive synthesis of finite-state controllers for pomdps. In: Uncertainty in Artificial Intelligence, Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, UAI 2022, 1-5 August 2022, Eindhoven, The Netherlands. Proceedings of Machine Learning Research, vol. 180, pp. 85–95. PMLR (2022), `https://proceedings.mlr.press/v180/andriushchenko22a.html`
4. Andriushchenko, R., Ceska, M., Junges, S., Katoen, J., Stupinský, S.: PAYNT: A tool for inductive synthesis of probabilistic programs. In: Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12759, pp. 856–869. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_40, `https://doi.org/10.1007/978-3-030-81685-8_40`
5. Angluin, D.: Learning regular sets from queries and counterexamples. Information and computation **75**(2), 87–106 (1987), `https://doi.org/10.1016/0890-5401(87)90052-6`
6. Ashok, P., Jackermeier, M., Jagtap, P., Kretínský, J., Weininger, M., Zamani, M.: dtcontrol: decision tree learning algorithms for controller representation. In: HSCC. pp. 17:1–17:7. ACM (2020), `https://dl.acm.org/doi/abs/10.1145/3365365.3383468`
7. Ashok, P., Jackermeier, M., Křetínský, J., Weinhuber, C., Weininger, M., Yadav, M.: dtcontrol 2.0: Explainable strategy representation via decision tree learning steered by experts. In: TACAS (2). Lecture Notes in Computer Science, vol. 12652, pp. 326–345. Springer (2021), `https://doi.org/10.1007/978-3-030-72013-1_17`
8. Bork, A., Chakraborty, D., Grover, K., Kretinsky, J., Mohr, S.: Learning Explainable and Better Performing Representations of POMDP Strategies. arXiv preprint arXiv:2401.07656 (2024), `https://doi.org/10.48550/arXiv.2401.07656`
9. Bork, A., Chakraborty, D., Grover, K., Mohr, S., Kretinsky, J.: Artifact for Paper: Learning Explainable and Better Performing Representations of POMDP Strategies, `https://doi.org/10.5281/zenodo.10437018`
10. Bork, A., Junges, S., Katoen, J., Quatmann, T.: Verification of indefinite-horizon pomdps. In: Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12302, pp. 288–304. Springer (2020), `https://doi.org/10.1007/978-3-030-59152-6_16`
11. Bork, A., Katoen, J.P., Quatmann, T.: Under-approximating expected total rewards in pomdps. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 22–40. Springer (2022), `https://doi.org/10.1007/978-3-030-99527-0_2`
12. Brázdil, T., Chatterjee, K., Chmelik, M., Fellner, A., Křetínský, J.: Counterexample explanation by learning small strategies in markov decision processes. In: CAV

(1). Lecture Notes in Computer Science, vol. 9206, pp. 158–177. Springer (2015), https://doi.org/10.1007/978-3-319-21690-4_10

13. Carr, S., Jansen, N., Wimmer, R., Serban, A.C., Becker, B., Topcu, U.: Counterexample-guided strategy improvement for pomdps using recurrent neural networks. In: Kraus, S. (ed.) Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019. pp. 5532–5539. ijcai.org (2019), https://doi.org/10.24963/ijcai.2019/768
14. Chatterjee, K., Chmelik, M., Tracol, M.: What is decidable about partially observable markov decision processes with ω-regular objectives. Journal of Computer and System Sciences **82**(5), 878–911 (2016), https://doi.org/10.1016/j.jcss.2016.02.009
15. Cubuktepe, M., Jansen, N., Junges, S., Marandi, A., Suilen, M., Topcu, U.: Robust finite-state controllers for uncertain pomdps. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021. pp. 11792–11800. AAAI Press (2021), https://doi.org/10.1609/aaai.v35i13.17401
16. Hansen, E.A.: Solving pomdps by searching in policy space. In: Cooper, G.F., Moral, S. (eds.) UAI '98: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, University of Wisconsin Business School, Madison, Wisconsin, USA, July 24-26, 1998. pp. 211–219. Morgan Kaufmann (1998), https://dl.acm.org/doi/abs/10.5555/2074094.2074119
17. Hauskrecht, M.: Incremental methods for computing bounds in partially observable markov decision processes. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA. pp. 734–739. AAAI Press / The MIT Press (1997), https://dl.acm.org/doi/10.5555/1867406.1867520
18. Hauskrecht, M.: Value-function approximations for partially observable markov decision processes. J. Artif. Intell. Res. **13**, 33–94 (2000), https://doi.org/10.1613/jair.678
19. Heck, L., Spel, J., Junges, S., Moerman, J., Katoen, J.: Gradient-descent for randomized controllers under partial observability. In: Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13182, pp. 127–150. Springer (2022), https://doi.org/10.1007/978-3-030-94583-1_7
20. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022), https://doi.org/10.1007/s10009-021-00633-z
21. Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Winterer, L., Katoen, J., Becker, B.: Finite-state controllers of pomdps using parameter synthesis. In: Globerson, A., Silva, R. (eds.) Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018. pp. 519–529. AUAI Press (2018)
22. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artificial Intelligence **101**(1), 99–134 (1998), https://doi.org/10.1016/S0004-3702(98)00023-X

23. Kurniawati, H., Hsu, D., Lee, W.S.: SARSOP: efficient point-based POMDP planning by approximating optimally reachable belief spaces. In: Brock, O., Trinkle, J., Ramos, F. (eds.) Robotics: Science and Systems IV, Eidgenössische Technische Hochschule Zürich, Zurich, Switzerland, June 25-28, 2008. The MIT Press (2008), https://doi.org/10.15607/RSS.2008.IV.009
24. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011), https://doi.org/10.1007/978-3-642-22110-1_47
25. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and related stochastic optimization problems. Artificial Intelligence **147**(1-2), 5–34 (2003), https://doi.org/10.1016/S0004-3702(02)00378-8
26. Meuleau, N., Kim, K., Kaelbling, L.P., Cassandra, A.R.: Solving pomdps by searching the space of finite policies. In: Laskey, K.B., Prade, H. (eds.) UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30 - August 1, 1999. pp. 417–426. Morgan Kaufmann (1999), https://dl.acm.org/doi/10.5555/2073796.2073844
27. Neider, D., Topcu, U.: An automaton learning approach to solving safety games over infinite graphs. In: TACAS. Lecture Notes in Computer Science, vol. 9636, pp. 204–221. Springer (2016), https://doi.org/10.1007/978-3-662-49674-9_12
28. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. Real Time Syst. **53**(3), 354–402 (2017), https://doi.org/10.1007/s11241-017-9269-4
29. Pineau, J., Gordon, G.J., Thrun, S.: Point-based value iteration: An anytime algorithm for pomdps. In: Gottlob, G., Walsh, T. (eds.) IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003. pp. 1025–1032. Morgan Kaufmann (2003)
30. Russell, S.J.: Artificial intelligence a modern approach. Pearson Education, Inc. (2010), https://dl.acm.org/doi/book/10.5555/1671238
31. Shahbaz, M., Groz, R.: Inferring mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 207–222. Springer (2009), https://doi.org/10.1007/978-3-642-05089-3_14
32. Shani, G., Pineau, J., Kaplow, R.: A survey of point-based pomdp solvers. Autonomous Agents and Multi-Agent Systems **27**, 1–51 (2013), https://doi.org/10.1007/s10458-012-9200-2
33. Simão, T.D., Suilen, M., Jansen, N.: Safe policy improvement for pomdps via finite-state controllers. In: Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence. AAAI'23/IAAI'23/EAAI'23, AAAI Press (2023), https://doi.org/10.1609/aaai.v37i12.26763
34. Smallwood, R.D., Sondik, E.J.: The optimal control of partially observable markov processes over a finite horizon. Oper. Res. **21**(5), 1071–1088 (1973), https://doi.org/10.1287/opre.21.5.1071
35. Spaan, M.T.J., Vlassis, N.: Perseus: Randomized point-based value iteration for pomdps. J. Artif. Intell. Res. **24**, 195–220 (2005), https://doi.org/10.1613/jair.1659

36. Thomas, P., Theocharous, G., Ghavamzadeh, M.: High-confidence off-policy evaluation. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 29 (2015), https://dl.acm.org/doi/10.5555/2888116.2888134

Simulations

Dissipative quadratizations of polynomial ODE systems *

Yubo Cai[1](✉) and Gleb Pogudin[2]

[1] École Polytechnique, Institute Polytechnique de Paris, Palaiseau, France
yubo.cai@polytechnique.edu
[2] LIX, CNRS, École Polytechnique, Institute Polytechnique de Paris, Palaiseau, France gleb.pogudin@polytechnique.edu

Abstract. Quadratization refers to a transformation of an arbitrary system of polynomial ordinary differential equations to a system with at most quadratic right-hand side. Such a transformation unveils new variables and model structures that facilitate model analysis, simulation, and control and offer a convenient parameterization for data-driven approaches. Quadratization techniques have found applications in diverse fields, including systems theory, fluid mechanics, chemical reaction modeling, and mathematical analysis.
In this study, we focus on quadratizations that preserve the stability properties of the original model, specifically dissipativity at given equilibria. This preservation is desirable in many applications of quadratization including reachability analysis and synthetic biology. We establish the existence of dissipativity-preserving quadratizations, develop an algorithm for their computation, and demonstrate it in several case studies.

Keywords: differential equations · quadratization · stability · variable transformation

1 Introduction

Systems of ordinary differential equations (ODEs) are the standard choice when it comes to modeling processes happening in continuous time, for example, in the sciences and engineering. For a given dynamical process, one can derive different ODE models, in particular, by choosing different sets of variables. It has been observed in a variety of areas and contexts that these choices may have a significant impact on the utility and relevance of the resulting model, and a number of different types of variable transformations have been studied.

In this paper, we will study one such transformation, *quadratization*, which aims at transforming an ODE system to a system where the right-hand side

* This work has partly been supported by the French ANR-22-CE48-0008 OCCAM and ANR-22-CE48-0016 NODE projects. We would like to thank Andrey Bychkov, Marcelo Forets, Christian Schilling, Boris Kramer for helpful discussions and the referees for careful reading and detailed comments.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 323–342, 2024.
https://doi.org/10.1007/978-3-031-57249-4_16

consists of polynomials of degree at most two. Let us illustrate this transformation on a toy example: we start with a scalar ODE $x' = x^3$ in a single variable $x = x(t)$ with cubic right-hand side. If we now augment the state space with an additional coordinate $y = x^2$, we can write the original equation as $x' = xy$ with quadratic right-hand side, and we can do the same for y':

$$y' = 2xx' = 2x^4 = 2y^2.$$

So, the transformation in this case is the following:

$$x' = x^3 \quad \to \quad \begin{cases} x' = xy, \\ y' = 2y^2. \end{cases}$$

It turns out that every polynomial ODE system can be similarly lifted to an at most quadratic one: this fact has been established at least 100 years ago [2,27] and has been rediscovered several times since then [8,10,11,17,24]. In the recent years quadratization has been used in a number of application areas including model order reduction [5,6,17,25,26], synthetic biology [13,20,21], numerical integration [16,18,19], and reachability analysis [14]. While it has been shown in [21] that the problem of finding the minimal number of extra variables necessary for quadratization is NP-hard, at least two practically useful software packages have been developed for performing quadratization: BioCham [21] and QBee [7].

In the majority of the applications mentioned above, the constructed quadratic ODE model is further used in the context of *numerical* simulations. It is, therefore, a natural question whether one can not only guarantee that the transformed model is at most quadratic, but also that it preserves some desirable dynamical/numerical properties of the original ODE system. To the best of our knowledge, this question has not been studied systematically, and in this paper, we initiate this line of research by studying *dissipativity-preserving quadratizations.*

We will say that an ODE system is *dissipative* at an equilibrium point if the real parts of the eigenvalues of the linearization of the system around this point are negative. In particular, dissipativity implies that the system is *asymptotically stable* at this point [23, Theorem 8.2.2]. The main contribution of the paper is two-fold. First, we prove that, for every polynomial system dissipative at several equilibrium points, there exists a quadratization which is also dissipative at all these points. Second, we design and implement an algorithm to search automatically for such quadratization attempting to minimize the dimension. Our algorithm is based on a combinatorial condition on the new variables which is sufficient to guarantee that the resulting quadratic model can be made dissipative as well. This combinatorial condition can be viewed as a generalization and formalization of the artificial stabilization used in [26, Section 4.1]. We implemented the new algorithm and we illustrate it in several case studies including an application for reachability analysis (in combination with the algorithm from [14]). Our implementation together with the examples from this paper is available at [1].

The rest of the paper is organized as follows. In Section 2, we introduce the main notions, quadratization, and dissipativity, and show that quadratization performed straightforwardly may not preserve dissipativity (and, thus, render the model into a numerically unstable one). Section 3 contains the statement and the proof of the main theoretical result of the paper (Theorem 1) that there always exists a dissipativity-preserving quadratization for any collection of dissipative equilibria. Based on the ideas from the proof, we give an algorithm (Algorithm 2) for constructing such a quadratization in Section 4. We showcase our implementation of this algorithm on several case studies in Section 5. Concluding remarks are contained in Section 6.

2 Preliminaries

Throughout this section, we will consider a polynomial ODE system, that is, a system of differential equations

$$\mathbf{x}' = \mathbf{p}(\mathbf{x}), \tag{1}$$

where $\mathbf{x} = \mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$ is a vector of unknown functions and $\mathbf{p} = (p_1, \ldots, p_n)$ is a vector of n-variate polynomials $p_1, \ldots, p_n \in \mathbb{R}[\mathbf{x}]$.

Definition 1 (Quadratization). *For a system* (1), quadratization *is a pair consisting of*

- *a list of new variables*

$$y_1 = g_1(\mathbf{x}), \ldots, y_m = g_m(\mathbf{x})$$

- *and two lists*

$$\mathbf{q}_1(\mathbf{x}, \mathbf{y}) = (q_{1,1}(\mathbf{x}), \ldots, q_{1,n}(\mathbf{y})) \quad \textit{and} \quad \mathbf{q}_2(\mathbf{x}, \mathbf{y}) = (q_{2,1}(\mathbf{x}, \mathbf{y}), \ldots, q_{2,m}(\mathbf{x}, \mathbf{y}))$$

of $m + n$-variate polynomials in $\mathbf{x}$ and $\mathbf{y} = (y_1, \ldots, y_m)$

such that the degree of each of of $\mathbf{q}_1$ and $\mathbf{q}_2$ is at most two and

$$\mathbf{x}' = \mathbf{q}_1(\mathbf{x}, \mathbf{y}) \quad \textit{and} \quad \mathbf{y}' = \mathbf{q}_2(\mathbf{x}, \mathbf{y}). \tag{2}$$

If all the polynomials $g_1, \ldots, g_m$ are monomials, the quadratization is called monomial quadratization*.*

Note that unlike, for example, [7, Definition 1], by *quadratization* we mean not just the set of new variables but also the quadratic ODE system (2). The reason for this is that, for a fixed set of new variables, there may be many different systems of the shape (see Example 1) prescribed by (2) exhibiting different numerical behaviors (see Example 2).

Example 1 (Quadratization). Consider the following scalar ODE

$$x' = -x + x^3.$$

Here we have $n = 1$ and $p_1(x) = -x + x^3$. Consider $y = g_1(x) = x^2$. Then we can write

$$\begin{aligned} x' &= -x + x^3 = -x + xy, \\ y' &= 2xx' = -2x^2 + 2x^4 = -2y + 2y^2. \end{aligned}$$

Therefore, one possible quadratization is given by

$$g_1(x) = x^2, \quad q_{1,1}(x, y) = -x + xy, \quad q_{2,1}(x, y) = -2y + 2y^2.$$

As we have mentioned above, there may be different $\mathbf{q}$'s corresponding to the same $\mathbf{g}$. In this example, we could take, for example, $q_{2,1} = -2y+2y^2+2(y-x^2) = -2x^2 + 2y^2$ or $q_{2,1} = y - 3x^2 + 2y^2$. As we will see in Example 2, such choices may have a dramatic impact on the numerical properties of the resulting ODE system.

Definition 2 (Equilibrium). *For a polynomial ODE system* (1)*, a point* $\mathbf{x}^* \in \mathbb{R}^n$ *is called an* equilibrium *if* $\mathbf{p}(\mathbf{x}^*) = 0$*.*

Definition 3 (Dissipativity). *An ODE system* (1) *is called* dissipative *at an equilibrium point* $\mathbf{x}^*$ *if all the eigenvalues of the Jacobian* $J(\mathbf{p})|_{\mathbf{x}=\mathbf{x}^*}$ *of* $\mathbf{p}$ *and* $\mathbf{x}^*$ *have negative real part.*

It is known that a system which is dissipative at an equilibrium point $\mathbf{x}^*$ *is* asymptotically stable *at* $\mathbf{x}^*$ *[23, Theorem 8.2.2], that is, any trajectory starting in a small enough neighborhood of* $\mathbf{x}^*$ *will converge to* $\mathbf{x}^*$ *exponentially fast.*

Note that if $\mathbf{x}^* = 0$*, then the Jacobian at this point is simply the matrix of the linear part of* $\mathbf{p}(\mathbf{x})$*.*

Assume that $\mathbf{x}^* \in \mathbb{R}^n$ is an equilibrium of $\mathbf{x}' = \mathbf{p}(\mathbf{x})$, and consider a quadratization of this system as in Definition 1. Then a direct computation shows that $(\mathbf{x}^*, \mathbf{g}(\mathbf{x}^*))$ is an equilibrium point of the resulting quadratic system (2).

Definition 4 (Dissipative quadratization). *Assume that a system* (1) *is dissipative at an equilibrium point* $\mathbf{x}^* \in \mathbb{R}^n$*. Then a quadratization given by* $\mathbf{g}, \mathbf{q}_1$ *and* $\mathbf{q}_2$ *(see Definition 1) is called* dissipative at $\mathbf{x}^*$ *if the system*

$$\mathbf{x}' = \mathbf{q}_1(\mathbf{x}, \mathbf{y}), \quad \mathbf{y}' = \mathbf{q}_2(\mathbf{x}, \mathbf{y})$$

is dissipative at a point $(\mathbf{x}^*, \mathbf{g}(\mathbf{x}^*))$*.*

The following example shows that, even for the same new variables $\mathbf{y} = \mathbf{g}(\mathbf{x})$, different quadratizations may have significantly different stability properties.

Example 2 (Stable and unstable quadratizations). Consider the scalar ODE $x' = -x + x^3$ from Example 1. We have already found a quadratization for it using a new variable $y = g_1(x) = x^2$ with the resulting quadratic system being

$$x' = -x + xy \quad \text{and } y' = -2y + 2y^2. \tag{3}$$

We notice that we can add/subtract $y - x^2$ with any coefficients from the right-hand sides of the system. For example, we can obtain:

$$x' = -x + xy \quad \text{and} \quad y' = -2y + 2y^2 + 12(y - x^2) = 10y - 12x^2 + 2y^2. \qquad (4)$$

Both systems above are quadratizations of the original system and, thus, mathematically, for any initial condition (x_0, y_0) satisfying $y_0^2 = x_0^2$, they must follow the same trajectory. However, (3) is stable at $(0,0)$ while (4) is not. By numerically integrating them, we can observe in Figure 1 that in practice (3) reflects the dynamics of the original equation accurately and (4) heavily suffers from numerical instability.

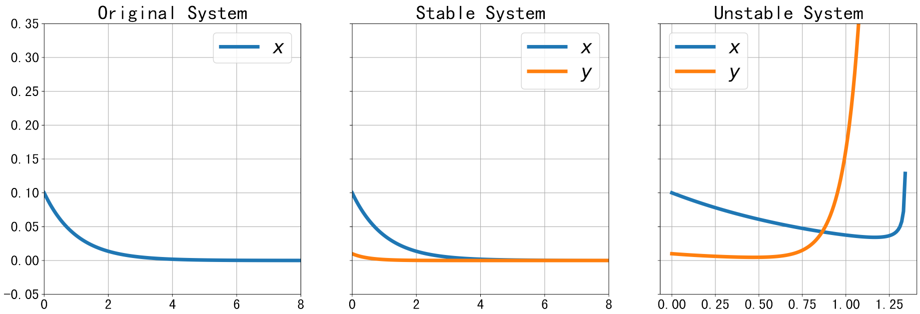


Fig. 1: Plot of the original equation, (3), and (4) with initial condition $\mathcal{X}_0 = [x_0, y_0 = x_0^2] = [0.1, 0.01]$. Numerical method: "LSODA" (uses hybrid Adams/BDF method with automatic stiffness detection) in scipy.integrate.solve_ivp package [22,29].

3 Existence of dissipativity-preserving quadratizations

The main result of this section is the following theorem. Its proof is constructive and is used to design an algorithm in Section 4.

Theorem 1. *For every polynomial ODE system* $\mathbf{x}' = \mathbf{p}(\mathbf{x})$*, there exists a quadratization that is dissipative at all the dissipative equilibria of* $\mathbf{x}' = \mathbf{p}(\mathbf{x})$.

Remark 1. In fact, the key ingredients of the proof, Propositions 1 and 2, imply a stronger statement: for every set of finitely many equilibria, there is a quadratization such that the number of nonnegative eigenvalues of the Jacobian of the quadratic system at these points is the same as for the original system.

The rest of the section will be devoted to proving Theorem 1. The main technical notion will be an *inner-quadratic* set of polynomials.

Definition 5 (Inner-quadratic set). *As finite set* $g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ *of nonconstant polynomials in* $\mathbf{x} = (x_1, \ldots, x_n)$ *is called* inner-quadratic *if, for every* $1 \leqslant i \leqslant m$*, there exist (not necessarily distinct)* $a, b \in \{x_1, \ldots, x_n, g_1, \ldots, g_m\}$ *such that* $g_i = ab$.

A quadratization will be called inner-quadratic *if the set of new variables* $\mathbf{g}$ *is inner-quadratic. We will also always assume that the new variables are sorted by degree, that is,* $\deg g_1 \leqslant \deg g_2 \leqslant \ldots \leqslant \deg g_m$.

The rationale behind the notion of inner-quadratic quadratization is that at most quadratic relations between the new variables give us the flexibility to "tune" the right-hand side of the resulting quadratic system in the same fashion as we added a multiple of $y - x^2$ in Example 2. These additional terms force the trajectory to stay on the image of the map $\mathbf{x} \to (\mathbf{x}, \mathbf{y})$, on which the properties of the original dynamics (such as dissipativity) are preserved. The following definition formalizes this observation.

Definition 6 (Stabilizers). *Consider a polynomial ODE system* $\mathbf{x}' = \mathbf{p}(\mathbf{x})$ *and its inner-quadratic quadratization given by* m *new variables* $\mathbf{y} = \mathbf{g}(\mathbf{x})$ *and right-hand side* $\mathbf{q}_1(\mathbf{x}, \mathbf{y}), \mathbf{q}_2(\mathbf{x}, \mathbf{y})$ *of the resulting quadratic system as in Definition 1. For every* $1 \leqslant i \leqslant m$*, by the definition of inner-quadratic set, there exist* $a_i, b_i \in \{\mathbf{x}, y_1, \ldots, y_{i-1}\}$ *such that the equality* $y_i = a_i b_i$ *holds if we replace each* y_j *with* $g_j(\mathbf{x})$*. We define* the i-th stabilizer *by* $h_i(\mathbf{x}, \mathbf{y}) := y_i - a_i b_i$.

Since each stabilizer is at most quadratic and vanishes under the substitution $\mathbf{y} = \mathbf{g}(\mathbf{x})$*, adding any stabilizer to any of* $\mathbf{q}_1, \mathbf{q}_2$ *still yields a quadratization of* $\mathbf{x}' = \mathbf{p}(\mathbf{x})$.

Example 3 (Stabilizers). Let us give an example of the stabilizers. Consider a system:

$$x_1' = -3x_1 + x_2^4, \quad x_2' = -2x_2 + x_1^2.$$

By applying **Algorithm 1**, we introduce the following new variables to obtain an inner-quadratic quadratization:

$$y_1 = x_1^2, \quad y_2 = x_2^2, \quad y_3 = x_1 x_2, \quad y_4 = x_2^3 = x_2 y_2.$$

Then the corresponding stabilizers, according to the definition above, will be:

$$\begin{aligned} h_1(\mathbf{x}, \mathbf{y}) &= y_1 - x_1^2, & h_2(\mathbf{x}, \mathbf{y}) &= y_2 - x_2^2, \\ h_3(\mathbf{x}, \mathbf{y}) &= y_3 - x_1 x_2, & h_4(\mathbf{x}, \mathbf{y}) &= y_4 - x_2 y_2 \end{aligned}$$

Theorem 1 follows directly from the following two properties of inner-quadratic quadratizations:

- every polynomial ODE system has an inner-quadratic quadratization (Proposition 1);
- for any inner-quadratic quadratization, one can modify the right-hand sides of the quadratic system (but not the new variables) using the stabilizers in order to obtain a dissipativity-preserving quadratization (Proposition 2).

Proposition 1. *Every polynomial ODE system* $\mathbf{x}' = \mathbf{p}(\mathbf{x})$ *admits an inner-quadratic quadratization. Furthermore, it can be chosen to be a monomial quadratization.*

Proof. We will show that the quadratization which is typically used to prove the existence of a quadratization for every polynomial ODE system (see, e.g. [9, Theorem 1]) is in fact inner-quadratic. For every $1 \leqslant i \leqslant n$, we introduce $d_i = \max\limits_{1\leqslant j\leqslant n} \deg_{x_i} p_j$. Then it is proven in [9, Theorem 1] that the following set of new variables yields a quadratization of $\mathbf{x}' = \mathbf{p}(\mathbf{x})$:

$$\mathcal{M} = \{x_1^{i_1} \ldots x_n^{i_n} \mid \forall j\colon 0 \leqslant i_j \leqslant d_j,\ \sum i_j > 1\}.$$

Let $g \in \mathcal{M}$. Then there exists $1 \leqslant j \leqslant n$ such that $\deg_{x_j} g > 0$. Then we can write $g = (g/x_j)\cdot x_j$, where g/x_j is either in $\mathcal{M}$ or belongs to $\{x_1, \ldots, x_n\}$. Thus, $\mathcal{M}$ is an inner-quadratic set.

Proposition 2. *Consider a system* $\mathbf{x}' = \mathbf{p}(\mathbf{x})$ *and its inner-quadratic quadratization defined by new variables* $\mathbf{g}(\mathbf{x})$ *and the new right-hand side* $\mathbf{q}_1, \mathbf{q}_2$ *as in Definition 1. Let* $\mathbf{x}_1^*, \ldots, \mathbf{x}_\ell^*$ *be a finite subset of the equilibria of the system. Then there exist vectors of quadratic polynomials* $\mathbf{r}_1(\mathbf{x}, \mathbf{y}), \mathbf{r}_2(\mathbf{x}, \mathbf{y})$ *such that* $\mathbf{g}, \mathbf{r}_1, \mathbf{r}_2$ *define a quadratization for which the eigenvalues of the Jacobian at each equilibrium point of the form* $(\mathbf{x}_i^*, \mathbf{g}(\mathbf{x}_i^*))$ *are the union of the eigenvalues of* $J(\mathbf{p})|_{\mathbf{x}=\mathbf{x}_i^*}$ *and a set of numbers with negative real part.*

Corollary 1. *Consider a system* $\mathbf{x}' = \mathbf{p}(\mathbf{x})$ *and its inner-quadratic quadratization defined by new variables* $\mathbf{g}(\mathbf{x})$ *and the new right-hand side* $\mathbf{q}_1, \mathbf{q}_2$ *as in Definition 1. Then there exist vectors of quadratic polynomials* $\mathbf{r}_1(\mathbf{x}, \mathbf{y}), \mathbf{r}_2(\mathbf{x}, \mathbf{y})$ *such that* $\mathbf{g}, \mathbf{r}_1, \mathbf{r}_2$ *define a quadratization which is dissipative at every dissipative equilibrium of* $\mathbf{x}' = \mathbf{p}(\mathbf{x})$.

Proof (Proof of Corollary 1). Since each dissipative equilibrium of the system is an isolated root of the polynomial system obtained by equating the right-hand side to zero, there are only finitely many of them. So we apply Proposition 2 to this finite set of equilibria and obtain the desired quadratization.

Before proving Proposition 2, we establish a useful linear-algebraic lemma.

Lemma 1. *Let* $A \in \mathbb{R}^{n\times n}$ *be a square matrix and* $B \in \mathbb{R}^{n\times n}$ *be an upper triangular matrix with ones on the diagonal. Then there exists* $\lambda_0 \in \mathbb{R}$ *such that, for every* $\lambda > \lambda_0$*, the real parts of all the eigenvalues of* $A - \lambda B$ *are negative.*

Proof. Consider the characteristic polynomial of $A - \lambda B$. It can be written as

$$\det(A - \lambda B - tI) = (-\lambda)^n \det(B - A/\lambda + (t/\lambda I)).$$

We set $T = t/\lambda$ and rewrite the latter determinant as $Q(T, 1/\lambda) := \det((B + T\cdot I) - A/\lambda)$. Since Q is the determinant of a matrix with the entries linear in T and $1/\lambda$, it is a bivariate polynomial in T and $1/\lambda$ of total degree at most n. Furthermore, if we set $\lambda = \infty$ (equivalently, if we set $1/\lambda = 0$), we have $Q(T, 0) = \det(B + T\cdot I)$. Since B is upper-triangular with ones on the diagonal, this determinant is equal to $\det(B+T\cdot I) = (T+1)^n$, so $Q(T, 1/\lambda)$ can be written as $Q(T, 1/\lambda) = (T+1)^n + \frac{1}{\lambda}p\left(T+1, \frac{1}{\lambda}\right)$, where p is a bivariate polynomial of the total degree at most $n-1$ in $T+1$ and $\frac{1}{\lambda}$. Let C be an upper bound for the absolute value of the coefficients of p. Then, for $\lambda > 1$, we can bound:

$$\left|p\left(T+1, \frac{1}{\lambda}\right)\right| < Cn^2 \max(|T+1|^{n-1}, 1).$$

Let T_0 be any root of $Q(T, 1/\lambda)$. Then we have

$$|T_0+1|^n \leqslant \frac{1}{\lambda} Cn^2 \max(|T_0+1|^{n-1}, 1).$$

Let us take $\lambda > Cn^2$. Then

$$|T_0+1|^n < \max(|T_0+1|^{n-1}, 1) \implies |T_0+1| < 1.$$

So, in this case, the real part of any root of Q will be negative. Then the same is true for the characteristic polynomial of $A - \lambda B$ because these two polynomials differ by scaling by a positive number λ. Therefore, λ_0 can be taken to be $\max(1, Cn^2)$.

We will also use the following folklore analytic lemma.

Lemma 2. *Let $\mathbf{x}' = \mathbf{f}(\mathbf{x})$ a system of polynomial differential equations of dimension n with an equilibrium point $\mathbf{x}^*$. Let $\varphi\colon \mathbb{R}^n \to \mathbb{R}^n$ be an invertile change of coordinates, and let $\mathbf{y}' = \mathbf{g}(\mathbf{y})$ be the image of the system under the coordinate change. Then the matrices $J_{\mathbf{x}}(\mathbf{f})|_{\mathbf{x}=\mathbf{x}^*}$ and $J_{\mathbf{y}}(\mathbf{g})|_{\mathbf{y}=\varphi(\mathbf{x}^*)}$ are conjugate. In particular, they have the same eigenvalues.*

Proof. By the chain rule, we have

$$\mathbf{y}' = (\varphi(\mathbf{x}))' = J_{\mathbf{x}}(\varphi)\mathbf{x}' = J_{\mathbf{x}}(\varphi)|_{\mathbf{x}=\varphi^{-1}(\mathbf{y})} f(\varphi^{-1}(\mathbf{y})) = \mathbf{g}(\mathbf{y}).$$

Then we can write $J_{\mathbf{y}}(\mathbf{g})$ as

$$J_{\mathbf{x}}(\varphi)|_{\mathbf{x}=\varphi^{-1}(\mathbf{y})} J_{\mathbf{x}}(f)|_{\mathbf{x}=\varphi^{-1}(\mathbf{y})} J_{\mathbf{y}}(\varphi^{-1}) + \sum_{i=1}^{n} A_i f_i(\varphi^{-1}(\mathbf{y})),$$

where A_i is the Jacobian of the i-th column of $J_{\mathbf{x}}(\varphi)|_{\mathbf{x}=\varphi^{-1}(\mathbf{y})}$. If we plug $\varphi(\mathbf{x}^*)$ for $\mathbf{y}$, since $\mathbf{f}(\mathbf{x}^*) = 0$, the latter sum will vanish, so we get

$$J_{\mathbf{x}}(\varphi)|_{\mathbf{x}=\mathbf{x}^*} J_{\mathbf{x}}(f)|_{\mathbf{x}=\mathbf{x}^*} J_{\mathbf{y}}(\varphi^{-1})|_{\mathbf{y}=\varphi(\mathbf{x}^*)}.$$

By the chain rule, the matrices $J_{\mathbf{x}}(\varphi)|_{\mathbf{x}=\mathbf{x}^*}$ and $J_{\mathbf{y}}(\varphi^{-1})|_{\mathbf{y}=\varphi(\mathbf{x}^*)}$ are inverses to each other, so the Jacobians are indeed conjugated.

Proof (Proof of Proposition 2). Before starting the proof, we would like to point at Example 6 in our extended version of the paper [3] which illustrates the main steps of the proof.

[3] https://arxiv.org/pdf/2311.02508.pdf

We define a map $\varphi\colon \mathbb{R}^{n+m} \to \mathbb{R}^{n+m}$ from a space with coordinates $(\mathbf{x}, \mathbf{y})$ to a space with coordinates $(\mathbf{x}, \mathbf{z})$, where $\mathbf{z} = (z_1, \ldots, z_m)$, by

$$\begin{aligned}&\varphi_i(\mathbf{x}, \mathbf{y}) = x_i \quad \text{for } 1 \leqslant i \leqslant n,\\ &\varphi_{n+j}(\mathbf{x}, \mathbf{y}) = y_j - g_j(\mathbf{x}) \quad \text{for } 1 \leqslant j \leqslant m.\end{aligned}$$

This map is invertible with the inverse given by $(\varphi^{-1})_i(\mathbf{x}, \mathbf{z}) = x_i$ for $1 \leqslant i \leqslant n$ and $(\varphi^{-1})_{n+j}(\mathbf{x}, \mathbf{z}) = z_j + g_j(\mathbf{x})$ for $1 \leqslant j \leqslant m$, so φ is bijective. Note that $\varphi(\mathbf{x}^\circ, \mathbf{g}(\mathbf{x}^\circ)) = (\mathbf{x}^\circ, \mathbf{0})$ for every $\mathbf{x}^\circ \in \mathbb{R}^n$. We apply a change of coordinates defined by φ to the quadratic system $\mathbf{x}' = \mathbf{q}_1(\mathbf{x}, \mathbf{y}),\ \mathbf{y}' = \mathbf{q}_2(\mathbf{x}, \mathbf{y})$ and obtain a (not necessarily quadratic) system of the form:

$$\mathbf{x}' = \widetilde{\mathbf{q}}_1(\mathbf{x}, \mathbf{z}) \quad \text{and} \quad \mathbf{z}' = \widetilde{\mathbf{q}}_2(\mathbf{x}, \mathbf{z}), \tag{5}$$

where $\widetilde{q}_1 = q_1(\mathbf{x}, \mathbf{z} + \mathbf{g}(\mathbf{z}))$ and $\widetilde{q}_2$ can be found using the chain rule as follows:

$$\mathbf{z}' = (\mathbf{y} - \mathbf{g}(\mathbf{x}))' = q_2(\mathbf{x}, \mathbf{z} + \mathbf{g}(\mathbf{x})) - J_{\mathbf{x}}(\mathbf{g})\mathbf{x}' = q_2(\mathbf{x}, \mathbf{z} + \mathbf{g}(\mathbf{x})) - J_{\mathbf{x}}(\mathbf{g}) q_1(\mathbf{x}, \mathbf{z} + \mathbf{g}(\mathbf{x})). \tag{6}$$

Since the variety $\{(\mathbf{x}^\circ, \mathbf{g}(\mathbf{x}^\circ)) \mid \mathbf{x}^\circ \in \mathbb{R}^n\}$ was an invariant variety of $\mathbf{x}' = \mathbf{q}_1(\mathbf{x}, \mathbf{y}),\ \mathbf{y}' = \mathbf{q}_2(\mathbf{x}, \mathbf{y})$ by construction, the linear space $\{(\mathbf{x}^\circ, \mathbf{0}) \mid \mathbf{x}^\circ \in \mathbb{R}^n\}$ is invariant for (5) and the restriction of (5) to this space coincides with the original system $\mathbf{x}' = \mathbf{p}(\mathbf{x})$. This implies the following constraints on $\widetilde{\mathbf{q}}_1$ and $\widetilde{\mathbf{q}}_2$:

- $\widetilde{\mathbf{q}}_1(\mathbf{x}, \mathbf{z}) = \mathbf{p}(\mathbf{x}) + \mathcal{O}(\mathbf{z})$, where $\mathcal{O}(\mathbf{z})$ stands for a polynomial with each monomial containing at least one of the $\mathbf{z}$;
- $\widetilde{\mathbf{q}}_2(\mathbf{x}, \mathbf{z}) = \mathcal{O}(\mathbf{z})$.

Due to these constraints, for every $\mathbf{x}^\circ \in \mathbb{R}^n$, the Jacobian of $(\widetilde{\mathbf{q}}_1, \widetilde{\mathbf{q}}_2)$ at $(\mathbf{x}^\circ, \mathbf{0})$ is of the form

$$J_{\mathbf{x},\mathbf{z}}(\widetilde{\mathbf{q}}_1, \widetilde{\mathbf{q}}_2)|_{\mathbf{x}=\mathbf{x}^\circ, \mathbf{z}=\mathbf{0}} = \begin{pmatrix} J_{\mathbf{x}}(\mathbf{p})|_{\mathbf{x}=\mathbf{x}^\circ} & * \\ 0 & J_{\mathbf{z}}(\widetilde{\mathbf{q}}_2)|_{\mathbf{x}=\mathbf{x}^\circ, \mathbf{z}=\mathbf{0}} \end{pmatrix} \tag{7}$$

Let $h_1(\mathbf{x}, \mathbf{y}), \ldots, h_m(\mathbf{x}, \mathbf{y})$ be the stabilizers of the quadratization (see Definition 6). We take an arbitrary parameter $\lambda \in \mathbb{R}$ and consider $\mathbf{q}_{2,\lambda}(\mathbf{x}, \mathbf{y})$ defined by

$$\mathbf{q}_{2,\lambda}(\mathbf{x}, \mathbf{y}) = \mathbf{q}_2(\mathbf{x}, \mathbf{y}) - \lambda \mathbf{h}(\mathbf{x}, \mathbf{y}). \tag{8}$$

Since the h_i's are stabilizers, $\mathbf{g}, \mathbf{q}_1, \mathbf{q}_{2,\lambda}$ is a quadratization of the original system for any value of λ (see Definition 6). By using $q_{2,\lambda}$ instead of q_2 in (6), we obtain $\widetilde{\mathbf{q}}_{2,\lambda} = \widetilde{\mathbf{q}}_2 - \lambda \mathbf{h}(\mathbf{x}, \mathbf{z} + \mathbf{g}(\mathbf{x}))$. Then, as in (7), we get

$$J_{\mathbf{x},\mathbf{z}}(\widetilde{\mathbf{q}}_1, \widetilde{\mathbf{q}}_{2,\lambda})|_{\mathbf{x}=\mathbf{x}^\circ, \mathbf{z}=\mathbf{0}} = \begin{pmatrix} J_{\mathbf{x}}(\mathbf{p})|_{\mathbf{x}=\mathbf{x}^\circ} & * \\ 0 & (J_{\mathbf{z}}(\widetilde{\mathbf{q}}_2) - \lambda J_{\mathbf{z}}(\mathbf{h}))|_{\mathbf{x}=\mathbf{x}^\circ, \mathbf{z}=\mathbf{0}} \end{pmatrix} \tag{9}$$

Observe that, since every g_i is of the form z_i plus polynomial in $\mathbf{x}$ and z's with smaller indices, $J_{\mathbf{z}}(\mathbf{h})$ is a lower-triangular matrix with ones on the diagonal.

Having such a convenient expression for the Jacobian, we consider the given equilibria $\mathbf{x}_1^*, \ldots, \mathbf{x}_\ell^*$. For any $\mathbf{x}^\circ \in \{\mathbf{x}_1^*, \ldots, \mathbf{x}_\ell^*\}$, the eigenvalues of the Jacobian (9) are the union of the eigenvalues of $J_{\mathbf{x}}(\mathbf{p})|_{\mathbf{x}=\mathbf{x}^\circ}$ and the eigenvalues of $(J_{\mathbf{z}}(\widetilde{\mathbf{q}}_2) - \lambda J_{\mathbf{z}}(\mathbf{h}))|_{\mathbf{x}=\mathbf{x}^\circ, \mathbf{z}=\mathbf{0}}$. Applying Lemma 1 to ℓ pairs of matrices

$A = J_{\mathbf{z}}(\widetilde{\mathbf{q}}_2)|_{\mathbf{x}=\mathbf{x}_i^*,\mathbf{z}=0}$ and $B = J_{\mathbf{z}}(\mathbf{h})|_{\mathbf{x}=\mathbf{x}_i^*,\mathbf{z}=0}$, we choose λ to be larger than any of the λ_0's provided by the lemma. Then all the eigenvalues of this block will also have negative real parts. By Lemma 2, the same true for the Jacobian of $\mathbf{x}' = \mathbf{q}_1(\mathbf{x}, \mathbf{y}),\ \mathbf{y}' = \mathbf{q}_{2,\lambda}(\mathbf{x}, \mathbf{y})$.

4 Algorithms

Based on the proof of Theorem 1, finding a dissipativity-preserving quadratization can be done in two following steps:

(Step 1) finding an inner-quadratic quadratization
(Step 2) modifying the corresponding quadratic system to achieve dissipativity at the given equilibria.

In this section, we give algorithms for both steps. Section 4.1 shows how to modify the quadratization algorithm from [7] to search for inner-quadratic quadratizations. Using this algorithm as a building block, we give a general algorithm for computing dissipativity-preserving quadratizations in Section 4.2.

4.1 Computing inner-quadratic quadratization

Our algorithm follows the general *Branch-and-Bound (B&B)* paradigm [28] and is implemented based on the optimal monomial quadratization algorithm from [7, Section 4]. Therefore, we will describe the algorithm briefly, mainly focusing on the differences with the algorithm from [7].

We define each subproblem [7, Definition 3.3] as a set of new monomial variables $\{y_1(\mathbf{x}), \ldots, y_\ell(\mathbf{x})\}$, and the subset of the *search space* [7, Definition 3.1] for the subproblem will be the set of all *quadratizations* including these new variables. To each subproblem $\{y_1(\mathbf{x}), \ldots, y_\ell(\mathbf{x})\}$, the algorithm from [7] assigns a set of generalized variables V (new variables, $\mathbf{x}$'s, and 1) and a set of nonsquares NS (monomials in the right-hand side which are not quadratic in the generalized variables [7, Definition 4]). Additionally, we define the set of non-inner-quadratic new variables NQ which consist of all the monomials among $y_1(\mathbf{x}), \ldots, y_\ell(\mathbf{x})$ which are not quadratic in $\{y_1(\mathbf{x}), \ldots, y_\ell(\mathbf{x}), x_1, \ldots, x_n\}$. In particular, a subproblem is an inner-quadratic quadratization *if and only if* $\mathrm{NS} = \varnothing$ and $\mathrm{NQ} = \varnothing$. Note that NS and NQ are disjoint since $\mathrm{NQ} \subseteq \mathrm{V}$ and $\mathrm{V} \cap \mathrm{NS} = \varnothing$.

Example 4. The notation previously introduced will now be demonstrated through the system $x' = x^4 + x^3$ (taken from [7, Example 4], to display the difference between two algorithms). We consider a subproblem with one already added new variable $y_1(x) := x^3$ (so, $y_1' = 3x^2x' = 3x^6 + 3x^5$). In this case, we have

$$V = \{1, x, x^3\}, \quad V^2 = \left\{1, x, x^2, x^3, x^4, x^6\right\}, \quad NS = \{x^5\}, \quad \underline{NQ = \{x^3\}}.$$

The algorithm starts from the subproblem $\varnothing$. For every iteration, we select one element m from $\mathrm{NS} \cup \mathrm{NQ}$ (using a heuristic score function [7, Section 4.1]) and compute all the decompositions of the form $m = m_1 m_2$, where m_1 and m_2 are monomials. If $m \in \mathrm{NS}$, for every such decomposition, we create a new subproblem by adding the elements of $\{m_1, m_2\} \setminus V$ and at least one new variable will be added due to the property of NS. If $m \in \mathrm{NQ}$, we only do this for the decompositions with $m_1 \neq 1$ and $m_2 \neq 1$.

We apply this operation recursively and stop when $\mathrm{NS} \cup \mathrm{NQ} = \varnothing$ for each branch. Therefore, we can find all the possible inner-quadratic quadratization of the system. To improve the efficiency, we do not consider branches with more new variables than in already found answers and use versions of domain-specific pruning rules from [7]. The algorithm is summarized as Algorithm 1.

Algorithm 1: Computing optimal inner-quadratic quadratization

Input

- polynomial ODEs system $\mathbf{x}' = \mathbf{p}(\mathbf{x})$.
- a set of already chosen new variables $y_1(\mathbf{x}), \dots, y_\ell(\mathbf{x})$ (at the first call, $\varnothing$).
- the order N of the smallest inner-quadratic quadratization found so far (at the first call, $N = \infty$).

Output a more optimal inner-quadratic quadratization containing $y_1(\mathbf{x}), \dots, y_\ell(\mathbf{x})$ if such quadratization exists.

(Step 1) If $y_1(\mathbf{x}), \dots, y_\ell(\mathbf{x})$ is a inner-quadratic quadratization, that is, $\mathrm{NS} = \varnothing$ and $\mathrm{NQ} = \varnothing$, and $\ell < N$, **return** $y_1, \dots, y_\ell$.

(Step 2) Select the element $m \in \mathrm{NS} \cup \mathrm{NQ}$ with the smallest *score*, compute all the decompositions $m = m_1 m_2$ as a product of two monomials. If $m \in NQ$, we only consider the decompositions with $m_1 \neq 1$ and $m_2 \neq 1$.

(Step 3) For each decomposition $m = m_1 m_2$ from the previous step, we consider a subproblem $\{z_1, \dots, z_\ell\} \cup (\{m_1, m_2\} \setminus V)$. If its size is less than N and none of the pruning rules apply, we run recursively on this subproblem and update N if a more optimal inner-quadratic quadratization has been found by the recursive call.

4.2 Computing dissipativity-preserving quadratization

Based on the proof of Theorem 1, the main idea behind the search for dissipativity-preserving quadratization is to start with any inner-quadratic quadratization, and replace the right-hand side for the new variables, $\mathbf{q}_2(\mathbf{x}, \mathbf{y})$, by $\mathbf{q}_2(\mathbf{x}, \mathbf{y}) - \lambda \mathbf{h}(\mathbf{x}, \mathbf{y})$ (see (8)) for increasing values of λ until the desired quadratization is found. The detailed algorithm is given as Algorithm 2, the proof of its correctness and termination is provided by Proposition 3, and a step-by-step example is given in Example 5.

Proposition 3. *Algorithm 2 always terminates and produces a correct output.*

Proof. We will start with proving the correctness. Note that since $\mathbf{g}, \mathbf{q}_1, \mathbf{q}_2$ computed in **(Step 1)** yield a quadratization of the input system, and $\mathbf{h}$ vanishes if

Algorithm 2: Computing a quadratization dissipative at all provided equilibria

```
Input  polynomial ODE system $\mathbf{x} \,=' \mathbf{p}(\mathbf{x})$ and a list of its dissipative equilibria
       $\mathbf{x}_1^*, \ldots, \mathbf{x}_\ell^*$;
Output  a quadratization of the system which is dissipative at $\mathbf{x}_1^*, \ldots, \mathbf{x}_\ell^*$.

(Step 1) Compute an inner-quadratic quadratization of $\mathbf{x}' = \mathbf{p}(\mathbf{x})$ using
         Algorithm 1. Let the introduced variables be $y_1 = g_1(\mathbf{x}), \ldots, y_m = g_m(\mathbf{x})$.
         Let $\mathbf{q}_1(\mathbf{x}, \mathbf{y})$ and $\mathbf{q}_2(\mathbf{x}, \mathbf{y})$ be the right-hand sides of the quadratic system as
         in Definition 1. If the corresponding quadratic system is dissipative at
         $\mathbf{x}_1^*, \ldots, \mathbf{x}_\ell^*$, return it.
(Step 2) Construct the stabilizers $\mathbf{h}(\mathbf{x}, \mathbf{y})$ for the quadratization from (Step 1) as in
         Definition 6, and set $\lambda = 1$.
(Step 3) While True
           (a) Construct a quadratic system $\Sigma_\lambda$
                   $\mathbf{x}' = \mathbf{q}_1(\mathbf{x}, \mathbf{y}),\ \mathbf{y}' = \mathbf{q}_2(\mathbf{x}, \mathbf{y}) - \lambda \mathbf{h}(\mathbf{x}, \mathbf{y}).$
           (b) Check if $\Sigma_\lambda$ is dissipative at $(\mathbf{x}_i^*, \mathbf{g}(\mathbf{x}_i^*))$ for every $1 \leqslant i \leqslant \ell$ (using the
               Routh-Hurwitz criterion [15, Chapter XV]). If yes, return
               quadratization defined by $\mathbf{g}(\mathbf{x}), \mathbf{q}_1(\mathbf{x}, \mathbf{y}), \mathbf{q}_2(\mathbf{x}, \mathbf{y}) - \lambda \mathbf{h}(\mathbf{x}, \mathbf{y})$.
               Otherwise, set $\lambda = 2\lambda$     .
```

$\mathbf{y}$ is replaced with $\mathbf{g}(\mathbf{x})$, then $\mathbf{g}, \mathbf{q}_1, \mathbf{q}_2 - \lambda \mathbf{h}$ yield a quadratization of the original system as well. Furthermore, if the algorithm returned at **(Step 3)**b, then this quadratization is dissipative at $\mathbf{x}_1^*, \ldots, \mathbf{x}_\ell^*$.

The termination of the algorithm follows from the proof of Proposition 2. We observe that the constructed $\mathbf{q}_2 - \lambda \mathbf{h}$ is exactly $\mathbf{q}_{2,\lambda}$ in the notation of the proof, and it is shown that there exists λ_0 such that, for every $\lambda > \lambda_0$, $\mathbf{g}, \mathbf{q}_1, \mathbf{q}_{2,\lambda}$ is dissipative at $\mathbf{x}_1^*, \ldots, \mathbf{x}_\ell^*$. Since λ in the algorithm is doubled on each iteration of the while loop, it will at some point exceed λ_0, and the algorithm will terminate.

Example 5. We will illustrate how Algorithm 2 works with the following differential equation:

$$x' = -x(x - a)(x - 2a) \tag{10}$$

where a is a positive scalar parameter. The system's equilibria are $0, a, 2a$, and, among them, $x = 0$ and $x = 2a$ are dissipative. Regardless of the value of a, Algorithm 1 called at **(Step 1)** will produce an inner-quadratic quadratization with one new variable $y = x^2$ and quadratic system:

$$\begin{cases} x' = -xy + 3ax^2 - 2a^2 x, \\ y' = -2y^2 + 6axy - 4a^2 x^2 \end{cases}$$

The stabilizer computed at **(Step 2)** will be $h(x, y) = y - x^2$.

Now let us fix $a = 1$ and continue with **(Step 3)**. At **(Step 3)**a, we form a new quadratic system Σ_λ:

$$\begin{cases} x' = -xy + 3x^2 - 2x, \\ y' = -2y^2 + 6xy - 4x^2 - \lambda(y - x^2) \end{cases}$$

For $\lambda = 1, 2, 4, 8, \ldots$ we check the eigenvalues of its Jacobian at points $(0, 0)$ and $(2, 4)$. The Jacobian of Σ_λ is

$$J = \begin{bmatrix} -y + 6x - 2 & -x \\ 6y + 2\lambda x - 8x & -4y - \lambda + 6x \end{bmatrix}$$

The eigenvalues we get on different iterations of the while-loop are summarized in Table 1 (the ones with nonnegative real parts are bold).

λ	at $(0, 0)$	at $(2, 4)$
1	-2, -1	-2, $\mathbf{3}$
2	-2, -2	-2, $\mathbf{2}$
4	-2, -4	-2, $\mathbf{0}$
8	-2, -8	-2, -4

Table 1: Eigenvalues of the Jacobian of Σ_λ at **(Step 3)**b

From the table we see, that the algorithm will stop and return at $\lambda = 8$. Note that our implementation offers three way of verifying the dissipativity: by computing the eigenvalues directly numerically (with NUMPY) or symbolically (with SYMPY) or by using the Routh-Hurwitz criterion [15, Chapter XV], [4, Chapter 3] (via TBCONTROL package [30]). The numerical evaluation of eigenvalues is the fastest (see Tables 2 and 3) but does not yield fully rigorous guarantees, the other two methods may be slower but provide such guarantees.

As the value of a increases, the original system is more unstable at equilibrium $x_{eq} = 2a$, which requires a larger value of λ in order to make the system dissipative at (x_{eq}, x_{eq}^2). We compute the dissipative quadratization of the system (10) with different values of a and the running time for each method, which is presented in Table 2.

5 Case studies

The code for reproducing the results of the case studies below is available in the "Examples" folder of [1].

a	λ (output)	time (*Numpy*)	time (*Routh-Hurwitz*)	time (*Sympy*)
1	8	33.63	36.89	40.98
5	128	34.04	38.36	39.84
10	512	33.07	41.91	43.66
50	16384	33.38	41.18	54.70
100	65536	34.90	43.06	54.31

Table 2: Output λ value and runtimes (in milliseconds) with different methods for a of the system 10, results were obtained on a laptop Apple M2 Pro CPU @ 3.2 GHz, MacOS Ventura 13.3.1, CPython 3.9.1. Runtime is averaged over 10 executions.

5.1 Application to reachability analysis

The reachability problem is: given an ODE system $\mathbf{x}' = \mathbf{p}(\mathbf{x})$, a set $S \subseteq \mathbb{R}^n$ of possible initial conditions, and a time $t \in \mathbb{R}_{>0}$, compute a set containing the set

$$\{\mathbf{x}(t) \mid \mathbf{x}' = \mathbf{p}(\mathbf{x}) \ \& \ \mathbf{x}(0) \in S\} \subseteq \mathbb{R}^n$$

of all points reachable from S at time t. One recent approach to this problem in the vicinity of a dissipative equilibrium $\mathbf{x}^*$ proposed by Forets and Schilling in [14] is to use Carleman linearization to reduce the problem to the linear case which is well-studied. However, the approach described in [14] relied on explicit bounds available only for quadratic systems under the assumption of dissipativity and weak nonlinearity (see [14, definition 1 and 2]). Algorithm 2 allows this restriction to be relaxed by computing a quadratization which preserves the dissipativity of $\mathbf{x}^*$.

We will illustrate this idea using the *Duffing* equation

$$x'' = kx + ax^3 + bx'$$

which describes a damped oscillator with non-linear restoring force. The equation can be written as a first-order system by introducing $x_1 := x, x_2 := x'$ as follows

$$x_1' = x_2, \quad x_2' = kx_1 + ax_1^3 + bx_2.$$

We take $a = 1, b = -1, k = 1$. Then the system will have three equilibria $\mathbf{x}^* = (0,0), (-1,0), (1,0)$, among which it will be dissipative only at the origin. Algorithm 1 finds an inner-quadratic quadratization for the system using a new variable $y(\mathbf{x}) = x_1^2$ resulting in the following quadratic system:

$$x_1' = x_2, \quad x_2' = ax_1y + bx_2 + kx_1, \quad y' = 2x_1x_2.$$

Obviously, the quadratization is an inner-quadratic quadratization as well. By applying Algorithm 2, we get $\lambda = 1$ with the following dissipative quadratization:

$$\begin{cases} x_1' = x_2 \\ x_2' = x_1y + x_1 - x_2 \\ y' = -y + x_1^2 + 2x_1x_2 \end{cases} \tag{11}$$

For the initial conditions $x_1(0) = 0.1,\ x_2(0) = 0.1,\ y(0) = x_1(0)^2 = 0.01$, system (11) satisfies the requirement of the algorithm from [14]. We apply the algorithm with truncation order $N = 5$ and report the result of the reachability analysis in Figure 2. The grey curve is the computed trajectory and the blue area is an upper bound for the reachable set.

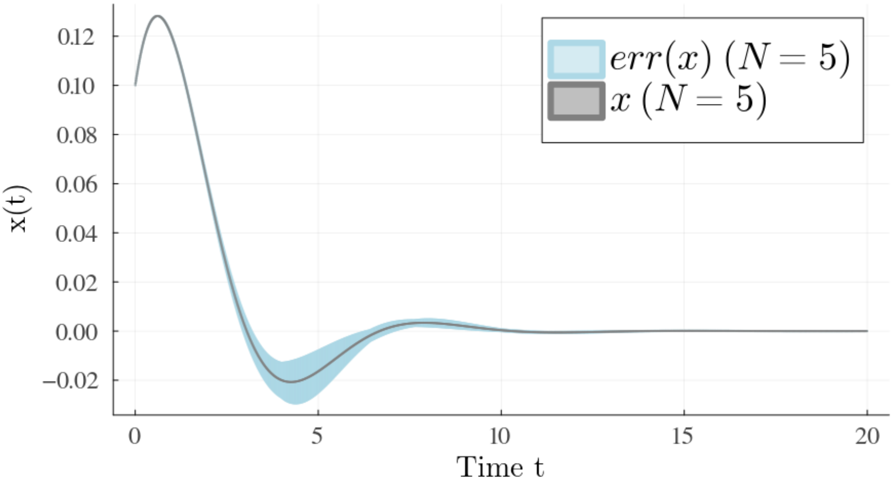


Fig. 2: Reachability analysis results with the computed trajectory (gray) and overapproximation of the reachable set (light blue). Initial condition $\mathcal{X}_0 = [0.1, 0.1, 0.01]$, truncation order $N = 5$, and the estimate reevaluation time $t = 4$ (see [14, Section 6.1]).

5.2 Preserving bistability

An ODE model is called *bistable* (or multistable) if it has at least two stable equilibria. This is a fundamental property for models in life sciences since such a model describes a system that can exhibit a switch-like behaviour, in other words, “make a choice” [12]. One of the smallest possible bistable models arising from a simple chemical reaction network [31, Table 1] is given by the following scalar ODE:

$$x' = k_1 x^2 - k_2 x^3 - k_3 x,$$

where k_1, k_2, k_3 are positive reaction rate constants. The equation has always one dissipative equilibrium at $x = 0$. It has two more equilibria as long as $k_1^2 > 4k_2k_3$, and in this case, the largest of them will be dissipative as well. For any nonzero parameter values, the inner-quadratic quadratization computed by Algorithm 1 will consist of a single new variable $y(x) := x^2$ and the quadratic system:

$$x' = k_1 w - k_2 x w - k_3 x, \quad y' = 2k_1 x y - 2k_2 y^2 - 2k_3 y. \tag{12}$$

For the case-study, we pick $k_1 = 0.4, k_2 = 1, k_3 = 0.03$. For these parameter values, the dissipative equilibria are $x = 0$ and $x = 0.3$, and Algorithm 2 finds that (12) is dissipative at them already. The plot below shows that, indeed, the trajectories of (12) staring in the neighbouthoods of $(0, 0)$ and $(0.3, 0.09)$ converge to the respective equilibria (Fig. 3).

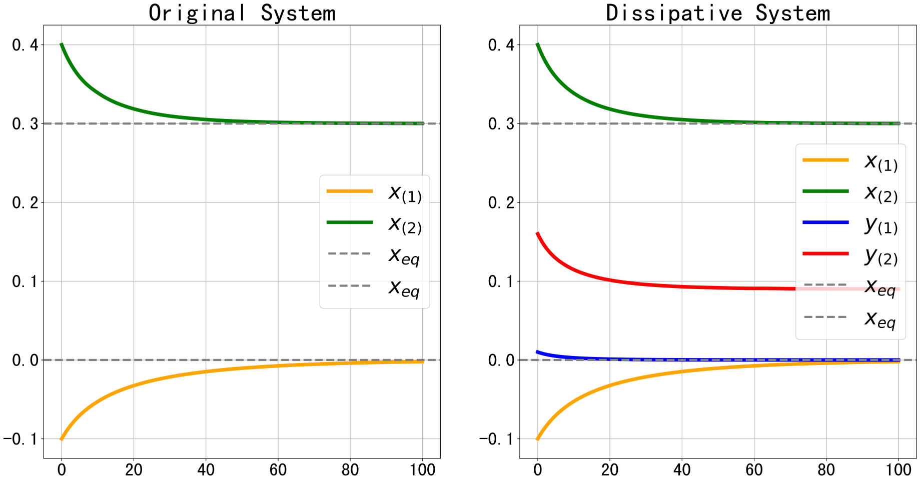


Fig. 3: Plot of the original equation and system (12) with initial state $\mathcal{X}_0 = [x_0, w_0] = [-0.1, 0.01]$ $(x_{(1)}, w_{(1)})$ and $\mathcal{X}_0 = [0.4, 0.16]$ $(x_{(2)}, w_{(2)})$.

5.3 Coupled Duffing oscillators

For a larger example, we will consider an ensemble consisting of Duffing oscillators from Section 5.1 which is an extended version of a pair of coupled oscillators from [3]. The model consisting of n oscillators is parametrized by a number $\delta \in \mathbb{R}$ and a matrix $A \in \mathbb{R}^{n \times n}$, and is defined by the following system:

$$\mathbf{x}'' = A\mathbf{x} - (A\mathbf{x})^3 - \delta\mathbf{x}',$$

where $\mathbf{x} = [x_1, \ldots, x_n]^\top$ are the positions of the oscillators, and $(A\mathbf{x})^3$ is the component-wise cube of vector $A\mathbf{x}$. Similarly to Section 5.1, we can rewrite this as a first-order system of dimension $2n$ by introducing new variables $\mathbf{z} = [z_1, \ldots, z_n]^\top$ for the derivatives of $\mathbf{x}$:

$$\dot{\mathbf{x}} = \mathbf{z}, \quad \mathbf{z}' = A\mathbf{x} - (A\mathbf{x})^3 - \delta\mathbf{z}.$$

Similarly, to [3, Table 1], if the eigenvalues of A are positive real numbers, then this system has 2^n dissipative equilibria. We run our code for $n = 1, \ldots, 8$ taking $\delta = 2$ and A being the tridiagonal matrix with ones on the diagonal and $\frac{1}{3}$ on the adjacent diagonals. Table 3 reports, for each n, the number of introduced variables and the times for computing inner-quadratic quadratization (Algorithm 1) and making it dissipative at all 2^n equilibria (Algorithm 2 using NUMPY for the eigenvalue computation or the symbolics Routh-Hurwitz criterion). We can observe that numerical methods for checking the dissipativity scale well (given that the number of points grows exponentially) while symbolic methods become very costly as the dimension grows.

n	dimension	# equilibria	# new vars	time (inner-quadratic)	time (dissipative)	
					NUMPY	ROUTH-HURWITZ
1	2	2	1	0.02	0.05	0.07
2	4	4	2	0.07	0.19	0.65
3	6	8	4	0.20	0.74	36.57
4	8	16	5	0.39	1.62	1179.33
5	10	32	7	0.72	4.30	> 2000
6	12	64	9	1.20	11.28	> 2000
7	14	128	10	1.75	28.23	> 2000
8	16	256	12	2.63	78.70	> 2000

Table 3: Runtimes (in seconds) for n coupled Duffing oscillators, results were obtained on a laptop with the following parameters: Apple M2 Pro CPU @ 3.2 GHz, MacOS Ventura 13.3.1, CPython 3.9.1.

6 Conclusions

While various quadratization techniques have been used recently in a number of application areas, and in most of the cases this was primarily involving numerical simulations, we are not aware of prior general results on the stability properties of the quadratized systems. In this paper, we studied quadratizations that preserve dissipativity at prescribed equilibria. First, we have shown that, for any set of dissipative equilibria such a quadratization exists. Then we have presented an algorithm capable of computing a quadratization with this property with dimension low enough to be of interest for applications. We showcase the algorithm on several case studies, including examples from reachability analysis and chemical reaction network theory.

The key ingredient of our algorithm is the computation of a quadratization (we call it inner-quadratic) which gives us substantial control over the stability properties of the quadratized system. We expect that this construction will be useful for further research in this direction.

In future research, we plan to extend the results of the paper in different directions. One natural problem is to extend the results and algorithms from the present paper beyond polynomial systems, for example, by designing an algorithm for dissipativity-preserving polynomialization. Additionally, exploring the preservation of other stability properties, such as limit cycles, attractors, and Lyapunov functions, is another promising avenue for research.

References

1. Dissipative-Quadratization Package (2023), https://github.com/yubocai-poly/DQbee
2. Appelroth, G.G.: Основная форма системы алгебраическихъ дифференціальныхъ уравненій. Sbornik: Mathematics **23**(1), 12–23 (1902), http://mi.mathnet.ru/sm6683
3. Balamurali, R., Kengne, L.K., Rajagopal, K., Kengne, J.: Coupled non-oscillatory Duffing oscillators: Multistability, multiscroll chaos generation and circuit realization. Physica A: Statistical Mechanics and its Applications **607**, 128174 (2022), https://doi.org/10.1016/j.physa.2022.128174
4. Bavafa-Toosi, Y.: Introduction to Linear Control Systems. Elsevier (2019). https://doi.org/10.1016/c2016-0-03896-2, http://dx.doi.org/10.1016/C2016-0-03896-2
5. Benner, P., Breiten, T.: Two-sided projection methods for nonlinear model order reduction. SIAM Journal on Scientific Computing **37**(2), B239–B260 (2015)
6. Bychkov, A., Issan, O., Pogudin, G., Kramer, B.: Exact and optimal quadratization of nonlinear finite-dimensional non-autonomous dynamical systems (2023), https://arxiv.org/abs/2303.10285
7. Bychkov, A., Pogudin, G.: Optimal monomial quadratization for ODE systems. In: Flocchini, P., Moura, L. (eds.) Combinatorial Algorithms. pp. 122–136. Springer International Publishing, Cham (2021)
8. Carothers, D.C., Parker, G.E., Sochacki, J.S., Warne, P.G.: Some properties of solutions to polynomial systems of differential equations. Electron. J. Diff. Eqns. **2005**(40), 1–17 (2005), https://eudml.org/doc/125330
9. Carothers, D.C., Parker, G., Sochacki, J.S., Warne, P.G.: Some properties of solutions to polynomial systems of differential equations. Electronic Journal of Differential Equations (EJDE) p. Paper No. 40 (2005), http://eudml.org/doc/125330
10. Carravetta, F.: Global exact quadratization of continuous-time nonlinear control systems. SIAM Journal on Control and Optimization **53**(1), 235–261 (2015), https://doi.org/10.1137/130915418
11. Carravetta, F.: On the solution calculation of nonlinear ordinary differential equations via exact quadratization. Journal of Differential Equations **269**(12), 11328–11365 (2020), https://doi.org/10.1016/j.jde.2020.08.028
12. Craciun, G., Tang, Y., Feinberg, M.: Understanding bistability in complex enzyme-driven reaction networks. Proceedings of the National Academy of Sciences **103**(23), 8697–8702 (2006), https://doi.org/10.1073/pnas.0602767103
13. Fages, F., Le Guludec, G., Bournez, O., Pouly, A.: Strong Turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In: Computational Methods in Systems Biology: 15th International Conference, CMSB 2017, Darmstadt, Germany, September 27–29, 2017, Proceedings 15. pp. 108–127. Springer (2017), https://doi.org/10.1007/978-3-319-67471-1_7
14. Forets, M., Schilling, C.: Reachability of weakly nonlinear systems using Carleman linearization. In: Lecture Notes in Computer Science, pp. 85–99. Springer International Publishing (2021), https://doi.org/10.1007%2F978-3-030-89716-1_6
15. Gantmacher, F.R.: The theory of matrices. Chelsea Publishing Company (1984)
16. Gözükirmizi, C., Demiralp, M.: Solving ODEs by obtaining purely second degree multinomials via branch and bound with admissible heuristic. Mathematics **7**(4) (2019), https://www.mdpi.com/2227-7390/7/4/367

17. Gu, C.: QLMOR: A projection-based nonlinear model order reduction approach using quadratic-linear representation of nonlinear systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **30**(9), 1307–1320 (2011), https://doi.org/10.1109/TCAD.2011.2142184
18. Guillot, L., Cochelin, B., Vergez, C.: A generic and efficient Taylor series–based continuation method using a quadratic recast of smooth nonlinear systems. International Journal for numerical methods in Engineering **119**(4), 261–280 (2019), https://doi.org/10.1002/nme.6049
19. Guillot, L., Cochelin, B., Vergez, C.: A Taylor series-based continuation method for solutions of dynamical systems. Nonlinear Dynamics **98**(4), 2827–2845 (2019), https://doi.org/10.1007/s11071-019-04989-5
20. Hemery, M., Fages, F.: Algebraic biochemistry: A framework for analog online computation in cells. In: Computational Methods in Systems Biology, pp. 3–20 (2022), https://doi.org/10.1007/978-3-031-15034-0_1
21. Hemery, M., Fages, F., Soliman, S.: On the complexity of quadratization for polynomial differential equations. In: Abate, A., Petrov, T., Wolf, V. (eds.) Computational Methods in Systems Biology. pp. 120–140. Springer International Publishing, Cham (2020), https://doi.org/10.1007/978-3-030-60327-4_7
22. Hindmarsh, A.C.: ODEPACK, a systemized collection of ODE solvers. Scientific computing (1983)
23. Hubbard, J.H., West, B.H.: Differential Equations: A Dynamical Systems Approach. High-dimensional Systems. Springer (1995)
24. Kerner, E.H.: Universal formats for nonlinear ordinary differential systems. Journal of Mathematical Physics **22**(7), 1366–1371 (1981)
25. Kramer, B., Willcox, K.: Nonlinear model order reduction via lifting transformations and proper orthogonal decomposition. AIAA Journal **57**(6), 2297–2307 (2019), https://doi.org/10.2514/1.J057791
26. Kramer, B., Willcox, K.: Balanced truncation model reduction for lifted nonlinear systems. In: Beattie, C., Benner, P., Embree, M., Gugercin, S., Lefteriu, S. (eds.) Realization and Model Reduction of Dynamical Systems: A Festschrift in Honor of the 70th Birthday of Thanos Antoulas, pp. 157–174. Springer International Publishing, Cham (2022), https://doi.org/10.1007/978-3-030-95157-3_9
27. Lagutinskii, M.N.: Къ вопросу о простѣ̆йшей формѣ̆ системы обыкновенныхъ дифференцiальныхъ уравненій. Sbornik: Mathematics **27**(4), 420–423 (1911), http://mi.mathnet.ru/sm6583
28. Morrison, D.R., Jacobson, S.H., Sauppe, J.J., Sewell, E.C.: Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. Discrete Optimization **19**, 79–102 (2016), https://doi.org/10.1016/j.disopt.2016.01.005
29. Petzold, L.: Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. SIAM journal on scientific and statistical computing **4**(1), 136–148 (1983)
30. Sandrock, C.: tbcontrol package, https://github.com/alchemyst/Dynamics-and-Control
31. Wilhelm, T.: The smallest chemical reaction system with bistability. BMC Systems Biology **3**(1) (2009), https://doi.org/10.1186/1752-0509-3-90

Forward and Backward Constrained Bisimulations for Quantum Circuits

A. Jiménez-Pastor[1(✉)], K. G. Larsen[1], M. Tribastone[2], and M. Tschaikowski[1]

[1] Aalborg University, Aalborg, Denmark
{ajpa,kgl,tschaikowski}@cs.aau.dk
[2] IMT Lucca, Lucca, Italy
mirco.tribastone@imtlucca.it

Abstract. Efficient methods for the simulation of quantum circuits on classic computers are crucial for their analysis due to the exponential growth of the problem size with the number of qubits. Here we study lumping methods based on bisimulation, an established class of techniques that has been proven successful for (classic) stochastic and deterministic systems such as Markov chains and ordinary differential equations. Forward constrained bisimulation yields a lower-dimensional model which exactly preserves quantum measurements projected on a linear subspace of interest. Backward constrained bisimulation gives a reduction that is valid on a subspace containing the circuit input, from which the circuit result can be fully recovered. We provide an algorithm to compute the constraint bisimulations yielding coarsest reductions in both cases, using a duality result relating the two notions. As applications, we provide theoretical bounds on the size of the reduced state space for well-known quantum algorithms for search, optimization, and factorization. Using a prototype implementation, we report significant reductions on a set of benchmarks. Furthermore, we show that constraint bisimulation complements state-of-the-art methods for the simulation of quantum circuits based on decision diagrams.

Keywords: bisimulation · quantum circuits · lumpability

1 Introduction

Quantum computers can solve certain problems more efficiently than classic computers. Earlier instances are Grover's quantum search [28] and Shor's factorization [47]; more recent works address the efficient solution of linear equations [30] and the simulation of differential equations [36]. Despite its potential and increasing interest from a commercial viewpoint [41], quantum computing is still in its infancy. The number of qubits of current quantum computers is prohibitively small; furthermore, low coherence times and quantum noise lead to high error rates. Further research and improvement of quantum circuits thus hinges on the availability of efficient simulation algorithms on classic computers.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 343–362, 2024.
https://doi.org/10.1007/978-3-031-57249-4_17

Being described by a unitary complex matrix, any quantum circuit can be simulated by means of array structures and the respective matrix operations [32,50,38]. Unfortunately, direct array approaches are subject to the curse of dimensionality [41] because the size of the matrix is exponential in the number of qubits. This motivated the introduction of techniques that try to overcome the exponential growth, resting for instance upon the stabilizer formalism [1], tensor networks [55,56], path sum reductions [3] and decision diagrams [57,41,29].

Here we study bisimulation relations for quantum circuits. Bisimulation has a long tradition in computer science [46]. For the purpose of this paper, the most relevant strand of research on this topic regards bisimulations for quantitative models such as probabilistic bisimulation [34,9]. This is closely related to ordinary lumpability for Markov chains [13], also known as *forward* bisimulation [25], which yields an aggregated Markov chain by means of partitioning the original state space, such that the probability of being in each macro-state/block is equal to the sum of the probabilities of each state in that block. Exact lumpability [13], also known as *backward* bisimulation [49], exploits a specific linear invariant induced by a partition of the state space such that states in the same partition block have the same probability at all time points [13]. In an analogous fashion, forward and backward bisimulations have been developed for chemical reaction networks [16,51,15], rule-based systems [25,24], and ordinary differential equations [14,19].

In all these cases, lumping can be mathematically expressed as a specific linear transformation of the original state space into a reduced one that is induced by a partition. In general, however, lumping allows for arbitrary linear transformation [49,12]. Since this may introduce loss of information, *constrained lumping* allows one to specify a subspace of interest that ought to be preserved in the reduction [53,12,42,31]. In partition-based bisimulations, constraints can be specified as suitable user-defined initial partitions of states for which lumping is computed as their (coarsest) refinement [17,20]. Bisimulation relations for dynamical systems [43,11] and the notions of constrained linear lumping in [53,42,31], instead, can be understood as linear projections (also known as "lumping schemes") into a lower-dimensional system that preserves an arbitrary linear constraint subspace.

The aim of this paper is to boost simulation of quantum circuits via forward- and backward-type bisimulations that can be constrained to subspaces.[3] Analogously to the cited literature, with *forward constrained bisimulation* (FCB) the aim is to obtain a lower-dimensional circuit which (exactly) preserves the behavior of the original circuit on the subset of interest. In *backward constrained bisimulation* (BCB), the reduction is valid on the constraint subspace; in this manner, the original quantum state can be fully recovered from the reduced circuit. Overall, this setting has complementary interpretation with respect to the analysis of a quantum circuit. It is known that a quantum state can only be accessed by means of a quantum measurement, mathematically expressed as

[3] In the following, the simulation of quantum circuits refers to their execution on a classic computer and not to the notion of one-sided bisimulation.

a projection onto a given subspace. FCB, in general, preserves any projection onto the constraint subspace. That is, if the constraint subspace contains the measurement subspace, FCB will exactly preserve the quantum measurement, but the full quantum state cannot be recovered in general. Instead, constraining the invariant set of BCB to contain the input of the circuit ensures that the full circuit result can be recovered from the reduced circuit.

We show that FCB and BCB are related by a duality property stating that a lumping scheme is an FCB if and only if its complex conjugate transpose is a BCB. Interestingly, this is analogous to the duality established between ordinary (forward) and exact (backward) lumpability for Markov chains [20,18], although it does not carry over to other models in general [7,52]. A relevant implication of this result is that one needs only one algorithm to compute both FCB and BCB. As a further contribution of this paper, we present such an algorithm, developed an as adaptation of the CLUE method for the constrained lumping of systems of ordinary differential equations with polynomial right-hand sides [42] of which it inherits the polynomial-time complexity in matrix size.

To show the applicability of our constrained bisimulations, we analyze several case studies for which we report both theoretical and experimental results. Specifically, we first study three classic quantum circuits for search (Grover's algorithm [40, Section 6.2]), optimization [21], and factorization [40, Section 5.3.2], respectively. In Grover's algorithm, the cardinality of the search domain is exponential in the number of qubits; we prove that BCB can always reduce the circuit matrix to a 2×2 matrix while exactly preserving the output of interest. Next, we consider quantum approximate optimization algorithm for solving SAT and MaxCut instances [21]; in this setting, our main theoretical result is an upper bound on the size of the reduced (circuit) matrix by the number of clauses (SAT) or edges (MaxCut). Finally, for quantum factorization we prove that the size of the reduced matrix gives the multiplicative order, that is, solves the order finding problem to which the factorization problem can be reduced [40].

From an experimental viewpoint, using a prototype based on a publicly available implementation of CLUE, we compare the aforementioned theoretical bounds against the actual reductions on a set of randomly generated instances. Moreover, we conduct a large-scale evaluation on common quantum algorithms collected in the repository [44], showing considerable reductions in all cases. Finally, we demonstrate that constrained bisimulation complements state-of-the-art methods for quantum circuit simulation based on decision diagrams [41], as implemented in the tool DDSIM [57].

Further related work. Probabilistic bisimulations [9,5,6] have been considered for quantum extensions of process calculi, see [26,23] and references therein. Similar to their classic counterparts, these seek to identify concurrent (quantum) processes with similar behavior. The current work, instead, is about boosting the simulation of quantum circuits and is in line with [8,18,54]. Specifically, it operates directly over quantum circuits rather than processes and exploits general linear invariants in the (complex) state space. In engineering, invariant-based reductions of linear systems are known under the names of proper orthogonal decomposition [4,39], Krylov methods [4], and dynamic mode decomposition [45,33,27]. Linear invariants describe also safety properties [10] in quantum model checking [59,58], without being used for reduction though. $\mathcal{L}$-bisimulation [12] and [33,27] yield the same reductions, with the difference being that the former obtains the smallest reduction up to a given initial constraint, while the latter computes the smallest reduction up to an initial condition. While relying similarly to us on reduction techniques, [33,27] focus on the reduction of quantum Hamiltonian dynamics, with applications mostly in quantum physics and chemistry. Instead, we study the reduction of quantum circuits which are the prime citizens of quantum computing. Moreover, we provide a prototype implementation of our approach and perform a large-scale numerical evaluation.

Paper outline. The paper is structured as follows. After a review of core concepts, Section 2 introduces forward and backward constrained bisimulation of (quantum) circuits. There, we also provide an algorithm for the computation of constrained bisimulations by extending [42,35] to circuits. Section 3 then derives bounds on the reduction sizes of quantum search [28], quantum optimization [21] and quantum order finding [40]. Section 4, instead, conducts a large-scale evaluation on published quantum benchmarks [44] and compares constrained bisimulations against DDSIM with respect to the possibility of speeding up circuit simulations. The paper concludes in Section 5.

2 Constrained Bisimulations for Quantum Circuits

Notation. We shall denote by n the number of qubits and set $N = 2^n$ for convenience. Column vectors are denoted by the *ket* notation $|z\rangle$, while the complex conjugate transpose of $|z\rangle$ is denoted by $|z\rangle^\dagger = \langle z|$, i.e., $\langle z| = |\bar{z}\rangle^T$ with $\bar{\cdot}$ and $\cdot^T$ denoting complex conjugation and transpose, respectively. In a similar vein, $\langle z| \, |z\rangle = \langle z|z\rangle$, where $\langle \cdot \mid \cdot \rangle$ is the standard scalar product over $\mathbb{C}^N$. Following standard notation, the canonical basis vectors of $\mathbb{C}^N$ are expressed using tensor products and bit strings $x \in \{0,1\}^n$; specifically, writing $\otimes$ for the Kronecker product, we have $|x_n\rangle \otimes |x_{n-1}\rangle \otimes \ldots \otimes |x_1\rangle = |x_n\rangle |x_{n-1}\rangle \ldots |x_1\rangle = |x_n x_{n-1} \ldots x_1\rangle = |d\rangle$, where $0 \leq d \leq 2^n - 1$ is a decimal representation of x, see [40] for details. We usually denote by $|x\rangle$ canonical basis vectors with $x \in \{0,1\}^n$, whereas $|u\rangle, |v\rangle, |w\rangle, |z\rangle \in \mathbb{C}^N$ refer to linear combinations in the form $|z\rangle = \sum_{x \in \{0,1\}^n} c_x |x\rangle$ with $c_x \in \mathbb{C}$. For any canonical basis vector, we have $|x\rangle = |\bar{x}\rangle$. To avoid confusion, forward constrained bisimulations (FCB) are denoted by row matrices $L \in \mathbb{C}^{d \times N}$ with $d \leq N$, while backward constrained bisimulations (BCB) are denoted by column matrices $L^\dagger \in \mathbb{C}^{N \times d}$.

Preliminaries. We begin by introducing core concepts from linear algebra and quantum computing [37,40].

Definition 1 (Core Concepts).

- *The column space of a matrix M are all linear combinations of its columns and is denoted by $\langle M \rangle_c$. One says, the columns of M span $\langle M \rangle_c$.*

- *The row space of a matrix is the set of all linear combinations of its rows and is denoted by $\langle M \rangle_r$. One says, the rows of M span $\langle M \rangle_r$.*
- *A (quantum) circuit over n qubits is described by a unitary map $U \in \mathbb{C}^{N \times N}$, that is, $U^{-1} = U^{\dagger}$.*
- *A (quantum) state $|z\rangle \in \mathbb{C}^N$ is a vector with (Euclidian) norm one.*
- *A matrix $P \in \mathbb{C}^{N \times N}$ is an orthogonal projection if $P \circ P = P = P^{\dagger}$.*
- *Any vector $|z\rangle \in \mathbb{C}^N$ generates the linear subspace $S_{|z\rangle} = \langle |z\rangle \rangle_c$.*

Throughout the paper, we do not work at the higher level where quantum circuits are defined by means of a quantum gate compositions [40]. Instead, we work directly at the level of the unitary maps that are induced by such compositions. With this in mind, we use the terms “unitary map” and “quantum circuit” interchangeably.

We distinguish between one- and multi-step applications of a quantum circuit [40]. For an input state $|w_0\rangle \in \mathbb{C}^N$, the full quantum state after *one-step* application is $U|w_0\rangle$. Instead, the full quantum state after a *multi-step* application is given by $U^k|w_0\rangle$, where $k > 1$ is the number of steps.

These definitions justify interpreting a quantum circuit as a discrete-time dynamical system as follows.

Definition 2 (Dynamical System). *A circuit $U \in \mathbb{C}^{N \times N}$ with input state $|w_0\rangle$ induces the discrete time dynamical system (DS) $|w_{k+1}\rangle = U|w_k\rangle$, with $k \geq 0$. We call $|w_k\rangle$ the* full quantum state *at step k.*

Example 1 *The one-qubit circuit $U = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ is known as the Pauli X-gate [40]. In the case of $k \geq 1$ steps and input $|w_0\rangle = |\phi\rangle$, where $|\phi\rangle = (1, -1)/\sqrt{2}$, the induced DS can be shown to be $|w_k\rangle = (-1)^k|\phi\rangle$.*

The result of a quantum computation is not directly accessible and is usually queried using quantum measurements [40]. These can be described by projective measurements [40], formally given by a family of orthogonal projections $\{P_1, \ldots, P_m\}$ satisfying $P_1 + \ldots + P_m = I$. When a quantum state $|z\rangle \in \mathbb{C}^N$ is measured, the probability of outcome $1 \leq i \leq m$ is $\pi_i = \langle z| P_i |z\rangle$. In case of outcome i, the quantum state after the measurement is $P_i|z\rangle/\sqrt{\pi_i}$. We will be mostly concerned with the case $\{P, I - P\}$ for a given orthogonal projection P.

Often, one is interested in querying states from a specific subspace S. For instance, the result of the HHL algorithm [30], considered in Section 4, is stored in a subset of all qubits, i.e., in a subspace. To this end, it suffices to use a projective measurement identifying S.

Definition 3. *Given an orthogonal projection P, we call $P|z\rangle$ the P-measurement of $|z\rangle$. A subspace $S \subseteq \mathbb{C}^N$ is identifiable by P if $P|z\rangle = |z\rangle$ for all $|z\rangle \in S$.*

A particularly simple yet useful class of projective measurements are those that measure a single state $|w\rangle$, i.e., identify the space $S_{|w\rangle}$ spanned by $|w\rangle$. This is given by the orthogonal projection $P_{|w\rangle} := |w\rangle\langle w|$.

Example 2 *Assume that we are interested in measuring the result of Example 1 using measurement $P_{|\phi\rangle}$ that identifies $S_{|\phi\rangle}$. Then, for $|w_0\rangle = \binom{1}{0}$, it holds that $P_{|\phi\rangle}|w_k\rangle = (-1)^k|\phi\rangle/\sqrt{2}$.*

Forward Constrained Bisimulation. We next introduce FCB.

Definition 4 (Forward Constrained Bisimulation, FCB). *Fix a circuit defined by $U \in \mathbb{C}^{N \times N}$ with initial state $|w_0\rangle$ and a matrix $L \in \mathbb{C}^{d \times N}$ with orthonormal rows.*

a) The DS reduced by L is given by $|\hat{w}_{k+1}\rangle = \hat{U} |\hat{w}_k\rangle$, where $\hat{U} = LUL^\dagger$, and initial state $|\hat{w}_0\rangle = L |w_0\rangle$.
b) L is called forward constrained bisimulation of DS $|w_{k+1}\rangle = U |w_k\rangle$ wrt constraint subspace $S \subseteq \mathbb{C}^N$ when $S \subseteq \langle L^\dagger \rangle_c$ and $L |w_k\rangle = |\hat{w}_k\rangle$ for all $k \geq 1$.

Before commenting on the definition, we establish the following.

Lemma 1. *The reduced map $\hat{U}$ in Definition 4 is unitary.*

Proof. See proof of Theorem 2.

We remark that the reduction holds for any choice of initial state $|w_0\rangle$, analogously to the aforementioned forward-type bisimulations [14,18] for (real-valued) dynamical systems. The assumption of orthonormality of rows of L implies that $d \leq N$, i.e., L is a transformation onto a possibly smaller-dimensional state space. Although it can be dropped without loss of generality [42], it allows for a more immediate relation to projective measurements. Indeed, matrix L induces the orthogonal projection P_L defined as $P_L = L^\dagger L$. This projective measurement identifies S because $S \subseteq \langle L^\dagger \rangle_c$. Moreover, $P_L |w_k\rangle$ is preserved in the reduced system for any k. To see this, it suffices to multiply $L |w_k\rangle = |\hat{w}_k\rangle$ by $L^\dagger$ from the left and to note that this yields $P_L |w_k\rangle = L^\dagger |\hat{w}_k\rangle$.

Example 3 *Continuing Example 1, it can be shown that the 2×1 matrix $L = |\phi\rangle^\dagger = (1, -1)/\sqrt{2}$ is an FCB wrt $S_{|\phi\rangle}$. Indeed, since $\hat{U} = LUL^\dagger = -1$, we obtain $|\hat{w}_{k+1}\rangle = -|\hat{w}_k\rangle$, while a direct calculation confirms that $L |w_0\rangle = |\hat{w}_0\rangle$ implies $L |w_k\rangle = |\hat{w}_k\rangle$ for all $k > 0$. Multiplying both sides by $L^\dagger$ from the left yields $L^\dagger L U^k |w_0\rangle = (-1)^k L^\dagger L |w_0\rangle$. Consequently, the $P_{|\phi\rangle}$-measurement of the original map can be obtained from the $P_{|\phi\rangle}$-measurement of the reduced map.*

Algorithm 1 adapts the algorithm for (real-valued) systems of ordinary differential equations with polynomial derivatives developed in [42,35] to the complex domain and yields the minimal FCB wrt subspace S, i.e., it returns an orthonormal $L \in \mathbb{C}^{d \times N}$ whose dimension d is minimal.

Theorem 1 (Minimal FCB). *Algorithm 1 computes a minimal FCB $L \in \mathbb{C}^{d \times N}$ wrt subspace S, i.e., the rowspace of any FCB L' wrt S contains that of L. The complexity of Algorithm 1 is polynomial in N.*

Proof. See proof of Theorem 2.

We briefly comment on Algorithm 1. The idea behind it exploits that L can be shown to be an FCB whenever $L^\dagger$ is an invariant set of the map U, that is, if the column space of $L^\dagger U$ is contained in the column space of $L^\dagger$. The

Algorithm 1 Computation of an FCB L wrt subspace S

```
Require: Unitary map $U \in \mathbb{C}^{N \times N}$ and subspace $S \subseteq \mathbb{C}^N$.
 1: compute orthonormal basis of $S$, store it in column matrix $L^\dagger \in \mathbb{C}^{N \times d_0}$
 2: repeat
 3:    for all columns $|z\rangle$ of $L^\dagger$ do
 4:       compute $|\pi\rangle = P_L U |z\rangle$
 5:       if $|\pi\rangle \neq U |z\rangle$ then
 6:          $|w\rangle = U |z\rangle - |\pi\rangle$
 7:          append column $|w\rangle \, / \, \langle w|w\rangle$ to $L^\dagger$
 8:       end if
 9:    end for
10: until no columns have been appended to $L^\dagger$
11: return matrix $L^{\dagger\dagger}$.
```

algorithm begins by initializing $L^\dagger$ with a basis of S in line 1. This ensures that S is contained in the column space of the final result. For every column $|z\rangle$ of $L^\dagger$, the main loop in line 2 checks whether $U|z\rangle$ is in the column space of $L^\dagger$ (line 5) by computing its projection $|\pi\rangle$ onto the column space of $L^\dagger$ (line 4). If it is not in the column space, the projection will differ from $U|z\rangle$ and the residual $|w\rangle$ must be added to $L^\dagger$. This shows the correctness, while the minimality of FCB L follows from the fact that only the necessary residuals are being added to $L^\dagger$. The complexity of the algorithm, instead, follows by noting that at most N columns can be added to $L^\dagger$ and that all computations of the main loop require, similarly the computation in line 1, at most $\mathcal{O}(N^3)$ operations.

Remark 1. As can be noticed in Algorithm 1, e.g., line 4, the computation of an FCB subsumes the computation of a single step of the circuit. For practical applications to single-step circuits where the modeler is interested in only a single input, FCB may be as expensive as simulating the original circuit directly. Hence, it is obvious that that the effectiveness of constrained bisimulations is particularly relevant when simulating the circuit with respect to several inputs, or when considering multi-step applications. Examples of this are provided in Section 3 and a numerical evaluation is carried out in Section 4.

Example 4 *Consider the FCB $L = |\phi\rangle^\dagger$ wrt subspace $S_{|\phi\rangle}$ from Example 3. Then, noting that $(I - P_L)\,|\phi\rangle = 0$, we infer that Algorithm 1 terminates in Line 5. Hence, L is a minimal FCB wrt $S_{|\phi\rangle}$.*

Backward Constrained Bisimulation. BCB yields a reduced system through the identification of an invariant set, i.e., a subspace S such that $U^k|z\rangle \in S$ for any $|z\rangle \in S$ and $k \geq 1$. Whereas in FCB the reduced model can recover projective measurements onto the constraint set *for any* initial set, here one can recover the full quantum state, so long as the initial states belong to the invariant set.

Definition 5 (Backward Constrained Bisimulation, BCB). *Let U, L and $\hat{U}$ be as in Definition 4. Then, $L^\dagger$ is a BCB of the dynamical system $|w_{k+1}\rangle =$*

$U\,|w_k\rangle$ wrt a subspace of inputs $S \subseteq \mathbb{C}^N$ when $S \subseteq \langle L^\dagger \rangle_c$ and whenever $|w_0\rangle = L^\dagger\,|\hat{w}_0\rangle$ implies $|w_k\rangle = L^\dagger\,|\hat{w}_k\rangle$ for all $k \geq 1$.

Similarly to FCB, we assume without loss of generality that $L \in \mathbb{C}^{d\times N}$ has orthonormal rows. As anticipated above, FCB and BCB are not comparable in general. Indeed, an FCB L makes no assumption on the initial condition $|w_0\rangle$, while a BCB $L^\dagger$ does so by requiring $L^\dagger L\,|w_0\rangle = |w_0\rangle$. Conversely, a BCB $L^\dagger$ allows one to obtain $|w_k\rangle$, while an FCB L allows to obtain $L\,|w_k\rangle$ instead of $|w_k\rangle$ itself.

Example 5 *Fix $|\phi\rangle = (1,-1)^T/\sqrt{2}$ from Example 4 and recall that $L = |\phi\rangle^\dagger$, $U\,|\phi\rangle = -\,|\phi\rangle$ and $\hat{U} = -1$. Then, $L^\dagger$ is a BCB of U wrt $S_{|\phi\rangle}$. Indeed, $L^\dagger L\,|w_0\rangle = |w_0\rangle$ implies $|w_0\rangle = |\phi\rangle$, while*

$$L^\dagger\,|\hat{w}_k\rangle = (-1)^k L^\dagger\,|\hat{w}_0\rangle = (-1)^k L^\dagger L\,|w_0\rangle = (-1)^k\,|w_0\rangle = U^k\,|\phi\rangle = |w_k\rangle\,.$$

Example 5 anticipates the next result that states FCB and BCB are dual notions. This generalizes the known duality of ordinary and exact lumpability of Markov chains [20,18].

Theorem 2 (Duality). *Fix a unitary map $U \in \mathbb{C}^{N\times N}$ and a subspace $S \subseteq \mathbb{C}^N$. L is an FCB wrt S if and only if $L^\dagger$ is a BCB wrt S.*

Proof. Let $S_0 \subseteq S$ be a basis of some fixed $S \subseteq \mathbb{C}^N$. We first note that the discussion of [42,31,35] and [45,4] can be extended to the complex field in a direct manner. With this, we obtain:

1. $L \in \mathbb{C}^{d\times N}$ is an FCB wrt S if and only if $\langle LU\rangle_r \subseteq \langle L\rangle_r$ with $\langle S_0^\dagger\rangle_r \subseteq \langle L\rangle_r$.
2. $D \in \mathbb{C}^{N\times d}$ is a BCB wrt S if and only if $\langle UD\rangle_c \subseteq \langle D\rangle_c$ with $\langle S_0\rangle_c \subseteq \langle D\rangle_c$.

Moreover, we observe the following:

$$\begin{aligned}
\langle LU\rangle_r \subseteq \langle L\rangle_r &\Leftrightarrow [U \text{ bijection}]\\
\langle LU\rangle_r = \langle L\rangle_r &\Leftrightarrow [\text{daggering}]\\
\langle U^\dagger L^\dagger\rangle_c = \langle L^\dagger\rangle_c &\Leftrightarrow [U \text{ unitary}]\\
\langle U^{-1} L^\dagger\rangle_c = \langle L^\dagger\rangle_c &\Leftrightarrow [U \text{ bijection}]\\
\langle L^\dagger\rangle_c = \langle UL^\dagger\rangle_c &\Leftrightarrow [U \text{ bijection}]\\
\langle UL^\dagger\rangle_c \subseteq \langle L^\dagger\rangle_c &
\end{aligned}$$

This yields Theorem 2, i.e., $L \in \mathbb{C}^{d\times N}$ is an FCB of U wrt constraint S if and only if $L^\dagger \in \mathbb{C}^{N\times d}$ is a BCB of U wrt S (because $S_0^{\dagger\dagger} = S_0$). Moreover, if $L^\dagger$ is computed by Algorithm 1, then $L^\dagger$ is a BCB wrt S, while $L^{\dagger\dagger}$ is an FCB wrt S. This follows by noting that in such a case $L^\dagger \in \mathbb{C}^{N\times d}$ satisfies

$$\begin{aligned}
\langle L^\dagger\rangle_c &= \langle U^k\,|z\rangle \mid 0 \leq k \leq N-1, |z\rangle \in S\rangle_c\\
&= \langle U^k\,|z\rangle \mid 0 \leq k \leq N-1, |z\rangle \in S_0\rangle_c
\end{aligned}$$

The complexity follows from the discussion after Theorem 1. A detailed complexity discussion can be obtained in [42]. Exploiting that an FCB L satisfies $LUL^\dagger L = LU$ by [53], we obtain

$$(LUL^\dagger)^\dagger(LUL^\dagger) = (LU^\dagger L^\dagger)(LUL^\dagger) = LU^\dagger UL^\dagger = LL^\dagger = I_{d\times d},$$

showing that $\hat{U}$ is unitary. □

In light of the above result, we often speak of a (constrained bisimulation) reduction. Moreover, we note that Theorem 2 ensures that a BCB reduction up to input yields an FCB reduction up result, a discussed next.

Remark 2. Let $L^\dagger$ be the BCB of U wrt $S_{|w_0\rangle}$, where $|w_0\rangle$ is the input. Then, $L = L^{\dagger\dagger}$ is an FCB wrt $S_{|w_0\rangle}$, implying that $P_L = L^\dagger L$ identifies the column space of $L^\dagger$, see discussion after Definition 4. At the same time, result $U^k\,|w_0\rangle$ is in the column span of $L^\dagger$ because $U^k\,|w_0\rangle = |w_k\rangle = L^\dagger\,|\hat{w}_k\rangle = L^\dagger\hat{U}L\,|w_0\rangle$.

We end the section by pointing out that, thanks to Theorem 2, Algorithm 1 can be used to compute a minimal BCB $L^\dagger$ wrt subspace S. Indeed, the only difference is that one should return $L^\dagger$ rather than $L^{\dagger\dagger}$ in the last line of the algorithm. With this change, we notice that Algorithm 1 coincides, in the case of a one dimensional subspace $S \subseteq \mathbb{C}^N$, with the Krylov subspace [4] that can be obtained by the Arnoldi iteration [45].

3 Applications

In this section we demonstrate that established quantum algorithms enjoy substantial bisimulation reductions. For each application, we provide a brief description of the quantum algorithm and a theoretical bound on its reduction.

3.1 Quantum Search

Let us assume we are given a non-zero function $f : \{0,1\}^n \to \{0,1\}$ and that we are asked to find some $x \in \{0,1\}^n$ such that $f(x) = 1$. Grover's seminal algorithm describes how this can be achieved in $\mathcal{O}(\sqrt{N})$ steps on a quantum computer [40, Section 6.2], thus yielding a quadratic speed-up over a classic computer. For any $x \in \{0,1\}^n$, the Grover map is given by

$$G\,|x\rangle = (-1)^{f(x)}(I - 2\,|\psi\rangle\,\langle\psi|)\,|x\rangle\,, \quad \text{with} \quad |\psi\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle \tag{1}$$

The Grover map yields the following celebrated result.

Theorem 3 (Quantum Search [40]). *Map G is unitary. Moreover, if the number of sought solutions $M = |\{x \mid f(x) = 1\}|$ satisfies $M \leq N/2$, then measuring $G^\kappa\,|\psi\rangle$ for $\kappa = \lceil \frac{\pi}{4}\sqrt{N/M}\rceil$ yields a state $|x\rangle$ satisfying $f(x) = 1$ with probability at least $\frac{1}{2}$.*

The next result allows one to compute result $G^\kappa |\psi\rangle$ from Theorem 3 using a map over a single qubit.

Theorem 4 (Reduced Grover). *The BCB $L^\dagger \in \mathbb{C}^{N\times d}$ of G wrt $S_{|\psi\rangle}$ has dimension $d = 2$ and a column space spanned by $|\psi\rangle$ and $G|\psi\rangle$.*

Proof. The claim follows by noting that the column space of an BCB wrt $S_{|\psi\rangle}$ is spanned by $|\psi\rangle, G|\psi\rangle, G^2|\psi\rangle, \ldots, G^{N-1}|\psi\rangle$ and so on. This, in turn, is known to have as basis [40, Section 6.2]

$$|\alpha\rangle = \frac{1}{\sqrt{M}} \sum_{x:f(x)=1} |x\rangle \qquad \text{and} \qquad |\beta\rangle = \frac{1}{\sqrt{N-M}} \sum_{x:f(x)=0} |x\rangle,$$

where M is as above, while $|\alpha\rangle$ is the superposition (i.e., sum) of all solution states and $|\beta\rangle$ is the superposition of all non-solution states. □

Remark 3. While the BCB $L^\dagger$ wrt $S_{|\psi\rangle}$ has always dimension 2, its column space depends on the oracle function f. This is because f appears in G, see (1).

3.2 Quantum Optimization

Quantum approximate optimization algorithm (QAOA) [21] is a computational model that has the same expressive power as the common quantum circuit model [22,21,2]. It is described by two matrices. The first one is the *problem Hamiltonian* H_P for which we are interested to compute a maximal eigenstate, i.e., an eigenvector for a maximal eigenvalue of H_P. The second is the *begin Hamiltonian* H_B for which a maximal eigenstate $|\psi\rangle$ is known already. With this, a maximal eigenstate of H_P can be obtained by conducting the QAOA introduced next.

Definition 6 (QAOA [21]). *For a problem Hamiltonian H_P and a begin Hamiltonian H_B, fix the unitary matrices*

$$U_B(\delta) = \exp(-i\delta H_B) \qquad \textit{and} \qquad U_P(\delta) = \exp(-i\delta H_P)$$

where $\delta > 0$ is a sufficiently small time step and $\exp(A)$ *is the matrix exponential. For a sequence of natural numbers $(k_i, l_i)_{i=1}^{\kappa}$ of length $\kappa \geq 1$, we define*

$$|w_\kappa\rangle = U_B(\delta)^{k_\kappa} U_P(\delta)^{l_\kappa} \cdot \ldots \cdot U_B(\delta)^{k_1} U_P(\delta)^{l_1} |\psi\rangle \tag{2}$$

The QAOA with $\kappa \geq 1$ stages is then given by $\max\{\langle w_\kappa| H_P |w_\kappa\rangle \mid (k_i, l_i)_{i=1}^{\kappa}\}$.

While the problem Hamiltonian H_P depends on the task or problem we are solving, the choice of the begin Hamiltonian H_B is informed by the so-called adiabatic theorem, a result that identifies conditions QAOA returns a global optimum. A common heuristic is to pick H_B such that H_B and H_P do not diagonalize over a common basis [22,21] and to assume without loss of generality that $|\psi\rangle = \sum_x |x\rangle / \sqrt{N}$ is the unique maximal eigenvector of H_B.

We next demonstrate bisimulation can be reduce QAOA when it is applied to SAT and MaxCut, two NP-complete problems [48]. We start by introducing the problem Hamiltonians H_P for both cases.

Definition 7 (SAT and MaxCut Problem Hamiltonians).

- *For a boolean formula $\phi = \bigwedge_{i=1}^{M} C_i$, where C_i is a clause over n boolean variables, the problem Hamiltonian is given by $H_P = \sum_i H_i$, where*

$$H_i \left|x\right\rangle = \begin{cases} \left|x\right\rangle & , C_i(x) \text{ is true} \\ 0 & , C_i(x) \text{ is false} \end{cases}$$

  for any $x \in \{0,1\}^n$ representing a boolean assignment.
- *For an undirected unweighted graph $G = (V, E)$ with vertices $V = \{1, \ldots, n\}$ and edges $E \subseteq V \times V$, we define the problem Hamiltonian $H_P = \sum_{(i,j)\in E} H_{i,j}$, where*

$$H_{i,j} \left|x\right\rangle = \begin{cases} \left|x\right\rangle & , x_i \neq x_j \\ 0 & , x_i = x_j \end{cases}$$

  for any $x \in \{0,1\}^n$ that represents a cut $C \subseteq \{1, \ldots, n\}$ by setting $i \in C$ if and only if $x_{i-1} = 1$.

Following this definition, it can be shown that the QAOA $\left\langle w_\kappa\right| H_P \left|w_\kappa\right\rangle$ from Definition 6 corresponds to a quantum measurement reporting either the expected number of satisfied clauses or the expected size of the cut. It is possible to guarantee that QAOA finds a global optimum for a sufficiently high κ [22,21].

The next result ensures that H_P has BCB $L^\dagger$ wrt $S_{|\psi\rangle}$ whose reduced map is provably small. Moreover, for any such L, it ensures that there exists a begin Hamiltonian H_B for which $L^\dagger$ is a BCB too, thus ensuring substantial reductions of the entire QAOA calculation (2).

Theorem 5 (Reduced QAOA). *Fix H_P as in Definition 7, any $\delta > 0$ and let $L^\dagger \in \mathbb{C}^{N\times d}$ be a BCB of $U_P(\delta)$ wrt $S_{|\psi\rangle}$. Then*

1. *The column space of $L^\dagger$ is spanned by*

$$\big(\left|\psi\right\rangle, U_P(\delta)\left|\psi\right\rangle, U_P^2(\delta)\left|\psi\right\rangle, \ldots, U_P^{M-1}(\delta)\left|\psi\right\rangle \big)^\dagger, \tag{3}$$

   where M is the number of clauses (SAT) or edges (MaxCUT). Specifically, the dimension of the BCB d is bounded by M.
2. *Then, for any Hamiltonian $\hat{H}_B \in \mathbb{C}^{d\times d}$ (i.e., Hermitian matrix), there is a Hamiltonian $H_B \in \mathbb{C}^{N\times N}$ such that*
   - *$L^\dagger$ is a BCB of $U_B(\delta) = \exp(-i\delta H_B)$ wrt $S_{|\psi\rangle}$, while its reduced map is $\hat{U}_B(\delta) = \exp\left(-i\delta \hat{H}_B\right)$*
   - *The computation (2) satisfies*

$$\begin{aligned} \left|w_\kappa\right\rangle &= U_B(\delta)^{k_\kappa} U_P(\delta)^{l_\kappa} \cdot \ldots \cdot U_B(\delta)^{k_1} U_P(\delta)^{l_1} \left|\psi\right\rangle \\ &= L^\dagger \hat{U}_B(\delta)^{k_\kappa} \hat{U}_P(\delta)^{l_\kappa} \cdot \ldots \cdot \hat{U}_B(\delta)^{k_1} \hat{U}_P(\delta)^{l_1} L \left|\psi\right\rangle \end{aligned} \tag{4}$$

   The QAOA in $\mathbb{C}^N$ thus corresponds to a QAOA in the reduced space $\mathbb{C}^d$.

Proof. We begin by proving 1. For SAT, it can be noticed that $H_P |x\rangle = \nu |x\rangle$, where $0 \leq \nu \leq M$ is the number of clauses that are satisfied by assignment x. A similar formula holds for MaxCut, with the difference being that ν is the size of the cut x. It is worth noting that H_P is in diagonal form for both SAT and MaxCut. If m denotes the number of distinct eigenvalues of H_P, then $m \leq M$, where M is in the case of SAT or MaxCUT, respectively, the number of clauses or edges. The same can be said concerning its matrix exponential $U_P(\delta)$ which, being unitary, enjoys an eigendecomposition, allowing us to write $|\psi\rangle = \sum_{i=1}^{m} c_i |z_i\rangle$, where $|z_i\rangle$ is an eigenvector for eigenvalue λ_i of $U_P(\delta)$. This yields

$$U^k |\psi\rangle = \sum_{i=1}^{m} c_i \lambda_i^k |z_i\rangle$$

for all $k \geq 0$. Without lost of generality, consider $d \leq m$ such that $c_k = 0$ for all $k > d$ and $c_k \neq 0$ otherwise. Writing vectors $\{U^k |\psi\rangle \mid 0 \leq k \leq m-1\}$ in basis $|z_1\rangle, \ldots, |z_d\rangle$ gives rise to a regular Vandermonde matrix [45] in $\mathbb{C}^{d\times d}$. This shows that $\{U^k |\psi\rangle \mid d \leq k \leq M-1\}$ are linear combinations of $\{U^k |\psi\rangle \mid 0 \leq k \leq d-1\}$, completing the proof of 1. Instead, 2. follows from the definition of BCB and Lemma 2 from below. □

The auxiliary result below is needed in the proof of Theorem 5.

Lemma 2. *Pick any $L \in \mathbb{C}^{d\times N}$ and $Q \in \mathbb{C}^{(N-d)\times N}$ so that the rows of L and Q comprise an orthonormal basis of $\mathbb{C}^N$, and define*

$$U_B = L^\dagger \hat{U}_B L + Q^\dagger \tilde{U}_B Q, \qquad \hat{U}_B = \exp\Big(-i\delta \hat{H}_B\Big), \qquad \tilde{U}_B = \exp\Big(-i\delta \tilde{H}_B\Big)$$

for any Hamiltonian $\hat{H}_B \in \mathbb{C}^{d\times d}$ and $\tilde{H}_B \in \mathbb{C}^{(N-d)\times(N-d)}$. Then, U_B is unitary, L is an FCB of it wrt $S_{|\psi\rangle}$, and $\hat{U}_B$ is its reduced map. Further, there exists a Hamiltonian $H_B \in \mathbb{C}^{N\times N}$ satisfying $U_B = \exp(-i\delta H_B)$.

Proof. We first show that $LU_B = LU_B L^\dagger L$ as this implies that L is an FCB of U by [53]. To see this, we note that

$$LU_BL^\dagger L = L\big(L^\dagger \hat{U}_B L + Q^\dagger \tilde{U}_B Q\big)L^\dagger L = LL^\dagger \hat{U}_B LL^\dagger L + LQ^\dagger \tilde{U}_B QL^\dagger L = \hat{U}_B L$$
$$LU_B = L\big(L^\dagger \hat{U}_B L + Q^\dagger \tilde{U}_B Q\big) = LL^\dagger \hat{U}_B L + LQ^\dagger \tilde{U}_B Q = \hat{U}_B L$$

where we have used that $LL^\dagger = 0$ and $LQ^\dagger = 0$, which follows from the choice of Q. From the calculation, we can also infer that $\hat{U}_B = LU_BL^\dagger$, i.e., $\hat{U}_B$ in indeed the reduced map. In a similar fashion, one can note that Q is also an FCB of U_B and that $\tilde{U}_B$ is the respective reduced map. Since both $\hat{U}_B$ and $\tilde{U}_B$ are unitary, we infer that also U_B is unitary (alternatively, a direct calculation yields $I = U_B U_B^\dagger$). Since any unitary matrix can be written as a matrix exponential of a Hamiltonian, there exists a Hamiltonian H_B satisfying $U_B = \exp(-i\delta H_B)$. □

3.3 Quantum Factorization and Order Finding

Let us assume that we are given a composite number N which we seek to factorize. As argued in [40, Section 5.3.2], this problem can be solved in randomized polynomial time, provided the same holds true for the order finding problem. Given some randomly picked $x \in \{2, 3 \ldots, N-1\}$, the latter asks to compute the multiplicative order of x modulo N, i.e., the smallest $r \geq 1$ satisfying $x^r \mod N = 1$. Following [40, Section 5.3.1], we consider the quantum algorithm defined by the unitary map

$$U\left|y\right\rangle = \begin{cases} \left|xy \mod N\right\rangle & , \ 0 \leq y < N \\ \left|y\right\rangle & , \ N \leq y < 2^l \end{cases}$$

Here, $l \geq 1$ is the smallest number satisfying $N \leq 2^l$.

The next result allows us to relate the order of x to the dimension of the BCB wrt $S_{|1\rangle}$. This fact is exploited in Shor's factorization algorithm [40].

Theorem 6 (Reduced Order Finding). *The dimension of the BCB of U wrt $S_{|1\rangle}$ coincides with the order of x modulo N.*

Proof. If can be shown [40] that the U from above is unitary and that $U\left|u_s\right\rangle = e^{2\pi is/r}\left|u_s\right\rangle$ for all $0 \leq s \leq r-1$, where

$$\left|u_s\right\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} \exp\left[\frac{-2\pi isk}{r}\right]\left|x^k \mod N\right\rangle \qquad \text{and} \qquad \frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}\left|u_s\right\rangle = \left|1\right\rangle.$$

With this, $U^k\left|1\right\rangle = \frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}(e^{2\pi is/r})^k u_s$ for any $p \geq 0$. Hence, the minimal BCB with respect to $S_{|1\rangle}$ is contained in the span of $u_0, \ldots, u_{r-1}$. To see that the dimension is exactly r, we note that vectors $\{U^k\left|\psi\right\rangle \mid 0 \leq k \leq r-1\}$, written in basis $u_0, \ldots, u_{r-1}$, constitute a regular Vandermonde matrix [45] in $\mathbb{C}^{r\times r}$. □

4 Numerical Experiments

We evaluate our approach on the applications from Section 3 and the quantum benchmark repository [44]. The approach has been implemented in Python by extending the publicly available implementation of the CLUE algorithm from [42,35]. The prototype is accessible at https://www.doi.org/10.5281/zenodo.8431443. All results reported were executed on a machine with i7-8665U CPU, 32GB RAM and 1024GB SSD. The reduced circuits of CLUE were simulated using Python's `numpy` libraries. All simulations using quantum circuits were done with `qiskit` 0.44.1 and DDSIM simulations were performed with `mqt.ddsim` version 1.19.0. All libraries are available using the default `pip` command in Python.

In our prototype, we have implemented Algorithm 1 by changing in CLUE the domain of definition from the real numbers to complex numbers and, instead of using Gaussian elimination [42,31] for deciding membership properties, we used orthogonal projections.

4.1 Applications from Section 3

Here we report the results of numerical experiments on the applications discussed in Section 3 and a comparison against DDSIM. For this, we considered circuits ranging from 5 to 15 qubits and fixed a timeout of 500 seconds. To allow for a representative evaluation, we averaged runtimes over 5 independent runs of Grover's circuit; instead, in case of quantum optimization, we averaged over 50 independent runs because SAT formulas and graphs were picked randomly in each run. Specifically, the instances were generated as follows:

- *Grover algorithm* (Sec. 3.1): following the convention of [44], we set up a search function f where $f(x) = 0$ for all $x \in \{0,1\}^n$ except for $f(11...1) = 1$. This can be realized via an oracle using the Toffoli gate.
- *Quantum Optimization for SAT* (Sec. 3.2): for each number of qubits n, we generate a random formula with m clauses (m is randomly picked between n and $3n$), where each clause has 3 variables at most. We guarantee that every formula contains all n variables.
- *Quantum Optimization for MaxCut* (Sec. 3.2): for each number of qubits n, we generate an Erdős-Rényi graph with n nodes and edge probability $\frac{1}{3}$.

For Grover's algorithm the experiments confirmed the two-dimensional bisimulation reduction theoretically demonstrated in Theorem 4. Instead, for quantum optimization we measured the (average) achieved reduction against the theoretical bounds developed in Theorem 5. For the comparison against DDSIM, we measured DDSIM's wallclock execution time for each circuit instance against CLUE's corresponding end-to-end runtime consisting of both computing the constrained bisimulation and simulating the reduced circuit. For quantum optimization, the number of steps κ was set to the smallest integer greater or equal to $\sqrt{N}$. The choice of κ is motivated by Theorem 3 and the discussion around the so-called adiabatic theorem in [22,21].

Discussion. For quantum optimization, Table 1 reports logarithmic CLUE reductions, reducing in particular $2^{15} \times 2^{15}$ matrices to 15×15 matrices in less than 4 s. Overall, DDSIM was faster than CLUE in case of Grover, while CLUE outperformed DDSIM on quantum optimization. We explain this by the diagonal form of the quantum optimization circuit. The results for quantum optimization and Grover confirm the observation made in Remark 1 that CLUE reductions may be practically useful in multi-step applications.

4.2 Benchmark Circuits

In this section, we report a numerical evaluation of the quantum benchmarks from [44], available at https://www.cda.cit.tum.de/mqtbench/. For each number $0 \leq x \leq N-1$, we computed $U\,|x\rangle$ by computing the bisimulation wrt subspace $S_{|x\rangle}$ and the respective reduced circuit; we report only circuit families which could be reduced, which were 9 out of 17. As before, we used 5 instances for each model and a timeout of 500 s; in the computation of the average reduction dimension d across all subspaces $S_{|x\rangle}$, a timeout was reached when the computation across all N subspaces $S_{|x\rangle}$ took more than 500 s.

	Grover		**SAT**			**MaxCut**		
qubits	DDSIM	CLUE	DDSIM	CLUE	d	DDSIM	CLUE	d
5	0.292	0.482	0.313	0.001	4.93/15	0.162	0.001	4.28/20
6	0.109	2.271	0.529	0.002	5.51/18	0.368	0.001	5.33/30
7	0.184	7.254	2.267	0.006	6.83/21	0.645	0.002	7.15/42
8	0.272	22.787	5.417	0.014	7.11/24	1.128	0.003	9.05/56
9	0.431	111.920	20.319	0.031	8.77/27	3.873	0.006	10.61/72
10	0.896	369.531	232.948	0.072	9.15/30	6.069	0.013	13.13/90
11	1.262	>500	>500	0.147	9.74/33	105.713	0.027	15.26/110
12	1.574	>500	>500	0.361	10.92/36	287.671	0.059	18.24/132
13	2.431	>500	>500	0.738	11.63/39	442.855	0.114	20.62/156
14	3.583	>500	>500	1.496	11.74/42	>500	0.232	24.24/182
15	5.452	>500	>500	3.232	13.08/45	>500	0.528	26.21/210

Table 1: Comparison of simulation times between DDSIM and the reduced model by CLUE. The latter includes the runtimes for computing the bisimulations by Algorithm 1. For SAT and MaxCut, column d reports the average size of the reduced circuit and its theoretical bounds from Theorem 5, separated by backslash.

Table 2 differentiates between reduction dimension wrt subspace $S_{|0\rangle}$ and the average reduction dimension across all subspaces $S_{|x\rangle}$. This is because the former can be interpreted as a BCB since $|0\rangle$ is the default input for most quantum circuits. Instead, the latter is meant to study the average reduction power of FCB, since FCB preserves quantum measurements. We remark that some circuits were only available for specific number of qubits (e.g., HHL and price calls). The reduction ratio d/N is given by the quotient between the dimension of the reduction dimension d and $N = 2^n$ (unlike Table 1 no bounds on d were available).

Overall, it can be noticed that substantial reductions could be obtained for a number of benchmark families. However, given that the benchmarks from Table 2 are all single-step applications, DDSIM was consistently faster than CLUE, once again confirming the observation from Remark 1.

5 Conclusion

We introduced forward and backward constrained bisimulations for quantum circuits which allow by means of reduction to preserve an invariant subspace of interest. The applicability of the approach was demonstrated by obtaining substantial reductions of common quantum algorithms, including, in particular, quantum search, quantum approximate optimization algorithms for SAT and MaxCut, as well as a number of benchmark circuits. Overall, the results suggest that constrained bisimulation can be used as a tool for speeding up the simulation of quantum circuits on classic computers, complementing state-of-

Circuit name	#-qubits	$\frac{d}{N}$ wrt $S_{\lvert 0\rangle}$	Avg. $\frac{d}{N}$ across $S_{\lvert x\rangle}$	Avg. time (s)	DDSIM time
Deutsch-Jozsa	3	50.00%	47.22%	0.046	0.019
	4	25.00%	24.26%	0.226	0.021
	5	12.50%	12.31%	1.127	0.023
	6	6.25%	6.20%	5.294	0.024
	7	3.12%	**TO**	**TO**	0.026
GHZ	3	75.00%	69.44%	0.025	0.088
	4	87.50%	83.08%	0.339	0.070
	5	50.00%	48.67%	1.802	0.073
	6	25.00%	24.66%	5.766	0.078
	7	12.50%	**TO**	**TO**	0.082
Graph State	3	50.00%	66.67%	0.073	0.102
	4	25.00%	23.23%	0.242	0.086
	5	25.00%	28.98%	2.636	0.093
	6	9.38%	10.96%	8.981	0.104
	7	6.25%	**TO**	**TO**	0.116
HHL algorithm	5	12.50%	78.79%	1.874	0.032
Pricing Call Option	5	25.00%	27.27%	0.256	0.564
	7	12.50%	**TO**	**TO**	0.736
	9	6.25%	**TO**	**TO**	0.996
Pricing Put Option	5	25.00%	27.27%	0.256	0.564
	7	12.50%	**TO**	**TO**	0.801
	9	6.25%	**TO**	**TO**	1.207
QFT	3	25.00%	41.67%	0.041	0.115
	4	12.50%	22.79%	0.159	0.108
	5	6.25%	11.93%	1.312	0.130
	6	3.12%	6.11%	5.971	0.157
	7	1.56%	**TO**	**TO**	0.184
Quantum Walk	3	75.00%	65.00%	0.026	0.123
	4	50.00%	45.83%	0.150	0.331
	5	50.00%	47.43%	0.968	0.739
	6	50.00%	48.58%	6.885	0.762
	7	50.00%	**TO**	**TO**	0.784
Travelling Salesman	4	87.50%	87.50%	1.014	0.178
	9	**TO**	**TO**	**TO**	0.316

Table 2: Evaluation of (single-step) quantum benchmarks from repository [44]. The simulation times of DDSIM refer to the computation with respect to input $|0\rangle$, while the third and fourth columns report dimensions of bisimulation reductions. Instead, the fifth column reports the average computation time of $U\,|x\rangle$ via a reduction wrt $S_{|x\rangle}$, including the computation time of the bisimulation. A cumulative timeout of 500 s is denoted by **TO**.

the-art methods based on decision diagrams especially when the circuit is to be simulated under several initial conditions or for *multi-step* applications.

In line with the relevant literature on bisimulations for dynamical systems, constrained bisimulations introduce loss of information due to their underlying projection onto a smaller dimensional state space; the information that is preserved, however, is exact. A relevant issue for future work is to consider approximate variants of bisimulation for quantum circuits, in order to find more aggressive reductions or to capture quantum-specific phenomena such as quantum noise. Another line of research considers the combination with complementary circuit simulation approaches, in particular those based on decision diagrams.

Acknowledgments This work was partially supported by the Poul Due Jensen Foundation grant 883901, the Villum Investigator Grant S4OS and the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

1. Aaronson, S., Gottesman, D.: Improved simulation of stabilizer circuits. Physical Review A **70**(5), 052328 (2004)
2. Aharonov, D., van Dam, W., Kempe, J., Landau, Z., Lloyd, S., Regev, O.: Adiabatic quantum computation is equivalent to standard quantum computation. In: 45th IEEE Symposium on Foundations of Computer Science. pp. 42–51 (2004)
3. Amy, M.: Towards large-scale functional verification of universal quantum circuits. In: Selinger, P., Chiribella, G. (eds.) QPL. vol. 287, pp. 1–21 (2018)
4. Antoulas, A.: Approximation of Large-Scale Dynamical Systems. Advances in Design and Control, SIAM (2005)
5. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: Complete axiomatization for the bisimilarity distance on markov chains. In: Desharnais, J., Jagadeesan, R. (eds.) CONCUR. LIPIcs, vol. 59, pp. 21:1–21:14 (2016)
6. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: A complete quantitative deduction system for the bisimilarity distance on markov chains. Log. Methods Comput. Sci. **14**(4) (2018)
7. Bacci, G., Bacci, G., Larsen, K.G., Tribastone, M., Tschaikowski, M., Vandin, A.: Efficient local computation of differential bisimulations via coupling and up-to methods. In: Symposium on Logic in Computer Science, LICS. pp. 1–14 (2021)
8. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: The bisimdist library: Efficient computation of bisimilarity distances for markovian models. In: Joshi, K.R., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST. pp. 278–281 (2013)
9. Baier, C., Hermanns, H.: Weak bisimulation for fully probabilistic processes. In: CAV. pp. 119–130 (1997)
10. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
11. Boreale, M.: Algebra, coalgebra, and minimization in polynomial differential equations. Log. Methods Comput. Sci. **15**(1) (2019)
12. Boreale, M.: Complete algorithms for algebraic strongest postconditions and weakest preconditions in polynomial odes. Sci. Comput. Program. **193**, 102441 (2020)
13. Buchholz, P.: Exact and ordinary lumpability in finite Markov chains. Journal of Applied Probability **31**(1), 59–75 (1994)

14. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Maximal aggregation of polynomial dynamical systems. Proceedings of the National Academy of Sciences **114**(38), 10029 – 10034 (2017)
15. Cardelli, L., Pérez-Verona, I.C., Tribastone, M., Tschaikowski, M., Vandin, A., Waizmann, T.: Exact maximal reduction of stochastic reaction networks by species lumping. Bioinform. **37**(15), 2175–2182 (2021)
16. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Forward and backward bisimulations for chemical reaction networks. In: CONCUR. pp. 226–239 (2015)
17. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Comparing chemical reaction networks: A categorical and algorithmic perspective. In: Symposium on Logic in Computer Science, LICS. pp. 485–494 (2016)
18. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Symbolic computation of differential equivalences. In: POPL. pp. 137–150 (2016)
19. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Guaranteed error bounds on approximate model abstractions through reachability analysis. In: QEST. pp. 104–121 (2018)
20. Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal state-space lumping in Markov chains. Information Processing Letters **87**(6), 309 – 315 (2003)
21. Farhi, E., Goldstone, J., Gutmann, S.: A quantum approximate optimization algorithm. arXiv preprint arXiv:1411.4028 (2014)
22. Farhi, E., Goldstone, J., Gutmann, S., Sipser, M.: Quantum computation by adiabatic evolution. arXiv preprint quant-ph/0001106 (2000)
23. Feng, Y., Duan, R., Ji, Z., Ying, M.: Probabilistic bisimulations for quantum processes. Information and Computation **205**(11), 1608–1639 (2007)
24. Feret, J., Danos, V., Krivine, J., Harmer, R., Fontana, W.: Internal coarse-graining of molecular systems. Proceedings of the National Academy of Sciences **106**(16), 6453–6458 (2009)
25. Feret, J., Henzinger, T., Koeppl, H., Petrov, T.: Lumpability abstractions of rule-based systems. Theoretical Computer Science **431**(0), 137 – 164 (2012)
26. Gay, S.J., Nagarajan, R.: Communicating quantum processes. In: POPL. p. 145–157 (2005)
27. Goldschmidt, A., Kaiser, E., DuBois, J.L., Brunton, S.L., Kutz, J.N.: Bilinear dynamic mode decomposition for quantum control. New Journal of Physics **23**(3), 033035 (2021)
28. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Righth Annual ACM Symposium on Theory of computing. pp. 212–219 (1996)
29. Grurl, T., Fuß, J., Hillmich, S., Burgholzer, L., Wille, R.: Arrays vs. decision diagrams: A case study on quantum circuit simulators. In: IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL). pp. 176–181 (2020)
30. Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. Physical Review Letters **103**(15), 150502 (2009)
31. Jiménez-Pastor, A., Jacob, J.P., Pogudin, G.: Exact Linear Reduction for Rational Dynamical Systems, pp. 198–216. Springer International Publishing (2022)
32. Khammassi, N., Ashraf, I., Fu, X., Almudever, C.G., Bertels, K.: Qx: A high-performance quantum computer simulation platform. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. pp. 464–469. IEEE (2017)
33. Kumar, A., Sarovar, M.: On model reduction for quantum dynamics: symmetries and invariant subspaces. Journal of Physics A: Mathematical and Theoretical **48**(1), 015301 (2014)

34. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. **94**(1), 1–28 (1991)
35. Leguizamon-Robayo, A., Jiménez-Pastor, A., Tribastone, M., Tschaikowski, M., Vandin, A.: Approximate Constrained Lumping of Polynomial Differential Equations, pp. 106–123. Springer Nature Switzerland (2023)
36. Liu, J.P., Kolden, H.Ø., Krovi, H.K., Loureiro, N.F., Trivisa, K., Childs, A.M.: Efficient quantum algorithm for dissipative nonlinear differential equations. Proceedings of the National Academy of Sciences **118**(35) (2021)
37. Meyer, C.D.: Matrix Analysis and Applied Linear Algebra. SIAM (2001)
38. Murali, P., McKay, D.C., Martonosi, M., Javadi-Abhari, A.: Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 1001–1016 (2020)
39. Nielsen, A.E., Hopkins, A.S., Mabuchi, H.: Quantum filter reduction for measurement-feedback control via unsupervised manifold learning. New Journal of Physics **11**(10), 105043 (2009)
40. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press (2000)
41. Niemann, P., Wille, R., Miller, D.M., Thornton, M.A., Drechsler, R.: Qmdds: Efficient quantum function representation and manipulation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(1), 86–99 (2016)
42. Ovchinnikov, A., Pérez Verona, I., Pogudin, G., Tribastone, M.: CLUE: exact maximal reduction of kinetic models by constrained lumping of differential equations. Bioinformatics **37**(19), 3385–3385 (08 2021)
43. Pappas, G.J., Lafferriere, G., Sastry, S.: Hierarchically consistent control systems. IEEE Trans. Automat. Contr. **45**(6), 1144–1160 (2000)
44. Quetschlich, N., Burgholzer, L., Wille, R.: MQT Bench: Benchmarking software and design automation tools for quantum computing (2022), MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/
45. Rowley, C.W., Mezič, I., Bagheri, S., Schlatter, P., Henningson, D.S.: Spectral analysis of nonlinear flows. Journal of Fluid Mechanics **641**, 115–127 (2009)
46. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (2011)
47. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Review **41**(2), 303–332 (1999)
48. Sipser, M.: Introduction to the theory of computation. ACM SIGACT News **27**(1), 27–29 (1996)
49. Sproston, J., Donatelli, S.: Backward bisimulation in Markov chain model checking. Software Engineering, IEEE Transactions on **32**(8), 531–546 (Aug 2006)
50. Steiger, D.S., Häner, T., Troyer, M.: Projectq: an open source software framework for quantum computing. Quantum **2**, 49 (2018)
51. Tognazzi, S., Tribastone, M., Tschaikowski, M., Vandin, A.: EGAC: a genetic algorithm to compare chemical reaction networks. In: Bosman, P.A.N. (ed.) GECCO. pp. 833–840. ACM (2017)
52. Tognazzi, S., Tribastone, M., Tschaikowski, M., Vandin, A.: Backward Invariance for Linear Differential Algebraic Equations. In: CDC. pp. 3771–3776 (2018)
53. Tomlin, A.S., Li, G., Rabitz, H., Tóth, J.: The effect of lumping and expanding on kinetic differential equations. SIAM Journal on Applied Mathematics **57**(6), 1531–1556 (1997)
54. Tschaikowski, M., Tribastone, M.: Spatial fluid limits for stochastic mobile networks. Perform. Evaluation **109**, 52–76 (2017)

55. Vidal, G.: Efficient classical simulation of slightly entangled quantum computations. Phys. Rev. Lett. **91**, 147902 (Oct 2003)
56. Villalonga, B., Boixo, S., Nelson, B., Henze, C., Rieffel, E., Biswas, R., Mandrà, S.: A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. npj Quantum Information **5**(1), 86 (2019)
57. Wille, R., Hillmich, S., Burgholzer, L.: Tools for quantum computing based on decision diagrams. ACM Transactions on Quantum Computing **3**(3) (jun 2022)
58. Ying, M., Feng, Y.: Model checking quantum systems - A survey (2018), arxiv
59. Ying, M., Li, Y., Yu, N., Feng, Y.: Model-checking linear-time properties of quantum systems. ACM Trans. Comput. Logic **15**(3) (2014)

A Parallel and Distributed Quantum SAT Solver Based on Entanglement and Teleportation

Shang-Wei Lin[1](✉), Tzu-Fan Wang[2], Yean-Ru Chen[2], Zhe Hou[3], David Sanán[4], and Yon Shin Teo[5]

[1] School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore
Shangweilin@gmail.com
[2] Department of Electrical Engineering, National Cheng Kung University, Tainan City, Taiwan
[3] School of Information and Communication Technology, Griffith University, Brisbane, Australia
[4] InfoComm Technology Cluster, Singapore Institute of Technology, Singapore, Singapore
[5] Continental Automotive, Singapore, Singapore

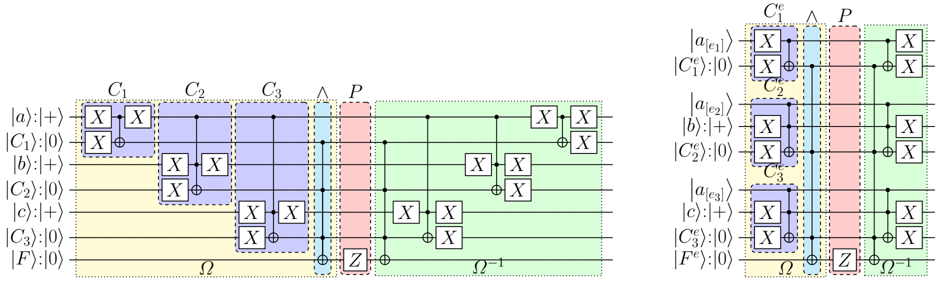
Abstract. Boolean satisfiability (SAT) solving is a fundamental problem in computer science. Finding efficient algorithms for SAT solving has broad implications in many areas of computer science and beyond. Quantum SAT solvers have been proposed in the literature based on Grover's algorithm. Although existing quantum SAT solvers can consider all possible inputs at once, they evaluate each clause in the formula one by one sequentially, making the time complexity $O(m)$, linear to the number of clauses m, *per Grover iteration*. In this work, we develop a *parallel* quantum SAT solver, which reduces the time complexity in each iteration to constant time $O(1)$ by utilising extra entangled qubits. To further improve the scalability of our solution in case of extremely large problems, we develop a distributed version of the proposed parallel SAT solver based on quantum teleportation such that the total qubits required are shared and distributed among a set of quantum computers (nodes), and the quantum SAT solving is accomplished collaboratively by all the nodes. We prove the correctness of our approaches and evaluate them in simulations and real quantum computers.

1 Introduction

Boolean satisfiability (SAT) solving is a fundamental problem in classical computing. Given a propositional formula, SAT determines whether there are truth assignments for propositional variables making the formula true. SAT has many applications: theorem proving, model checking, software/hardware verification, optimization, scheduling, etc. In addition, SAT is central in the computation and complexity theories as it is NP-complete. Finding efficient algorithms for SAT solving has broad implications for many areas of computer science and beyond.

Quantum computing generalizes classical computing from binary bits to quantum bits (qubits), which could represent both 0's and 1's simultaneously in superpositions. Another advantage of quantum computers is their innate ability to execute all the possible computational paths simultaneously, known as quantum parallelism. Qubits can become entangled with each other, which is a strictly

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 363–382, 2024.
https://doi.org/10.1007/978-3-031-57249-4_18



(a) Conventional (sequential) oracle

(b) Parallel oracle

Fig. 1: Different oracles for formula $\mathcal{F}$.

quantum mechanical phenomenon with no classical analogue, and is also a computing resource enabling quantum computers to achieve *quantum supremacy* over their classical counterparts. These properties of quantum computing lead to substantial speed-up compared to certain classical computing algorithms.

In quantum computing, Grover's algorithm [23] is able to search for targets (e.g., satisfying assignments in SAT) in a huge search space with a *quadratic* speed-up compared to classical searching algorithms. Applying it to SAT problems has significant theoretical and practical implications. Although the quadratic speedup still yields an exponential time complexity for SAT solving, it is a more systematic improvement than heuristics. The real-world benefit compared to modern algorithms such as CDCL [28] is hard to measure now. Grover's algorithm has two essential components: (1) an *oracle*, and (2) the *diffuser*. The oracle answers the "yes/no" question about whether an object in the search space is the target we are looking for. The diffuser tries to maximize the probability of obtaining the targets in the search when measuring the qubits. In a nutshell, if one wants to use Grover's algorithm for a search problem, the key is to provide the oracle. As long as the oracle can correctly identify the targets in the search space, the diffuser, which is standard and independent from the search problem, can help to "extract" the targets. Example 1 gives a running example:

Example 1. Consider the following Boolean formula $\mathcal{F}$ over three Boolean variables, where $a = 1, b = 1, c = 1$ is the only assignment that makes $\mathcal{F}$ true.

$$\mathcal{F} : (a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$$

To solve the SAT problem of formula $\mathcal{F}$ by Grover's algorithm, Fernandes et al. [21] proposed an oracle, as shown in Fig. 1a, where the C_1 (cyan) block processes the first clause (a), the C_2 block processes the second clause $(\overline{a} \vee b)$, and C_3 processes the third clause $(\overline{a} \vee c)$. Even though the three variables a, b, c are put, respectively, in the $|+\rangle$ superposition state, i.e., $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, to consider all possible inputs at once, the oracle still needs to process each clause one by one sequentially because variable a appears in all the three clauses, and thus the clauses have data dependency. Theoretically, this sequential oracle takes $O(m)$

time complexity, where m is the number of clauses. The readers need not worry about the technical detail here, as it will be briefly introduced in Section 2.2.

In this work, we propose a quantum oracle that processes each clause in parallel, as shown in Fig. 1b, which brings a significant improvement in time complexity from linear time $O(m)$ to constant time $O(1)$. One can observe that the circuit depth in Fig. 1b is much shorter than that in Fig. 1a, which implies a shorter execution time [14]. The idea behind our approach is the strategy of "trade space for time". We use two additional qubits for variables a so that each clause C_i has its own variable $a_{[e_i]}$ for $i \in \{1, 2, 3\}$, which makes each clause able to be processed independently in parallel, as the three cyan blocks C_1^e, C_2^e, C_3^e in Fig. 1b. However, the values of the three variables $a_{[e_1]}$, $a_{[e_2]}$, $a_{[e_3]}$ cannot be arbitrary. They must have the same value as they represent the (single) value of variable a in the formula $\mathcal{F}$. Here comes an interesting question: how do we make sure that the three variables always have the same value? The answer is *entanglement*! If we prepare for the three variables the following entangled state

$$\left|a_{[e_1]}\right\rangle \left|a_{[e_2]}\right\rangle \left|a_{[e_3]}\right\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle),$$

then their values will be all 1 with $\frac{1}{2}$ probability or all 0 with $\frac{1}{2}$ probability, which captures the exact semantics when solving formula $\mathcal{F}$. The technical details about the proposed parallel oracle and its corresponding diffuser are introduced in Section 3. To the best of our knowledge, this is the first work that proposes a parallel quantum SAT solving technique based on entanglement.

The proposed parallel SAT solver gains the improvement in time complexity by paying more (entangled) qubits. What if the SAT problem is extremely complex and requires substantial resources? In such a scenario, *distributed quantum computing* [17, 18, 24], adopting the strategy of "divide and conquer", emerges as a sub-branch of quantum computing. Analogously, we develop a distributed version of our parallel SAT solver. In this distributed version, the total qubits required are shared and distributed among a set of quantum computers (nodes), and the quantum SAT solving is accomplished collaboratively by all the nodes involved based on quantum teleportation [9, 11, 33, 36]. The technical details of the proposed distributed quantum SAT solver is introduced in Section 4. To the best of our knowledge, this is also the first work that proposes a distributed quantum SAT solving technique based on quantum teleportation.

2 Preliminaries

We assume that the readers have basic knowledge in quantum computing, e.g., the *tensor product* operation, primitive quantum gates (e.g., X, Z, H, etc), and *quantum entanglement*. We use the *ket* notation $|\cdot\rangle$ to denote the (column) vector representing a quantum state. Given two vectors $|v_1\rangle$ and $|v_2\rangle$, we use $|v_1\rangle \otimes |v_2\rangle$ to denote their tensor product, which may be abbreviated as $|v_1\rangle |v_2\rangle$, or even $|v_1 v_2\rangle$ for simplicity. When applying an operation on a vector $|v\rangle$, we use $|v'\rangle$ to denote the state of $|v\rangle$ after the operation, or $|v\rangle_t$ to denote the state of $|v\rangle$ at step t during the operation, where $t \in \mathbb{N}$.

2.1 Grover's algorithm

Grover's algorithm [23] is one of the most well-known quantum algorithms. It is used to solve the search problem for finding M target elements in an unsorted database with N elements ($M < N$). Grover's algorithm is widely used in many applications, such as cryptography [22], pattern matching [38], etc.

The two main operations of it are *phase inversion* and *inversion about the average*, which are handled by the oracle and diffuser, respectively. Initially, the input will be placed in superposition ($|x\rangle$) to evaluate all elements in the database at once. Next, the oracle function U_f considers all the possible inputs and marks the target element by applying phase inversion, i.e., $U_f|x\rangle = (-1)^{f(x)}|x\rangle$, in which $f(x) = 1$ for the target element and $f(x) = 0$ for the others. After the target element is marked, the diffuser applies the *inversion about the average* operation to amplify the probability of the target element so that one can obtain the result by measurement. In order to achieve the optimal/maximum probability for the target element to be measured, the two operations (called a Grover iteration) need to be repeated for $(\pi/4)\sqrt{N/M}$ iterations. The oracle is problem-dependent, while the diffuser is not. Thus, designing the correct oracle is the key to applying Grover's algorithm. Usually, the number of target elements is unknown before the search, but there are several ways to resolve this issue. The most common one is to apply quantum counting [12] to obtain the (approximate) number of target elements before using Grover's algorithm. It is a quadratic speed-up compared with classical methods requiring $O(N)$ operations.

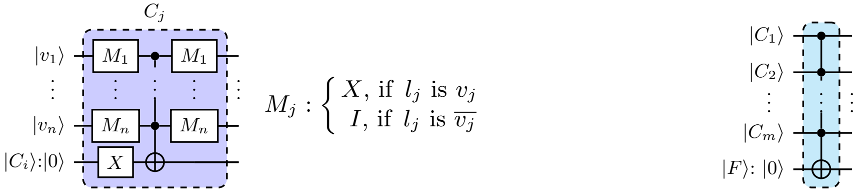
2.2 Conventional Quantum SAT Solving

Consider the following syntax for SAT formulas in conjunctive normal form (CNF) over a set of Boolean variables V:

$$F \simeq C_1 \wedge C_2 \wedge \cdots \wedge C_m \qquad C \simeq l_1 \vee l_2 \vee \cdots \vee l_n \qquad l \simeq v \mid \overline{v}$$

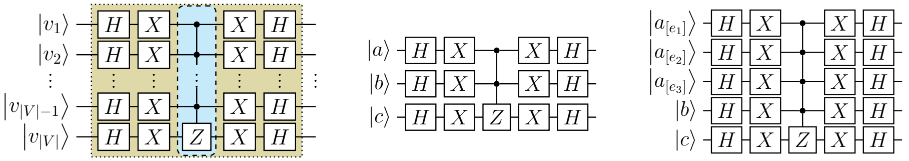
A formula F is a conjunction of m clauses $C_1, C_2, \ldots, C_m$, and each clause C_i is a disjunction of n literals $l_1, l_2, \ldots, l_n$, where $m, n \in \mathbb{N}$. A literal l_j could be a Boolean variable v and called a *positive* literal, or the negation of a Boolean variable $\overline{v}$ and called a *negative* literal. A formula can be viewed as a function $F : \{0,1\}^{|V|} \mapsto \{0,1\}$ mapping an input vector $\boldsymbol{v} \in \{0,1\}^{|V|}$ to true/false (0/1), where $|V|$ denotes the cardinality of V. A formula F is *satisfiable* if there exists some $\boldsymbol{v} \in \{0,1\}^{|V|}$ such that $F(\boldsymbol{v}) = 1$, and we call such $\boldsymbol{v}$ a *solution* to F. A formula F is *unsatisfiable* if it does not have any solution. We do not include Boolean constants true/false in the syntax as they can be rewritten as $(v \vee \overline{v})$ and $(v \wedge \overline{v})$, respectively, and are usually eliminated before SAT solving.

Given a formula $F : C_1 \wedge C_2 \wedge \ldots \wedge C_m$, to apply Grover's algorithm, an oracle for F is required. The construction of the quantum circuit for the conventional oracle follows the bottom-up approach [21,26]. The circuit for each clause C_i is constructed first, and then all the clauses are conjuncted together. Fig. 2a shows how to construct the circuit for each clause $C_i : l_1 \vee l_2 \vee \ldots \vee l_n$, where the M_j


C_j
$|v_1\rangle$
$|v_n\rangle$
$|C_i\rangle$:$|0\rangle$
M_1
M_n
X
$M_j : \begin{cases} X, \text{ if } l_j \text{ is } v_j \\ I, \text{ if } l_j \text{ is } \overline{v_j} \end{cases}$
$|C_1\rangle$
$|C_2\rangle$
$|C_m\rangle$
$|F\rangle$: $|0\rangle$

(a) Clause $C_i : l_1 \vee \cdots \vee l_n$ (b) Formula F

Fig. 2: The quantum circuit construction scheme for classic oracle.



(a) General diffuser scheme. (b) A diffuser for $\mathcal{F}$. (c) Wrong Diffuser

Fig. 3: Classic Diffuser.

gate depends on literal l_j for $j \in \{1, 2, \ldots, n\}$. If l_j is positive, M_j is the X gate, while if l_j is negative, M_j is the I gate. The qubit $|C_i\rangle$ represents the truth value of clause C_i. Once the quantum circuits for all the m clauses are constructed, they are conjuncted by a CNOT gate (m-qubit Toffoli gate, to be more precise) to form the circuit for F, as shown in Fig. 2b, where $|F\rangle$ represents the truth value of formula F, which is controlled by $|C_i\rangle$ for all $i \in \{1, 2, \ldots, m\}$. Fig. 1a shows the conventional oracle for formula $\mathcal{F} : (a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$. The Ω block is constructed as mentioned previously to identify the solutions of formula $\mathcal{F}$. The P gate is used to give a "-1" phase to those solutions, and the Ω^{-1} block is the inverse operation of Ω to restore each input vector to its initial value for the following diffusion process.

The purpose of the diffuser is to amplify the amplitude of the solution vectors to increase/maximize the probability of them being measured. Fortunately, the diffusion process is independent of the input problems, i.e., different problems can share a general-purpose diffuser design. Fig. 3a shows a commonly used diffuser [23]. Fig. 3b shows the diffuser for formula $\mathcal{F} : (a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$, which has a same overall structure of the general scheme.

3 Parallel Quantum SAT Solver

In this section, we introduce how to parallelize a quantum SAT solver to speed up the SAT solving process. Section 3.1 introduces the proposed parallel oracle using entanglement, and Section 3.2 introduces the corresponding parallel diffuser. Note that, due to the page limit, we give all the proofs in the author version [27].

3.1 Parallel Oracle

Let V be a set of Boolean variables and $F : C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be a Boolean CNF formula over V with m clauses, where $m \in \mathbb{N}$. If a variable $v \in V$ is shared by k clauses in F where $k \in \mathbb{N}$, we call it a *shared* variable. For formula F, we define its *expanded* formula with respect to v, denoted by F^e_v, obtained by replacing each occurrence of variable v with a *(fresh) expanded* variable $v_{[e_i]}$ where $i \in \{1, 2, \ldots, k\}$ and $v_{[e_1]} = v$. Since $v_{[e_1]} = v$, we may use these two symbols interchangeably, and we use $[\![v]\!]$ to denote the set of expanded variables $\{v, v_{[e_2]}, \ldots, v_{[e_k]}\}$. We generalize the definition of expanded formulas to the whole set V, and the expanded formula is denoted by F^e_V or even F^e, in which *every* shared variable is treated in the above manner. We use $V^e = \bigcup_{v \in V} [\![v]\!]$ to denote the set of Boolean variables of F^e_V, and each clause in F^e_V is denoted by C^e_j where $j \in \{1, 2, \ldots, m\}$. Example 2 illustrates our definitions.

Example 2. Consider formula $\mathcal{F}$ over $V = \{a, b, c\}$ in Example 1. The variable a appears in three clauses, so we can obtain the following expanded formula, where $a = a_{[e_1]}$:

$$\mathcal{F}^e_a : (a_{[e_1]}) \wedge (\overline{a_{[e_2]}} \vee b) \wedge (\overline{a_{[e_3]}} \vee c)$$

As a is the only shared variable, the expanded formula F^e_V would be F^e_a, where $C^e_1 = (a_{[e_1]})$, $C^e_2 = (\overline{a_{[e_2]}} \vee b)$, $C^e_3 = (\overline{a_{[e_3]}} \vee c)$, and $V^e = \{a_{[e_1]}, a_{[e_2]}, a_{[e_3]}, b, c\}$.

It is obvious that a Boolean formula F may not be logically equivalent to its expanded formula F^e_V. However, if F^e_V is *equivalently expanded*, i.e., it satisfies the following condition:

$$v_{[e_1]} \Leftrightarrow v_{[e_2]} \Leftrightarrow \cdots \Leftrightarrow v_{[e_k]} \text{ for all } v \in V$$

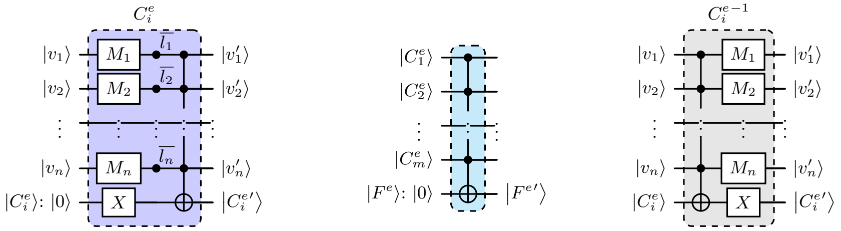
then an input vector $\boldsymbol{v} \in \{0, 1\}^{|V|}$ for formula F uniquely determines an input vector $\boldsymbol{v^e} \in \{0, 1\}^{|V^e|}$ for formula F^e_V, and vice versa. In such cases, Lemma 1 proves that $\boldsymbol{v}$ is a solution to F if and only if $\boldsymbol{v^e}$ is a solution to F^e_V. Let us consider $\mathcal{F}$ in Example 1 again. If $a_{[e_1]} \Leftrightarrow a_{[e_2]} \Leftrightarrow a_{[e_3]}$, then $\mathcal{F}^e_V \Leftrightarrow \mathcal{F}$.

Lemma 1. *Given a formula F over V, if F is equivalently expanded to F^e_V, then $\boldsymbol{v}$ is a solution to F iff $\boldsymbol{v^e}$ is a solution to F^e_V.*

From Lemma 1, given a CNF formula F over V, our parallel oracle operates on its equivalently expanded formula F^e. But how can we ensure that those expanded variables are logically equivalent? The answer is *entanglement*! For each variable $v \in V$ shared among k clauses, we prepare the following entangled state initially for v and its expanded variables:

$$|v_{[e_1]}\rangle\, |v_{[e_2]}\rangle \cdots |v_{[e_k]}\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes k} + |1\rangle^{\otimes k})$$

In this setting, each shared variable and its expanded variables will be all $|0\rangle$ with $\frac{1}{2}$ probability or be all $|1\rangle$ with $\frac{1}{2}$ probability. This state is actually the GHZ



(a) Clause C_i^e (b) Formula F^e (c) Inverse Circuit

Fig. 4: Quantum circuit construction scheme for clauses and formula.

state for k qubits and can be generated by a quantum circuit efficiently [19] with depth $O(\log_2 k)$.

The proposed parallel oracle construction is a bottom-up approach. Suppose the expanded formula is $F_V^e : C_1^e \wedge C_2^e \wedge \cdots \wedge C_m^e$. The quantum circuit of each clause C_i^e is constructed first for all $i \in \{1, 2, \ldots, m\}$, and all the m clause circuits are then conjuncted. Fig. 4a shows how to construct the circuit for each clause $C_i^e : l_1 \vee l_2 \vee \ldots \vee l_n$, where the qubit $|C_i^e\rangle$ represents the truth value (initially $|0\rangle$) of clause C_i^e. Notice that the M_j gate here depends on literal l_j for $j \in \{1, 2, \ldots, n\}$, exactly the same as in Fig. 2a, i.e., if l_j is negative, M_j would be the I gate ; otherwise, M_j would be the X gate. Lemma 2 proves the correctness of the clause construction, where $\left|C_i^{e\prime}\right\rangle$ gives the result of the clause after the computation in Fig. 4a.

Lemma 2 (Clause Correctness). $\left|C_i^{e\prime}\right\rangle = |1\rangle \Leftrightarrow$ *clause* C_i^e *is true.*

Once all the m clauses are constructed, they are conjuncted by a m-qubit Toffoli gate, as shown in Fig. 4b, in which $|F^e\rangle$ is the qubit (initially 0) representing the truth value of formula F^e controlled by the m qubits $|C_i^e\rangle$ for $i \in \{1, 2, \ldots, m\}$. Lemma 3 proves the correctness of the formula construction.

Lemma 3 (Formula Correctness). $\left|F^{e\prime}\right\rangle = |1\rangle \Leftrightarrow$ *formula* F^e *is true.*

Fig. 5 shows the construction for the whole parallel oracle $\mathcal{O} = \Omega^{-1}(P(\Omega))$, where the Ω block is constructed by composing the building blocks of clause circuits and their conjunction; the P block applies a Z gate on the $|F^e\rangle$ qubit to give a "-1" phase to the input vector when $|F^e\rangle$ is $|1\rangle$, i.e., when formula F^e evaluates to true; the Ω^{-1} block is the inverse operation of Ω to restore the input vector back to its initial value for the following diffusion process. The $C_i^{e^{-1}}$ circuit is the inverse operation of C_i^e. Its construction is shown in Fig. 4c. The correctness of the proposed parallel oracle $\mathcal{O}$ is proved in Theorem 1.

Theorem 1 (Parallel Oracle Correctness). *Let* $\boldsymbol{v^e}$ *be the input vector of formula* F^e*. Our parallel oracle* $\mathcal{O}$ *ensures the following:*

$$\mathcal{O}(|\boldsymbol{v^e}\rangle) = \begin{cases} |\boldsymbol{v^e}\rangle \ , \ \textit{if} \ \ F^e(\boldsymbol{v^e}) = 0 \\ -|\boldsymbol{v^e}\rangle \ , \ \textit{if} \ \ F^e(\boldsymbol{v^e}) = 1 \end{cases}$$

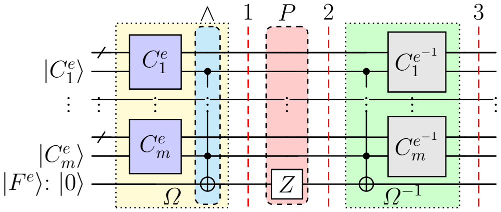


Fig. 5: The parallel oracle construction scheme.

Let us get back to our running example $\mathcal{F} : (a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$. After the conventional oracle O, the state of $|\boldsymbol{v}\rangle : |a\rangle\,|b\rangle\,|c\rangle$ becomes

$$\frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle - |111\rangle),$$

where $|111\rangle$ has a "-1" phase as it is the solution to formula $\mathcal{F}$. In our approach, the input vector $|\boldsymbol{v}\rangle$ is equivalently expanded into $|\boldsymbol{v^e}\rangle : \left|a_{[e_1]}\right\rangle \left|a_{[e_2]}\right\rangle \left|a_{[e_3]}\right\rangle |b\rangle\,|c\rangle$. After applying our parallel oracle $\mathcal{O}$, the state of the input vector $|\boldsymbol{v^e}\rangle$ becomes

$$\frac{1}{\sqrt{8}}(\left|\tilde{\mathbf{0}}00\right\rangle + \left|\tilde{\mathbf{0}}01\right\rangle + \left|\tilde{\mathbf{0}}10\right\rangle + \left|\tilde{\mathbf{0}}11\right\rangle + \left|\tilde{\mathbf{1}}00\right\rangle + \left|\tilde{\mathbf{1}}01\right\rangle + \left|\tilde{\mathbf{1}}10\right\rangle - \left|\tilde{\mathbf{1}}11\right\rangle),$$

where $\tilde{\mathbf{0}}$ denotes 000, $\tilde{\mathbf{1}}$ denotes 111, and $\left|\tilde{\mathbf{1}}11\right\rangle$ is the solution to the expanded formula $\mathcal{F}^e : (a_{[e_1]}) \wedge (\overline{a_{[e_2]}} \vee b) \wedge (\overline{a_{[e_3]}} \vee c)$.

3.2 Parallel Diffuser

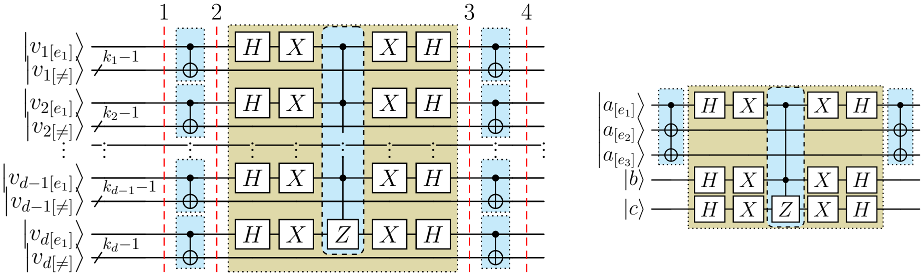
The purpose of the diffuser is to amplify the amplitude of the solution vectors to increase/maximize the probability of the solution being measured. The classic diffuser used in Grover's algorithm adopts the so-called *inversion about the average* approach to achieve this goal. However, the classic diffuser does not work directly in our parallel setting. Let us use the running example $\mathcal{F}$: $(a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$ again for illustration. Fig. 3c shows the case when the classic diffuser is directly applied to $\mathcal{F}^e$ on all qubits (including the expanded ones $\left|a_{[e_2]}\right\rangle$ and $\left|a_{[e_3]}\right\rangle$), which generates the wrong result. This is because the classic diffuser assumes all the combinations of the input values have an equal probability of occurring, i.e., $\left|a_{[e_1]}\right\rangle \left|a_{[e_2]}\right\rangle \left|a_{[e_3]}\right\rangle$ could be $|000\rangle$, $|001\rangle$, $|010\rangle$, $\ldots$, $|111\rangle$ with equal probability $\frac{1}{8}$. This violates the invariant we want to preserve at all times, i.e., $\left|a_{[e_1]}\right\rangle \left|a_{[e_2]}\right\rangle \left|a_{[e_3]}\right\rangle$ can only be either $|000\rangle$ or $|111\rangle$. The correct parallel diffuser for $\mathcal{F}$ is the one shown in Fig. 6b.

Now, let us see what adjustments have to be made to utilize the classic diffuser in our parallel setting. Here, we omit the details of the classic diffuser, which is out of scope. Instead, let us assume that $|\boldsymbol{v}\rangle$ in $\mathcal{F}$ is amplified as

$$(\alpha_0\,|000\rangle + \alpha_1\,|001\rangle + \alpha_2\,|010\rangle + \alpha_3\,|011\rangle + \alpha_4\,|100\rangle + \alpha_5\,|101\rangle + \alpha_6\,|110\rangle + \alpha_7\,|111\rangle),$$

where $\alpha_i \in \mathbb{C}$, $i \in \{0, 1, \ldots, 7\}$, and $\sum_{i=0}^{7} |\alpha_i|^2 = 1$. Our parallel diffuser is designed to achieve the same effect, i.e., to amplify $|\boldsymbol{v^e}\rangle$ in formula $\mathcal{F}^e$ as

$$(\alpha_0\left|\tilde{\mathbf{0}}00\right\rangle + \alpha_1\left|\tilde{\mathbf{0}}01\right\rangle + \alpha_2\left|\tilde{\mathbf{0}}10\right\rangle + \alpha_3\left|\tilde{\mathbf{0}}11\right\rangle + \alpha_4\left|\tilde{\mathbf{1}}00\right\rangle + \alpha_5\left|\tilde{\mathbf{1}}01\right\rangle + \alpha_6\left|\tilde{\mathbf{1}}10\right\rangle + \alpha_7\left|\tilde{\mathbf{1}}11\right\rangle),$$



(a) General diffuser scheme.

(b) Diffuser for formula $\mathcal{F}$

Fig. 6: The parallel diffuser scheme.

where $\tilde{\mathbf{0}}$ denotes 000, $\tilde{\mathbf{1}}$ denotes 111. Fig. 6a shows the quantum circuit construction for the proposed parallel diffuser. Suppose a CNF formula F is over V, where $|V| = d$. For each variable $v_j \in V$ for $j \in \{1, 2, \ldots, d\}$, if v_j appears in k_j clauses in F, we use the following notation

$$\left|v_{j[\neq]}\right\rangle = \left|v_{j[e_2]}\right\rangle \left|v_{j[e_3]}\right\rangle \cdots \left|v_{j[e_{k_j}]}\right\rangle$$

to denote the tensor product of all expanded variables except $v_{j[e_1]}$. In Step 1 of Fig. 6a, each shared variable $|v_j\rangle_1$ is entangled with its expanded variables, i.e., $|v_j\rangle_1 \left|v_{j[\neq]}\right\rangle_1 = \alpha_j |0\rangle^{\otimes k_j} + \beta_j |1\rangle^{\otimes k_j}$, where $\alpha_j, \beta_j \in \mathbb{C}$.

In Step 2, each expanded variable is disentangled with $|v_j\rangle$ by a CNOT gate with one control (i.e, $|v_j\rangle$) and $(k_j - 1)$ targets (i.e., $\left|v_{j[\neq]}\right\rangle$). Thus, $|v_j\rangle_2 \left|v_{j[\neq]}\right\rangle_2 = (\alpha_j |0\rangle + \beta_j |1\rangle) \otimes |0\rangle^{\otimes k_j - 1}$, i.e., $\left|v_{j[e_q]}\right\rangle_2$ becomes $|0\rangle$ and is independent from $\left|v_{j[e_1]}\right\rangle$ for $q \in \{2, 3, \ldots, k_j\}$.

In Step 3, only $\left|v_{j[e_1]}\right\rangle_2$ is selected as the representative for the diffusion process for all $j \in \{1, 2, \ldots, d\}$, and the classic diffuser can be utilized. Actually, the selected representatives $\left|v_{1[e_1]}\right\rangle_2 \left|v_{2[e_1]}\right\rangle_2 \cdots \left|v_{d[e_1]}\right\rangle_2$ are exactly the input of the classic diffuser $|v_1\rangle |v_2\rangle \cdots |v_d\rangle$, as shown in Fig. 3.

Assume $\left|v_{j[e_1]}\right\rangle_3$ is amplified as $\alpha'_j |0\rangle + \beta'_j |1\rangle$ after the diffusion process. In Step 4, the expanded variables $\left|v_{j[\neq]}\right\rangle$ are entangled back with $|v_j\rangle$ by a CNOT gate with one control (i.e, $|v_j\rangle$) and $(k_j - 1)$ targets (i.e., $\left|v_{j[\neq]}\right\rangle$). Thus, $|v_j\rangle_4 \left|v_{j[\neq]}\right\rangle_4 = \alpha'_j |0\rangle^{\otimes k_j} + \beta'_j |1\rangle^{\otimes k_j}$. Theorem 2 shows the details step by step and proves that our parallel diffuser has the same effect as the classic diffuser.

Theorem 2 (Parallel Diffuser Correctness). *Let $|\boldsymbol{v}\rangle : |v_1\rangle |v_2\rangle \cdots |v_d\rangle$ be the input vector of F and D be the classic diffuser such that*

$$D(|\boldsymbol{v}\rangle) = \sum_{i=0}^{2^d-1} \alpha'_i \left(|b_1\rangle |b_2\rangle \cdots |b_d\rangle\right),$$

where the index i is represented as the binary string $|b_1\rangle |b_2\rangle \cdots |b_d\rangle \in \{0,1\}^d$. If the input vector of F^e is $|\boldsymbol{v}^e\rangle$, our parallel diffuser $\mathcal{D}$ ensures the following:

$$\mathcal{D}(|\boldsymbol{v}^e\rangle) = \sum_{i=0}^{2^d-1} \alpha_i' \left(|b_1\rangle^{\otimes k_1} |b_2\rangle^{\otimes k_2} \cdots |b_d\rangle^{\otimes k_d} \right).$$

4 Distributed Quantum SAT Solver

In this section, we consider the scenario where one quantum computer has insufficient qubits to handle the whole SAT problem. To overcome this issue, we follow the "divide and conquer" strategy and develop a *distributed* quantum SAT solver, including a distributed oracle (Sect. 4.1) and a distributed diffuser (Sect. 4.2).

4.1 Distributed Oracle

Let us recall the proposed parallel oracle in Fig. 5. The circuit for processing each clause C_i^e is independent of each other for $i \in \{1, 2, \ldots, m\}$ and thus can be naturally handled by one dedicated quantum computer. The critical question here is "how to handle the conjunction distributedly", i.e., how to distributedly perform the CNOT gate with m control qubits and one target qubit.

Sarvaghad-Moghaddam et al. proposed a protocol for distributed quantum gates [30] based on quantum teleportation [9,11,33,36]. However, the correctness of the protocol was not proved in their paper. We generalise their work to a distributed controlled-U gate with any number of nodes and further prove the correctness of our design. Fig. 7 shows the design of the protocol. Suppose we want to perform a controlled U gate with m control qubits, as shown in the right side of Fig. 7, where $|C_i\rangle$ is the control qubit for $i \in \{1, 2, \ldots, m\}$ and $|t\rangle$ is the target qubit. The proposed distributed protocol is designed in a way that the m control qubits need not be in the same quantum computer (node) where the target qubit $|t\rangle$ is located. Let us assume that the control qubit $|C_i\rangle$ is located on node i where $i \in \{1, 2, \ldots, m\}$, and the target qubit $|t\rangle$ is located on a master node, as shown in the left side of Fig. 7. To perform the controlled U gate remotely, initially, each node i shares, with the master node, a pair of the following entangled qubits:

$$|e_i\rangle |\widehat{e_i}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \text{ for all } i \in \{1, 2, \ldots, m\},$$

where node i holds qubit $|e_i\rangle$ and the master node holds qubit $|\widehat{e_i}\rangle$.

In step 1, each node i performs a CNOT gate on $|C_i\rangle |e_i\rangle$, measures qubit $|e_i\rangle$ in the standard ($|0\rangle$ and $|1\rangle$) basis and then sends the measurement outcome to node master via a non-quantum channel (e.g., TCP/IP, etc.). After receiving the measurement outcome, the node master applies an X gate on qubit $|\widehat{e_i}\rangle$ if the measurement outcome is $|1\rangle$; otherwise, nothing is performed. After this step, the qubit $|e_i\rangle$ collapses, and the two qubits $|C_i\rangle |\widehat{e_i}\rangle$ become entangled in the same state, i.e., they are either in state $|00\rangle$ or $|11\rangle$.

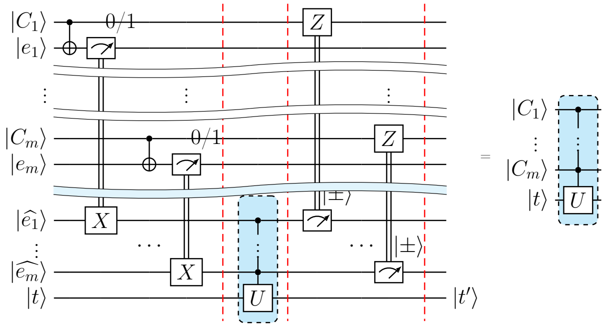


Fig. 7: The distributed m-controlled-U gate scheme.

In step 2, since $|C_i\rangle$ and $|\widehat{e_i}\rangle$ have the same state, applying the controlled U gate with $|\widehat{e_i}\rangle$ as the m control qubits is equivalent to that with $|C_i\rangle$ as the m control qubits for $i \in \{1, 2, \dots, m\}$.

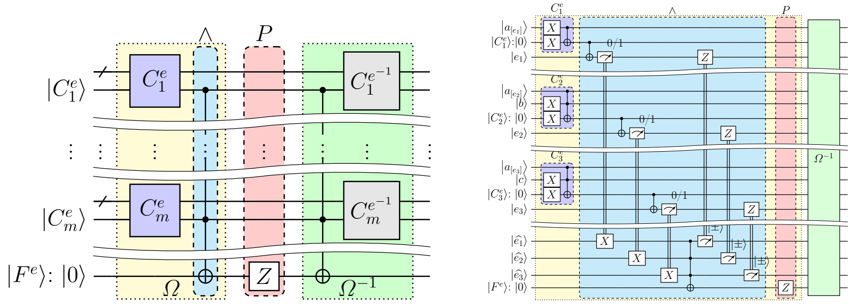
Step 3 disentangles $|C_i\rangle$ from $|\widehat{e_i}\rangle$. To do so, the node master measures the qubit $|\widehat{e_i}\rangle$ in the $|+\rangle$ and $|-\rangle$ basis and then sends the measurement outcome to node i via a non-quantum channel. After receiving the measurement outcome, node i performs a Z gate on qubit $|C_i\rangle$ if the outcome is $|-\rangle$; otherwise, nothing is performed. Once node i finishes this step for each $i \in \{1, 2, \dots, m\}$, the operation of the controlled U gate is accomplished distributedly among the $m + 1$ nodes. Theorem 3 proves the correctness of the distributed protocol.

Theorem 3 (Distributed Protocol Correctness of Fig. 7).
Let $|C_i\rangle = x_i |0\rangle + y_i |1\rangle$, where $x_i, y_i \in \mathbb{C}$ and $i \in \{1, , 2, \dots, m\}$. The following three conditions hold:
(1) In Step 1, $|\widehat{e_i}\rangle |C_i\rangle = x_i |00\rangle + y_i |11\rangle$ for all $i \in \{1, 2, \dots, m\}$.
(2) In Step 2, $|t'\rangle = U(|t\rangle)$ iff $|C_i\rangle = |1\rangle$ for all $i \in \{1, 2, \dots, m\}$.
(3) In Step 3, $|C_i\rangle = x_i |0\rangle + y_i |1\rangle$ for all $i \in \{1, 2, \dots, m\}$.

With this developed protocol, we can perform the conjunction of m clauses distributedly. The design of the distributed oracle is shown in Fig. 8a, where each clause C_i^e is handled by node i, and the node master interacts with node i on qubit $|C_i^e\rangle$ for all $i \in \{1, 2, \dots, m\}$ as the m control qubits to accomplish the conjunction based on the distributed protocol. Notice that there are two conjunction operations to be performed distributedly: one is in the Ω block, and the other is in the Ω^{-1} block. The correctness of the proposed distributed oracle follows from Theorem 1 and Theorem 3.

Let us use the running example for illustration. Fig. 8b shows[6] the distributed oracle for the formula $\mathcal{F} : (a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$. Since there are three clauses, we need four nodes involved (one for each clause and one for the master node). Each

[6] Due to the space limit, this figure is shrunk to show the structure only. The full-size version can be found in the author version [27].



(a) General scheme.

(b) Oracle for formula $\mathcal{F}$.

Fig. 8: The parallel and distributed oracle construction scheme.

node i shares the pair $|e_i\rangle\,|\widehat{e_i}\rangle$ with the node master for $i \in \{1, 2, 3\}$ such that node i holds qubit $|e_i\rangle$, while the node master holds qubit $|\widehat{e_i}\rangle$. The conjunction is performed based on the proposed distributed protocol, as shown in the $\wedge$-block in cyan color. The other conjunction operation in the Ω^{-1} is identical, which is omitted here due to the space limit.

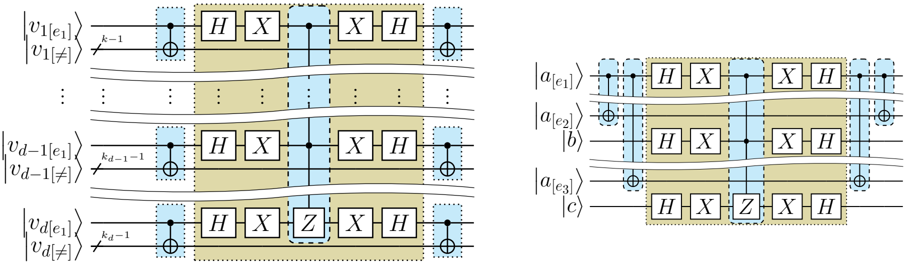
4.2 Distributed Diffuser

Let us recall the design of our parallel diffuser in Fig. 6a. Since only variables $\left|v_{j[e_1]}\right\rangle$ for $j \in \{1, 2, \ldots, d\}$ are selected as the representative for the diffusion process, it is natural to host each of $\left|v_{j[e_1]}\right\rangle$ on a different node for the distributed diffusion. Fig. 9a shows the design of our distributed diffuser. The critical operation is the controlled Z gate (the center block in cyan color), which can be accomplished based on the proposed distributed protocol, as introduced in Section 4.1. Except for the controlled Z gate, there are two other types of operations needed to be performed distributedly:

1. $\left|v_{j[e_1]}\right\rangle$ disentangles with $\left|v_{j[\neq]}\right\rangle$ for all $j \in \{1, 2, \ldots, d\}$, and
2. $\left|v_{j[e_1]}\right\rangle$ entangles back with $\left|v_{j[\neq]}\right\rangle$ for all $j \in \{1, 2, \ldots, d\}$,

as shown in the leftmost and rightmost cyan blocks of Fig. 9a, respectively. These operations can be accomplished based on the proposed distributed protocol as well. Notice that we do not unfold the distributed protocol for each operation to be performed distributedly in Fig. 9a due to the space limit. Instead, we mark those operations that can be accomplished by the proposed distributed protocol in cyan color to highlight the high-level structure of our design. The correctness of our distributed diffuser follows from Theorem 2 and Theorem 3.

We illustrate our approach using the running example. Fig. 9b shows the distributed diffuser for formula $\mathcal{F} : (a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$. Generally, we create a node for each variable in the formula. Since there are three variables in $\mathcal{F}$, we need three nodes, where the first node holds $\left|a_{[e_1]}\right\rangle$, the second holds qubits $|b\rangle$ and $\left|a_{[e_2]}\right\rangle$, and the third holds qubits $|c\rangle$ and $\left|a_{[e_3]}\right\rangle$. For the controlled Z



(a) General diffuser scheme.

(b) Diffuser for formula $\mathcal{F}$.

Fig. 9: The parallel and distributed diffuser scheme.

gate in the diffusion process, the third node can serve as the node master in the distributed protocol. Before (resp. after) the diffusion process, $|a_{[e_1]}\rangle$ needs to disentangle (resp. entangle back) with $|a_{[e_2]}\rangle$ and $|a_{[e_3]}\rangle$. These operations can be accomplished by our distributed protocol as well. Note that our distributed protocol works only when there is one target qubit, while the structure of the disentangling/entangling operations here has one control qubit with multiple target qubits. Thus, instead of performing the disentangling/entangling in one shot, we need to perform them sequentially, e.g., $|a_{[e_1]}\rangle$ first disentangles (entangles back) with $|a_{[e_2]}\rangle$ then with $|a_{[e_3]}\rangle$, as the leftmost (rightmost) cyan blocks in Fig. 9b. Interestingly, the order does not matter. One can easily check that different orders give the same result.

5 Complexity Analysis and Evaluation

Now, we theoretically compare the time complexity of our quantum SAT solvers with the conventional (sequential) quantum SAT solver w.r.t. the circuit depth, as longer circuit depth corresponds to longer execution time. Notice that different types of quantum computers have different ways to realize quantum mechanisms and thus have different complexities. For superconducting quantum computers, each operation has to be decomposed and implemented based on basic (one or two inputs) quantum gates, e.g., X, CNOT, etc. In this setting, the m-qubit toffoli gate has to be implemented using $O(\log_2 m)$ basic gates [31], i.e., it requires $O(\log_2 m)$ depth. However, for non-superconducting quantum computers, e.g., trapped ion quantum computers [29,35], the m-qubit Toffoli gate can be implemented as one single gate and only requires $O(1)$ depth. In the following, we discuss the complexity in two settings: superconducting and non-superconducting. The comparisons are summarized in Table 1.

Superconducting. Given a formula F with m clauses, the conventional oracle requires $O(m)$ time complexity to process each clause and $O(\log_2 m)$ time complexity to perform the conjunction by the m-qubit Toffoli gate, which suggests an overall $O(m)$ time complexity. Our parallel oracle, as well as our distributed oracle, require $O(1)$ time complexity to process all clauses and $O(\log_2 m)$

	superconducting		non-superconducting	
	Oracle	Diffuser	Oracle	Diffuser
Conventional	$O(m)+O(\log_2 m)$	$O(\log_2 n)$	$O(m)$	$O(1)$
Parallel	$O(1)+O(\log_2 m)$	$O(\log_2 k_{\texttt{max}})+O(\log_2 n)$	$O(1)$	$O(1)$
Distributed	$O(1)+O(\log_2 m)$	$O(k_{\texttt{max}})+O(\log_2 n)$	$O(1)$	$O(k_{\texttt{max}})$

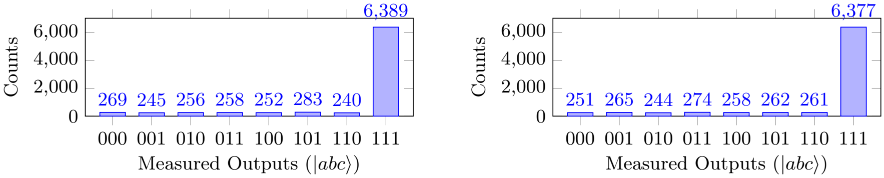
Table 1: Time (depth) complexity comparisons

time complexity to perform the conjunction, which suggests an overall $O(\log_2 m)$ depth. For the diffusion process, the conventional diffuser requires $O(\log_2 n)$ time complexity [31] because of the n-qubit controlled-Z gate, where n is the number of variables. Our parallel diffuser, as well as our distributed diffuser, have a small extra overhead because they need to disentangle or entangle a variable with each of its expended variables. Assume that variable $v_j \in V$ is shared by k_j clauses for $j \in \{1, 2, \ldots, d\}$ and $|V| = d$. Let $k_{\texttt{max}}$ be the maximum value among k_j for all the shared variables. This extra overhead would be bounded by $O(\log_2 k_{\texttt{max}})$ for our parallel diffuser [31], and bounded by $O(k_{\texttt{max}})$ for our distributed diffuser. Notice that the disentangling/entangling process for each variable v_j is independent and can be performed in parallel. In practice, this extra overhead would be negligible as $k_{\texttt{max}}$ would be much smaller than n, the total number of variables, as well as m, the totoal number of clauses. Also notice that we safely omit the time complexity of preparing the initial GHZ states for all variables because the initialization for each variable [19] can be performed in parallel and is thus bounded by $O(\log_2 k_{\texttt{max}})$ in overall, which is subsumed by other factors.

Non-superconducting. In non-superconducting quantum computers, the m-qubit Toffoli gate for clause conjunction and the n-qubit controlled Z gate in all the diffusers can be done in one operation (gate) and thus reduced to $O(1)$ time complexity [29, 35]. However, our distributed diffuser still requires $O(k_{\texttt{max}})$ time complexity because the operation to disentangle or entangle a variable with each of its expended variables is distributed and has to be done sequentially based on the proposed distributed protocol (Fig. 7). Notice that this operation in our parallel diffuser only takes $O(1)$ time complexity because it can be done by one single operation in a (centralized) quantum computer [29, 35]. The complexity of preparing initial GHZ states is safely omitted here because the initialization for each variable takes $O(1)$ and can be performed in parallel, which makes the overall complexity in depth still $O(1)$.

#Iterations. How many iterations are required for our parallel and distributed solvers to obtain the solutions? The answer is $O(\sqrt{N/M})$, the same as that of the conventional one, where N is the size of the search space and M is the number of solutions (c.f. Section 2.1). Notice that although additional expanded variables are introduced, they have the same values and are entangled with the original variables. Furthermore, only original variables are involved in the diffusion process, i.e., the size of the search space remains the same.

Simulation. We have implemented the proposed parallel and distributed SAT solver for our running example $\mathcal{F} : (a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$ in Qiskit [6].



(a) Parallel approach

(b) Distributed approach

Fig. 10: Simulation results for formula $\mathcal{F}$.

The implementation can be obtained in [1]. For the parallel SAT solver, a total of 9 qubits are required (three for variable a, two for variables b and c, three for all the clauses, and one for formula $\mathcal{F}$). For the distributed SAT solver, a total of 36 qubits are required (9 for formula $\mathcal{F}$ itself and 27 for performing the proposed distributed quantum protocol). Only one Grover iteration is required for both solvers. Fig. 10a and Fig. 10b show the simulation results of performing the two implementations for 8, 192 shots, respectively. The x-axis shows the measured outcome of $|abc\rangle$, while the y-axis shows the count of each outcome being measured. One can observe that $|111\rangle$, the solution to formula $\mathcal{F}$, has an overwhelmingly higher probability over other non-solution inputs that are almost negligible. Thus, this experiment shows the correct result as expected from our proofs. We have tried more and larger examples, and they all yielded correct results through cross-checking with classical SAT answers. Due to the page limit, we refer the readers to [2] for more details. Given an arbitrary Boolean formula, we have also developed a tool [3] that can automatically generate its corresponding quantum circuit of the proposed parallel solver for SAT solving.

Execution on Real Quantum Computers. We have also tried to implement the conventional SAT solver and the proposed parallel solver for our running example $\mathcal{F} : (a) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$ on a real quantum computer, "ibm_brisbane" provided by IBM [25]. This quantum computer is based on superconducting and offers at most 127 qubits for usage, but it uses a special set of universal basic gates, which is different from the regular Clifford set. Thus, both designs of the conventional and our parallel solvers have to be transpiled first to be executed on this quantum computer. After the transpilation, the circuit depth of the conventional solver is 480, while that of our parallel solver is only 328, which is around 31.7% reduction. The execution time of the conventional solver for 1, 024 shots takes 35 seconds, while that (with the same number of shots) of our parallel solver takes 30 seconds, which is around 14.3% reduction. This result also confirms our assumption that shorter circuit depth has a shorter execution time. Although the 14.3% reduction in execution time does not correspond to the 31.7% reduction in circuit depth, we found that this could be due to the fact that the example is small and the initialization time for the quantum computer is included. Comprehensive experiments on large examples are extremely costly[7]. Thus, we defer it to future work. Another reason for not experimenting more is that the current state-of-the-art quantum computer does not natively support multiple processing units running in parallel, so it cannot fully reflect the advantage of our theories.

6 Related Work

Quantum search. Improving the proof search in SAT solving using quantum computing is a promising and broadly discussed direction. Barreto et al.'s method [8] adopts Shenvi's quantum random walk search algorithm [37] in a local search setting and applies it to 3-SAT. Their method enables parallel simulation of the quantum SAT solving algorithm, though it is different from our notion of performing and coordinating multiple quantum SAT solving instances in parallel. Another prominent example is to use Grover's algorithm to search for a satisfiable truth assignment for Boolean variables [20].

Hybrid methods. A straightforward application of Grover's algorithm in SAT solving requires a large number of qubits. Consequently, several *hybrid* approaches are proposed to reduce the number of qubits by combining quantum computing with classical computing algorithms. For example, quantum cooperative search replaces some qubits with classical bits and solves the classical bits using traditional SAT solving [15]. Zhang et al.'s approach optimize the data structures in SAT solving to take advantage of Grover's algorithm and DPLL [40]. Another venue is to focus on a parameterized area of the search space and then Grover's search [39]. These hybrid approaches achieved varied theoretical improvements in the time complexity of SAT solving.

Quantum heuristics. Quantum walk [16] could be applied in heuristics to improve SAT solving. Campos et al. [13] presented an algorithm for solving k-SAT, where each clause has exactly k variables. Their approach leverages continuous time quantum walk over a hypercube graph with potential barriers for exploiting the properties of quantum tunnelling to obtain the possibility of getting out of local minima. Their simulation shows a reasonable success rate, though heuristic methods may not guarantee that a solution is found. Similarly, research on classical algorithms for quantum SAT solving [5] is also in a different vein.

Quantum annealing. Quantum annealers [7] are another widely used optimization technique that minimizes objective functions over discrete variables using quantum fluctuation. Bian et al.'s method [10] encodes SAT solving into a quadratic unconstrained binary optimization (QUBO) problem and applies quantum annealing to solve it.

Applications. Quantum SAT solving has found numerous applications. For instance, Quantum SAT solving may be applied to speed up integer factorization. Mosca et al. [32] showed how to design SAT circuits for finding smooth numbers, which is an essential step in Number Field Sieve (NFS) — the best-known classical solution. Assuming that there is a quantum SAT solver that performs

[7] IBM's "Pay-As-You-Go" plan [25] charges USD $1.6 per second for execution.

better than classical solvers, their method would lead to a factorization method that outperforms NFS. The maximum satisfiability (MAX-SAT) problem asks for the maximum number of clauses that are satisfiable in a conjunctive normal form. Alasow and Perkowski [4] apply Grover's search with a customized oracle to perform SAT solving, which also leads to an efficient solution to MAX-SAT.

Qiu et al. proposed a distributed Grover's algorithm [34], which decomposes the original SAT formula into a set of 2^k subformulas (obtained by instantiating k Boolean variables). Each of the 2^k subformulas is then solved by one quantum computer running Grover's algorithm, and the final solution depends on the subsolutions to the subformulas. Their "divide and conquer" strategy does not utilize any quantum characteristics, while ours utilizes quantum teleportation.

7 Conclusion and Future Work

This work is the first to propose a parallel quantum SAT solver using entanglement. Compared to the sequential quantum SAT solver, our parallel solver reduces the time complexity of each Grover iteration from linear time $O(m)$ to constant time $O(1)$ by using more qubits. To scale to complex problems, we also propose the first distributed quantum SAT solver using quantum teleportation such that the total qubits required are shared and distributed among a set of quantum computers (nodes), and the quantum SAT solving is accomplished collaboratively by all the nodes. We prove the correctness of our methods. They are also evaluated in simulations via Qiskit, and the results are correct. In the future, we plan to extend our parallel and distributed quantum SAT solvers to handle satisfiability modulo theories (SMT) problems.

Acknowledgements

This research is mainly supported by the project MOE-T2EP20123-0004 (funded by Singapore's Ministry of Education) and partially supported by the project NSTC 112-2221-E-006-184 (funded by National Science and Technology Council, Taiwan), the project MOE-T1-1/2022-43 (funded by Singapore's Ministry of Education), and Griffith University IIIS RSP fund.

References

1. `https://anonymous.4open.science/r/Examples-in-Paper-7461`.
2. `https://anonymous.4open.science/r/Examples-for-General-SAT-Solver-6211`.
3. `https://anonymous.4open.science/r/Quantum-General-SAT-Solver-86FF`.
4. Abdirahman Alasow and Marek Perkowski. Quantum algorithm for maximum satisfiability. In *2022 IEEE 52nd International Symposium on Multiple-Valued Logic (ISMVL)*, pages 27–34, 2022.
5. Marco Aldi, Niel de Beaudrap, Sevag Gharibian, and Seyran Saeedi. On efficiently solvable cases of quantum k-sat. *Communications in Mathematical Physics*, 381(1):209–256, 2021.

6. Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, et al. Qiskit: An Open-source Framework for Quantum Computing, January 2019.
7. Bruno Apolloni, Nicolò Cesa-Bianchi, and Diego De Falco. A numerical implementation of "quantum annealing". In *Stochastic Processes, Physics and Geometry: Proceedings of the Ascona-Locarno Conference*, pages 97–111, 1990.
8. M Barreto, G Abal, and S Nesmachnow. A parallel spatial quantum search algorithm applied to the 3-sat problem. In *Proc. of XII Argentine Symposium on Artificial Intelligence*, pages 1–12, 2011.
9. Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Physical Review Letters*, 70(13):1895–1899, 1993.
10. Zhengbing Bian, Fabian Chudak, William Macready, Aidan Roy, Roberto Sebastiani, and Stefano Varotti. Solving sat and maxsat with a quantum annealer: Foundations and a preliminary report. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems*, pages 153–171, Cham, 2017. Springer International Publishing.
11. Dik Bouwmeester, Jian-Wei Pan, Klaus Mattle, Manfred Eibl, Harald Weinfurter, and Anton Zeilinger. Experimental quantum teleportation. *Nature*, 390(6660):575–579, 1997.
12. Gilles Brassard, Peter Hoyer, and Alain Tapp. Quantum Counting. *arXiv e-prints*, pages quant–ph/9805082, May 1998.
13. Ernesto Campos, Salvador E Venegas-Andraca, and Marco Lanzagorta. Quantum tunneling and quantum walks as algorithmic resources to solve hard k-sat instances. *Scientific Reports*, 11(1):16845, 2021.
14. Fernando R. Cardoso, Daniel Yoshio Akamatsu, Vivaldo Leiria Campo Junior, Eduardo I. Duzzioni, Alfredo Jaramillo, and Celso Villas-Boas. Detailed account of complexity for implementation of circuit-based quantum algorithms. *Frontiers in Physics*, 9, 2021.
15. Sheng-Tzong Cheng and Ming-Hung Tao. Quantum cooperative search algorithm for 3-sat. *Journal of Computer and System Sciences*, 73(1):123–136, 2007.
16. Andrew M. Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, page 59–68, New York, NY, USA, 2003. Association for Computing Machinery.
17. J. I. Cirac, A. K. Ekert, S. F. Huelga, and C. Macchiavello. Distributed quantum computation over noisy channels. *Physical Review A*, 59(6):4249, 1999.
18. Richard Cleve and Harry Buhrman. Substituting quantum entanglement for communication. *Physical Review A*, 56(2):1201, 1997.
19. Diogo Cruz, Romain Fournier, Fabien Gremion, Alix Jeannerot, Kenichi Komagata, Tara Tosic, Jarla Thiesbrummel, Chun Lam Chan, Nicolas Macris, Marc-André Dupertuis, and Clément Javerzac-Galy. Efficient quantum algorithms for GHZ and W states, and implementation on the ibm quantum computer. *Advanced Quantum Technologies*, 2(5–6):1–13, 2019.
20. Diogo Fernandes and Inês Dutra. Using grover's search quantum algorithm to solve boolean satisfiability problems: Part i. *XRDS: Crossroads, The ACM Magazine for Students*, 26(1):64–66, 2019.
21. Diogo Fernandes, Carla Silva, and Inês Dutra. Using grover's search quantum algorithm to solve boolean satisfiability problems, part 2. *XRDS*, 26(2):68–71, nov 2019.

22. Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying grover's algorithm to aes: Quantum resource estimates. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography*, pages 29–43, Cham, 2016. Springer International Publishing.
23. Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.
24. Lov K. Grover. Quantum telecomputation. *arXiv preprint*, arXiv:quant-ph/9704012, 1997.
25. IBM. `https://www.ibm.com/quantum`.
26. Shang-Wei Lin, Si-Han Chen, Tzu-Fan Wang, and Yean-Ru Chen. A quantum SMT solver for bit-vector theory. *arXiv preprint*, arXiv:2303.09353, 2023.
27. Shang-Wei Lin, Tzu-Fan Wang, Yean-Ru Chen, Zhe Hou, David Sanán, and Yon Shin Teo. A parallel and distributed quantum SAT solver based on entanglement and quantum teleportation. *arXiv preprint*, arXiv:2308.03344, 2023.
28. J.P. Marques Silva and K.A. Sakallah. Grasp–a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.
29. Dmitri Maslov1 and Yunseong Nam. Use of global interactions in efficient quantum circuit constructions. *New Journal of Physics*, 20(3):033018, 2018.
30. Sarvaghad-Moghaddam Moein and Mariam Zomorodi. A general protocol for distributed quantum gates. *Quantum Information Processing*, 20(8):265, 2021.
31. Cristopher Moore and Martin Nilsson. Parallel quantum computation and quantum codes. *SIAM Journal on Computing*, 31(3):799–815, 2001.
32. Michele Mosca, João Marcos Vensi Basso, and Sebastian R Verschoor. On speeding up factoring with quantum sat solvers. *Scientific Reports*, 10(1):1–8, 2020.
33. M. A. Nielsen, E. Knill, and R. Laflamme. Complete quantum teleportation using nuclear magnetic resonance. *Nature*, 396(6706):52–55, 1998.
34. Daowen Qiu, Le Luo, and Ligang Xiao. Distributed grover's algorithm. *arXiv preprint*, arXiv:2204.10487v4, 2022.
35. S. E. Rasmussen, K. Groenland, R. Gerritsma, K. Schoutens, and N. T. Zinner. Single-step implementation of high-fidelity n-bit toffoli gates. *PHYSICAL REVIEW A*, 101(2):022308, 2020.
36. M. Riebe, H. Häffner, C. F. Roos, W. Hänsel, J. Benhelm, G. P. T. Lancaster, T. W. Körber, C. Becher, F. Schmidt-Kaler, D. F. V. James, and R. Blatt. Deterministic quantum teleportation with atoms. *Nature*, 429(6993):734–737, 2004.
37. Neil Shenvi, Julia Kempe, and K. Birgitta Whaley. Quantum random-walk search algorithm. *Phys. Rev. A*, 67:052307, May 2003.
38. Hiroyuki Tezuka, Kouhei Nakaji, Takahiko Satoh, and Naoki Yamamoto. Grover search revisited: Application to image pattern matching. *Phys. Rev. A*, 105:032440, Mar 2022.
39. Charles Moudina Varmantchaonala, Jean Louis Kedieng Ebongue Fendji, Jean Pierre Tchapet Njafa, and Marcellin Atemkeng. Quantum hybrid algorithm for solving sat problem. *Engineering Applications of Artificial Intelligence*, 121:106058, 2023.
40. Runkai Zhang, Jing Chen, Huiling Zhao, et al. Procedure of solving 3-sat problem by combining quantum search algorithm and dpll algorithm. *Computing, Performance and Communication Systems*, 4:14–24, 2020.

Author Index

A
Abdulla, Parosh Aziz III-276
Ádám, Zsófia III-129, III-330, III-371, III-412
Akshay, S. I-123
Aldughaim, Mohannad III-376
Aniva, Leni I-311
Artho, Cyrille II-3
Atig, Mohamed Faouzi III-276
Avni, Guy III-153
Ayaziová, Paulína III-341, III-406

B
Backes, John I-3
Badings, Thom II-258
Baier, Daniel III-359
Bajczi, Levente III-330, III-371, III-412
Barbosa, Haniel I-311
Barrett, Clark I-311
Basa, Eliyahu I-123
Bayless, Sam I-3
Beckert, Bernhard I-268
Bentele, Manuel III-418
Beutner, Raven II-196
Beyer, Dirk III-129, III-299, III-359
Blanchard, Allan I-331
Bocchi, Laura I-207
Bodden, Eric I-229
Boillot, Jérôme III-387
Bork, Alexander II-299
Bozhilov, Stanimir III-335, III-381
Brauße, Franz III-376

C
Cai, Yubo II-323
Chakraborty, Debraj II-299
Chakraborty, Supratik I-123, II-175, III-393
Chalupa, Marek III-353
Chatterjee, Prantik II-155
Chen, Xiaohong I-350
Chen, Yean-Ru II-363
Chen, Yu-Fang I-24
Chen, Zhenbang III-347
Chien, Po-Chun III-129, III-359, III-365
Chocholatý, David I-24, II-130
Chowdhury, Md Solimul I-34
Cimatti, Alessandro II-44
Codel, Cayden R. I-34
Cordeiro, Lucas C. III-376
Correnson, Loïc I-331
Cosler, Matthias III-45

D
D'Souza, Deepak II-175
Dacík, Tomáš I-188
Dahlsen-Jensen, Mikael Bisgaard III-194
de Pol, Jaco van III-194
Dierl, Simon II-87
Dietsch, Daniel III-418
Djoudi, Adel I-331
Dobos-Kovács, Mihály III-371, III-412
Dubslaff, Clemens III-255
Duong, Hai III-24
Dwyer, Matthew B. III-24

E
Ehlers, Rüdiger I-83
Eisenbarth, Thomas III-399
Erhard, Julian III-335, III-381

F
Farias, Bruno III-376
Fassbender, Dennis II-44
Fedyukovich, Grigory II-175
Feng, Nick I-3
Fiedor, Tomáš II-130
Fievet, Baptiste III-194
Fiterau-Brostean, Paul II-87

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 383–386, 2024.
https://doi.org/10.1007/978-3-031-57249-4

Fleury, Mathias I-311
Fried, Dror I-123
Furbach, Florian III-276

G
Gadelha, Mikhail R. III-376
Galgali, Varadraj II-3
Garcia-Contreras, Isabel I-43
Garg, Shashwat III-276
Griggio, Alberto II-44
Grover, Kush II-299
Gurfinkel, Arie I-43

H
Hahn, Christopher III-45
Hanselmann, Michael II-44
Hari Govind, V. K. I-43
Hasuo, Ichiro II-279
Havlena, Vojtěch I-24, II-130
He, Dongjie I-229
Heinzemann, Christian II-44
Heizmann, Matthias III-418
Henze, Franziska II-44
Hermanns, Holger III-255
Heule, Marijn J. H. I-34, I-61
Hilaire, Thibault I-370
Holík, Lukáš I-24, II-130
Holter, Karoliine III-335, III-381
Hou, Zhe II-363
Howar, Falk II-87
Hruška, Martin II-130
Hu, Alan J. I-3
Huerta y Munive, Jonathan Julián I-288
Huisman, Marieke III-71
Husung, Nils III-255

I
Ilcinkas, David I-370
Iqbal, Syed M. I-3

J
Jakobsen, Anna Blume III-110
Jankola, Marek III-359
Jansen, Nils II-258
Jeannin, Jean-Baptiste I-248
Jiang, Xinyu III-418
Jiménez-Pastor, A. II-343
Jonáš, Martin III-90, III-406
Jonsson, Bengt II-87
Jørgensen, Rasmus Skibdahl Melanchton III-110
Jung, Jean Christoph I-167
Junges, Sebastian II-109, II-258, II-279

K
Kabra, Aditi I-144
Kapritsos, Manos I-248
Karakaya, Kadiray I-229
Karmarkar, Hrishikesh III-393
Katoen, Joost-Pieter II-237
Kettl, Matthias III-359
Khalimov, Ayrat I-83
King, Andy I-207
Klauck, Michaela II-44
Klauke, Jonas I-229
Klumpp, Dominik III-418
Köhl, Maximilian A. III-255
Kokologiannakis, Michalis II-66
König, Lukas II-44
Korovin, Konstantin III-376
Kosmatov, Nikolai I-331
Křetínský, Jan II-299
Kruger, Loes II-109
Kumor, Kristián III-406
Küperkoch, Stefan II-44
Kwiatkowska, Marta III-3

L
Lachnitt, Hanna I-311
Lahav, Ori III-235
Lal, Akash II-155
Larsen, K. G. II-343
Laurent, Jonathan I-144
Lee, Nian-Ze III-129, III-359, III-365
Lemberger, Thomas III-359
Lengál, Ondřej I-24, II-130
Leroux, Jérôme I-370
Li, Jianxin II-217
Li, Xianzhiyu III-376
Lima, Leonardo I-288
Lin, Shang-Wei II-363
Lingsch-Rosenfeld, Marian III-359
Loose, Nils III-399
Luo, Linghui I-229

M
Mächtle, Felix III-399
Madhukar, Kumar III-393

Majumdar, Rupak II-66, III-213
Mallik, Kaushik III-153
Manino, Edoardo III-376
Menezes, Rafael Sá III-376
Mertens, Hannah II-237
Metta, Ravindra III-393
Micskei, Zoltán III-330
Milanese, Marco III-387
Miné, Antoine III-387
Mitsch, Stefan I-144
Mohr, Stefanie II-299
Molnár, Vince III-371, III-412
Monat, Raphaël III-387
Mondok, Milán III-371, III-412
Mozumder, Nusrat Jahan III-24
Murgia, Maurizio I-207

N
Nayak, Satya Prakash III-173
Neider, Daniel I-167
Neurohr, Christian I-167
Nötzli, Andres I-311
Novák, Jakub III-406

O
Omar, Ayham III-45
Osama, Muhammad II-23
Ouadjaout, Abdelraouf III-387

P
Panagou, Dimitra I-248
Parízek, Pavel II-3
Parolini, Francesco III-387
Pavlogiannis, Andreas III-110
Petrucci, Laure III-194
Pike, Lee I-3
Platzer, André I-144
Podelski, Andreas III-418
Pogudin, Gleb II-323

Q
Qu, Daohan II-3
Quatmann, Tim II-237

R
Reynolds, Andrew I-311
Richter, Cedric III-353
Rodrigues, Nishant I-350
Rogalewicz, Adam I-188
Roşu, Grigore I-350
Rot, Jurriaan II-109, II-279
Roy, Subhajit II-155

S
S, Sumanth Prabhu II-175
Saan, Simmo III-335, III-381
Sadhukhan, Suman III-153
Sağlam, Irmak III-213
Sagonas, Konstantinos II-87
Sanán, David II-363
Sanders, Peter I-268
Scheucher, Manfred I-61
Schmidt, Markus I-229
Schmitt, Frederik III-45
Schmuck, Anne-Kathrin III-173
Schott, Stefan I-229
Schüssele, Frank III-418
Schwarz, Michael III-335, III-381
Sebe, Mircea Octavian I-350
Sedláček, Jindřich III-406
Seidl, Helmut III-335, III-381
Shmarov, Fedor III-376
Shoham, Sharon I-43
Síč, Juraj I-24, II-130
Sieck, Florian III-399
Singh, Abhishek Kr III-235
Sirrenberg, Nils III-129
Solanki, Mayank II-155
Somorjai, Márk III-371, III-412
Song, Kunjian III-376
Spiessl, Martin III-359
Stoelinga, Marielle II-258
Strejček, Jan III-90, III-341, III-406
Szekeres, Dániel III-371, III-412

T
Tachna-Fram, Avi I-248
Tåquist, Fredrik II-87
Tekriwal, Mohit I-248
Telbisz, Csanád III-371, III-412
Temel, Mertcan I-340
Teo, Yon Shin II-363
Thejaswini, K. S. III-213
Tihanyi, Norbert III-376

Tilscher, Sarah III-335, III-381
Tinelli, Cesare I-311
Tomov, Naum I-103
Tonetta, Stefano II-44
Traytel, Dmitriy I-288
Trentin, Patrick I-3
Tribastone, M. II-343
Trtík, Marek III-90, III-406
Tschaikowski, M. II-343

U
Ulbrich, Mattias I-268
Urban, Lukáš III-90

V
Vafeiadis, Viktor II-66
van Abbema, Feije I-103
van de Pol, Jaco III-110
van den Brand, Mark III-71
van den Haak, Lars B. III-71
van der Vegt, Marck II-279
van Dijk, Tom I-103
Venkatesh, R. II-175, III-393
Vojdani, Vesal III-335, III-381
Vojnar, Tomáš I-188
Volk, Matthias II-258

W
Wachowitz, Henrik III-359
Wang, Benjie III-3
Wang, Tzu-Fan II-363
Wang, Zhen III-347
Watanabe, Kazuki II-279
Wendler, Philipp III-359
Westhofen, Lukas I-167
Whalen, Mike I-3
Wiesler, Julian I-268
Wijs, Anton II-23, III-71
Winkler, Tobias II-237
Witt, Sascha I-268

X
Xu, Dong III-24

Y
Yi, Pu (Luke) II-3

Z
Zaoral, Lukáš III-406
Zhang, Leping II-217
Zhang, Xiyue III-3
Zhao, Yongwang II-217
Zuleger, Florian I-188