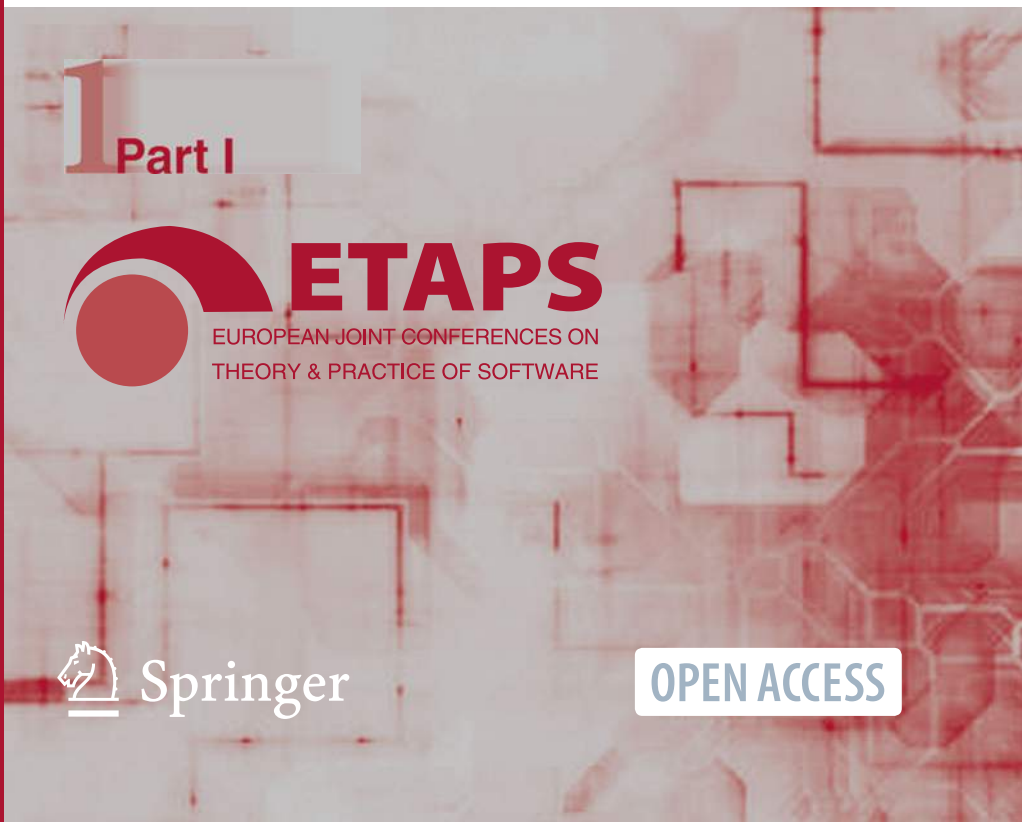
Sriram Sankaranarayanan
Natasha Sharygina (Eds.)

LNCS 13993 ARCoSS

Tools and Algorithms for the Construction and Analysis of Systems

29th International Conference, TACAS 2023
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022
Paris, France, April 22–27, 2023
Proceedings, Part I



Sriram Sankaranarayanan ·
Natasha Sharygina
Editors

Tools and Algorithms for the Construction and Analysis of Systems

29th International Conference, TACAS 2023
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2022
Paris, France, April 22–27, 2023
Proceedings, Part I

Springer

Editors
Sriram Sankaranarayanan
University of Colorado
Boulder, CO, USA

Natasha Sharygina
University of Lugano
Lugano, Switzerland

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-30822-2 ISBN 978-3-031-30823-9 (eBook)
https://doi.org/10.1007/978-3-031-30823-9

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

ETAPS Foreword

Welcome to the 26th ETAPS! ETAPS 2023 took place in Paris, the beautiful capital of France. ETAPS 2023 was the 26th 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 2023 received 361 submissions in total, 124 of which were accepted, yielding an overall acceptance rate of 34.3%. 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 2023 featured the unifying invited speakers Véronique Cortier (CNRS, LORIA laboratory, France) and Thomas A. Henzinger (Institute of Science and Technology, Austria) and the conference-specific invited speakers Mooly Sagiv (Tel Aviv University, Israel) for ESOP and Sven Apel (Saarland University, Germany) for FASE. Invited tutorials were provided by Ana-Lucia Varbanescu (University of Twente and University of Amsterdam, The Netherlands) on heterogeneous computing and Joost-Pieter Katoen (RWTH Aachen, Germany and University of Twente, The Netherlands) on probabilistic programming.

As part of the programme we had the second edition of TOOLympics, an event to celebrate the achievements of the various competitions or comparative evaluations in the field of ETAPS.

ETAPS 2023 was organized jointly by Sorbonne Université and Université Sorbonne Paris Nord. Sorbonne Université (SU) is a multidisciplinary, research-intensive and worldclass academic institution. It was created in 2018 as the merge of two first-class research-intensive universities, UPMC (Université Pierre and Marie Curie) and Paris-Sorbonne. SU has three faculties: humanities, medicine, and 55,600 students (4,700 PhD students; 10,200 international students), 6,400 teachers, professor-researchers and 3,600 administrative and technical staff members. Université Sorbonne Paris Nord is one of the thirteen universities that succeeded the University of Paris in 1968. It is a major teaching and research center located in the north of Paris. It has five campuses, spread over the two departments of Seine-Saint-Denis and Val

d'Oise: Villetaneuse, Bobigny, Saint-Denis, the Plaine Saint-Denis and Argenteuil. The university has more than 25,000 students in different fields, such as health, medicine, languages, humanities, and science. The local organization team consisted of Fabrice Kordon (general co-chair), Laure Petrucci (general co-chair), Benedikt Bollig (workshops), Stefan Haar (workshops), Étienne André (proceedings and tutorials), Céline Ghibaudo (sponsoring), Denis Poitrenaud (web), Stefan Schwoon (web), Benoît Barbot (publicity), Nathalie Sznajder (publicity), Anne-Marie Reytier (communication), Hélène Pétridis (finance) and Véronique Criart (finance).

ETAPS 2023 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), EASST (European Association of Software Science and Technology), Lip6 (Laboratoire d'Informatique de Paris 6), LIPN (Laboratoire d'informatique de Paris Nord), Sorbonne Université, Université Sorbonne Paris Nord, CNRS (Centre national de la recherche scientifique), CEA (Commissariat à l'énergie atomique et aux énergies alternatives), LMF (Laboratoire méthodes formelles), and Inria (Institut national de recherche en informatique et en automatique).

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

Other members of the steering committee are: Dirk Beyer (Munich), Luís Caires (Lisboa), Ana Cavalcanti (York), Bernd Finkbeiner (Saarland), Reiko Heckel (Leicester), Joost-Pieter Katoen (Aachen and Twente), Naoki Kobayashi (Tokyo), Fabrice Kordon (Paris), Laura Kovács (Vienna), Orna Kupferman (Jerusalem), Leen Lambers (Cottbus), Tiziana Margaria (Limerick), Andrzej Murawski (Oxford), Laure Petrucci (Paris), Elizabeth Polgreen (Edinburgh), Peter Ryan (Luxembourg), Sriram Sankaranarayanan (Boulder), Don Sannella (Edinburgh), Natasha Sharygina (Lugano), Pawel Sobocinski (Tallinn), Sebastián Uchitel (London and Buenos Aires), Andrzej Wasowski (Copenhagen), Stephanie Weirich (Pennsylvania), Thomas Wies (New York), 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-Verlag GmbH for their support. I hope you all enjoyed ETAPS 2023.

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

April 2023

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

We are pleased to present the proceedings of TACAS 2023, the 29th edition of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems held as part of the 26th European Joint Conferences on Theory and Practice of Software (ETAPS 2023), April 24–28, 2023 in Paris, France. TACAS brings together a community of researchers, developers, and end-users who are broadly interested in rigorous algorithmic techniques for the construction and analysis of systems. The conference is a venue that interleaves various disciplines including formal verification of software and hardware systems, static analysis, program synthesis, verification of machine learning/autonomous systems, probabilistic programming, SAT/SMT solving, constraint solving, static analysis, automated theorem proving and Cyber-Physical Systems.

There were five submission categories for TACAS 2023:

1. **Regular research papers** advancing the theoretical foundations for the construction and analysis of systems.
2. **Case study papers** describing the application of state-of-the-art research techniques on real-world applications.
3. **Regular tool papers** presenting a new tool, a new tool component, or novel extensions to an existing tool of interest to the community.
4. **Tool demonstration papers** focusing on the usage aspects of tools.
5. **SV-COMP competition tool papers** organized as a separate conference track.

Regular research, case study, and regular tool papers were restricted to a total of sixteen pages, and tool demonstration papers to six pages, exclusive of references.

This year 169 papers were submitted to TACAS, consisting of 119 regular research papers, 34 regular tool and case study papers, and 16 tool demonstration papers. Each paper was reviewed by three Program Committee (PC) members, who made use of subreviewers. As a result, the PC accepted in total 62 papers, among which there were 45 regular papers, 11 regular tool/case-study papers and 6 tool demonstration papers. The PC members were pleasantly surprised by an unusually large number of strong submissions. Almost all accepted papers had either all positive reviews or a "championing" program committee member who argued in favor of accepting the paper. Furthermore, all accepted papers had a positive average score. One paper was accepted conditionally and successfully "shepherded" by the PC.

Similarly to previous years, it was possible to submit an artifact alongside a paper, which was mandatory for regular tool and tool demonstration papers. An artifact might consist of tools, models, proofs, or other data required for validation of the results

of the paper. The Artifact Evaluation Committee (AEC) reviewed the artifacts based on their documentation, ease of use, and, most importantly, whether the results presented in the corresponding paper could be accurately reproduced. The evaluation was carried out using a standardized virtual machine to ensure consistency of the results, except for 4 artifacts that had special hardware or software requirements. The evaluation had two rounds. The first round was carried out in parallel with the work of the PC and evaluated the artifacts for all the submitted regular tool and tool demo papers. The judgment of the AEC was communicated to the PC and weighed in their discussion (the PC rejected a total of 4 papers in this phase). The second round took place after the paper acceptance notifications were sent out so the authors of accepted research and case-study papers could submit their artifacts. In both rounds, the AEC provided 3 reviews per artifact and communicated with the authors to resolve apparent technical issues. In total, 69 artifacts were submitted (51 in the first round and 18 in the second), and the AEC evaluated a total of 64 artifacts regarding their availability, functionality, and/or reusability. Finally, among the 62 accepted papers, the AEC awarded 32 functional badges, 21 reusable badges, and 33 available badges. Such badges appear on the first page of each paper to certify the properties of each artifact.

As a separate conference track, TACAS 2023 hosted the 12th Competition on Software Verification (SV-COMP 2023). SV-COMP is the annual comparative evaluation of tools for automatic software verification and witness validation. The TACAS proceedings contain a selection of 13 short papers that describe participating verification systems and a report presenting the results of the competition. These papers were reviewed by a separate program committee (the competition jury); each of the papers was assessed by at least three reviewers. A total of 52 verification systems were systematically evaluated, with 34 developer teams from ten countries, including five submissions from industry. Two sessions in the TACAS program were reserved for the competition: presentations by the competition chair and the participating development teams in the first session and an open community meeting in the second session.

We would like to thank all the people who helped to make TACAS 2023 successful. First, we would like to thank the authors for submitting their papers to TACAS 2023. The PC members and additional reviewers did a great job in reviewing papers: they contributed informed and detailed reports and engaged in the PC discussions. We also thank the steering committee, and especially its chair, Joost-Pieter Katoen, for his valuable advice. Lastly, we would like to thank the overall organization team of ETAPS 2023.

April 2023

Sriram Sankaranarayanan
Natasha Sharygina
Grigory Fedyukovich
Sergio Mover
Dirk Beyer

Organization

Program Committee Chairs

Sriram Sankaranarayanan	University of Colorado Boulder, USA
Natasha Sharygina	University of Lugano, Switzerland

Program Committee

Christel Baier	TU Dresden, Germany
Haniel Barbosa	Universidade Federal de Minas Gerais, Brazil
Ezio Bartocci	TU Wien, Austria
Dirk Beyer	LMU Munich, Germany
Armin Biere	Freiburg, Germany
Nikolaj Bjørner	Microsoft, USA
Roderick Bloem	Graz University of Technology, Austria
Ahmed Bouajjani	IRIF, Université Paris Cité, France
Hana Chockler	King's College London, UK
Alessandro Cimatti	Fondazione Bruno Kessler, Italy
Rance Cleaveland	University of Maryland, USA
Javier Esparza	TU Munich, Germany
Chuchu Fan	MIT, USA
Grigory Fedyukovich	Florida State University, USA
Bernd Finkbeiner	CISPA Helmholtz Center for Information Security, Germany
Martin Fränzle	Carl von Ossietzky Universität Oldenburg, Germany
Khalil Ghorbal	Inria, France
Laure Gonnord	Grenoble-INP/LCIS, France
Orna Grumberg	Technion - Israel Institute of Technology, Israel
Kim Guldstrand Larsen	Aalborg University, Denmark
Arie Gurfinkel	University of Waterloo, Canada
Ranjit Jhala	University of California, San Diego, USA
Laura Kovacs	TU Wien, Austria
Alexander Kulikov	St. Petersburg Department of Steklov Institute of Mathematics, Russia
Bettina Könighofer	Graz University of Technology, Austria
Wenchao Li	Boston University, USA
Sergio Mover	Ecole Polytechnique, France
Peter Müller	ETH Zurich, Switzerland
Kedar Namjoshi	Nokia Bell Labs, USA
Aina Niemetz	Stanford University, USA
Corina Pasareanu	CMU, NASA, KBR, USA
Nir Piterman	University of Gothenburg, Sweden

Philipp Ruemmer	University of Regensburg, Germany
Krishna S.	Indian Institute of Technology Bombay, India
Cesar Sanchez	IMDEA Software Institute, Spain
Sharon Shoham	Tel Aviv University, Israel
Fabio Somenzi	University of Colorado Boulder, USA
Cesare Tinelli	University of Iowa, USA
Stavros Tripakis	Northeastern University, USA
Frits Vaandrager	Radboud University, Netherlands
Yakir Vizel	Technion, Israel
Tomas Vojnar	Brno University of Technology, Czechia
Naijun Zhan	Chinese Academy of Sciences, China
Lijun Zhang	Chinese Academy of Sciences, China
Florian Zuleger	Vienna University of Technology, Austria

Artifact Evaluation Committee Chairs

Grigory Fedyukovich	Florida State University, USA
Sergio Mover	Ecole Polytechnique, France

Artifact Evaluation Committee

Timothy A. Thijm	Princeton University, USA
Leonardo Alt	Ethereum Foundation, Germany
Pedro H. A. de Amorim	Cornell University, USA
Martin Blicha	University of Lugano, Switzerland
Alexander Bork	RWTH Aachen, Germany
Priyanka Darke	Tata Consultancy Services, India
Emanuele De Angelis	IASI-CNR, Rome, Italy
Jip J. Dekker	Monash University, Australia
Zafer Esen	Uppsala University, Sweden
Aleksandr Fedchin	Tufts University, USA
Hadar Frenkel	CISPA – Helmholtz Center for Information Security, Germany
Pamina Georgiou	Vienna University of Technology, Austria
Thomas Møller Grosen	Aalborg University, Denmark
Ahmed Irfan	SRI International, USA
Martin Jonas	Fondazione Bruno Kessler, Italy
Dongjoo Kim	Seoul National University, South Korea
Satoshi Kura	National Institute of Informatics, Japan
Denis Mazzucato	Ecole Normale Superieure, France
Baoluo Meng	GE Global Research, USA
Federico Mora	University of California, Berkeley, USA
Dmitry Mordvinov	Saint-Petersburg State University, JetBrains Research, Russia
Srinidhi Nagendra	Chennai Mathematical Institute, India
Andres Noetzli	Stanford University, USA

Jiří Pavela	FIT VUT, Czechia
Sumanth Prabhu	TRDDC, India
Felipe R. Monteiro	Amazon Web Services, USA
Olli Saarikivi	Microsoft Research, USA
Saeid Tizpaz Niari	University of Texas at El Paso, USA
Hari Govind Vediramana Krishnan	University of Waterloo, Canada
Jingbo Wang	University of Southern California, USA
Anton Xue	University of Pennsylvania, USA
Hansol Yoon	Republic of Korea Air Force, South Korea

Program Committee and Jury—SV-COMP

Dirk Beyer (Chair)	LMU Munich, Germany
Viktor Malík (2LS)	TU Brno, Czechia
Lei Bu (BRICK)	Nanjing University, China
Marek Chalupa (Bubaak)	ISTA, Austria
Michael Tautschnig (CBMC)	Queen Mary University London, UK
Henrik Wachowitz (CPAchecker)	LMU Munich, Germany
Hernán Ponce de León (Dartagnan)	Huawei Dresden Research, Germany
Fei He (Deagle)	Tsinghua University, China
Fatimah Aljaafari (EBF)	University of Manchester, UK
Rafael Sá Menezes (ESBMC-kind)	University of Manchester, UK
Martin Spiessl (Frama-C-SV)	LMU Munich, Germany
Falk Howar (GDart, GDart-LLVM)	TU Dortmund, Germany
Simmo Saan (Goblint)	University of Tartu, Estonia
William Leeson (Graves-CPA, Graves-Par)	University of Virginia, USA
Soha Hussein (Java-Ranger)	University of Minnesota, USA
Peter Schrammel (JBMC)	University of Sussex/Diffblue, UK
Gidon Ernst (Korn)	LMU Munich, Germany
Tong Wu (LF-checker)	University of Manchester, UK
Vesal Vojdani (Locksmith)	University of Tartu, Estonia
Lei Bu (MLB)	Nanjing University, China
Raphaël Monat (Mopsa)	Inria and University of Lille, France
Cedric Richter (PeSCo-CPA)	University of Oldenburg, Germany
Jie Su (PIChecker)	Xidian University, China
Marek Trtik (Symbiotic)	Masaryk University, Brno, Czechia
Levente Bajczi (Theta)	Budapest University of Technology and Economics, Hungary

Matthias Heizmann (UAutomizer)	University of Freiburg, Germany
Dominik Klumpp (UGemCutter)	University of Freiburg, Germany
Frank Schüssele (UKojak)	University of Freiburg, Germany
Daniel Dietsch (UTaipan)	University of Freiburg, Germany
Priyanka Darke (VeriAbs, VeriAbsL)	Tata Consultancy Services, India
Raveendra Kumar M. (VeriFuzz)	Tata Consultancy Services, India
HaiPeng Qu (VeriOover)	Ocean University of China, China

Steering Committee

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

Additional Reviewers

Abd Alrahman, Yehia
Ahmad, H. M. Sabbir
An, Jie
Asarin, Eugene
Azzopardi, Shaun
Bacci, Giorgio
Baier, Daniel
Balakrishnan, Gogul
Balasubramanian, A. R.
Baumeister, Jan
Becchi, Anna
Ben Shimon, Yoav
Berger, Guillaume
Beutner, Raven
Bily, Aurel
Blicha, Martin
Bombardelli, Alberto
Brieger, Marvin
Brizzio, Matías
Bunk, Thomas
Caillaud, Benoît
Cano Córdoba, Filip
Ceresa, Martin
Ceska, Milan
Chen, Mingshuai
Chen, Xin
Chen, Yilei
Chiari, Michele
Czerner, Philipp
Dardinier, Thibault
Dawson, Charles
De Masellis, Riccardo
Debrestian, Darin
Di Stefano, Luca
Egolf, Derek
Elad, Neta
Elashkin, Andrey
Esen, Zafer
Fazekas, Katalin
Feng, Shenghua
Ferres, Bruno
Fiedor, Jan
Fleury, Mathias
Fontaine, Pascal

Frenkel, Eden
Frenkel, Hadar
Froleyks, Nils
Fu, Feisi
Garcia-Contreras, Isabel
Garg, Kunal
Georgiou, Pamina
Gianola, Alessandro
Gigerl, Barbara
Goorden, Martijn
Gorostiaga, Felipe
Goyal, Srajan
Griggio, Alberto
Grosen, Thomas Møller
Gstrein, Bernhard
Gupta, Ashutosh
Habermehl, Peter
Hader, Thomas
Hadzic, Vedad
Hagemann, Willem
Hamza, Ameer
Haring, Johannes
Hausmann, Daniel
Havlena, Vojtěch
Hermo, Montserrat
Holík, Lukáš
Hozzová, Petra
Huang, Chao
Huang, Chengchao
Hyvärinen, Antti
Itzhaky, Shachar
Jacobs, Swen
Jaeger, Manfred
Jansen, David N.
Jensen, Nicolaj Østerby
Jha, Prabhat
Jonas, Martin
Junges, Sebastian
Kaki, Gowtham
Kaufmann, Daniela
Kenison, George
Kettl, Matthias
Khalimov, Ayrat
Kifetew, Fitsum
Kiourti, Panagiota
Klüppelholz, Sascha
Kröger, Paul
Käfer, Nikolai
Lal, Akash
Larrauri, Alberto
Larraz, Daniel
Lazic, Marijana
Le, Nham
Lee, Nian-Ze
Lengal, Ondrej
Li, Renjue
Lidell, David
Liu, Jiaxiang
Lopez-Miguel, Ignacio D.
Luttenberger, Michael
Macías, Fernando
Maderbacher, Benedikt
McClurg, Jedidiah
Meng, Yue
Metzger, Niklas
Michelland, Sebastien
Monniaux, David
Moosbrugger, Marcel
Nadel, Alexander
Nam, Seunghyeon
Nesterini, Eleonora
Neufeld, Emery
Nickovic, Dejan
Noetzli, Andres
Oliveira Da Costa, Ana
Otoni, Rodrigo
Parthasarathy, Gaurav
Paxian, Tobias
Pluska, Alexander
Poli, Federico
Pontiggia, Francesco
Prandi, Davide
Pranger, Stefan
Preiner, Mathias
Radanne, Gabriel
Rakow, Astrid
Rappoport, Omer
Rauh, Andreas
Rawson, Michael
Rebola Pardo, Adrian
Reynolds, Andrew
Riley, Daniel

Rodriguez, Andoni
Rogalewicz, Adam
Román Calvo, Enrique
Rubio, Rubén
Rutledge, Kwesi
Sallinger, Sarah
Sankaranarayanan, Sriram
Schlichtkrull, Anders
Schoisswohl, Johannes
Schultz, William
Schupp, Stefan
Schwammberger, Maike
Sextl, Florian
Siber, Julian
So, Oswin
Sogokon, Andrew
Spiessl, Martin
Steen, Alexander
Su, Yusen
Susi, Angelo
Síč, Juraj
Tappler, Martin
Thibault, Joan
Ting, Gan
Treml, Lilly Maria
Trivedi, Ashutosh
Turrini, Andrea
Varanasi, Sarat Chandra
Vediramana Krishnan, Hari Govind
Visconti, Ennio
Wachowitz, Henrik
Wand, Michael
Wardega, Kacper
Weininger, Maximilian
Wendler, Philipp
Wienhöft, Patrick
Wu, Hao
Wu, Haoze
Xue, Anton
Yadav, Drishti
Yang, Pengfei
Yang, Ruixiao
Yu, Chenning
Yu, Mingxin
Zavalia, Lucas
Zhan, Bohua
Zhang, Hanwei
Zhang, Songyuan
Zhou, Weichao
Zhou, Yuhao
Zimmermann, Martin
Zlatkin, Ilia

Contents – Part I

Machine Learning/Neural Networks

Automata

Proofs

Constraint Solving/Blockchain

Markov Chains/Stochastic Control

Verification

Contents – Part II

Tools (Regular Papers)

Synthesis

Graphs/Probabilistic Systems

Runtime Monitoring/Program Analysis

12th Competition on Software Verification — SV-COMP 2023

Invited Talk

A Learner-Verifier Framework for Neural Network Controllers and Certificates of Stochastic Systems*

Krishnendu Chatterjee[1], Thomas A. Henzinger[1](✉), Mathias Lechner[2], and Đorđe Žikelić[1]

[1] Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
{krishnendu.chatterjee,tah,djordje.zikelic}@ist.ac.at
[2] Massachusetts Institute of Technology (MIT), Cambridge, MA, USA
mlechner@mit.edu

Abstract. Reinforcement learning has received much attention for learning controllers of deterministic systems. We consider a learner-verifier framework for stochastic control systems and survey recent methods that formally guarantee a conjunction of reachability and safety properties. Given a property and a lower bound on the probability of the property being satisfied, our framework jointly learns a control policy and a formal certificate to ensure the satisfaction of the property with a desired probability threshold. Both the control policy and the formal certificate are continuous functions from states to reals, which are learned as parameterized neural networks. While in the deterministic case, the certificates are invariant and barrier functions for safety, or Lyapunov and ranking functions for liveness, in the stochastic case the certificates are supermartingales. For certificate verification, we use interval arithmetic abstract interpretation to bound the expected values of neural network functions.

Keywords: Learning-based control · Stochastic systems · Martingales. · Formal verification

1 Introduction

Learning-based control and verification of learned controllers. Learning-based control and reinforcement learning (RL) were empirically demonstrated to have enormous potential to solve highly non-linear control tasks. However, their deployment in safety-critical scenarios such as autonomous driving or healthcare requires safety assurances. Most safety-aware RL algorithms optimize expected reward while only empirically trying to maximize safety probability. This together with the non-explainable nature of neural network controllers obtained via deep RL raise questions about the trustworthiness of learning-based methods for safety-critical applications [9,27]. To that end, formal verification of learned

*This work was supported in part by the ERC-2020-AdG 101020093, ERC CoG 863818 (FoRM-SMArt) and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 665385.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 3–25, 2023.
https://doi.org/10.1007/978-3-031-30823-9_1

controllers as well as learning-based control with formal safety guarantees have become very active research topics.

Learning certificate functions. A classical approach to formally proving properties of dynamical systems is to compute a certificate function. A *certificate function* [26] is a function that assigns real values to system states and its defining conditions imply satisfaction of the property. Thus, in order to prove the property of interest, it suffices to compute a certificate function for that property. For instance, Lyapunov functions [46] and barrier functions [50] are standard certificate functions for proving reachability of some target set and avoidance of some unsafe set of system states, respectively, when the system dynamics are deterministic. While both Lyapunov and barrier functions are well-studied concepts in dynamical systems theory, early methods for their computation either required designing the certificates by hand or using computationally intractable numerical procedures. A more recent approach reduces certificate computation to a semi-definite programming problem by using sum-of-squares (SOS) techniques [33,49,37]. However, a limitation of this approach is that it is only applicable to polynomial systems and computation of polynomial certificate functions, whereas it is not applicable to systems with general non-linearities. Moreover, SOS methods do not scale well with the dimension of the system.

Learning-based methods are a promising approach to overcome these limitations and they have received much attention in recent years. These methods jointly learn a neural network control policy and a neural network certificate function, e.g. a Lyapunov function [53,18,3,17] or a barrier function [38,58,52,1], depending on the property of interest. The neural network certificate is then formally verified, ensuring that these methods provide formal guarantees. Both learning and verification procedures developed for verifying neural network certificates are not restricted to polynomial dynamical systems. See [26] for an overview of existing learning-based control methods that learn a certificate function to verify a system property in deterministic dynamical systems.

Prior works – deterministic dynamical systems. While the above works present significant advancements in learning-based control and verification of dynamical systems, they are predominantly restricted to *deterministic* dynamical systems. In other words, they assume that they have access to the exact dynamics function according to which the system evolves. However, for most control tasks, the underlying models used by control methods are imperfect approximations of real systems inferred from observed data. Thus, control and verification methods should also account for model uncertainty due to the noise in observed data and the approximate nature of model inference.

This survey – stochastic dynamical systems. In this work, we survey recent developments in learning-based methods for control and verification of discrete-time *stochastic* dynamical systems, based on [44,68]. Stochastic dynamical systems use probability distributions to quantify and model uncertainty. In stochastic dynamical systems, given a property of interest and a probability parameter $p \in [0, 1]$, the goal is to learn a control policy and a formal certificate which guarantees that the system under the learned policy satisfies the property of interest with probability at least p.

Supermartingale certificate functions. Lyapunov functions and barrier functions can be used to prove properties in deterministic dynamical systems, however they are not applicable to stochastic dynamical systems and do not allow reasoning about the probability of a property being satisfied. Instead, the learning-based methods of [44,68] use *supermartingale certificate functions* to formally prove properties in stochastic systems. Supermartingales are a class of stochastic processes that decrease in expected value at every time step [66]. Their nice convergence properties and concentration bounds allow their use in designing certificate functions for stochastic dynamical systems. In particular, *ranking supermartingales (RSMs)* [15,44] were used to verify probability 1 reachability and *stochastic barrier functions (SBFs)* [50] were used to verify safety with the specified probability $p \in [0, 1]$. *Reach-avoid supermartingales (RASMs)* [68] unify and extend these two concepts and were used to verify reach-avoidance properties with the specified probability $p \in [0, 1]$, i.e. a conjunction of reachability and safety properties. We define and compare these concepts in Section 3.

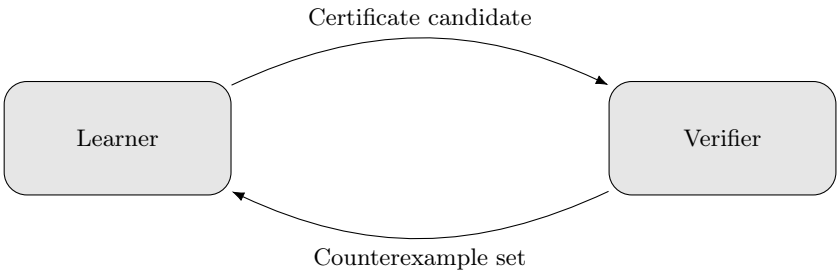


Fig. 1: Schematic illustration of the learner-verifier loop.

Learner-verifier framework for stochastic dynamical systems. In Section 4, we then present a *learner-verifier framework* of [44,68] for learning-based control and for the verification of learned controllers in stochastic dynamical systems in a counterexample guided inductive synthesis (CEGIS) fashion [55]. The algorithm jointly learns a neural network control policy and a neural network supermartingale certificate function. It consists of two modules – the learner, which learns a policy and a supermartingale certificate function candidate, and the verifier, which then formally verifies the candidate supermartingale certificate function. If the verification step fails, the verifier computes counterexamples and passes them back to the learner, which tries to learn a new candidate. This loop is repeated until a candidate is successfully verified, see Fig. 1.

This framework builds on the existing learner-verifier methods for learning-based control in deterministic dynamical systems [18,2,26]. However, the extension of this framework to stochastic dynamical systems and the synthesis of supermartingale certificate functions is far from straightforward. In particular, the methods of [18,2] use knowledge of the deterministic dynamics function to reduce the verification task to a decision procedure and use an off-the-shelf solver. However, verification of the expected decrease condition of supermartin-

gale certificates by reduction to a decision procedure would require being able to compute a closed-form expression of the expected value of a neural network function over a probability distribution and provide it to the solver. It is not clear how the closed-form expression can be computed, and it is not known whether the closed-form expression exists in the general case.

This challenge is solved by using a method for efficient computation of tight *upper and lower bounds on the expected value* of a neural network function. The verifier module then verifies the expected decrease condition by *discretizing* the state space and formally verifying a slightly stricter condition at the discretization points by using the computed expected value bounds. By carefully choosing the mesh of the discretization and adding an additional error term, we obtain a sound verification method applicable to general Lipschitz continuous systems. The expected value bound computation for neural network functions relies on interval arithmetic and abstract interpretation, and since it is of independent interest, we discuss it in detail in Section 5. We are not aware of any existing methods that tackle this problem.

Extension to general stochastic certificates. We conclude this survey with a discussion of possible extensions of the learner-verifier framework in Section 6 and of related work in Section 7.

2 Preliminaries

We consider discrete-time stochastic dynamical systems defined via

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t, \omega_t),\ \mathbf{x}_0 \in \mathcal{X}_0.$$

The function $f : \mathcal{X} \times \mathcal{U} \times \mathcal{N} \to \mathcal{X}$ is the dynamics function of the system and $t \in \mathbb{N}_0$ is the time index. We use $\mathcal{X} \subseteq \mathbb{R}^m$ to denote the system state space, $\mathcal{U} \subseteq \mathbb{R}^n$ the control action space and $\mathcal{N} \subseteq \mathbb{R}^p$ the stochastic disturbance space. For each $t \in \mathbb{N}_0$, $\mathbf{x}_t \in \mathcal{X}$ the state of the system, $\mathbf{u}_t \in \mathcal{U}$ the action and $\omega_t \in \mathcal{N}$ the stochastic disturbance vector at time t. The set $\mathcal{X}_0 \subseteq \mathcal{X}$ is the set of initial states. In each time step, $\mathbf{u}_t$ is chosen according to a control policy $\pi : \mathcal{X} \to \mathcal{U}$, i.e. $\mathbf{u}_t = \pi(\mathbf{x}_t)$, and ω_t is sampled according to some specified probability distribution d over $\mathbb{R}^p$. The dynamics function f, control policy π and probability distribution d together define a stochastic feedback loop system.

A trajectory of the system is a sequence $(\mathbf{x}_t, \mathbf{u}_t, \omega_t)_{t \in \mathbb{N}_0}$ such that, for each $t \in \mathbb{N}_0$, we have $\mathbf{u}_t = \pi(\mathbf{x}_t)$, $\omega_t \in \mathsf{support}(d)$ and $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t, \omega_t)$. For each initial state $\mathbf{x}_0 \in \mathcal{X}$, the system induces a Markov process. This gives rise to the probability space over the set of all trajectories of the system that start in $\mathbf{x}_0$ [51]. We denote the probability measure and the expectation in this probability space by $\mathbb{P}_{\mathbf{x}_0}$ and $\mathbb{E}_{\mathbf{x}_0}$, respectively.

Assumptions. We assume that $\mathcal{X} \subseteq \mathbb{R}^m$, $\mathcal{X}_0 \subseteq \mathbb{R}^m$, $\mathcal{U} \subseteq \mathbb{R}^n$ and $\mathcal{N} \subseteq \mathbb{R}^p$ are all Borel-measurable. This is necessary for the probability space of the set of all system trajectories starting in some initial state to be mathematically well-defined. We also assume that $\mathcal{X} \subseteq \mathbb{R}^m$ is compact (i.e. closed and bounded) and that the dynamics function f is Lipschitz continuous, which are common assumptions in

control theory. Finally, we assume that the probability distribution d is a product of independent univariate probability distributions, which is necessary for efficient sampling and expected value computation.

2.1 Brief Overview of Martingale Theory

In this subsection, we provide a brief overview of definitions and results from martingale theory that lie at the core of formal reasoning about supermartingale certificate functions. We assume that the reader is familiar with the mathematical definitions of probability space, measurability and random variables, see [66] for the necessary background. The results in this subsection will help in building an intuition on supermartingale certificate functions, but omitting them would not prevent the reader from following the rest of this paper.

Probability space. A probability space is a triple $(\Omega, \mathcal{F}, \mathbb{P})$ where Ω is a state space, $\mathcal{F}$ is a sigma-algebra and $\mathbb{P}$ is a probability measure which is required to satisfy Kolmogorov axioms [66]. A random variable is a function $X : \Omega \to \mathbb{R}$ that is $\mathcal{F}$-measurable. We use $\mathbb{E}[X]$ to denote the *expected value* of X. A *(discrete-time) stochastic process* is a sequence $(X_i)_{i=0}^{\infty}$ of random variables in $(\Omega, \mathcal{F}, \mathbb{P})$.

Conditional expectation. Let X be a random variable in a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Given a sub-$\sigma$-algebra $\mathcal{F}' \subseteq \mathcal{F}$, a *conditional expectation* of X given $\mathcal{F}'$ is an $\mathcal{F}'$-measurable random variable Y such that, for each $A \in \mathcal{F}'$, we have

$$\mathbb{E}[X \cdot \mathbb{I}(A)] = \mathbb{E}[Y \cdot \mathbb{I}(A)].$$

Here, $\mathbb{I}(A) : \Omega \to \{0, 1\}$ is an *indicator function* of A defined via $\mathbb{I}(A)(\omega) = 1$ if $\omega \in A$, and $\mathbb{I}(A)(\omega) = 0$ if $\omega \notin A$. Intuitively, conditional expectation of X given $\mathcal{F}'$ is an $\mathcal{F}'$-measurable random variable that behaves like X whenever its expected value is taken over an event in $\mathcal{F}'$. Conditional expectation of a random variable X given $\mathcal{F}'$ is guaranteed to exist if X is real-valued and non-negative [66]. Moreover, for any two conditional expectations Y and Y' of X given $\mathcal{F}'$, we have that $\mathbb{P}[Y = Y'] = 1$. Therefore, the conditional expectation is almost-surely unique and we may pick one such random variable as a canonical conditional expectation and denote it by $\mathbb{E}[X \mid \mathcal{F}']$.

Supermartingales. Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space and $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \cdots \subseteq \mathcal{F}$ be an increasing sequence of sub-σ-algebras in $\mathcal{F}$ with respect to inclusion. A non-negative *supermartingale* with respect to $(\mathcal{F}_i)_{i=0}^{\infty}$ is a stochastic process $(X_i)_{i=0}^{\infty}$ such that each X_i is $\mathcal{F}_i$-measurable, and $X_i(\omega) \geq 0$ and $\mathbb{E}[X_{i+1} \mid \mathcal{F}_i](\omega) \leq X_i(\omega)$ hold for each $\omega \in \Omega$ and $i \geq 0$. Intuitively, the second condition says that the expected value of X_{i+1} given the value of X_i has to decrease. This condition is formalized by using conditional expectation.

The following two results that will be key technical ingredients in our design of supermartingale certificate functions. The first theorem shows that nonnegative supermartingales have nice convergence properties and converge almost-surely to some finite value. The second theorem bounds the probability that the value of the supemartingale ever exceeds some threshold, and it will allow us to bound from above the probability of occurrence of some bad event.

Theorem 1 (Supermartingale convergence theorem [66]). *Let $(X_i)_{i=0}^{\infty}$ be a nonnegative supermartingale with respect to $(\mathcal{F}_i)_{i=0}^{\infty}$. Then, there exists a random variable X_∞ in $(\Omega, \mathcal{F}, \mathbb{P})$ to which the supermartingale converges to with probability 1, i.e. $\mathbb{P}[\lim_{i\to\infty} X_i = X_\infty] = 1$.*

Theorem 2 ([41]). *Let $(X_i)_{i=0}^{\infty}$ be a nonnegative supermartingale with respect to $(\mathcal{F}_i)_{i=0}^{\infty}$. Then, for every real $\lambda > 0$, we have $\mathbb{P}[\sup_{i\geq 0} X_i \geq \lambda] \leq \mathbb{E}[X_0]/\lambda$.*

2.2 Problem Statement

We now formally define the properties and control tasks that we focus on in this work. In what follows, let $\mathcal{X}_t, \mathcal{X}_u \subseteq \mathcal{X}$ be disjoint Borel-measurable sets and $p \in [0, 1]$ be a lower bound on the probability with which the system under the learned controller needs to satisfy the property:

- *Reachability.* Let $\text{Reach}(\mathcal{X}_t) = \{(\mathbf{x}_t, \mathbf{u}_t, \omega_t)_{t\in\mathbb{N}_0} \mid \exists t \in \mathbb{N}_0 . \mathbf{x}_t \in \mathcal{X}_t\}$ be the set of all trajectories that reach the target set $\mathcal{X}_t$. The goal is to learn a control policy under which the system reaches $\mathcal{X}_t$ with probability at least p, i.e. $\mathbb{P}_{\mathbf{x}_0}[\text{Reach}(\mathcal{X}_t)] \geq p$ holds for every initial state $\mathbf{x}_0 \in \mathcal{X}_0$.
- *Safety (or avoidance).* Let $\text{Safe}(\mathcal{X}_u) = \{(\mathbf{x}_t, \mathbf{u}_t, \omega_t)_{t\in\mathbb{N}_0} \mid \forall t' \leq t . \mathbf{x}_{t'} \notin \mathcal{X}_u\}$ be the set of all trajectories that do not visit the unsafe set $\mathcal{X}_u$. The goal is to learn a control policy under which the system stays away from $\mathcal{X}_u$ with probability at least p, i.e. $\mathbb{P}_{\mathbf{x}_0}[\text{Safe}(\mathcal{X}_u)] \geq p$ holds for every initial state $\mathbf{x}_0 \in \mathcal{X}_0$.
- *Reach-avoidance.* Let $\text{ReachAvoid}(\mathcal{X}_t, \mathcal{X}_u) = \{(\mathbf{x}_t, \mathbf{u}_t, \omega_t)_{t\in\mathbb{N}_0} \mid \exists t \in \mathbb{N}_0 . \mathbf{x}_t \in \mathcal{X}_t \wedge (\forall t' \leq t . \mathbf{x}_{t'} \notin \mathcal{X}_u)\}$ be the set of all trajectories that reach $\mathcal{X}_t$ without reaching $\mathcal{X}_u$. The goal is to learn a control policy under which the system reaches $\mathcal{X}_t$ while staying away from $\mathcal{X}_u$ with probability at least p, i.e. $\mathbb{P}_{\mathbf{x}_0}[\text{ReachAvoid}(\mathcal{X}_t, \mathcal{X}_u)] \geq p$ holds for every initial state $\mathbf{x}_0 \in \mathcal{X}_0$.

3 Supermartingale Certificate Functions

We now overview three classes of supermartingale certificate functions that formally prove reachability, safety and reach-avoidance properties. *Supermartingale certificate functions* do not refer to a single class of certificate functions. Rather, we use this term to refer to all certificate functions that exhibit a supermartingale-like behavior and can formally verify properties in stochastic dynamical systems. In what follows, we assume that the control policy π is fixed. In the following section, we will then present a learner-verifier framework for jointly learning a control policy and a supermartingale certificate function.

RSMs for probability 1 reachability. We start with *ranking supermartingales (RSMs)*, which can prove probability 1 reachability of some target set $\mathcal{X}_t$. Intuitively, an RSM is a continuous function that maps system states to nonnegative real values and is required to strictly decrease in expectation by some $\epsilon > 0$ in

every time step until the target $\mathcal{X}_t$ is reached. Due to the strict expected decrease as well as the Supermartingale Convergence Theorem (Theorem 1), one can show that the existence of an RSM guarantees that the system under policy π reaches $\mathcal{X}_t$ with probability 1. RSMs can be viewed as a stochastic extension of Lyapunov functions. Note that RSMs can only be used to prove probability 1 reachability, but cannot be used to reason about probabilistic reachability. RSMs were originally used for proving almost-sure termination in probabilistic programs [15] and were used to certify probability 1 reachability in stochastic dynamical systems in [44].

Definition 1 (Ranking supermartingales [44]). *Let $\mathcal{X}_t \subseteq \mathcal{X}$ be a target set. A continuous function $V : \mathcal{X} \to \mathbb{R}$ is a* ranking supermartingale (RSM) *with respect to $\mathcal{X}_t$ if it satisfies:*

1. Nonnegativity condition. $V(\mathbf{x}) \geq 0$ *for each* $\mathbf{x} \in \mathcal{X}$.
2. Expected Decrease condition. *There exists* $\epsilon > 0$ *such that, for each* $\mathbf{x} \in \mathcal{X} \backslash \mathcal{X}_t$, *we have* $V(\mathbf{x}) \geq \mathbb{E}_{\omega \sim d}[V(f(\mathbf{x}, \pi(\mathbf{x}), \omega))] + \epsilon$.

Theorem 3 ([44]). *Suppose that there exists an RSM with respect to $\mathcal{X}_t$. Then, for every $\mathbf{x}_0 \in \mathcal{X}_0$, we have $\mathbb{P}_{\mathbf{x}_0}[\text{Reach}(\mathcal{X}_t)] = 1$.*

SBFs for probabilistic safety. On the other hand, *stochastic barrier functions (SBFs)* can prove probabilistic safety. Given an unsafe set $\mathcal{X}_u$ and probability $p \in [0, 1)$, an SBF is also a continuous function mapping system states to nonnegative real values, which is required to decrease in expectation at each time step. However, unlike RSMs, the expected decrease need not be strict and there is no target set. In addition, its initial value must be at most 1, whereas its value upon reaching an unsafe set must be at least $1/(1-p)$. Thus, for the system under policy π to violate the safety constraint, the value of the SBF needs to increase from at most 1 to at least $1/(1-p)$ even though it is required to decrease in expectation. The probability of this event can be bounded from above and shown to be at most $1-p$ by using Theorem 2. We highlight the assumption that $p < 1$, which is necessary for the safety constraint to be mathematically defined. As the name suggests, SBFs are a stochastic extension of barrier functions.

Definition 2 (Stochastic barrier functions [50]). *Let $\mathcal{X}_u \subseteq \mathcal{X}$ be an unsafe set and $p \in [0, 1)$. A continuous function $V : \mathcal{X} \to \mathbb{R}$ is a* stochastic barrier function (SBF) *with respect to $\mathcal{X}_u$ and p if it satisfies:*

1. Nonnegativity condition. $V(\mathbf{x}) \geq 0$ *for each* $\mathbf{x} \in \mathcal{X}$.
2. Initial condition. $V(\mathbf{x}) \leq 1$ *for each* $\mathbf{x} \in \mathcal{X}_0$.
3. Safety condition. $V(\mathbf{x}) \geq \frac{1}{1-p}$ *for each* $\mathbf{x} \in \mathcal{X}_u$.
4. Expected Decrease condition. *For each* $\mathbf{x} \in \mathcal{X}$, *if* $V(\mathbf{x}) \leq \frac{1}{1-p}$ *then* $V(\mathbf{x}) \geq \mathbb{E}_{\omega \sim d}[V(f(\mathbf{x}, \pi(\mathbf{x}), \omega))]$.

Theorem 4 ([50]). *Suppose that there exists an SBF with respect to $\mathcal{X}_u$ and p. Then, for every $\mathbf{x}_0 \in \mathcal{X}_0$, we have $\mathbb{P}_{\mathbf{x}_0}[\text{Safe}(\mathcal{X}_u)] \geq p$.*

RASMs for probabilistic reach-avoidance. Finally, *reach-avoid supermartingales (RASMs)* unify and extend RSMs and SBFs in the sense that they allow simultaneous reasoning about reachability and safety and proving a conjunction of

these properties, i.e. reach-avoid properties. Let $\mathcal{X}_t$ and $\mathcal{X}_u$ be disjoint target and unsafe sets and let $p \in [0, 1)$. Similarly to SBFs, an RASM is a continuous nonnegative function which is required to be initially at most 1 but needs to attain a value that is at least $1/(1-p)$ for the unsafe region to be reached. On the other hand, similarly to RSMs, it is required to strictly decrease in expectation by $\epsilon > 0$ at every time step until either the target set $\mathcal{X}_t$ or a state in which the value is at least $1/(1-p)$ is reached. Thus, RASMs can be viewed as a stochastic extension of both Lyapunov functions and barrier functions, which combines the strict decrease of Lypaunov functions and the level-set reasoning of barrier functions.

Definition 3 (Reach-avoid supermartingales [68]). *Let $\mathcal{X}_t \subseteq \mathcal{X}$ and $\mathcal{X}_u \subseteq \mathcal{X}$ be a target set and an unsafe set, respectively, and let $p \in [0, 1]$ be a probability threshold. Suppose that either $p < 1$ or that $p = 1$ and $\mathcal{X}_u = \emptyset$. A continuous function $V : \mathcal{X} \to \mathbb{R}$ is a* reach-avoid supermartingale (RASM) *with respect to $\mathcal{X}_t$, $\mathcal{X}_u$ and p if it satisfies:*

1. Nonnegativity condition. $V(\mathbf{x}) \geq 0$ *for each* $\mathbf{x} \in \mathcal{X}$.
2. Initial condition. $V(\mathbf{x}) \leq 1$ *for each* $\mathbf{x} \in \mathcal{X}_0$.
3. Safety condition. $V(\mathbf{x}) \geq \frac{1}{1-p}$ *for each* $\mathbf{x} \in \mathcal{X}_u$.
4. Expected Decrease condition. *There exists* $\epsilon > 0$ *such that, for each* $\mathbf{x} \in \mathcal{X} \backslash \mathcal{X}_t$ *at which* $V(\mathbf{x}) \leq \frac{1}{1-p}$, *we have* $V(\mathbf{x}) \geq \mathbb{E}_{\omega \sim d}[V(f(\mathbf{x}, \pi(\mathbf{x}), \omega))] + \epsilon$.

Theorem 5 ([68]). *Suppose that there exists an RASM with respect to $\mathcal{X}_t$, $\mathcal{X}_u$ and p. Then, for every $\mathbf{x}_0 \in \mathcal{X}_0$, we have $\mathbb{P}_{\mathbf{x}_0}[\text{ReachAvoid}(\mathcal{X}_t, \mathcal{X}_u)] \geq p$.*

Note that RASMs indeed unify and generalize the definitions of RSMs and SBFs. First, by setting $\mathcal{X}_u = \emptyset$ and $p = 1$ (so $1/(1-p) = \infty$), RASMs reduce to RSMs as the Initial condition that can be enforced without loss of generality by rescaling. Second, by setting $\mathcal{X}_t = \emptyset$, RASMs reduce to SBFs. In this case, the Expected Decrease condition is strengthened as it requires strict decrease by $\epsilon > 0$. However, the proof of Theorem 5 which we outline below also implies Theorem 4 and $\epsilon > 0$ is only necessary to reason about the reachability of $\mathcal{X}_t$.

We also note that RASMs strictly extend the applicability of RSMs, since RASMs can be used to prove reachability with any lower bound $p \in [0, 1]$ on probability and not only probability 1 reachability. Indeed, if we set $\mathcal{X}_u = \emptyset$ and $p \in [0, 1]$, in order to prove reachability of $\mathcal{X}_t$ with probability at least p the RASMs require strict expected decrease in expectation by $\epsilon > 0$ until either $\mathcal{X}_t$ is reached or the RASM value exceeds $1/(1-p)$ (with $1/(1-p) = \infty$ if $p = 1$).

In the rest of this section, we outline the proof of Theorem 5 that was presented in [68]. This proof also implies Theorem 3 and Theorem 4. We do this to highlight the connection of RSMs, SBFs and RASMs to the mathematical notion of supermartingale processes. We also do this to illustrate the tools from martingale theory that are used in proving soundness of supermatingale certificate functions, as we envision that they may be useful in designing supermatingale certificate functions for more general classes of properties.

Proof (proof sketch of Theorem 5). Here we outline the main ideas behind the proof, and for the full proof we refer the reader to [68]. Let $\mathbf{x}_0 \in \mathcal{X}_0$. We need to

show that $\mathbb{P}_{\mathbf{x}_0}[\text{ReachAvoid}(\mathcal{X}_t, \mathcal{X}_u)] \geq p$. To do this, we consider the probability space $(\Omega_{\mathbf{x}_0}, \mathcal{F}_{\mathbf{x}_0}, \mathbb{P}_{\mathbf{x}_0})$ of trajectories that start in $\mathbf{x}_0$ and for each time step $t \in \mathbb{N}_0$ define a random variable in this probability space via

$$X_t(\rho) = \begin{cases} V(\mathbf{x}_t), & \text{if } \mathbf{x}_i \notin \mathcal{X}_t \text{ and } V(\mathbf{x}_i) < \frac{1}{1-p} \text{ for each } 0 \leq i \leq t \\ 0, & \text{if } \mathbf{x}_i \in \mathcal{X}_t \text{ for some } 0 \leq i \leq t, V(\mathbf{x}_j) < \frac{1}{1-p} \text{ for each } 0 \leq j \leq i \\ \frac{1}{1-p}, & \text{otherwise} \end{cases}$$

for each trajectory $\rho = (\mathbf{x}_t, \mathbf{u}_t, \omega_t)_{t \in \mathbb{N}_0} \in \Omega_{\mathbf{x}_0}$. Hence, $(X_t)_{t=0}^{\infty}$ defines a stochastic process whose value at each time step is equal to the value of V at the current system state unless either the target set $\mathcal{X}_t$ has been reached after which future values of $\mathcal{X}_t$ are set to 0, or a state in which V exceeds $1/(1-p)$ has been reached after which future values of $\mathcal{X}_t$ are set to $1/(1-p)$. It can be shown that $(X_t)_{t=0}^{\infty}$ is a nonnegative supermartingale $(\Omega_{\mathbf{x}_0}, \mathcal{F}_{\mathbf{x}_0}, \mathbb{P}_{\mathbf{x}_0})$. This claim can be proved by using the Nonnegativity and the Expected Decrease condition of RASMs. Here we do not yet need that the expected decrease is strict, i.e. $\epsilon \geq 0$ in the Expected Decrease condition of RASMs is sufficient.

Since $(X_t)_{t=0}^{\infty}$ is a nonnegative supermartingale, substituting $\lambda = 1/(1-p)$ into the inequality in Theorem 2 shows that

$$\mathbb{P}_{\mathbf{x}_0}\Big[\sup_{i \geq 0} X_i \geq \frac{1}{1-p}\Big] \leq (1-p) \cdot \mathbb{E}_{\mathbf{x}_0}[X_0] \leq 1-p.$$

The second inequality follows since $X_0(\rho) = V(\mathbf{x}_0) \leq 1$ for every $\rho \in \Omega_{\mathbf{x}_0}$ by the Initial condition of RASMs. Hence, by the Safety condition of RASMs it follows that the system under policy π reaches the unsafe set $\mathcal{X}_u$ with probability at most $1-p$. Note that here we can already conclude the claim of Theorem 4.

Finally, as $(X_t)_{t=0}^{\infty}$ is a nonnegative supermartingale, by Theorem 1 its value converges with probability 1. One can then prove that this value has to be either 0 or $\geq 1/(1-p)$ by using the fact that the expected decrease in the Expected Decrease condition of RASMs is strict. But we showed above that a state in which V is $\geq 1/(1-p)$ is reached with probability at most $1-p$. Hence, the probability that the system under policy π reaches the target set $\mathcal{X}_t$ without reaching the unsafe set $\mathcal{X}_u$ is at least p, i.e. $\mathbb{P}_{\mathbf{x}_0}[\text{ReachAvoid}(\mathcal{X}_t, \mathcal{X}_u)] \geq p$. □

4 Learner-Verifier Framework for Stochastic Systems

We now present the learner-verifier framework of [44,68] for the learning-based control and verification of learned controllers in stochastic dynamical systems. We focus on the probabilistic reach-avoid problem, assume that we are given a target set $\mathcal{X}_t$, unsafe set $\mathcal{X}_u$ and a probability parameter $p \in [0,1]$, and learn a control policy π and an RASM which certifies that $\mathbb{P}_{\mathbf{x}_0}[\text{ReachAvoid}(\mathcal{X}_t, \mathcal{X}_u)] \geq p$ for all $\mathbf{x}_0 \in \mathcal{X}_0$. The algorithm for learning RSMs and SBFs can be obtained analogously, since we showed that RASMs unify and generalize RSMs and SBFs.

The algorithm behind the learner-verifier framework consists of two modules – the learner, which learns a neural network control policy π_θ and a neural

network supermartingale certificate function V_ν, and the verifier, which then formally verifies the learned candidate function. If the verification step fails, the verifier produces counterexamples that are passed back to the learner to fine-tune its loss function. Here, θ and ν are vectors of neural network parameters. The loop is repeated until either a certificate function is successfully verified, or some specified timeout is reached. By incorporating feedback from the verifier, the learner is able to tune the policy and the certificate function towards ensuring that the resulting policy meets the desired reach-avoid specification.

Applications. As outlined above, the learner-verifier framework can be used for *learning-based control* with formal guarantees that a property of interest is satisfied by jointly learning a control policy and a supermartingale certificate function for the property. On the other hand, it can also be used to *formally verify* a previously learned control policy by fixing policy parameters and only learning a supermartingale certificate function. Finally, if one uses a different method to learn a policy that turns out to violate the desired property, one can use the learner-verifier framework to *fine-tune an unsafe policy* towards repairing it and obtaining a safe policy for which a supermartingale certificate function certifies that the property of interest is satisfied.

4.1 Algorithm Initialization

As mentioned in Section 1, the key challenge for the verifier is to check the Expected Decrease condition of supermartingale certificates. Our algorithm solves this challenge by discretizing the state space and verifying a slightly stricter condition at discretization vertices which we show to imply the Expected Decrease condition over the whole region required by Definition 3. On the other hand, learning two neural networks in parallel while simultaneously optimizing several objectives can be unstable due to inherent dependencies between two networks. Thus, proper initialization of networks is important. We allow all neural network architectures so long as all activation functions are continuous functions. Furthermore, we apply the softplus activation function to the output neuron of V_ν, in order to ensure that the value of V_ν is always nonnegative.

Discretization. A *discretization* $\tilde{\mathcal{X}}$ of $\mathcal{X}$ with mesh $\tau > 0$ is a set of states such that, for every $\mathbf{x} \in \mathcal{X}$, there exists a state $\tilde{\mathbf{x}} \in \tilde{\mathcal{X}}$ such that $||\mathbf{x} - \tilde{\mathbf{x}}||_1 < \tau$. The algorithm takes mesh τ as a parameter and computes a finite discretization $\tilde{\mathcal{X}}$ with mesh τ by simply taking a hyper-rectangular grid of the sufficiently small cell size. Since $\mathcal{X}$ is compact, this yields a finite discretization.

Network initialization. The policy network π_θ is initalized by running proximal policy optimization (PPO) [54] on the Markov decision process (MDP) defined by the stochastic dynamical system with a reward function $r_t = 1[\mathcal{X}_t](\mathbf{x}_t) - [\mathcal{X}_u](\mathbf{x}_t)$.

The discretization $\tilde{\mathcal{X}}$ is used to define three sets of states which are then used by the learner to initialize the certificate network V_ν and to which counterexamples computed by the verifier will be added later. In particular, the algorithm initializes $C_{\text{init}} = \tilde{\mathcal{X}} \cap \mathcal{X}_0$, $C_{\text{unsafe}} = \tilde{\mathcal{X}} \cap \mathcal{X}_u$ and $C_{\text{decrease}} = \tilde{\mathcal{X}} \cap (\mathcal{X} \backslash \mathcal{X}_t)$.

4.2 The Learner module

The Learner updates the parameters θ of the policy and ν of the neural network certificate function candidate V_ν with the objective of the candidate satisfying the supermartingle certificate conditions. The parameter updates happen incrementally via gradient descent of the form $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}(\theta,\nu)}{\partial \theta}$ and $\nu \leftarrow \nu - \alpha \frac{\partial \mathcal{L}(\theta,\nu)}{\partial \nu}$, where $\alpha > 0$ is the learning rate and $\mathcal{L}$ is a loss function that corresponds to a differentiable optimization objective of the supermartingle certificate conditions. Ideally, the global minimum of $\mathcal{L}$ should correspond to a policy π and a neural network V_ν that fulfills all certificate conditions. In practice, however, due to the non-convexity of the network V_ν, gradient descent is not guaranteed to converge to the global minimum. As a result, the learner is not monotone, i.e. a new iteration does not guarantee improvement over the previous iteration. The training process usually applies a fixed number of gradient descent iterations or, alternatively, continues until a certain threshold on the loss value is achieved.

Loss functions. The particular type of loss function $\mathcal{L}$ depends on the type of supermartingale certificate function that should be learned by the network, but is of the general form

$$\mathcal{L}(\theta,\nu) = \mathcal{L}_{\text{Certificate}}(\theta,\nu) + \lambda \cdot \big(\mathcal{L}_{\text{Lipschitz}}(\theta) + \mathcal{L}_{\text{Lipschitz}}(\nu)\big), \tag{1}$$

where $\mathcal{L}_{\text{Certificate}}$ is the specification-specific loss. The auxiliary loss terms $\mathcal{L}_{\text{Lipschitz}}$ regularize the training to obtain networks π_θ and V_ν that have a low upper bound of their Lipschitz constant. The purpose of this regularization is that networks with low Lipschitz upper bound are easier to check by the verifier module, i.e. requiring a coarser discretization grid. The value of $\lambda > 0$ decides the strength of the regularization that is applied. The regularization loss is based on the upper bound derived in [57] and defined as

$$\mathcal{L}_{\text{Lipschitz}}(\theta) = \max\Big\{L_{V_\theta} - \frac{\delta}{\tau \cdot (L_f \cdot (L_\pi + 1) + 1)}, 0\Big\}. \tag{2}$$

In the case of a reach-avoid specification, the RASM certificate loss is

$$\mathcal{L}_{\text{Certificate}}(\theta,\nu) = \mathcal{L}_{\text{Expected}}(\theta,\nu) + \mathcal{L}_{\text{Unsafe}}(\nu) + \mathcal{L}_{\text{Init}}(\nu), \tag{3}$$

with

$$\begin{aligned}
\mathcal{L}_{\text{Expected}}(\theta,\nu) = \frac{1}{|C_{\text{decrease}}|} \cdot \sum_{\mathbf{x}\in C_{\text{expected}}} \Big(\max\Big\{ \\
\sum_{\omega_1,\ldots,\omega_N \sim \mathcal{N}} \frac{V_\nu\big(f(\mathbf{x},\pi_\theta(\mathbf{x}),\omega_i)\big)}{N} - V_\theta(\mathbf{x}) + \tau \cdot K, 0\Big\}\Big) \\
\mathcal{L}_{\text{Init}}(\nu) = \max_{\mathbf{x}\in C_{\text{init}}} \{V_\nu(\mathbf{x}) - 1, 0\} \\
\mathcal{L}_{\text{Unsafe}}(\nu) = \max_{\mathbf{x}\in C_{\text{unsafe}}} \{\frac{1}{1-p} - V_\nu(\mathbf{x}), 0\}.
\end{aligned}$$

The sets C_{expected}, C_{init} and C_{unsafe} are the training sets for achieving the expected decrease, initial and unsafe RASM conditions. Each of the three sets is

initialized with a coarse discretization of the state space to guide the learning toward learning a correct RASM already in the first loop iteration. In the subsequent calls to the learner, these sets are extended by counterexamples computed by the verifier. In [68] it was shown that, if V_θ is a RASM and satisfies all conditions checked by the verifier below, then $\mathcal{L}_{\text{Certificate}}(\theta, \nu) \to 0$ as the number of samples N used to estimate expected values in $\mathcal{L}_{\text{Expected}}(\theta, \nu)$ increases.

4.3 The Verifier module

Verification task. The verifier now formally checks whether the learned RASM candidate V_ν satisfies the four RASM defining conditions in Definition 3. Since we applied the softplus activation function to the output neuron of V_ν, we know that the Nonnegativity condition is satisfied by default. Thus, the verifier only needs to check the Initial, Safety and Expected Decrease conditions in Definition 3.

Expected Decrease condition. To check the Expected Decrease condition, we utilize the fact that the dynamics function f is Lipschitz continuous and that the state space $\mathcal{X}$ is compact to show that it suffices to check a slightly stricter condition at the discretization points. Let L_f be a Lipschitz constant of f. Since π_θ and V_ν are continuous functions defined over the compact domain $\mathcal{X}$, we know that they are also Lipschitz continuous. Let L_π and L_V be their Lipschitz constants. We assume that L_f is provided to the algorithm, and use the method of [57] for computing neural network Lipschitz constants to compute L_π and L_V.

To verify the Expected Decrease condition, the verifier collects a subset $\tilde{\mathcal{X}}_e \subseteq \tilde{\mathcal{X}}$ of all discretization vertices whose adjacent grid cells contain a non-target state and over which V_ν attains a value that is smaller than $\frac{1}{1-p}$. To compute this set, the algorithm first collects all grid cells that intersect $\mathcal{X} \backslash \mathcal{X}_t$. For each collected cell, it then uses interval arithmetic abstract interpretation (IA-AI) [24,30] to propagate interval bounds across neural network layers towards bounding from below the minimal value that V_ν attains over the cell. Finally, it adds to $\tilde{\mathcal{X}}_e$ vertices of those cells at which the computed lower bound is less than $1/(1-p)$.

Finally, the verifier checks if the following condition is satisfied at each $\tilde{\mathbf{x}} \in \tilde{\mathcal{X}}_e$

$$\mathbb{E}_{\omega \sim d}\Big[V_\nu\Big(f(\tilde{\mathbf{x}}, \pi_\theta(\tilde{\mathbf{x}}), \omega)\Big)\Big] < V_\nu(\tilde{\mathbf{x}}) - \tau \cdot K, \tag{4}$$

where $K = L_V \cdot (L_f \cdot (L_\pi + 1) + 1)$. Note that this condition is a strengthened version of the Expected Decrease condition, where instead of strict decrease by arbitrary $\epsilon > 0$ we require strict decrease by at least $\tau \cdot K$ which depends on the discretization mesh τ and Lipschitz constants of f, π_θ and V_ν. To compute $\mathbb{E}_{\omega \sim d}[V_\nu(f(\tilde{\mathbf{x}}, \pi_\theta(\tilde{\mathbf{x}}), \omega))]$ in eq. (4), we cannot simply evaluate the expected value in state $\tilde{\mathbf{x}}$ by substituting $\tilde{\mathbf{x}}$ into some expression, as we do not know a closed-form expression for the expected value of a neural network function. Instead, the algorithm uses the method of [44] to compute upper and lower bounds on the expected value of a neural network function, which we describe in Section 5. This upper bound is then plugged it into eq. (4).

If no violations to eq. (4) are found, the verifier concludes that the Expected Decrease condition is satisfied. Otherwise, for any counterexample $\tilde{\mathbf{x}}$ to eq. (4),

the algorithm checks if $\tilde{\mathbf{x}} \in \mathcal{X} \backslash \mathcal{X}_t$ and $V_\nu(\mathbf{x}) < 1/(1-p)$ and if so adds it to the counterexample set C_{decrease}.

Initial and safety conditions. The Initial and Safety conditions are checked using IA-AI. To check the Initial condition, the verifier collects the set $\text{Cells}_{\mathcal{X}_0}$ of all grid cells that intersect the initial set $\mathcal{X}_0$, and for each cell in $\text{Cells}_{\mathcal{X}_0}$ checks if

$$\sup_{\mathbf{x} \in \text{cell}} V_\nu(\mathbf{x}) > 1. \tag{5}$$

The supremum is bounded from above via IA-AI by propagating interval bounds across neural network layers. If no violations are found, the verifier concludes that V_ν satisfies the Initial condition. Otherwise, vertices of any grid cells which are counterexamples to eq. (5) and which are contained in $\mathcal{X}_0$ are added to C_{init}. Analogously, to check the Safety condition, the verifier collects the set $\text{Cells}_{\mathcal{X}_u}$ of all grid cells that intersect the unsafe set $\mathcal{X}_u$, and for each cell checks if

$$\inf_{\mathbf{x} \in \text{cell}} V_\nu(\mathbf{x}) < \frac{1}{1-p}. \tag{6}$$

If no violations are found, the verifier concludes that V_ν satisfies the Safety condition. Otherwise, vertices of any grid cells which are counterexamples to eq. (6) and which are contained in $\mathcal{X}_u$ are added to C_{unsafe}.

Algorithm output and correctness. If all three checks are successful and no counterexample is found, the algorithm concludes that π_θ guarantees reach-avoidance with probability at least p and outputs the policy p_θ. Otherwise, it proceeds to the next learner-verifier iteration where computed counterexamples are added to sets C_{init}, C_{unsafe} and C_{decrease} to be used by the learner. The following theorem establishes correctness of the verifier module, and its proof can be found in [68].

Theorem 6 ([68]). *Suppose that the verifier verifies that the certificate V_ν satisfies eq. (4) for each $\tilde{\mathbf{x}} \in \tilde{\mathcal{X}}_e$, eq. (5) for each cell $\in \text{Cells}_{\mathcal{X}_0}$ and eq. (6) for each cell $\in \text{Cells}_{\mathcal{X}_u}$. Then the function V_ν is an RASM for the system with respect to $\mathcal{X}_t$, $\mathcal{X}_u$ and p.*

Optimizations. The verification task can be made more efficient by a discretization refinement procedure. In particular, the verifier may start with a coarse grid and decomposes each grid cell on demand into a finer discretization in case the check when some RASM condition fails. This procedure can be used recursively to refine further in the case when elements of the decomposed grid cannot be verified. In case the recursion encounters a grid element that violates Eq. 4 even for $\tau = 0$, the refinement procedure terminates unsuccessfully with the grid center point as a counterexample of the RASM condition. This optimization with a maximum recursion depth of 1 has been applied in [68].

5 Bounding Expected Values of Neural Networks

We now present the method for computing upper and lower bounds on the expected value of a neural network function over a given probability distribution.

We are not aware of any existing methods for solving this problem, so believe that this is a result of independent interest.

To define the setting of the problem at hand, let $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^n$ be a system state and suppose that we want to compute upper and lower bounds the expected value $\mathbb{E}_{\omega \sim d}[V(f(\mathbf{x}, \pi(\mathbf{x}), \omega))]$. Here d is a probability distribution over the stochastic disturbance space $\mathcal{N} \subseteq \mathbb{R}^p$ from which the stochastic disturbance is sampled independently at each time step. As noted in Section 2, we assume that d is a product of independent univariate probability distributions. Alternatively, the method is also applicable if the support of d is bounded.

The method first partitions the stochastic disturbance space $\mathcal{N} \subseteq \mathbb{R}^p$ into finitely many cells $\text{cell}(\mathcal{N}) = \{\mathcal{N}_1, \ldots, \mathcal{N}_k\}$. Let $\text{maxvol} = \max_{\mathcal{N}_i \in \text{cell}(\mathcal{N})} \mathsf{vol}(\mathcal{N}_i)$ and $\text{minvol} = \min_{\mathcal{N}_i \in \text{cell}(\mathcal{N})} \mathsf{vol}(\mathcal{N}_i)$ denote the maximal and the minimal volume of any cell in the partition with respect to the Lebesgue measure over $\mathbb{R}^p$, respectively. Also, for each $\omega \in \mathcal{N}$ let $F(\omega) = V(f(\mathbf{x}, \pi(\mathbf{x}), \omega))$. The upper and the lower boundd on the expected value are computed as follows

$$\mathbb{E}_{\omega \sim d}\Big[V\Big(f(\mathbf{x}, \pi(\mathbf{x}), \omega)\Big)\Big] \leq \sum_{\mathcal{N}_i \in \text{cell}(\mathcal{N})} \text{maxvol} \cdot \sup_{\omega \in \mathcal{N}_i} F(\omega),$$

$$\mathbb{E}_{\omega \sim d}\Big[V\Big(f(\mathbf{x}, \pi(\mathbf{x}), \omega)\Big)\Big] \geq \sum_{\mathcal{N}_i \in \text{cell}(\mathcal{N})} \text{minvol} \cdot \inf_{\omega \in \mathcal{N}_i} F(\omega).$$

Each supremum (resp. infimum) in the sum is then bounded from above (resp. from below) via interval arithmetic abstract interpretation by using the method of [30].

If the support of d is bounded, then no further adjustments are needed. However, if the support of d is unbounded, maxvol and minvol may not be finite. In this case, since we assume that d is a product of univariate distributions, the method first applies the probability integral transform [48] to each univariate probability distribution in d in order to reduce the problem to the case of a probability distribution of bounded support.

6 Discussion on Extension to General Certificates

The focus of this survey has primarily been on three concrete classes of supermartingale certificate functions in stochastic systems, namely RSMs, SBFs and RASMs, and the learner-verifier framework for their computation. For each class of supemartingale certificate functions, the learner module encodes the defining conditions of the certificate as a differentiable loss function whose minimization leads to a candidate certificate function. The verifier module then formally checks whether the defining conditions of the certificate function are satisfied. These checks are performed by discretizing the state space and using interval arithmetic abstract interpretation and the previously discussed method for computing bounds on expected values of neural network functions.

It should be noted that the design of both the learner and the verifier modules was not specifically tailored to any of the three certificate functions. Rather, both the learner and the verifier follow very general design principles that we envision

are applicable to more general classes of certificate functions. In particular, we hypothesize that as long as the state space of the system is compact and a certificate function can be defined in terms of

- exact and expected value evaluations of Lipschitz continuous functions, and
- inequalities between such evaluations imposed over state space regions,

then the learner-verifier framework in Section 4 may present a promising approach to learning and verifying the certificate function. In particular, the learner-verifier framework presents a natural candidate for automating the computation of *any supermartingale certificate function* that may be designed for other properties in the future. Furthermore, while RSMs, SBFs and RASMs exhibit a supermartingale-like behavior which is fundamental for their soundness, the learner-verifier framework does not rely or depend on their supermartingale-like behavior. Hence, we envision that the learner-verifier framework could also be used to compute other classes of *stochastic certificate functions*.

Even more generally, note that all certificate functions that we have considered so far are of the type $\mathcal{X} \to \mathbb{R}$. One could also consider extensions of the learner-verifier framework to learning certificate functions of different datatypes. For instance, the work [43] uses a learner-verifier framework to learn an inductive transition invariant of type $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$ that certifies safety in deterministic systems. On the other hand, lexicographic ranking supermartingales are a multi-dimensional generalization of RSMs of type $\mathcal{X} \to \mathbb{R}^k$ that provide a more efficient and compositional approach to proving probability 1 termination in probabilistic programs [5,22]. Studying possible extensions of the learner-verifier framework for stochastic systems to learn certificate functions of different arity of both domain and codomain is a very interesting direction of future work.

7 Related Work

Existing learning-based methods for learning and verification of certificate functions in deterministic and stochastic systems have been discussed in Section 1. In this section, we overview some other existing methods for verification and control of stochastic dynamical systems, as well as some other uses of martingale theory in stochastic system verification.

Abstraction-based methods. Another class of approaches to stochastic dynamical system control with formal safety guarantees are abstraction based methods [56,42,14,63,60,25]. These methods consider finite-time horizon systems and approximate them via a finite-state Markov decision process (MDP). The control problem is then solved for the obtained MDP and the computed policy is used to exhibit a policy for the original stochastic dynamical system. The key difference in applicability between abstraction based methods and our framework is that abstraction based methods consider *finite-time horizon* systems, whereas we consider *infinite-time horizon* systems.

Safe control via shielding. Shielding is an RL framework that ensures safety in the context of avoidance of unsafe regions by computing two control policies – the main policy that optimizes the expected reward, and the backup policy that the system falls back to whenever the safety constraint may be violated [7,36,29].

Constrained MDPs. A standard approach to safe RL is to solve constrained MDPs (CMDPs) [8,28] which impose hard constraints on expected cost for one or more auxiliary cost functions. Several efficient RL algorithms for solving CMDPs have been proposed [59,4], however their constraints are only satisfied in expectation, hence constraint satisfaction is not formally guaranteed.

RL reward specification and neurosymbolic methods. There are several works on solving model-free RL tasks under logic specifications. In particular, several works propose methods for designing reward functions that encode temporal logic specifications [6,12,32,31,45,34,13,40,39]. Formal methods have also been used for extraction of interpretable policies [62,61,35] and safe RL [10,67,11].

Deterministic systems with stochastic controllers. Another way to give rise to a stochastic dynamical system is to consider a dynamical system with deterministic dynamics function and use a stochastic controller, which helps in quantifying uncertainty in the controller's prediction. Formal verification of deterministic dynamical systems with Bayesian neural network controllers has been considered in [43]. In particular, this work also uses a learner-verifier method to learn an inductive invariant for the deterministic system which formally proves safety.

Supermartingales for probabilistic program analysis. Supermartingales have also been used for the analysis of probabilistic programs (PPs). In particular, RSMs were originally introduced in the setting of PPs to prove almost-sure termination [15] and have since been extensively used, see e.g. [19,20,5,47,22]. The work [1] proposed a learner-verifier method to learn an RSM in the PP. Supermartingales were also used for safety [23,64,21], cost [65] and recurrence and persistence [16] analysis in PPs.

8 Conclusion

This paper presents a framework for learning-based control with formal reachability, safety and reach-avoidance guarantees in stochastic dynamical systems. We present a learner-verifier framework in which a neural network control policy is learned together with a neural network certificate function that formally proves that the property of interest holds with at least some desired probability $p \in [0, 1]$. For certification, we use supermartingale certificate functions. The learner module encodes the defining certificate function conditions into a differentiable loss function which is then minimized to learn a candidate certificate function. The verifier then formally verifies the candidate by using interval arithmetic abstract interpretation and a novel method for computing bounds on expected values of neural networks.

The learner-verifier framework presented in this work opens several interesting directions for future work. The first is the design of supermartingale certificates for more general properties of stochastic systems and the use of our learner-verifier framework for their computation. The second is to study and understand the general class of certificate functions in stochastic systems that the learner-verifier can be used to compute, possibly going beyond supermartingale certificate functions. Finally, on the practical side, a venue for future work is to explore methods for reducing the computational cost of the framework and extensions that can handle more complex and higher dimensional systems.

References

1. Abate, A., Ahmed, D., Edwards, A., Giacobbe, M., Peruffo, A.: FOSSIL: a software tool for the formal synthesis of lyapunov functions and barrier certificates using neural networks. In: Bogomolov, S., Jungers, R.M. (eds.) HSCC '21: 24th ACM International Conference on Hybrid Systems: Computation and Control, Nashville, Tennessee, May 19-21, 2021. pp. 24:1–24:11. ACM (2021). https://doi.org/10.1145/3447928.3456646, https://doi.org/10.1145/3447928.3456646
2. Abate, A., Ahmed, D., Giacobbe, M., Peruffo, A.: Formal synthesis of lyapunov neural networks. IEEE Control. Syst. Lett. **5**(3), 773–778 (2021). https://doi.org/10.1109/LCSYS.2020.3005328, https://doi.org/10.1109/LCSYS.2020.3005328
3. Abate, A., Giacobbe, M., Roy, D.: Learning probabilistic termination proofs. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 3–26. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_1, https://doi.org/10.1007/978-3-030-81688-9_1
4. Achiam, J., Held, D., Tamar, A., Abbeel, P.: Constrained policy optimization. In: International Conference on Machine Learning. pp. 22–31. PMLR (2017)
5. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. Proc. ACM Program. Lang. **2**(POPL), 34:1–34:32 (2018). https://doi.org/10.1145/3158122, https://doi.org/10.1145/3158122
6. Aksaray, D., Jones, A., Kong, Z., Schwager, M., Belta, C.: Q-learning for robust satisfaction of signal temporal logic specifications. In: 55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016. pp. 6565–6570. IEEE (2016). https://doi.org/10.1109/CDC.2016.7799279, https://doi.org/10.1109/CDC.2016.7799279
7. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. pp. 2669–2678. AAAI Press (2018), https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17211
8. Altman, E.: Constrained Markov decision processes, vol. 7. CRC Press (1999)
9. Amodei, D., Olah, C., Steinhardt, J., Christiano, P.F., Schulman, J., Mané, D.: Concrete problems in AI safety. CoRR **abs/1606.06565** (2016), http://arxiv.org/abs/1606.06565
10. Anderson, G., Verma, A., Dillig, I., Chaudhuri, S.: Neurosymbolic reinforcement learning with formally verified exploration. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020), https://proceedings.neurips.cc/paper/2020/hash/448d5eda79895153938a8431919f4c9f-Abstract.html
11. Bacci, E., Giacobbe, M., Parker, D.: Verifying reinforcement learning up to infinity. In: Zhou, Z. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. pp. 2154–2160. ijcai.org (2021). https://doi.org/10.24963/ijcai.2021/297, https://doi.org/10.24963/ijcai.2021/297

12. Brafman, R.I., Giacomo, G.D., Patrizi, F.: Ltlf/ldlf non-markovian rewards. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. pp. 1771–1778. AAAI Press (2018), https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17342
13. Camacho, A., Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: LTL and beyond: Formal languages for reward function specification in reinforcement learning. In: Kraus, S. (ed.) Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019. pp. 6065–6073. ijcai.org (2019). https://doi.org/10.24963/ijcai.2019/840, https://doi.org/10.24963/ijcai.2019/840
14. Cauchi, N., Abate, A.: Stochy-automated verification and synthesis of stochastic processes. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 258–259 (2019)
15. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 511–526. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_34, https://doi.org/10.1007/978-3-642-39799-8_34
16. Chakarov, A., Voronin, Y., Sankaranarayanan, S.: Deductive proofs of almost sure persistence and recurrence properties. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 260–279. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_15, https://doi.org/10.1007/978-3-662-49674-9_15
17. Chang, Y., Gao, S.: Stabilizing neural control using self-learned almost lyapunov critics. In: IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, May 30 - June 5, 2021. pp. 1803–1809. IEEE (2021). https://doi.org/10.1109/ICRA48506.2021.9560886, https://doi.org/10.1109/ICRA48506.2021.9560886
18. Chang, Y., Roohi, N., Gao, S.: Neural lyapunov control. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 3240–3249 (2019), https://proceedings.neurips.cc/paper/2019/hash/2647c1dba23bc0e0f9cdf75339e120d2-Abstract.html
19. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through positivstellensatz's. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779, pp. 3–22. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_1, https://doi.org/10.1007/978-3-319-41528-4_1
20. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 327–

342. ACM (2016). https://doi.org/10.1145/2837614.2837639, https://doi.org/10.1145/2837614.2837639
21. Chatterjee, K., Goharshady, A.K., Meggendorfer, T., Zikelic, D.: Sound and complete certificates for quantitative termination analysis of probabilistic programs. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 55–78. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_4, https://doi.org/10.1007/978-3-031-13185-1_4
22. Chatterjee, K., Goharshady, E.K., Novotný, P., Zárevúcky, J., Zikelic, D.: On lexicographic proof rules for probabilistic termination. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13047, pp. 619–639. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_33, https://doi.org/10.1007/978-3-030-90870-6_33
23. Chatterjee, K., Novotný, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 145–160. ACM (2017). https://doi.org/10.1145/3009837.3009873, https://doi.org/10.1145/3009837.3009873
24. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973, https://doi.org/10.1145/512950.512973
25. Crespo, L.G., Sun, J.: Stochastic optimal control via bellman's principle. Autom. **39**(12), 2109–2114 (2003). https://doi.org/10.1016/S0005-1098(03)00238-3, https://doi.org/10.1016/S0005-1098(03)00238-3
26. Dawson, C., Gao, S., Fan, C.: Safe control with learned certificates: A survey of neural lyapunov, barrier, and contraction methods. CoRR **abs/2202.11762** (2022), https://arxiv.org/abs/2202.11762
27. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**, 1437–1480 (2015), http://dl.acm.org/citation.cfm?id=2886795
28. Geibel, P.: Reinforcement learning for mdps with constraints. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4212, pp. 646–653. Springer (2006). https://doi.org/10.1007/11871842_63, https://doi.org/10.1007/11871842_63
29. Giacobbe, M., Hasanbeig, M., Kroening, D., Wijk, H.: Shielding atari games with bounded prescience. In: Dignum, F., Lomuscio, A., Endriss, U., Nowé, A. (eds.) AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021. pp. 1507–1509. ACM (2021). https://doi.org/10.5555/3463952.3464141, https://www.ifaamas.org/Proceedings/aamas2021/pdfs/p1507.pdf
30. Gowal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Arandjelovic, R., Mann, T.A., Kohli, P.: On the effectiveness of interval bound propagation for training verifiably robust models. CoRR **abs/1810.12715** (2018), http://arxiv.org/abs/1810.12715
31. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-regular objectives in model-free reinforcement learning. In: Vojnar, T., Zhang,

L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11427, pp. 395–412. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_27, https://doi.org/10.1007/978-3-030-17462-0_27
32. Hasanbeig, M., Kantaros, Y., Abate, A., Kroening, D., Pappas, G.J., Lee, I.: Reinforcement learning for temporal logic control synthesis with probabilistic satisfaction guarantees. In: 58th IEEE Conference on Decision and Control, CDC 2019, Nice, France, December 11-13, 2019. pp. 5338–5343. IEEE (2019). https://doi.org/10.1109/CDC40024.2019.9028919, https://doi.org/10.1109/CDC40024.2019.9028919
33. Henrion, D., Garulli, A.: Positive polynomials in control, vol. 312. Springer Science & Business Media (2005)
34. Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: Using reward machines for high-level task specification and decomposition in reinforcement learning. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 2112–2121. PMLR (2018), http://proceedings.mlr.press/v80/icarte18a.html
35. Inala, J.P., Bastani, O., Tavares, Z., Solar-Lezama, A.: Synthesizing programmatic policies that inductively generalize. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net (2020), https://openreview.net/forum?id=S1l8oANFDH
36. Jansen, N., Könighofer, B., Junges, S., Serban, A., Bloem, R.: Safe reinforcement learning using probabilistic shields (invited paper). In: Konnov, I., Kovács, L. (eds.) 31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference). LIPIcs, vol. 171, pp. 3:1–3:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.3, https://doi.org/10.4230/LIPIcs.CONCUR.2020.3
37. Jarvis-Wloszek, Z., Feeley, R., Tan, W., Sun, K., Packard, A.: Some controls applications of sum of squares programming. In: 42nd IEEE international conference on decision and control (IEEE Cat. No. 03CH37475). vol. 5, pp. 4676–4681. IEEE (2003)
38. Jin, W., Wang, Z., Yang, Z., Mou, S.: Neural certificates for safe control policies. CoRR **abs/2006.08465** (2020), https://arxiv.org/abs/2006.08465
39. Jothimurugan, K., Alur, R., Bastani, O.: A composable specification language for reinforcement learning tasks. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 13021–13030 (2019), https://proceedings.neurips.cc/paper/2019/hash/f5aa4bd09c07d8b2f65bad6c7cd3358f-Abstract.html
40. Jothimurugan, K., Bansal, S., Bastani, O., Alur, R.: Compositional reinforcement learning from logical specifications. In: Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual. pp. 10026–10039 (2021), https://proceedings.neurips.cc/paper/2021/hash/531db99cb00833bcd414459069dc7387-Abstract.html
41. Kushner, H.J.: A partial history of the early development of continuous-time nonlinear stochastic systems theory. Autom. **50**(2), 303–334 (2014). https:

//doi.org/10.1016/j.automatica.2013.10.013, https://doi.org/10.1016/j.automatica.2013.10.013
42. Lavaei, A., Khaled, M., Soudjani, S., Zamani, M.: AMYTISS: parallelized automated controller synthesis for large-scale stochastic systems. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 461–474. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_24, https://doi.org/10.1007/978-3-030-53291-8_24
43. Lechner, M., Zikelic, D., Chatterjee, K., Henzinger, T.A.: Infinite time horizon safety of bayesian neural networks. In: Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual. pp. 10171–10185 (2021), https://proceedings.neurips.cc/paper/2021/hash/544defa9fddff50c53b71c43e0da72be-Abstract.html
44. Lechner, M., Zikelic, D., Chatterjee, K., Henzinger, T.A.: Stability verification in stochastic control systems via neural network supermartingales. In: Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022. pp. 7326–7336. AAAI Press (2022), https://ojs.aaai.org/index.php/AAAI/article/view/20695
45. Li, X., Vasile, C.I., Belta, C.: Reinforcement learning with temporal logic rewards. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017. pp. 3834–3839. IEEE (2017). https://doi.org/10.1109/IROS.2017.8206234, https://doi.org/10.1109/IROS.2017.8206234
46. Lyapunov, A.M.: The general problem of the stability of motion. International journal of control **55**(3), 531–534 (1992)
47. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2**(POPL), 33:1–33:28 (2018). https://doi.org/10.1145/3158121, https://doi.org/10.1145/3158121
48. Murphy, K.P.: Machine learning - a probabilistic perspective. Adaptive computation and machine learning series, MIT Press (2012)
49. Parrilo, P.A.: Structured semidefinite programs and semialgebraic geometry methods in robustness and optimization. California Institute of Technology (2000)
50. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Trans. Autom. Control. **52**(8), 1415–1428 (2007). https://doi.org/10.1109/TAC.2007.902736, https://doi.org/10.1109/TAC.2007.902736
51. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887, https://doi.org/10.1002/9780470316887
52. Qin, Z., Zhang, K., Chen, Y., Chen, J., Fan, C.: Learning safe multi-agent control with decentralized neural barrier certificates. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net (2021), https://openreview.net/forum?id=P6_q1BRxY8Q
53. Richards, S.M., Berkenkamp, F., Krause, A.: The lyapunov neural network: Adaptive stability certification for safe learning of dynamical systems. In: 2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings. Proceedings of Machine Learning Research, vol. 87, pp. 466–476. PMLR (2018), http://proceedings.mlr.press/v87/richards18a.html

54. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
55. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Shen, J.P., Martonosi, M. (eds.) Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. pp. 404–415. ACM (2006). https://doi.org/10.1145/1168857.1168907, https://doi.org/10.1145/1168857.1168907
56. Soudjani, S.E.Z., Gevaerts, C., Abate, A.: FAUST ^{2} : Formal abstractions of uncountable-state stochastic processes. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 272–286. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_23, https://doi.org/10.1007/978-3-662-46681-0_23
57. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. In: Bengio, Y., LeCun, Y. (eds.) 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings (2014), http://arxiv.org/abs/1312.6199
58. Taylor, A.J., Singletary, A., Yue, Y., Ames, A.D.: Learning for safety-critical control with control barrier functions. In: Bayen, A.M., Jadbabaie, A., Pappas, G.J., Parrilo, P.A., Recht, B., Tomlin, C.J., Zeilinger, M.N. (eds.) Proceedings of the 2nd Annual Conference on Learning for Dynamics and Control, L4DC 2020, Online Event, Berkeley, CA, USA, 11-12 June 2020. Proceedings of Machine Learning Research, vol. 120, pp. 708–717. PMLR (2020), http://proceedings.mlr.press/v120/taylor20a.html
59. Uchibe, E., Doya, K.: Constrained reinforcement learning from intrinsic and extrinsic rewards. In: 2007 IEEE 6th International Conference on Development and Learning. pp. 163–168. IEEE (2007)
60. Vaidya, U.: Stochastic stability analysis of discrete-time system using lyapunov measure. In: American Control Conference, ACC 2015, Chicago, IL, USA, July 1-3, 2015. pp. 4646–4651. IEEE (2015). https://doi.org/10.1109/ACC.2015.7172061, https://doi.org/10.1109/ACC.2015.7172061
61. Verma, A., Le, H.M., Yue, Y., Chaudhuri, S.: Imitation-projected programmatic reinforcement learning. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 15726–15737 (2019), https://proceedings.neurips.cc/paper/2019/hash/5a44a53b7d26bb1e54c05222f186dcfb-Abstract.html
62. Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5052–5061. PMLR (2018), http://proceedings.mlr.press/v80/verma18a.html
63. Vinod, A.P., Gleason, J.D., Oishi, M.M.K.: Sreachtools: a MATLAB stochastic reachability toolbox. In: Ozay, N., Prabhakar, P. (eds.) Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019. pp. 33–38.

ACM (2019). https://doi.org/10.1145/3302504.3311809, https://doi.org/10.1145/3302504.3311809
64. Wang, J., Sun, Y., Fu, H., Chatterjee, K., Goharshady, A.K.: Quantitative analysis of assertion violations in probabilistic programs. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 1171–1186. ACM (2021). https://doi.org/10.1145/3453483.3454102, https://doi.org/10.1145/3453483.3454102
65. Wang, P., Fu, H., Goharshady, A.K., Chatterjee, K., Qin, X., Shi, W.: Cost analysis of nondeterministic probabilistic programs. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 204–220. ACM (2019). https://doi.org/10.1145/3314221.3314581, https://doi.org/10.1145/3314221.3314581
66. Williams, D.: Probability with Martingales. Cambridge mathematical textbooks, Cambridge University Press (1991)
67. Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 686–701. ACM (2019). https://doi.org/10.1145/3314221.3314638, https://doi.org/10.1145/3314221.3314638
68. Zikelic, D., Lechner, M., Henzinger, T.A., Chatterjee, K.: Learning control policies for stochastic systems with reach-avoid guarantees. To appear at the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI-23) (2023)

Model Checking

Bounded Model Checking for Asynchronous Hyperproperties*

Tzu-Han Hsu[1], Borzoo Bonakdarpour[1](✉), Bernd Finkbeiner[2], and César Sánchez[3]

[1] Michigan State University, East Lansing, MI, USA {tzuhan,borzoo}@msu.edu
[2] CISPA Helmholtz Center, Saarbrücken, Germany finkbeiner@cispa.de
[3] IMDEA Software Institute, Madrid, Spain cesar.sanchez@imdea.org

Abstract. Many types of attacks on confidentiality stem from the nondeterministic nature of the environment that computer programs operate in. We focus on verification of confidentiality in nondeterministic environments by reasoning about *asynchronous hyperproperties*. We generalize the temporal logic A-HLTL to allow nested *trajectory* quantification, where a trajectory determines how different execution traces may advance and stutter. We propose a bounded model checking algorithm for A-HLTL based on QBF-solving for a fragment of A-HLTL and evaluate it by various case studies on concurrent programs, scheduling attacks, compiler optimization, speculative execution, and cache timing attacks. We also rigorously analyze the complexity of model checking A-HLTL.

1 Introduction

Motivation. Consider the concurrent program [10] shown in Fig. 1, where `h` is a secret variable, and `await` command is a conditional critical region. This program should satisfy the following information-flow policy: "Any sequences of observable outputs produced by an interleaving should be reproducible by some other interleaving for a different value of `h`". If this is the case, then an attacker cannot successfully guess the value of `h` from the sequence of observable outputs of the `print()` statements. For example, Fig. 2 shows how one can align two interleavings of threads `T1` and `T2` with respect to the observable sequence of outputs `'abcd'`, given two different values of secret `h`. Let us call such an alignment a *trajectory* (illustrated by the sequence of dashed lines). However, if

```
1  Thread T1() {
2    await sem>0 then
3      sem = sem - 1;
4      print('a');
5      v = v+1;
6      print('b');
7      sem = sem + 1;
8    }
9
10 Thread T2 (){
11   print('c');
12   if h then
13     await sem>0 then
14       sem = sem - 1;
15       v = v+2;
16       sem = sem + 1;
17   else
18     skip;
19   print('d');
20 }
```

Fig. 1: `T1` and `T2` leak the value of `h`.

* This research has been partially supported by the United States NSF SaTC Award 2100989, by the Madrid Regional Gov. Project BLOQUES-CM (S2018/TCS-4339), by Project PRODIGY (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the EU NextGenerationEU/PRTR, by the German Research Foundation (DFG) as part of TRR 248 (389792660), and by the European Research Council (ERC) Grant HYPER (101055412)

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 29–46, 2023.
https://doi.org/10.1007/978-3-031-30823-9_2

thread `T1` holds the semaphore and executes the critical region as an atomic operation. Then, output `'acdb'` arising due to concurrent execution of threads `T1` and `T2` reveals the value of `h` as 0, as the same output cannot be reproduced when `h=1`. Thus, the program in Fig. 1 violates the above policy.

The above policy is an example of a *hyperproperty* [5]; i.e., a set of sets of execution traces. In addition to information-flow requirements, hyperproperties can express other complex requirements such as linearizability [12] and control conditions in cyber-physical systems such as robustness and sensitivity. The temporal logic A-HLTL [1] can express hyperproperties whose sets of traces advance at different speeds, allowing stuttering steps. For example, the above policy can be expressed in A-HLTL by the following formula: $\varphi_{\mathsf{NI}} = \forall\pi.\exists\pi'.\mathsf{E}\tau.(\mathsf{h}_{\pi,\tau} \neq \mathsf{h}_{\pi',\tau}) \wedge \square(\mathsf{obs}_{\pi,\tau} = \mathsf{obs}_{\pi',\tau})$, where `obs` denotes the output observations, meaning that for all executions (i.e., interleavings) π, there should exist another execution π' and a trajectory τ, such that π and π' start from different values of `h` and τ can align all the observations along π and π' (see Fig. 2). A-HLTL can reason about *one* source of *nondeterminism* by the scheduler in the system that may lead to information leak. Indeed, the model checking algorithms proposed in [1] can discover the bug in the program in Fig. 1.

```
1  Thread T1 (){
2    while (true){
3      await sem>0 then
4        sem = sem - 1;
5        print('a');
6        v = v+1;
7        print('b');
8        sem = sem + 1;
9    }
10 }

12 Thread T2(){
13   while (true)
14     h = read(Channel1);
15 }

17 Thread T3(){
18   while (true){
19     print('c');
20     if (h == 1) then
21       await sem>0 then
22         sem = sem - 1;
23         v = v+2;
24         sem = sem + 1;
25     else
26       skip;
27     print('d');
28   }
29 }

31 Thread T4(){
32   while (true)
33     l = read(Channel2);
34 }
```

Fig. 3: `T1` and `T2` receive inputs from asynch. channels read by `T3` and `T4`.

Now, consider a more complex version of the same program shown in Fig. 3 inspired by modern programming languages such as `Go` and `P` that allow CSP-style concurrency. Here, new threads `T3` and `T4` read the values of secret input `h` and public input `l` from two asynchronous channels, rendering two different sources of nondeterminism: (1) the scheduler that results in different interleavings, and (2) data availability in the channels. This, in turn, means formula φ_{NI} no longer captures the following specification of the program, which should be:

> *"Any sequence of observable outputs produced by an interleaving should be reproducible by some other interleaving such that for all alignments of public inputs, there exists an alignment of the public outputs".*

Satisfaction of this policy (not expressible in A-HLTL as proposed in [1]) prohibits an attacker from successfully determining the sequence of values of `h`.

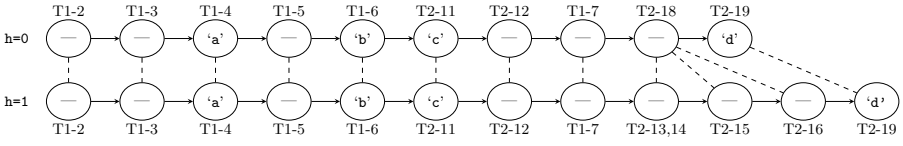


Fig. 2: Two secure interleavings for the program in Fig. 1

Contributions. In this paper, we strive for a general logic-based approach that enables model checking of a rich set of asynchronous hyperproperties. To this end, we concentrate on A-HLTL model checking for programs subject to multiple sources of nondeterminism. Our first contribution is a generalization of A-HLTL that allows nested *trajectory* quantification. For example, the above policy requires reasoning about two different trajectories that cannot be composed into one since their sources of nondeterminism are different. This observation motivates the need for enriching A-HLTL with the tools to quantify over trajectories. This generalization enables expressing policies such as follows:

$$\varphi_{\mathsf{NI}_{\mathsf{nd}}} = \forall\pi.\exists\pi'.\mathsf{A}\tau.\mathsf{E}\tau'.\big(\Diamond(\mathsf{h}_{\pi,\tau} \neq \mathsf{h}_{\pi',\tau}) \wedge \Box\,(\mathsf{I}_{\pi,\tau} = \mathsf{I}_{\pi',\tau})\big) \rightarrow \Box(\mathsf{obs}_{\pi,\tau'} = \mathsf{obs}_{\pi',\tau'}),$$

where A and E denote the universal (res., existential) trajectory quantifiers.

Our second contribution is a *bounded model checking* (BMC) algorithm for a fragment of the extended A-HLTL that allows an arbitrary number of trace quantifier alternations and up to one trajectory quantifier alternation. Following [15], we propose two bounded semantics (called *optimistic* and *pessimistic*) for A-HLTL based on the satisfaction of eventualities. We introduce a reduction to the satisfiability problem for quantified Boolean formulas (QBF) and prove that our translation provides decision procedures for A-HLTL BMC for *terminating systems*, i.e., those whose Kripke structure is acyclic. Our focus on terminating programs is due to the general undecidability of A-HLTL model checking [1]. As in the classic BMC for LTL, the power of our technique is in hunting bugs that are often in the shallow parts of reachable states.

Our third contribution is rigorous complexity analysis of A-HLTL model checking for terminating programs (see Table 1). We show that for formulas with only one trajectory quantifier the complexity is aligned with that of classic synchronous semantics of HyperLTL [4]. However, the complexity of A-HLTL model checking with multiple trajectory quantifiers is one step higher than HyperLTL model checking in the polynomial hierarchy. An interesting observation here is that the complexity of model checking a formula with two existential trajectory quantifiers is one step higher than one with only one existential quantifier although the plurality of the quantifiers does not change. Generally speaking, A-HLTL model checking for terminating programs remains in PSPACE.

Multiple Traces – Single Trajectory		
$\exists^+\mathsf{E}\ /\ \forall^+\mathsf{A}$	NL-complete (Theorem 2)	
$[\exists(\exists/\forall)^+(\mathsf{A}/\mathsf{E})]^k$	Σ_k^p-complete	Thm 3
$[\forall(\exists/\forall)^+(\mathsf{E}/\mathsf{A})]^k$	Π_k^p-complete	
Multiple Traces – Multiple Trajectories		
$[\exists(\exists/\forall)^+(\mathsf{E}^+\mathsf{E})]^k$	Σ_{k+1}^p-complete	Thm 4
$[\forall(\forall/\exists)^+(\mathsf{A}^+\mathsf{A})]^k$	Π_{k+1}^p-complete	
$[\exists(\exists/\forall)^+\mathsf{A}^+\mathsf{E}^+]^k$	Σ_{k+1}^p-complete	Thm 5
$[\forall(\forall/\exists)^+\mathsf{E}^+\mathsf{A}^+]^k$	Π_{k+1}^p-complete	
A-HLTL	PSPACE	

Table 1: A-HLTL model checking complexity for acyclic models.

Finally, we have implemented our BMC technique. We evaluate our implementation on verification of four case studies: (1) information-flow security in concurrent programs, (2) information leak in speculative executions, (3) preservation of security in compiler optimization, and (4) cache-based timing attacks. These case studies exhibit a proof of concept for the highly intricate nature of information-flow requirements and how our foundational theoretical results handle them.

Related Work. The concept of hyperproperties is due to Clarkson and Schneider [5]. HyperLTL [4] and A-HLTL are currently the only logics for which practical model checking algorithms are known [8,7,15,1]. For HyperLTL, the algorithms have been implemented in the model checkers MCHyper and bounded model checker HyperQB [14]. HyperLTL is limited to synchronous hyperproperties. The A-HLTL model checking problem is known to be undecidable in general [1]. However, decidable fragments that can express observational determinism, noninterference, and linearizability have been identified. This paper generalizes A-HLTL by allowing nested trajectory quantifiers and due to the general undecidability result focuses on terminating programs.

FOL[E] [6] can express a limited form of asynchronous hyperproperties. As shown in [6], FOL[E] is subsumed by HyperLTL with additional quantification over predicates. For $S1S[E]$ and H_μ, the model checking problem is in general undecidable; for H_μ, two fragments, the k-synchronous, k-context bounded fragments, have been identified for which model checking remains decidable [11]. Other logical extensions of HyperLTL with asynchronous capabilities are studied in [3], including their decidable fragments, but their model checking problems have not been implemented and the relative expressive power with respect to other asynchronous formalisms has not been studied.

2 Extended Asynchronous HyperLTL

Preliminaries. Given a natural number $k \in \mathbb{N}_0$, we use $[k]$ for the set $\{0, \ldots, k\}$. Let AP be a set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the *alphabet*, where we call each element of Σ a *letter*. A *trace* is an infinite sequence $\sigma = a_0 a_1 \cdots$ of letters from Σ. We denote the set of all infinite traces by Σ^ω. We use $\sigma(i)$ for a_i and σ^i for the suffix $a_i a_{i+1} \cdots$. A *pointed trace* is a pair (σ, p), where $p \in \mathbb{N}_0$ is a natural number (called the *pointer*). Pointed traces allow to traverse a trace by moving the pointer. Given a pointed trace (σ, p) and $n > 0$, we use $(\sigma, p) + n$ to denote the resulting trace $(\sigma, p + n)$. We denote the set of all pointed traces by $\mathsf{PTR} = \{(\sigma, p) \mid \sigma \in \Sigma^\omega \text{ and } p \in \mathbb{N}_0\}$.

A *Kripke structure* is a tuple $\mathcal{K} = \langle S, s_{init}, \delta, L \rangle$, where S is a set of states, $s_{init} \in S$ is the initial state, $\delta \subseteq S \times S$ is a transition relation, and $L : S \to \Sigma$ is a labeling function on the states of $\mathcal{K}$. We require that for each $s \in S$, there exists $s' \in S$, such that $(s, s') \in \delta$. □

A *path* of a Kripke structure $\mathcal{K}$ is an infinite sequence of states $s(0)s(1) \cdots \in S^\omega$, such that $s(0) = s_{init}$ and $(s(i), s(i+1)) \in \delta$, for all $i \geq 0$. A trace of $\mathcal{K}$ is a sequence $\sigma(0)\sigma(1)\sigma(2) \cdots \in \Sigma^\omega$, such that there exists a path $s(0)s(1) \cdots \in S^\omega$ with $\sigma(i) = L(s(i))$ for all $i \geq 0$. We denote by $\mathsf{Traces}(\mathcal{K}, s)$ the set of all traces of $\mathcal{K}$ with paths that start in state $s \in S$.

The directed graph $\mathcal{F} = \langle S, \delta \rangle$ is called the *Kripke frame* of the Kripke structure $\mathcal{K}$. A *loop* in $\mathcal{F}$ is a finite sequence $s_0 s_1 \cdots s_n$, such that $(s_i, s_{i+1}) \in \delta$, for all $0 \leq i < n$, and $(s_n, s_0) \in \delta$. We call a Kripke frame *acyclic*, if the only loops are self-loops on terminal states, i.e., on states that have no other outgoing transition. Acyclic Kripke structures model terminating programs.

Extended A-HLTL. The syntax of extended A-HLTL is:

$$\varphi ::= \exists \pi.\varphi \mid \forall \pi.\varphi \mid \mathsf{E}\tau.\varphi \mid \mathsf{A}\tau.\varphi \mid \psi$$
$$\psi ::= \mathit{true} \mid a_{\pi,\tau} \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \psi_1 \,\mathcal{U}\, \psi_2 \mid \psi_1 \,\mathcal{R}\, \psi_2$$

where $a \in \mathsf{AP}$, π is a trace variable from an infinite supply $\mathcal{V}$ of trace variables, τ is a *trajectory variable* from an infinite supply $\mathcal{J}$ of trajectory variables (see formula $\varphi_{\mathsf{NI_{nd}}}$ in Section 1 for an example). The intended meaning of $a_{\pi,\tau}$ is that proposition $a \in \mathsf{AP}$ holds in the current time in trace π and *trajectory* τ (explained later). Trace (respectively, trajectory) quantifiers $\exists\pi$ and $\forall\pi$ (respectively, $\mathsf{E}\tau$ and $\mathsf{A}\tau$) allow reasoning simultaneously about different traces (respectively, trajectories). The intended meaning of E is that there is a trajectory that gives an interpretation of the relative passage of time between the traces for which the temporal formula that relates the traces is satisfied. Dually, A means that all trajectories satisfy the inner formula. Given an A-HLTL formula φ, we use $\mathsf{Paths}(\varphi)$ (respectively, $\mathsf{Trajs}(\varphi)$) for the set of trace (respectively, trajectory) variables quantified in φ. A formula φ is *well-formed* if for all atoms $a_{\pi,\tau}$ in φ, π and τ are quantified in φ (i.e., $\tau \in \mathsf{Trajs}(\varphi)$ and $\pi \in \mathsf{Paths}(\varphi)$) and no trajectory/trace variable is quantified twice in φ. We use the usual syntactic sugar $\mathit{false} \triangleq \neg\mathit{true}$, and $\Diamond\varphi \triangleq \mathit{true}\,\mathcal{U}\varphi$, $\varphi_1 \rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, and $\Box\varphi \triangleq \neg\Diamond\neg\varphi$, etc. We choose to add $\mathcal{R}$ (release) and $\wedge$ to the logic to enable negation normal form (NNF). As our BMC algorithm cannot handle formulas that are not invariant under stuttering, the *next* operator is not included.

Semantics. A *trajectory* $t : t(0)t(1)t(2)\cdots$ for a formula φ is an infinite sequence of subsets of $\mathsf{Paths}(\varphi)$, i.e., each $t_i \subseteq \mathsf{Paths}(\varphi)$, for all $i \geq 0$. Essentially, in each step of the trajectory one or more of the traces make progress or all may stutter. A trajectory is *fair* for a trace variable $\pi \in \mathsf{Paths}(\varphi)$ if there are infinitely many positions j such that $\pi \in t(j)$. A trajectory is fair if it is fair for all trace variables in $\mathsf{Paths}(\varphi)$. Given a trajectory t, by t^i, we mean the suffix $t(i)t(i+1)\cdots$. Furthermore, for a set of trace variables $\mathcal{V}$, we use $\mathsf{TRJ}_{\mathcal{V}}$ for the set of all fair trajectories for indices from $\mathcal{V}$. We also use a *trajectory assignment* $\Gamma : \mathsf{Trajs}(\varphi) \rightharpoonup \mathsf{TRJ}_{Dom(\Gamma)}$, where $Dom(\Gamma)$ is the subset of $\mathsf{Trajs}(\varphi)$ for which Γ is defined. Given a trajectory assignment Γ, a trajectory variable τ, and a trajectory t, we denote by $\Gamma[\tau \mapsto t]$ the assignment that coincides with Γ for every trajectory variable except for τ, which is mapped to t.

For the semantics of extended A-HLTL, we need asynchronous trace assignments $\Pi : \mathsf{Paths}(\varphi) \times \mathsf{Trajs}(\varphi) \rightarrow T \times \mathbb{N}$ which map each pair (π, τ) formed by a path variable and trajectory variable into a pointed trace. Given (Π, Γ) where Π is an asynchronous trace assignment and Γ a trajectory assignment, we use $(\Pi, \Gamma) + 1$ for the successor of (Π, Γ) defined as (Π', Γ') where $\Gamma'(\tau) = \Gamma(\tau)^1$, and $\Pi'(\pi, \tau) = \Pi(\pi, \tau) + 1$ if $\pi \in \Gamma(\tau)(0)$ and $\Pi'(\pi, \tau) = \Pi(\pi, \tau)$ otherwise. Note that Π can assign the same π to different pointed traces depending on the trajectory. We use $(\Pi, \Gamma) + k$ as the k-th successor of (Π, Γ). Given an asynchronous trace assignment Π, a trace variable π, a trajectory variable τ a trace σ, and a pointer p, we denote by $\Pi[(\pi, \tau) \mapsto (\sigma, p)]$ the assignment that coincides

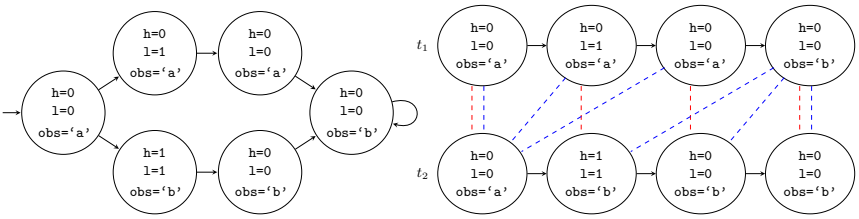


Fig. 4: Kripke structure $\mathcal{K}$ and traces t_1 and t_2 of $\mathcal{K}$, $\mathcal{K} \models \varphi_{\mathsf{NI}_{\mathsf{nd}}}$ but $\mathcal{K} \not\models \varphi_{\mathsf{NI}}$.

with Π for every pair except for (π, τ), which is mapped to (σ, p). The satisfaction of an A-HLTL formula φ over a trace assignment Π, a trajectory assignment Γ, and a set of traces T is defined as follows (we omit $\neg$, $\wedge$ and $\vee$ which are standard):

$$
\begin{array}{lll}
(\Pi, \Gamma) \models_T \exists \pi. \varphi & \text{iff} & \text{for some } \sigma \in T: \\
 & & (\Pi[(\pi,\tau) \mapsto (\sigma, 0)], \Gamma) \models_T \varphi \text{ for all } \tau \\
(\Pi, \Gamma) \models_T \forall \pi. \varphi & \text{iff} & \text{for all } \sigma \in T: \\
 & & (\Pi[(\pi,\tau) \mapsto (\sigma, 0)], \Gamma) \models_T \varphi \text{ for all } \tau \\
(\Pi, \Gamma) \models_T \mathsf{E}\tau. \psi & \text{iff} & \text{for some } t \in \mathsf{TRJ}_{Dom(\Pi)} : (\Pi, \Gamma[\tau \mapsto t]) \models \psi \\
(\Pi, \Gamma) \models_T \mathsf{A}\tau. \psi & \text{iff} & \text{for all } t \in \mathsf{TRJ}_{Dom(\Pi)} (\Pi, \Gamma[\tau \mapsto t]) \models \psi \\
(\Pi, \Gamma) \models a_{\pi,\tau} & \text{iff} & a \in \sigma(n) \text{ where } (\sigma, n) = \Pi(\pi, \tau) \\
(\Pi, \Gamma) \models \psi_1 \, \mathcal{U} \, \psi_2 & \text{iff} & \text{for some } i \geq 0 : (\Pi, \Gamma) + i \models \psi_2 \text{ and} \\
 & & \text{for all } j < i : (\Pi, \Gamma) + j \models \psi_1 \\
(\Pi, \Gamma) \models \psi_1 \, \mathcal{R} \, \psi_2 & \text{iff} & \text{for all } i \geq 0 : (\Pi, \Gamma) + i \models \psi_2, \text{ or} \\
 & & \text{for some } i \geq 0 : (\Pi, \Gamma) + i \models \psi_1 \text{ and} \\
 & & \quad \text{for all } j \leq i : (\Pi, \Gamma) + j \models \psi_2
\end{array}
$$

We say that a set T of traces satisfies a sentence φ, denoted by $T \models \varphi$, if $(\Pi_\emptyset, \Gamma_\emptyset) \models_T \varphi$. We say that a Kripke structure $\mathcal{K}$ satisfies an A-HLTL formula φ (and write $\mathcal{K} \models \varphi$) if and only if we have $\mathsf{Traces}(\mathcal{K}, S_{init}) \models \varphi$. An example is illustrated in Fig. 4.

3 Bounded Model Checking for A-HLTL

We first introduce the bounded semantics of A-HLTL (for at most one trajectory quantifier alternation but arbitrary trace quantifiers) which will be used to generate queries to a QBF solver to aid solving the BMC problem. The main result of this section is Theorem 1 which provides decision procedures for model checking A-HLTL for terminating systems.

3.1 Bounded Semantics of A-HLTL

The bounded semantics corresponds to the exploration of the system up to a certain bound. In our case, we will consider two bounds k and m (with $k \leq m$).

The bound k corresponds to the *maximum depth* of the unrolling of the Kripke structures and m is the *bound on trajectories length*. We start by introducing some auxiliary functions and predicates, for a given trace assignment and (Π, Γ). First, the family of functions $pos_{\pi,\tau} : \{0 \dots m\} \to \mathbb{N}$. The meaning of $pos_{\pi,\tau}(i)$ provides how many times π has been selected in $\{\tau(0), \dots, \tau(i)\}$. We assume that Kripke structures are equipped with an atomic proposition *halt* (one per trace variable π) which encodes whether the state is a halting state. Given (Π, Γ) we consider the predicate *halted* that holds whenever for all π and τ, $halt \in \sigma(j)$ for $(\sigma, j) = \Pi(\pi, \tau)$. In this case we write $(\Pi, \Gamma, n) \models halted$.

We define two bounded semantics which only differ in how they inspect beyond the (k, m) bounds: $\models^{hpes}_{k,m}$, called the *halting pessimistic semantics* and $\models^{hopt}_{k,m}$, called the *halting optimistic semantics*. We start by defining the bounded semantics of the quantifiers.

$$(\Pi, \Gamma, 0) \models_{k,m} \exists\pi.\ \psi \quad \text{iff} \quad \text{there is a } \sigma \in T_\pi, \text{ such that for all } \tau \ (\Pi[(\pi,\tau) \to (\sigma, 0)], \Gamma, 0) \models_{k,m} \psi \tag{1}$$

$$(\Pi, \Gamma, 0) \models_{k,m} \forall\pi.\ \psi \quad \text{iff} \quad \text{for all } \sigma \in T_\pi, \text{ for all } \tau : (\Pi[(\pi,\tau) \to (\sigma, 0)], \Gamma, 0) \models_{k,m} \psi \tag{2}$$

$$(\Pi, \Gamma, 0) \models_{k,m} \mathsf{E}\tau.\ \psi \quad \text{iff} \quad \text{there is a } t \in \mathsf{TRJ}_{Dom(\Pi)} : (\Pi, \Gamma[\tau \to t], 0) \models_{k,m} \psi \tag{3}$$

$$(\Pi, \Gamma, 0) \models_{k,m} \mathsf{A}\tau.\ \psi \quad \text{iff} \quad \text{for all } t \in \mathsf{TRJ}_{Dom(\Pi)} : (\Pi, \Gamma[\tau \to t], 0) \models_{k,m} \psi \tag{4}$$

For the Boolean operators, for $i \leq m$:

$$(\Pi, \Gamma, i) \models_{k,m} \mathtt{true} \tag{5}$$

$$(\Pi, \Gamma, i) \models_{k,m} a_{\pi,\tau} \quad \text{iff} \quad a \in (\sigma, j) \text{ where } (\sigma, j) = \Pi(\pi, \tau)(i) \text{ and } j \leq k \tag{6}$$

$$(\Pi, \Gamma, i) \models_{k,m} \neg a_{\pi,\tau} \quad \text{iff} \quad a \notin (\sigma, j) \text{ where } (\sigma, j) = \Pi(\pi, \tau)(i) \text{ and } j \leq k \tag{7}$$

$$(\Pi, \Gamma, i) \models_{k,m} \psi_1 \vee \psi_2 \quad \text{iff} \quad (\Pi, \Gamma, i) \models_{k,m} \psi_1 \text{ or } (\Pi, \Gamma, i) \models_{k,m} \psi_2 \tag{8}$$

$$(\Pi, \Gamma, i) \models_{k,m} \psi_1 \wedge \psi_2 \quad \text{iff} \quad (\Pi, \Gamma, i) \models_{k,m} \psi_1 \text{ and } (\Pi, \Gamma, i) \models_{k,m} \psi_2 \tag{9}$$

For the temporal operators, we must consider the cases of falling of the paths (beyond k) and falling of the traces (beyond m). We define the predicate *off* which holds for (Π, Γ, i) if for some (π, τ), $pos_{\pi,\tau}(i) > k$ and $halt_\pi \notin \sigma(k)$ where σ is the trace assigned to π. Note that *halted* implies that *off* does not hold because all paths (including those at k or beyond) satisfy *halt*.

We define two semantics that differ on how to interpret when the end of the unfolding of the traces and trajectories is reached. The *halting pessimistic* semantics, denoted by $\models^{hpes}_{k,m}$ take (1)-(9) above and add (10)-(13) together with $(\Pi, \Gamma, i) \not\models_{k,m} off$. Rules (10) and (11) define the semantics of the temporal operators for the case $i < m$, that is, before the end of the unrolling of the trajectories (recall that we do not consider $\bigcirc$):

$$(\Pi, \Gamma, i) \models_{k,m} \psi_1 \,\mathcal{U}\, \psi_2 \quad \text{iff} \quad (\Pi, \Gamma, i) \models_{k,m} \psi_2, \text{ or } (\Pi, \Gamma, i) \models_{k,m} \psi_1, \text{ and } (\Pi, \Gamma, i) + 1 \models_{k,m} \psi_1 \,\mathcal{U}\, \psi_2 \tag{10}$$

$$(\Pi, \Gamma, i) \models_{k,m} \psi_1 \mathcal{R} \psi_2 \quad \text{iff } (\Pi, \Gamma, i) \models_{k,m} \psi_2, \text{ and } (\Pi, \Gamma, i) \models_{k,m} \psi_1, \text{ or } (\Pi, \Gamma, i) + 1 \models_{k,m} \psi_1 \mathcal{R} \psi_2 \tag{11}$$

For the case of $i = m$, that is, at the bound of the trajectory:

$$(\Pi, \Gamma, m) \models^{hpes}_{k,m} \psi_1 \mathcal{U} \psi_2 \quad \text{iff } (\Pi, \Gamma, m) \models_{k,m} \psi_2 \tag{12}$$

$$(\Pi, \Gamma, m) \models^{hpes}_{k,m} \psi_1 \mathcal{R} \psi_2 \quad \text{iff } (\Pi, \Gamma, m) \models_{k,m} \psi_1 \wedge \psi_2, \text{ or } (\Pi, \Gamma, m) \models_{k,m} \mathit{halted} \wedge \psi_2 \tag{13}$$

The *halting optimistic* semantics, denoted by $\models^{hopt}_{k,m}$ take rules (1)-(11) and (12′)-(13′), but now if $(\Pi, \Gamma, i) \models^{hopt}_{k,m}$ *off* then $(\Pi, \Gamma, i) \models^{hopt}_{k,m} \varphi$ holds for every formula. Again, rules (10) and (11) define the semantics of the temporal operators for the case $i < m$. Then, for $i = m$:

$$(\Pi, \Gamma, m) \models^{hopt}_{k,m} \psi_1 \mathcal{U} \psi_2 \quad \text{iff } (\Pi, \Gamma, m) \models_{k,m} \psi_2, \text{ or } (\Pi, \Gamma, m) \not\models_{k,m} \mathit{halted} \wedge \psi_1 \tag{12′}$$

$$(\Pi, \Gamma, m) \models^{hopt}_{k,m} \psi_1 \mathcal{R} \psi_2 \quad \text{iff } (\Pi, \Gamma, m) \models_{k,m} \psi_2 \tag{13′}$$

Similar to [15] for the case of HyperLTL, the pessimistic semantics capture the case where we assume that pending eventualities will not become true in the future after the end of the trace (this is also assumed in LTL BMC). Dually, the optimistic semantics assume that all pending eventualities at the end of the trace will be fulfilled. Therefore, the following hold (proofs in [13]).

Lemma 1. *Let $k \leq k'$ and $m \leq m'$.*

1. *If $(\Pi, \Gamma, 0) \models^{hpes}_{k,m} \varphi$, then $(\Pi, \Gamma, 0) \models^{hpes}_{k',m'} \varphi$.*
2. *If $(\Pi, \Gamma, 0) \not\models^{hopt}_{k,m} \varphi$, then $(\Pi, \Gamma, 0) \not\models^{hopt}_{k',m'} \varphi$.*

Lemma 2. *The following hold for every k and m,*

1. *If $(\Pi, \Gamma, 0) \models^{hpes}_{k,m} \varphi$, then $(\Pi, \Gamma, 0) \models \varphi$.*
2. *If $(\Pi, \Gamma, 0) \not\models^{hopt}_{k,m} \varphi$, then $(\Pi, \Gamma, 0) \not\models \varphi$.*

3.2 From Bounded Semantics to QBF Solving

Let $\mathcal{K}$ be a Kripke structure and φ be an A-HLTL formula. Based on the bounded semantics introduced previously, our main approach is to generate a QBF query (with bounds k, m), which can use either the pessimistic or the optimistic semantics. We use $[\![\mathcal{K}, \varphi]\!]^{hpes}_{k,m}$ if the pessimistic semantics are used and $[\![\mathcal{K}, \varphi]\!]^{hopt}_{k,m}$ if the optimistic semantics are used. Our translations will satisfy that

(1) if $[\![\mathcal{K}, \varphi]\!]^{hpes}_{k,m}$ is SAT, then $\mathcal{K} \models \varphi$;
(2) if $[\![\mathcal{K}, \varphi]\!]^{hopt}_{k,m}$ is UNSAT, then $\mathcal{K} \not\models \varphi$;
(3) if the Kripke structure is unrolled to the diameter and the trajectories up to a maximum length (see below), then $[\![\mathcal{K}, \varphi]\!]^{hpes}_{k,m}$ is SAT if and only if $[\![\mathcal{K}, \varphi]\!]^{hopt}_{k,m}$ is SAT.

The first step to define $[\![\mathcal{K}, \varphi]\!]^{hopt}_{k,m}$ and $[\![\mathcal{K}, \varphi]\!]^{hpes}_{k,m}$ is to encode the unrolling of the models up-to a given depth k. For a path variable π corresponding to Kripke structure $\mathcal{K}$, we introduce $(k+1)$ copies $(x^0, \ldots, x^k)$ of the Boolean variables that define the state of $\mathcal{K}$ and use the initial condition I and the transition relation R of $\mathcal{K}$ to relate these variables. For example, for $k = 3$, we unroll the transition relation up-to 3 as follows:

$$[\![\mathcal{K}]\!]_3 = I(x^0) \wedge R(x^0, x^1) \wedge R(x^1, x^2) \wedge R(x^2, x^3).$$

Encoding positions. For each trajectory variable τ and given the bound m on the unrolling of trajectories, we add $\mathsf{Paths}(\varphi) \times (m+1)$ variables $t^0_\pi \ldots t^m_\pi$, for each π. The intended meaning of t^j_π is that t^j_π is true whenever $\pi \in t(j)$, that is, when t dictates that π moves at time instant j. In order to encode sanity conditions on trajectories, that are crucial for completeness, it is necessary to introduce a family of variables that captures how much π has moved according to τ after j steps. There is a variable *pos* for each trace variable π, each trajectory τ and each $i \leq k$ and $j \leq m$. We represent this variable by $pos^{i,j}_{\pi,\tau}$. The intention is that *pos* is true whenever after j steps trajectory τ has dictated that trace π progresses precisely i times. Fig. 5 shows encodings t^j_π and $pos^{i,j}_{\pi,\tau}$ for the traces w.r.t. the blue trajectory, τ' in Fig. 4. We will use the auxiliary definitions (for $i \in \{0 \ldots k\}$ and $j \in \{0 \ldots m\}$) to force that the path π has moved to position i after j moves from the trajectory and that π has not fallen off the trace (and does not change position when the paths fall off the trace):

Encodings of t^j_π and $t^j_{\pi'}$:

$[t^0_\pi, t^1_\pi, t^2_\pi, t^3_\pi, t^4_\pi, t^5_\pi, t^6_\pi]$
$[t^0_{\pi'}, t^1_{\pi'}, t^2_{\pi'}, t^3_{\pi'}, t^4_{\pi'}, t^5_{\pi'}, t^6_{\pi'}]$

Encodings of $pos^{i,j}_{\pi,\tau}$ and $pos^{i,j}_{\pi',\tau'}$

$[pos^{0,0}_{\pi,\tau'}, pos^{0,1}_{\pi,\tau'}, pos^{0,2}_{\pi,\tau'}, pos^{0,3}_{\pi,\tau'},$
$pos^{1,1}_{\pi,\tau'}, pos^{1,2}_{\pi,\tau'}, pos^{1,3}_{\pi,\tau'}, pos^{1,4}_{\pi,\tau'},$
$pos^{2,2}_{\pi,\tau'}, pos^{2,3}_{\pi,\tau'}, pos^{2,4}_{\pi,\tau'}, pos^{2,5}_{\pi,\tau'},$
$pos^{3,3}_{\pi,\tau'}, pos^{3,4}_{\pi,\tau'}, pos^{3,5}_{\pi,\tau'}, pos^{3,6}_{\pi,\tau'}]$

$[pos^{0,0}_{\pi',\tau'}, pos^{0,1}_{\pi',\tau'}, pos^{0,2}_{\pi',\tau'}, pos^{0,3}_{\pi',\tau'},$
$pos^{1,1}_{\pi',\tau'}, pos^{1,2}_{\pi',\tau'}, pos^{1,3}_{\pi',\tau'}, pos^{1,4}_{\pi',\tau'},$
$pos^{2,2}_{\pi',\tau'}, pos^{2,3}_{\pi',\tau'}, pos^{2,4}_{\pi',\tau'}, pos^{2,5}_{\pi',\tau'},$
$pos^{3,3}_{\pi',\tau'}, pos^{3,4}_{\pi',\tau'}, pos^{3,5}_{\pi',\tau'}, pos^{3,6}_{\pi',\tau'}]$

Fig. 5: Variables for encodings of the blue trajectory in Fig. 4, where green variables are *true* and gray variables are *false*.

$$setpos^{i,j}_{\pi,\tau} \stackrel{\text{def}}{=} pos^{i,j}_{\pi,\tau} \wedge \bigwedge_{n \in \{0..k\} \setminus \{i\}} \neg pos^{n,j}_{\pi,\tau} \wedge \neg off^j_{\pi,\tau}$$

$$nopos^j_{\pi,\tau} \stackrel{\text{def}}{=} off^j_{\pi,\tau} \wedge \bigwedge_{n \in \{0..k\}} \neg pos^{n,j}_{\pi,\tau}$$

Initially, $I_{pos} \stackrel{\text{def}}{=} \bigwedge_{\pi,\tau} setpos^{0,0}_{\pi,\tau}$, where $\pi \in \mathsf{Traces}(\varphi)$ and $\tau \in \mathsf{TRJ}_{Dom(\Pi)}$. I_{pos} captures that all paths are initially at position 0. Then, for every step $j \in \{0 \ldots m\}$, the following formulas relate the values of *pos* and *off*, depending on whether trajectory τ moves path π or not (and on whether π has reached the end k or halted):

$$step^j_{\pi,\tau} \stackrel{\text{def}}{=} \bigwedge_{i \in \{0..k-1\}} \left(pos^{i,j}_{\pi,\tau} \wedge t^j_\pi \rightarrow setpos^{i+1,j+1}_{\pi,\tau} \right)$$

$$stutters^{j}_{\pi,\tau} \stackrel{\text{def}}{=} \bigwedge_{i\in\{0..k\}} \left(pos^{i,j}_{\pi,\tau} \wedge \neg t^{j}_{\pi} \rightarrow setpos^{i,j+1}_{\pi,\tau}\right)$$

$$ends^{j}_{\pi,\tau} \stackrel{\text{def}}{=} (pos^{k,j}_{\pi,\tau} \wedge t^{j}_{\pi}) \rightarrow \left((\neg halt^{k}_{\pi} \rightarrow nopos^{j+1}_{\pi,\tau}) \wedge (halt^{k}_{\pi} \rightarrow setpos^{k,j+1}_{\pi,\tau})\right)$$

Then the following formula captures the correct assignment to the the *pos* variables, including the initial assignment:

$$\varphi_{pos} \stackrel{\text{def}}{=} I_{pos} \wedge \bigwedge_{j\in\{0..m\}} \bigwedge_{\pi,\tau} (step^{j}_{\pi,\tau} \wedge stutters^{j}_{\pi,\tau} \wedge ends^{j}_{\pi,\tau})$$

For example, Fig. 5 (w.r.t. Fig. 4) encodes the blue trajectory (τ') of π (i.e., t_1) and π' (i.e., t_2) as follows. First, for $j \in [0,3)$, it advances t_1 and stutters t_2. Therefore, $t^0_{\pi}, t^1_{\pi}, t^2_{\pi}$ are *true* and $t^0_{\pi'}, t^1_{\pi'}, t^2_{\pi'}$ are *false*. Notice that for *pos* encodings, the π position advances according to $step^{j}_{\pi,\tau'}$ (i.e., $pos^{0,0}_{\pi,\tau'}, pos^{1,1}_{\pi,\tau'}, pos^{2,2}_{\pi,\tau'}, pos^{3,3}_{\pi,\tau'}$); while π' stutters according to $stutters^{j}_{\pi',\tau'}$ (i.e., $pos^{0,0}_{\pi',\tau'}, pos^{0,1}_{\pi',\tau'}, pos^{0,2}_{\pi',\tau'}, pos^{0,3}_{\pi',\tau'}$). Then, for $j \in [3,5]$, it alternatively advances t_2 which makes $t^3_{\pi}, t^4_{\pi}, t^5_{\pi}$ *false* and $t^3_{\pi'}, t^4_{\pi'}, t^5_{\pi'}$ *true*. Similarly, the movements becomes $pos^{3,4}_{\pi,\tau'}, pos^{3,5}_{\pi,\tau'}, pos^{3,6}_{\pi,\tau'}$ and $pos^{1,4}_{\pi',\tau'}, pos^{2,5}_{\pi',\tau'}, pos^{3,6}_{\pi',\tau'}$. At the halting point (i.e., $j = k$), both trajectory trigger $ends^{j}$ and do not advance anymore.

Encoding the inner LTL formula. We will use the following auxiliary predicates:

$$halted^{j} \stackrel{\text{def}}{=} \bigwedge_{\tau} halted^{j}_{\tau} \qquad\qquad off^{j} \stackrel{\text{def}}{=} \bigvee_{\pi,\tau} off^{j}_{\pi,\tau}$$

We now give the encoding for the inner temporal formulas for a fix unrolling k and m as follows. For the atomic and Boolean formulas, the following translations are performed for $j \in \{0 \dots m\}$.

$$[\![p_{\pi,\tau}]\!]^{j}_{k,m} := \bigvee_{i\in\{0..k\}} (pos^{i,j}_{\pi,\tau} \wedge p^{i}_{\pi}) \tag{14}$$
$$[\![\neg p_{\pi,\tau}]\!]^{j}_{k,m} := \bigvee_{i\in\{0..k\}} (pos^{i,j}_{\pi,\tau} \wedge \neg p^{i}_{\pi}) \tag{15}$$
$$[\![\psi_1 \vee \psi_2]\!]^{j}_{k,m} := [\![\psi_1]\!]^{j}_{k,m} \vee [\![\psi_2]\!]^{j}_{k,m} \tag{16}$$
$$[\![\psi_1 \wedge \psi_2]\!]^{j}_{k,m} := [\![\psi_1]\!]^{j}_{k,m} \wedge [\![\psi_2]\!]^{j}_{k,m} \tag{17}$$

The halting pessimistic semantics translation uses $[\![\cdot]\!]_{hpes}$, taking (14)-(17) and (18)-(21) below. For the temporal operators and $j < m$:

$$[\![\psi_1 \,\mathcal{U}\, \psi_2]\!]^{j}_{k,m} := \neg off^{j} \wedge \left([\![\psi_2]\!]^{j}_{k,m} \vee ([\![\psi_1]\!]^{j}_{k,m} \wedge [\![\psi_1 \,\mathcal{U}\, \psi_2]\!]^{j+1}_{k,m})\right) \tag{18}$$
$$[\![\psi_1 \,\mathcal{R}\, \psi_2]\!]^{j}_{k,m} := \neg off^{j} \wedge \left([\![\psi_2]\!]^{j}_{k,m} \wedge ([\![\psi_1]\!]^{j}_{k,m} \vee [\![\psi_1 \,\mathcal{R}\, \psi_2]\!]^{j+1}_{k,m})\right) \tag{19}$$

For $j = m$:

$$[\![\psi_1 \,\mathcal{U}\, \psi_2]\!]^{m}_{k,m} := [\![\psi_2]\!]^{m}_{k,m} \tag{20}$$
$$[\![\psi_1 \,\mathcal{R}\, \psi_2]\!]^{m}_{k,m} := \left([\![\psi_1]\!]^{m}_{k,m} \wedge [\![\psi_2]\!]^{m}_{k,m}\right) \vee \left(halted^{m} \wedge [\![\psi_2]\!]^{m}_{k,m}\right) \tag{21}$$

The halting optimistic semantics translation uses $[\![\cdot]\!]_{hopt}$, taking (14)-(17) and (18′)-(21′) as follows, For the temporal operators and $j < m$:

$$[\![\psi_1 \,\mathcal{U}\, \psi_2]\!]^j_{k,m} := \mathit{off}^j \vee \left([\![\psi_2]\!]^j_{k,m} \vee ([\![\psi_1]\!]^j_{k,m} \wedge [\![\psi_1 \,\mathcal{U}\, \psi_2]\!]^{j+1}_{k,m})\right) \tag{18′}$$
$$[\![\psi_1 \,\mathcal{R}\, \psi_2]\!]^j_{k,m} := \mathit{off}^j \vee \left([\![\psi_2]\!]^j_{k,m} \wedge ([\![\psi_1]\!]^j_{k,m} \vee [\![\psi_1 \,\mathcal{R}\, \psi_2]\!]^{j+1}_{k,m})\right) \tag{19′}$$

For $j = m$:

$$[\![\psi_1 \,\mathcal{U}\, \psi_2]\!]^m_{k,m} := [\![\psi_2]\!]^m_{k,m} \vee \left(\mathit{halted}^m \wedge [\![\psi_1]\!]^m_{k,m}\right) \tag{20′}$$
$$[\![\psi_1 \,\mathcal{R}\, \psi_2]\!]^m_{k,m} := [\![\psi_2]\!]^m_{k,m} \tag{21′}$$

Combining the encodings. Let φ be a A-HLTL formula of the form $\varphi = \mathbb{Q}_A \pi_A. \ldots . \mathbb{Q}_Z \pi_Z . \mathbb{Q}_a \tau_a. \ldots . \mathbb{Q}_z \tau_z . \psi$. Combining all the components, the encoding of the A-HLTL BMC problem into QBF, for bounds k and m is:

$$[\![\mathcal{K}, \varphi]\!]_{k,m} = \mathbb{Q}_A \overline{x_A}. \cdots . \mathbb{Q}_Z \overline{x_Z} . \mathbb{Q}_a \overline{t_a}. \cdots . \mathbb{Q}_z \overline{t_z}.\ \exists \overline{\mathit{pos}}.\ \exists \overline{\mathit{off}}.$$
$$\left([\![\mathcal{K}]\!]_k \circ_A \cdots [\![\mathcal{K}]\!]_k \circ_Z (\varphi_{pos} \wedge \mathit{enc}(\psi))\right)$$

where $\circ_A = \rightarrow$ if $\mathbb{Q}_A = \forall$ (and $\circ_A = \wedge$ if $\mathbb{Q}_A = \exists$), and $\circ_B$, ... are defined similarly. The sets $\overline{\mathit{pos}}$ is the set of variables $\mathit{pos}^{i,j}_{\pi,\tau}$ that encode the positions and $\overline{\mathit{off}}$ is the set of variables $\mathit{off}^j_{\pi,\tau}$ that encode when a trace progress has fallen off its unrolling limit. We next define the encoding $\mathit{enc}(\psi)$ of the temporal formula ψ.

Encoding formulas with up to 1 trajectory quantifier alternations We consider the encoding into QBF of formulas with zero and one quantifier alternation separately. In the following, we say that at position j a collection of trajectories U "moves" whenever either all trajectories have moved all their paths to the halting state, or at least one of the trajectories in U makes one of the non-halted path move at position j. Formally,

$$\mathit{moves}^j_U \stackrel{\text{def}}{=} \mathit{halted}^j_U \vee \bigvee_{\tau \in U, \pi} (t^j_\pi \wedge \neg \mathit{halt}^j_{\pi,\tau})$$

– $\mathsf{E}^+ U.\psi$: In this case, the formula generated for $\mathit{enc}(\psi)$ is

$$\left(\bigwedge_{j \in \{0 \ldots m\}} \mathit{moves}^j_U\right) \wedge [\![\psi]\!]^0_{k,m}$$

This is correct since the positions at which all trajectories stutter all paths can be removed (obtaining a satisfying path), we can restrict the search to non-stuttering trajectory steps.

– $\mathsf{A}^+ U.\psi$: In this case, the formula generated for $\mathit{enc}(\psi)$ is

$$\left(\bigwedge_{j \in \{0 \ldots m\}} \mathit{moves}^j_U\right) \rightarrow [\![\psi]\!]^0_{k,m}$$

The reasoning is similar as the previous case.

– $\mathsf{A}^+U_A\mathsf{E}^+U_E.\psi$: In this case, the formula generated for $enc(\psi)$ is

$$\Big(\bigwedge_{j\in\{0\ldots m\}} moves^j_{U_A}\Big) \rightarrow \Big(\bigwedge_{j\in\{0\ldots m\}} (halted^j_{U_A} \rightarrow moves^j_{U_E}) \wedge [\![\psi]\!]^0_{k,m}\Big)$$

Universally quantified trajectories must explore all trajectories, which must be responded by the existential trajectories. Assume there is a strategy for U_E for the case that universal trajectories U_A never stutter at any position. This can be extended into a strategy for the case where U_A can possible stutter, by adding a stuttering step to the U_E trajectories at the same position. This guarantees the same evaluation. Therefore, we restrict our search for the outer U_A to non-stuttering trajectories. Finally, U_E is obliged to move after U_A has halted all paths to prevent global stuttering.

– $\mathsf{E}^+U_E\mathsf{A}^+U_A.\psi$: In this case, the formula generated for $enc(\psi)$ is similar,

$$\Big(\bigwedge_{j\in\{0\ldots m\}} moves^j_{U_E}\Big) \wedge \Big(\bigwedge_{j\in\{0\ldots m\}} (halted^j_{U_E} \rightarrow moves^j_{U_A}) \rightarrow [\![\psi]\!]^0_{k,m}\Big)$$

The rationale for this encoding is the following. It is not necessary to explore a non-moving step j for the existentially quantified trajectories U_E because if this stuttering step is successful it must work for all possible moves of the U_A trajectories at the same time step j. This includes the case that all trajectories in U_A make all paths stutter (which, if we remove j one still has all the legal trajectories for U_A). Since the logic does not contain the next operator, the evaluation for the given U_E and one of the trajectories for U_A that stutter at j will be the same as for $j+1$ for all logical formulas. Therefore, the trajectory that is obtained from removing step j from U_E is still a satisfying trajectory assignment. It follows that if there is a model for U_E there is a model that does not stutter. Finally, after all paths have halted according to the U_E trajectories, a step of U_A that stutters all paths that have not halted can be removed because, again the evaluation is the same in the previous and subsequent state. It follows that if the formula has a model, then it has a model satisfying the encoding.

Theorem 1. *Let φ be an A-HLTL formula with at most one trajectory quantifier alternation, let K be the maximum depth of a Kripke structure and let $M = K \times |\mathsf{Paths}(\varphi)| \times |\mathsf{Trajs}(\varphi)|$. Then, the following hold:*

– $[\![\mathcal{K}, \varphi]\!]^{hpes}_{K,M}$ *is satisfiable if and only if* $\mathcal{K} \models \varphi$.
– $[\![\mathcal{K}, \varphi]\!]^{hopt}_{K,M}$ *is satisfiable if and only if* $\mathcal{K} \models \varphi$.

Theorem 1 (proof in [13]) provides a model checking decision procedure. An alternative decision procedure is to iteratively increase the bound of the unrollings and invoke both semantics in parallel until the outcome coincides.

4 Complexity of A-HLTL Model Checking for Acyclic Frames

Our goal in this section is to analyze the complexity of the A-HLTL model checking problem in the size of an acyclic Kripke structure (all proofs in [13]).

Problem Formulation. We use $\mathsf{MC}[\mathsf{Fragment}]$ to distinguish different variations of the problem, where MC is the model checking decision problem, i.e., whether or not $\mathcal{K} \models \varphi$, and Fragment is one of the following for φ:

- '$[\exists(\exists/\forall)^+\mathsf{A}/\mathsf{E}]^k$', for $k \geq 0$, is the fragment with a lead existential trace quantifier, one outermost universal or existential trajectory quantifier, and k (counting *all*) quantifier alternations, where $k = 0$ means the existential alternation-free fragment '$\exists^+\mathsf{E}^+$'. Fragment '$[\forall(\forall/\exists)^+\mathsf{A}/\mathsf{E}]^k$' is defined similarly, where $k = 0$ is the universal alternation-free fragment '$\forall^+\mathsf{A}^+$'.
- Fragments '$[\exists(\exists/\forall)^+(\mathsf{E}^+\mathsf{A}^+/\mathsf{A}^+\mathsf{E}^+/\mathsf{E}\mathsf{E}^+/\mathsf{A}\mathsf{A}^+)]^k$', for $k \geq 1$ denotes the fragment with a lead existential trace quantifier, multiple outermost trajectory quantifiers with at most one alternation, and k quantifier alternations (counting *all* quantifiers), where $k = 1$ means fragment '$\exists\mathsf{E}\mathsf{A}$'. Fragment '$[\forall(\forall/\exists)^+(\mathsf{E}^+\mathsf{A}^+/\mathsf{A}^+\mathsf{E}^+/\mathsf{E}\mathsf{E}^+/\mathsf{A}\mathsf{A}^+)]^k$' is defined similarly, where $k = 1$ means fragment '$\forall\mathsf{A}\mathsf{E}$'.

The Complexity of A-HLTL Model Checking. We first show the A-HLTL model checking problem for the alternation-free fragment with only one trajectory quantifier is NL-complete. For example, verification of information leak in speculative execution in sequential programs renders a formula of the form $\forall^4\mathsf{A}$, which belongs to the alternation-free fragment (more details in Section 5).

Theorem 2. $\mathsf{MC}[\exists^+\mathsf{E}]$ *and* $\mathsf{MC}[\forall^+\mathsf{A}]$ *are NL-complete.*

We now switch to formulas with alternating trace quantifiers. The significance of the next theorem is that a single trajectory quantifier does not change the complexity of model checking as compared to the classic HyperLTL verification [2]. It is noteworthy to mention that several important classes of formulas belong to this fragment. For example, according to Theorem 3 while model checking *observational determinism* [20] ($\forall\forall\mathsf{E}$), *generalized noninference* [16] ($\forall\forall\exists\mathsf{E}$), and *non-inference* [5] ($\forall\exists\mathsf{E}$) with a single initial input are all coNP-complete.

Theorem 3. $\mathsf{MC}[\exists(\exists/\forall)^+(\mathsf{A}/\mathsf{E})]^k$ *is* Σ_k^p*-complete and* $\mathsf{MC}[\forall(\forall/\exists)^+(\mathsf{E}/\mathsf{A})]^k$ *is* Π_k^p*-complete in the size of the Kripke structure.*

We now focus on formulas with multiple trajectory quantifiers. We first show that alternation-free multiple trajectory quantifiers bumps the class of complexity by one step in the polynomial hierarchy.

Theorem 4. $\mathsf{MC}[\exists(\exists/\forall)^+\mathsf{E}\mathsf{E}^+]^k$ *is* Σ_{k+1}^p*-complete and* $\mathsf{MC}[\forall(\forall/\exists)^+\mathsf{A}\mathsf{A}^+]^k$ *is* Π_{k+1}^p*-complete in the Kripke structure.*

Theorem 5. *For* $k \geq 1$, $\mathsf{MC}[\exists(\exists/\forall)^+\mathsf{A}^+\mathsf{E}^+]^k$ *is* Σ_{k+1}^p*-complete and* $\mathsf{MC}[\forall(\forall/\exists)^+\mathsf{E}^+\mathsf{A}^+]^k$ *is* Π_{k+1}^p*-complete in the size of the Kripke structure.*

Finally, Theorems 3, 4, and 5 imply that the model checking problem for acyclic Kripke structures and A-HLTL formulas with an arbitrary number of trace quantifier alternation and only one trajectory quantifier is in PSPACE.

5 Case Studies and Evaluation

We now evaluate our technique. The encoding in Section 3 is implemented on top of the open-source bounded model checker HyperQB [15]. All experiments are executed on a MacBook Pro with 2.2GHz processor and 16GB RAM (https://github.com/TART-MSU/async_hltl_tacas23).

Non-interference in Concurrent Programs. We first consider the programs presented earlier in Figs. 1 and 3 together with A-HLTL formulas φ_{NI} and $\varphi_{\mathsf{NI}_{\mathsf{nd}}}$ from Section 1. We receive UNSAT (for the original formula and not its negation), which indicates that violations have been spotted. Indeed, our implementation successfully finds a counterexample with a specific trajectory that prints out `'acdb'` when the high-security value `h` is equal to zero (entries of `ACDB` and $\texttt{ACDB}_{\texttt{ndet}}$ in Table 3). Our other experiment is an extension of the example in [10] for multiple asynchronous channels (see Fig. 6) and the following formula: $\varphi_{\mathsf{OD}_{\mathsf{nd}}} = \forall\pi.\forall\pi'.\mathsf{A}\tau.\ \mathsf{E}\tau'.\ \square\ (\mathsf{I}_{\pi,\tau} \leftrightarrow \mathsf{I}_{\pi',\tau}) \rightarrow \square\ (\mathsf{obs}_{\pi,\tau'} \leftrightarrow \mathsf{obs}_{\pi',\tau'})$. The results for this case are entries of `ConcLeak` and $\texttt{ConcLeak}_{\texttt{ndet}}$ in Table 3. Details of the counterexample can be found in [13].

```
1  Thread T1(){
2    while (true){
3      x := 0;
4      y := 0;
5      if ( h == 1) then
6        x := 1;
7        y := 1;
8      else
9        y := 1;
10       x := 1;
11   }
12 }
13 Thread T2(){
14   while (true) {
15     print x;
16     print y;
17   }
18 }
19 Thread T3(){
20   while (true){
21     h := 0||1;
22     l := 0||1;
23   }
24 }
```

Fig. 6: Program with nondeterministic sequence of inputs.

Speculative Information Flow. *Speculative execution* is a standard optimization technique that allows branch prediction by the processor. *Speculative noninterference* (SNI) [9] requires that two executions with the same *policy* `p` (i.e., initial configuration) can be observed differently in speculative semantics (e.g., a possible branch), if and only if their non-speculative semantics with normal condition checks are also observed differently; i.e., the following A-HLTL formula:

$$\varphi_{\mathsf{SNI}} = \underbrace{\forall\pi_1.\forall\pi_2.}_{\text{speculative}}\ \underbrace{\forall\pi_1'.\forall\pi_2'}_{\text{nonspeculative}}\ .\ \mathsf{A}\tau.\Big(\square(\mathsf{obs}_{\pi_1,\tau} \leftrightarrow \mathsf{obs}_{\pi_2,\tau}) \wedge$$

$$(\mathsf{p}_{\pi_1,\tau} \leftrightarrow \mathsf{p}_{\pi_2,\tau}) \wedge (\mathsf{p}_{\pi_1,\tau} \leftrightarrow \mathsf{p}_{\pi_1',\tau}) \wedge (\mathsf{p}_{\pi_2,\tau} \leftrightarrow \mathsf{p}_{\pi_2',\tau})\Big) \rightarrow \square\big(\mathsf{obs}_{\pi_1',\tau} \leftrightarrow \mathsf{obs}_{\pi_2',\tau}\big)$$

where `obs` is the memory footprint, traces π_1 and π_2 range over the (nonspeculative) `C` code and traces π_1' and π_2' range over the corresponding (speculative) assembly code. We evaluate SNI on the translation from a `C` program (details in [13]), where `y` is the input policy `p` and multiple versions of `x86` assembly code [9]. The results of model checking speculative execution are in Table 3 (see entries from $\texttt{SpecExcu}_{V1}$ to $\texttt{SpecExcu}_{V7}$). Additional versions from $\texttt{SpecExcu}_{V3}$ to $\texttt{SpecExcu}_{V7}$ are under different compilation options. Our method correctly identify all the insecure and secure ones as stated in [9].

Compiler Optimization Security. Secure compiler optimization [17] aims at preserving input-output behaviors of a *source* program (original implementation)

and a *target* program (after applying optimization), including security policies. We investigate the following optimization strategies: Dead Branch Elimination (DBE), Loop Peeling (LP), and Expression Flattening (EF). To verify a secure optimization, we consider two scenarios: (1) one single I/O event (one trajectory, similar to [1]), and (2) a sequences of I/O events (two trajectories):

$$\varphi_{\mathsf{SC}} = \forall\pi.\forall\pi'.\mathsf{E}\tau.\ (\mathsf{in}_{\pi,\tau} \leftrightarrow \mathsf{in}_{\pi',\tau}) \rightarrow \square\ (\mathsf{out}_{\pi,\tau} \leftrightarrow \mathsf{out}_{\pi',\tau})$$
$$\varphi_{\mathsf{SC}_{\mathsf{nd}}} = \forall\pi.\forall\pi'.\mathsf{A}\tau.\ \mathsf{E}\tau'.\square\ (\mathsf{in}_{\pi,\tau} \leftrightarrow \mathsf{in}_{\pi',\tau}) \rightarrow \square\ (\mathsf{out}_{\pi,\tau'} \leftrightarrow \mathsf{out}_{\pi',\tau'}),$$

where in is the set of inputs and out is the set of outputs. Table 3 (cases DBE – $\texttt{EFLP}_{\texttt{ndet}}$) shows the verification results of each optimization strategy and different combination of the strategies (details in [13]).

Cache-Based Timing Attacks. Asynchrony also leads to attacks when system executions are confined to a single CPU and its cache [18]. A cache-based timing attack happens when an attacker is able to guess the values of high-security variables when cache operations (i.e., evict, fetch) influence the scheduling of different threads. Our case study is inspired by the cache-based timing attack example in [18] and we use the formula of observational determinism $\varphi_{\mathsf{OD}_{\mathsf{nd}}}$ introduced earlier in this section to find the potential attacks (see cases of CacheTA and $\texttt{CacheTA}_{\texttt{ndet}}$ in Table 3 with details in [13]).

5.1 Analysis of Experimental Results

Table 3 presents the diameter of the transition relation, length of trajectories m, state spaces, and the number of trajectory variables. We also present the total solving time of our algorithm as well as the break down: generating models (genQBF), building trajectory encodings (buildTr), and final QBF solving (solveQBF). Our two most complex cases are concurrent leak ($\texttt{ConcLeak}_{\texttt{ndet}}$) and loop peeling ($\texttt{LP}_{\texttt{ndet}}$). For concurrent leak, it is because there are three threads with many interleavings (i.e., asynchronous composition), takes longer time to build. For loop peeling, although there is no need to consider interleavings except for the nondeterministic inputs; however, the diameters of traces ($D_{\mathcal{K}_1}$, $D_{\mathcal{K}_2}$) are longer than other cases, which makes the length and size of trajectory variables (i.e., m and $|T|$) grow and increases the total solving time. Our encoding is able to handle a variety of cases with one or more trajectories, depending on whether multiple sources of non-determinism is present. To see efficiency, we compare the solving time for cases of compiler optimization with one trajectory with the results in [1]. This method reduces A-HLTL model checking to HyperLTL model checking for limited fragments and utilizes the model checker MCHyper. On the other hand, we directly handle asynchrony by trajectory encoding. Table 2 shows our algorithm considerably outperforms the approach in [1] in larger cases.

	MCHyper [1]	This paper	
Case	**Total**[s]	genQBF/ buildTr/ solveQBF[s]	**Total**[s]
DBE	**0.8**	0.9 / 0.07 / 0.01	**0.98**
LP	**365.9**	1.37 / 1.40 / 1.13	**3.90**
EFLP	**1315.2**	5.11 / 8.12 / 9.35	**22.58**

Table 2: Comparison of model checking compiler optimization with [1].

		(model checking spec and data)							(time took for solving)			
Models	φ	$D_{\mathcal{K}_1}$	$D_{\mathcal{K}_2}$	m	$\lvert S_{\mathcal{K}_1}\rvert$	$\lvert S_{\mathcal{K}_2}\rvert$	$\lvert T\rvert$	QBF	genQBF[s]	buildTr[s]	solveQBF[s]	**Total**[s]
ACDB	φ_{NI}	6	6	12	109	109	1378	UNSAT	2.80	0.32	0.23	**3.35**
$\text{ACDB}_{\text{ndet}}$	$\varphi_{\mathsf{NI}_{\text{nd}}}$	8	8	16	696	696	2754	UNSAT	7.74	2.54	3.73	**14.01**
ConcLeak	φ_{OD}	11	11	22	597	597	6118	UNSAT	14.85	7.10	8.29	**30.24**
$\text{ConcLeak}_{\text{ndet}}$	$\varphi_{\mathsf{OD}_{\text{nd}}}$	18	18	36	2988	2988	22274	UNSAT	127.09	53.14	731.48	**911.72**
SpecExcu_{V1}	φ_{SNI}	3	6	9	132	340	1112	UNSAT	7.45	1.72	3.07	**12.24**
SpecExcu_{V2}	φ_{SNI}	3	6	9	144	168	1112	SAT	5.61	1.28	2.44	**9.33**
SpecExcu_{V3}	φ_{SNI}	3	6	9	87	340	636	UNSAT	7.30	1.68	2.97	**11.95**
SpecExcu_{V4}	φ_{SNI}	3	6	9	93	340	636	UNSAT	7.37	1.71	4.50	**13.58**
SpecExcu_{V5}	φ_{SNI}	3	6	9	132	168	636	SAT	6.23	1.23	3.48	**10.94**
SpecExcu_{V6}	φ_{SNI}	3	7	10	132	340	766	UNSAT	7.47	1.82	3.26	**12.55**
SpecExcu_{V7}	φ_{SNI}	2	5	7	144	168	352	SAT	5.83	1.28	2.58	**9.69**
DBE	φ_{SC}	4	4	8	8	6	546	SAT	0.9	0.07	0.01	**0.98**
DBE_{ndet}	$\varphi_{\mathsf{SC}_{\text{nd}}}$	13	13	26	82	72	9414	SAT	1.60	0.56	9.61	**11.77**
DBE_{ndet} w/ bugs	$\varphi_{\mathsf{SC}_{\text{nd}}}$	13	13	26	82	72	9414	UNSAT	1.36	0.49	2.05	**3.90**
LP	φ_{SC}	22	22	44	80	76	3870	SAT	1.37	1.40	1.13	**3.90**
LP_{ndet}	$\varphi_{\mathsf{SC}_{\text{nd}}}$	17	17	34	558	811	19110	SAT	7.37	3.86	48.15	**59.38**
LP_{ndet} w/ loops	$\varphi_{\mathsf{SC}_{\text{nd}}}$	33	35	68	757	1591	128114	SAT	30.52	34.99	4165.54	**4231.05**
LP_{ndet} w/ bugs	$\varphi_{\mathsf{SC}_{\text{nd}}}$	17	17	34	558	661	19110	UNSAT	6.51	3.60	20.75	**30.86**
EFLP	φ_{SC}	32	32	64	80	248	108290	SAT	5.11	8.12	9.35	**22.58**
$\text{EFLP}_{\text{ndet}}$	$\varphi_{\mathsf{SC}_{\text{nd}}}$	18	22	40	582	1729	28986	SAT	15.92	8.90	135.48	**160.30**
$\text{EFLP}_{\text{ndet}}$ w/ loops	$\varphi_{\mathsf{SC}_{\text{nd}}}$	33	45	78	295	1996	178894	SAT	36.98	62.89	121.60	**221.47**
CacheTA	φ_{OD}	13	13	26	48	48	9414	UNSAT	1.49	0.53	0.38	**2.40**
$\text{CacheTA}_{\text{ndet}}$	$\varphi_{\mathsf{OD}_{\text{nd}}}$	58	58	16	16	32	16258	UNSAT	1.95	1.33	1.02	**4.30**
$\text{CacheTA}_{\text{ndet}}$ w/ loops	$\varphi_{\mathsf{OD}_{\text{nd}}}$	35	35	70	88	88	139302	UNSAT	5.50	27.65	125.92	**159.07**

Table 3: Case studies break down for Kripke structures: $\mathcal{K}_1, \mathcal{K}_2$ (all case studies have two, e.g.,one for high-level and one for assembly code), formula: φ, diameter: D, state space: $|S|$, trajectory depth: m, and size of trajectory variables: $|T|$.

6 Conclusion and Future Work

In this paper, we focused on the problem of A-HLTL model checking for *terminating* programs. We generalized A-HLTL to allow nested *trajectory* quantification, where a trajectory determines how different traces may advance and stutter. We rigorously analyzed the complexity of A-HLTL model checking for acyclic Kripke structures. The complexity grows in the polynomial hierarchy with the number of quantifier alternations, and, it is either aligned with that of HyperLTL or is one step higher in the polynomial hierarchy. We also proposed a BMC algorithm for A-HLTL based on QBF-solving and reported successful experimental results on verification of information flow security in concurrent programs, speculative execution, compiler optimization, and cache-based timing attacks.

Asynchronous hyperproperties enable logic-based verification for software programs. Thus, future work includes developing different abstraction techniques such as predicate abstraction, abstraction-refinement, etc, to develop software model checking techniques. We also believe developing synthesis techniques for A-HLTL creates opportunities to automatically generate secure programs and assist in areas such as secure compilation.

References

1. J. Baumeister, N. Coenen, B. Bonakdarpour, B. Finkbeiner, and C. Sánchez. A temporal logic for asynchronous hyperproperties. In *Proc. of the 33rd Int'l Conf. on Computer Aided Verification (CAV'21), Part I*, volume 12759 of *LNCS*, pages 694–717. Springer, 2021.
2. B. Bonakdarpour and B. Finkbeiner. The complexity of monitoring hyperproperties. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium CSF*, pages 162–174, 2018.
3. L. Bozzelli, A. Peron, and C. Sánchez. Asynchronous extensions of HyperLTL. In *Proc. of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'21)*, pages 1–13. IEEE, 2021.
4. M. R. Clarkson, F. Finkbeiner, K. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *Proceedings of the 3rd International Conference on Principles of Security and Trust (POST)*, pages 265–284, 2014.
5. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
6. N. Coenen, B. Finkbeiner, C. Hahn, and J. Hofmann. The hierarchy of hyperlogics. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2019.
7. N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup. Verifying hyperliveness. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification*, pages 121–139, Cham, 2019. Springer International Publishing.
8. B. Finkbeiner, M. Rabe, and C. Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *In Proc. of the 27th Int'l Conf. on Computer Aided Verification (CAV'15)*, volume 9206 of *LNCS*, pages 30–48. Springer, 2015.
9. M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. SPECTECTOR: Principled detection of speculative information flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, S&P 2020. IEEE, 2020.
10. G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, pages 218–232, 2007.
11. J. O. Gutsfeld, M. Müller-Olm, and C. Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.
12. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
13. T. Hsu, B. Bonakdarpour, B. Finkbeiner, and C. Sánchez. Bounded model checking for asynchronous hyperproperties. *CoRR*, abs/2301.07208, 2023.
14. T. Hsu and C. Sánchez. Hyperqube: A qbf-based bounded model checker for hyperproperties. *CoRR*, abs/2109.12989, 2021.
15. T.-H. Hsu, C. Sánchez, and B. Bonakdarpour. Bounded model checking for hyperproperties. In *Proceedings of the 27th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 94–112, 2021.
16. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 79–93, Apr. 1994.

17. K. S. Namjoshi and L. M. Tabajara. Witnessing secure compilation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 1–22. Springer, 2020.
18. D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security*, pages 718–735. Springer, 2013.
19. S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*, page 29, 2003.

Model Checking Linear Dynamical Systems under Floating-point Rounding*

Engel Lefaucheux[1](✉), Joël Ouaknine[2], David Purser[3,4], and Mohammadamin Sharifi[5]

[1] University of Lorraine, CNRS, Inria, LORIA, Nancy, France
engel.lefaucheux@inria.fr
[2] Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrücken, Germany
joel@mpi-sws.org
[3] University of Warsaw, Warsaw, Poland
[4] University of Liverpool, Liverpool, UK
D.Purser@liverpool.ac.uk
[5] Sharif University of Technology, Tehran, Iran
sharifim689@gmail.com

Abstract. We consider linear dynamical systems under floating-point rounding. In these systems, a matrix is repeatedly applied to a vector, but the numbers are rounded into floating-point representation after each step (i.e., stored as a fixed-precision mantissa and an exponent). The approach more faithfully models realistic implementations of linear loops, compared to the exact arbitrary-precision setting often employed in the study of linear dynamical systems.

Our results are twofold: We show that for non-negative matrices there is a special structure to the sequence of vectors generated by the system: the mantissas are periodic and the exponents grow linearly. We leverage this to show decidability of ω-regular temporal model checking against semi-algebraic predicates. This contrasts with the unrounded setting, where even the non-negative case encompasses the long-standing open Skolem and Positivity problems.

On the other hand, when negative numbers are allowed in the matrix, we show that the reachability problem is undecidable by encoding a two-counter machine. Again, this is in contrast with the unrounded setting where point-to-point reachability is known to be decidable in polynomial time.

Keywords: Model Checking · Floating-point · Dynamical Systems.

1 Introduction

Loops are a fundamental staple of any programming language, and the study of loops plays a pivotal role in many subfields of computer science, including automated verification, abstract interpretation, program analysis, semantics, etc. The focus of the present paper is on the algorithmic analysis of simple (i.e., non-nested) linear (or affine) while loops, such as the following:

* A long version of this paper is available as [19].

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 47–65, 2023.
https://doi.org/10.1007/978-3-031-30823-9_3

```
x = 3, y = 4, z = 2
while x+3y+z > 4:
    x = 3x +2z
    y = 3x + y
    z = y + z
```

We are interested in analysing how the loop evolves. A simple reachability query is to decide whether the loop variables ever satisfy a Boolean combination of polynomial inequalities, for example modelling a loop guard. More generally, one might seek to consider significantly more complex temporal properties, such as those expressible in linear temporal logic or monadic second-order logic: this gives rise to a model-checking problem.

Modelling the evolution of such a loop may require unbounded memory. That is, the number of bits needed to represent the numbers x, y, and z may grow larger and larger. However, most computer systems do not represent rational numbers to arbitrary precision, but rather use *floating-point rounding*, in which a number y is stored using two components: the mantissa $m \in \mathbb{Q}$ and the exponent $\alpha \in \mathbb{Z}$, such that $y = m \cdot 10^{\alpha}$.[6]

Typically floating-point numbers are specified using either 32 or 64 bits, with some of these reserved for the mantissa and some for the exponent, thus bounding both the mantissa and the exponent. **We do not do this**, and only place a bound on the number of bits representing the mantissa, allowing the exponent to grow unboundedly (in either direction). From a theoretical standpoint, bounding the number of bits of both the mantissa and the exponent would necessarily give rise to a finite-state system, for which essentially any decision problem would become decidable (at least in principle, if not necessarily in practice). Due to the unboundedness of exponents in our setting, we do not have to consider overflows ('NaN', 'infinity' or '-infinity' which are part of most floating-point specifications).

Formally, we model our programs using linear dynamical systems (LDS), which comprise a starting vector representing the initial state of each variable and a matrix describing the evolution of the program. An LDS generates an infinite sequence of vectors (the *orbit* of the system) by multiplying the matrix with the current vector and then applying floating-point rounding to the result.

Our results

We consider the *model-checking* problem for linear dynamical systems evolving under floating-point rounding. More formally, let $Y_1, \ldots, Y_k \subseteq \mathbb{R}^d$ be semi-algebraic targets. Given an orbit $(x^{(t)})_{t \in \mathbb{N}}$, we define the characteristic word $w = w_1, w_2, w_3, \ldots$ with respect to $Y_1, \ldots, Y_k$ over alphabet $2^{\{1,\ldots,k\}}$ such that $i \in w_t$ if and only if $x^{(t)} \in Y_i$. The model-checking problem asks whether w is in an ω-regular language, or equivalently satisfies a temporal specification given in monadic second-order logic (MSO).

[6] We work in base 10 throughout for simplicity of exposition. All our results carry over *mutatis mutandis* in any integer base, including base 2 as typically used in practice.

Our results show that analysing LDS under floating-point rounding is neither clearly easier nor harder than in the standard setting (without rounding). Our first contribution establishes *undecidability* of point-to-point reachability (and *a fortiori* model checking) under floating-point rounding, a surprising outcome given that point-to-point reachability is solvable in polynomial time without rounding [16]. On the other hand, in the standard setting neither decidability nor undecidability are known for full model checking (although mathematical hardness results exist); see [24,18,17].

Theorem 1. *The floating-point point-to-point reachability problem is undecidable.*

However, for non-negative matrices, we show that the full MSO model-checking problem is decidable in our setting, without restrictions on the dimensions of the predicates or the ambient space. This is in stark contrast to the standard setting, where assuming non-negativity does not simplify the problem. Model checking non-negative LDS without rounding would require (at a minimum) solving the longstanding open Skolem and Positivity problems [2].

Theorem 2. *Let (M, x) be a non-negative linear dynamical system, let $Y_1, \ldots, Y_k$ be semialgebraic targets and let ϕ be an MSO formula using predicates over $Y_1, \ldots, Y_k$. It is decidable whether the characteristic word under floating-point rounding satisfies ϕ.*

We place no dimension restriction on the predicates; in particular, showing that the Skolem and Positivity problems are *decidable* on non-negative systems under floating-point rounding. At this time we do not however have complexity upper bounds on our model-checking algorithm, or lower bounds on the model-checking problem.

Related work

There is a line of practical tools for the analysis, verification, and invariant synthesis for floating-point loops [7,20,1,22]. These tools typically work well in practice, but do not necessarily work in all cases. The analysis of concrete implementations of floating-point specifications requires careful analysis of edge cases around $\pm\infty$ and 'NaN'. In contrast to these tools which focus primarily on practical analysis, our work seeks to understand the theoretical possibilities and limitations of the exact analysis of (possibly long-running) floating-point loops in a generalised setting.

The study of linear dynamical systems explores the sequence of vectors induced by a matrix. Model checking is only known to be decidable for certain classes of semialgebraic predicates—in particular those with low dimension [18] or for prefix-independent properties [4]; see also [17]. The well-known Skolem and Positivity problems being special cases of model checking, they place technical limits on the dimensions that can be handled without first resolving long-standing open cases of these problems. Recent progress suggests that the Skolem

problem may be yet be conquered, at least for diagonalisable matrices [8,21], but Positivity requires solving particularly difficult problems in analytic number theory [24,12]. The non-negative case can be used to model sequences of distributions induced by Markov chains [6], although all hardness limitations apply already in the probabilistic setting [2].

Baier et al. [5] consider LDS under rounding to fixed-decimal precision, showing reachability is PSPACE-complete for hyperbolic systems (when no eigenvalue has modulus one) and decidable for certain other constrained classes of rounding. A notable difference of fixed-decimal precision is that it cannot allow arbitrarily small numbers, unlike the floating-point numbers we consider.

A recent line of work focusses on linear dynamical systems with perturbations at every step, with a view to understanding the robustness of reachability problems [13,14,3]. However, unlike rounding, the perturbation is chosen in order to assist hitting the target and the perturbation is arbitrarily small.

For linear while loops the reachability problem can be rephrased as a halting problem, asking whether a guard condition is eventually met from a given initial state. The related termination problem asks whether a guard condition is met from *every* initial state [26,10]. Issues arising from implementations using floating-point representations to solve the termination problem of unrounded (arbitrary precision) loops are considered in [27]. In contrast, we are interested in analysing programs in which the intended behaviour is to round the numbers to fixed-precision floating-point numbers at every step of the loop.

Organisation In Section 2, we formalise the model and problems and discuss some of the properties of floating-point rounding. In Section 3, we present our undecidability result for the general case. Finally, in Section 4 we establish some special periodic structure associated with the orbit and use this structure in Section 5 to show that model checking is decidable for non-negative LDS.

2 Preliminaries

2.1 Linear dynamical systems and rounding functions

Definition 1. *A d-dimensional linear dynamical system (LDS) (M, x) comprises a matrix $M \in \mathbb{Q}^{d\times d}$ and an initial vector $x \in \mathbb{Q}^d$.*

Given a rounding function $[\cdot] : \mathbb{Q}^d \to \mathbb{Q}^d$, and an LDS (M, x) the rounded orbit $\mathcal{O}$ is the sequence $(x^{(t)})_{t\in\mathbb{N}}$ such that $x^{(0)} = [x]$ and $x^{(t)} = [Mx^{(t-1)}]$ for all $t \geq 1$.

Given $p \in \mathbb{N}$, we say that a number x is a floating-point number with precision p if $x = m \cdot 10^\alpha$ such that $m \in \mathbb{Q}$ is a decimal number in $\{0\} \cup [0.1, 1)$ with p digits in the fractional part (after the decimal point) and $\alpha \in \mathbb{Z}$. In particular, we associate by convention the number with mantissa $m = 0$ to the exponent $-\infty$. Given a number $x = m \cdot 10^\alpha$ we define $\mathsf{mantissa}(x) = m$ and $\mathsf{exponent}(x) = \alpha$.

We are interested in the floating-point rounding function $[\cdot]$ with precision $p \in \mathbb{N}$. Given a real number $x \in \mathbb{R}$, we define $[x]$, the floating-point rounding of

x, as the closest floating-point number with precision p based on the first $p+1$ digits of x.

Where there are two possible choices, any deterministic choice that is consistent with the properties listed below is acceptable.[7] We denote by $\mathbb{FP}_{10}[p]$ the subset of $\mathbb{Q}$ representable in base 10 as a floating-point numbers with p digits. We use the following useful properties of the rounding function:

- it is *log-bounded*, *i.e.* there exists a constant $c \in \mathbb{R}_+$ such that $\forall x \in \mathbb{R}$, $\frac{|x|}{c} \leq |[x]| \leq c|x|$.
- it is *mantissa-based*, *i.e.* if $x = 10^\alpha x'$, then $[x] = 10^\alpha [x']$.
- it is $(p+1)$-*finite*, *i.e.* the output of the rounding is not dependent on the i-th digit of the mantissa, for each integer $i > p+1$. In other words, if x and x' agree on the first $p+1$ digits then $[x] = [x']$.
- it is *sign preserving*, *i.e.* $\text{sign}(x) = \text{sign}([x])$. The fact that $[x] = 0$ if and only if $x = 0$ also follows from the log-bounded property.

The floating-point rounding is defined above on a single real. It is extended straightforwardly to a vector x by applying it to each of its components $(x)_i$ where i ranges from 1 to the dimension of the vector. As such, the term $[Mx]$ is obtained by first computing exactly the the vector Mx and then by rounding each component $(Mx)_i$. An alternative approach could be to maintain each subcomputation in p-bits of precision, *but this is not the approach we take*. Such an orbit can be simulated in our setting by increasing the dimension so that operations can be staggered in a way that at most one operation (scalar product or variable addition) is used in each assignment.

2.2 Model checking

We consider the model-checking problem of an LDS over semialgebraic sets.

Definition 2. *A semialgebraic set $Y \subseteq \mathbb{R}^d$ is defined by a finite Boolean combination of polynomial inequalities.*

Let (M, x) be an LDS with rounded orbit $\mathcal{O}$ and $\mathcal{Y} = \{Y_1, \ldots, Y_k\}$ be a collection of semialgebraic sets. The characteristic word of $\mathcal{O}$ is $w = w_1 w_2 w_3 \ldots \in (2^{\{1,\ldots,k\}})^\omega$, such that $j \in w_t$ if and only if $x^{(t)} \in Y_j$.

The model-checking problem asks whether the characteristic word is contained within a given ω-regular language, usually specified in a temporal logic such as monadic second order logic (MSO), or often its LTL fragment. Without loss of generality we assume that the property is given as a Büchi automaton [11].

Problem 1 (Floating-point Model-checking Problem). Given an LDS (M, x) with rounded orbit $\mathcal{O}$, a collection of semialgebraic sets $\mathcal{Y} = \{Y_1, \ldots, Y_k\}$ and an ω-regular specification ϕ, the model-checking problem consists in deciding whether the characteristic word w of $\mathcal{O}$ satisfies the specification ϕ.

[7] For example, always rounding up, always rounding down, round to even, rounding towards zero, rounding away from zero are acceptable, providing the choice is fixed.

We will also consider the point-to-point reachability problem, which is a subcase of the model-checking problem (Problem 1):

Problem 2 (Floating-point Point-to-point Reachability Problem). Given a d-dimensional LDS (M, x), and a target vector $y \in \mathbb{Q}^d$, the point-to-point reachability problem consists in deciding whether y belongs to the rounded orbit $\mathcal{O}$.

Given a target $Y \subseteq \mathbb{R}^d$, we associate the set of hitting times $\mathcal{Z}(Y) = \{t \mid x^{(t)} \in Y\}$. Under this formulation, the reachability problem is reformulated as whether $\mathcal{Z}(Y)$ is empty. However, for model checking we will develop a more comprehensive understanding of the hitting times of each target $Y_1, \ldots, Y_k$.

2.3 Structure of M

Formally, M is a d-dimensional matrix indexed by the elements $\{1, \ldots, d\}$. However, we interpret M as an automaton over states $Q = \{q_1, \ldots, q_d\}$ and reference the entries of M by pairs of states. That is, we refer to M_{q_1,q_2} rather than $M_{1,2}$.

We denote by G_M the weighted directed graph whose adjacency matrix is M. That is, a graph with vertices Q and with an edge from q_j to q_i weighted by M_{q_i,q_j} if $M_{q_i,q_j} \neq 0$.[8]

Let $S_1, \cdots, S_s \subseteq Q$ be the strongly connected components (SCCs) of G_M. Our analysis will consider each strongly connected component separately, thus it will often be useful to consider the entries of $x \in \mathbb{FP}_{10}[p]^Q$ corresponding only to one strongly connected component. Without loss of generality, by reordering the states where necessary, we assume that the states in Q are ordered so that states within the same SCC appear next to one another, and the strongly connected components are topologically sorted, *i.e.* there is no edge from S_i to S_j where $i > j$. We split a vector x into s smaller vectors, denoted $x_{S_1}, \ldots, x_{S_s}$, each representing the entries of x corresponding to the SCC. Letting $x_{S_j} = (z_{1,j}, \cdots, z_{d_j,j})^T$ and $|S_j| = d_j$, we thus have x is partitioned as

$$x = (z_{1,1} \cdots z_{d_1,1}, \cdots, z_{1,s} \cdots z_{d_s,s})^T.$$

Moreover, for each pair of SCCs S_i, S_j, we denote by M_{S_i,S_j} the submatrix of M restricted to the rows related to S_i and columns related to S_j, which is a matrix with d_i rows and d_j columns. If $S_i = S_j$, we simply write M_{S_i}. In other words, M_{S_i,S_j} is the matrix that shows the dependency between S_i and S_j, and we have

$$M = \begin{pmatrix} M_{S_1} & M_{S_1,S_2} & \cdots & M_{S_1,S_s} \\ M_{S_2,S_1} & M_{S_2} & \cdots & M_{S_2,S_s} \\ \vdots & \vdots & \ddots & \vdots \\ M_{S_s,S_1} & M_{S_s,S_2} & \cdots & M_{S_s} \end{pmatrix}$$

We say S_i *feeds* S_j, and S_j is *fed by* S_i if there is some edge in G_M from some state in S_i to some state in S_j.

[8] Note that the orientation of the edge may appear switched from the reader's expectation. This is due to the convention that M is pre-multiplied with x at every step.

3 Undecidability of point-to-point reachability

In this section, we give a sketch of the proof of the undecidability of Problem 2 (and thus of Problem 1) in the general case. The full proof can be found in the long version of this paper [19].

Theorem 1. *The floating-point point-to-point reachability problem is undecidable.*

This result is obtained by reduction from the termination of a two-counter Minsky machine. We recall the definition of this model:

Definition 3. *A two-counter Minsky machine is defined by a finite set of states $\ell_1, \ldots, \ell_m$, a distinguished starting state (w.l.o.g. ℓ_1), a distinguished halting state (w.l.o.g. ℓ_m), two natural integer counters, here denoted as x and y, and a mapping deterministically associating to each state transition a particular action.*

Each transition takes one of the following forms: for $z \in \{x, y\}$,

increment $\mathrm{inc}_z(\ell_j)$*: add 1 to counter z, move to state ℓ_j.*
decrement $\mathrm{dec}_z(\ell_j)$*: remove 1 from counter z if $z > 0$, move to state ℓ_j.*
zero test $\mathrm{zero?}_z(\ell_j, \ell_k)$*: if $z = 0$ move to state ℓ_j else move to state ℓ_k.*

The configuration of a two-counter Minsky machine consists of the current state and the values of x and y.

Without loss of generality (by first using a zero test), one can assume a decrementation operation is never used in a configuration where the counter to be decreased has value 0, hence removing the need to check whether $z > 0$.

The halting problem asks whether, starting in configuration $(\ell_1, 0, 0)$, that is, in the distinguished starting state with both counters set to 0, whether the state ℓ_m is reached. The problem is undecidable [23].

We build an LDS with mantissa length $p = 1$ and base 10 that simulates a run of a given Minsky machine. The reduction happens to maintain the invariant that each mantissa always has the value 0 or 1 after rounding (although, as we operate in base 10, there are 10 possible values the mantissa could have taken). For ease of readability, we describe this LDS using variables to represent the dimensions and linear functions to represent the transition matrix. For each state of the Minsky machine, we use two variables corresponding to the two counters. Throughout the simulation, if the Minsky machine is in state j, the counter values are stored in the exponents of the variables associated with state j, and all other variables are zero.

The crux of our reduction lies in the handling of the zero test. More precisely, suppose we need to branch depending on whether x is equal to 0, then we need to define linear transitions that transfer the values of the two counters from one pair of variables to the appropriate new pair of variables. This is done using filter functions: the function $\mathrm{filter}_+(u, v)$ (resp. $\mathrm{filter}_-(u, v)$) is equal to v if $v \geq u$ (resp. $v < u$) and to 0 otherwise. We end this sketch with the construction of these functions and proof that they operate as advertised.

Lemma 1. *Given u, v of the form 10^c with $c \in \mathbb{N}$, one can compute the value $w = \text{filter}_+(u, v)$ in three linear operations with floating-point rounding.*

Proof. We compute $w = \text{filter}_+(u, v)$ in three successive operations using two temporary variables, *temp* and *temp*2, initially set at 0 (recall, rounding is applied after each step):

$$\begin{aligned} temp &\leftarrow u + v \\ temp2 &\leftarrow temp - u \\ w &\leftarrow 1.1 * temp2 \end{aligned}$$

Let $c_1, c_2 \in \mathbb{N}$ such that $u = 10^{c_1}$ and $v = 10^{c_2}$. Recall that the notation $[\cdot]$ is the floating-point rounding function.

First observe that if $c_1 = c_2$:

$$\begin{aligned} temp &\leftarrow [10^{c_1} + 10^{c_2}] = 2 \cdot 10^{c_1} \\ temp2 &\leftarrow [2 \cdot 10^{c_1} - 10^{c_1}] = 10^{c_1} (= v) \\ w &\leftarrow [1.1 \cdot 10^{c_1}] = 10^{c_1} = v \quad \text{as required.} \end{aligned}$$

Secondly, assume that $u > v$, and thus $c_1 > c_2$:

$$\begin{aligned} temp &\leftarrow [10^{c_1} + 10^{c_2}] = 10^{c_1} = u \\ temp2 &\leftarrow [10^{c_1} - 10^{c_1}] = 0 \\ w &\leftarrow [1.1 \cdot 0] = 0 \quad \text{as required.} \end{aligned}$$

We split the case that $v > u$, thus $c_2 > c_1$, into two cases. Suppose $c_2 > c_1 + 1$:

$$\begin{aligned} temp &\leftarrow [10^{c_1} + 10^{c_2}] = 10^{c_2} = v \\ temp2 &\leftarrow [10^{c_2} - 10^{c_1}] = [0.\underbrace{99\ldots99}_{c_2 - c_1 \geq 2} \cdot 10^{c_2}] = 1 \cdot 10^{c_2} = v \\ w &\leftarrow [1.1 \cdot 10^{c_2}] = 10^{c_2} = v \quad \text{as required.} \end{aligned}$$

Finally, $c_2 = c_1 + 1$:

$$\begin{aligned} temp &\leftarrow [10^{c_1} + 10^{c_2}] = 10^{c_2} = v \\ temp2 &\leftarrow [10^{c_2} - 10^{c_1}] = [0.9 \cdot 10^{c_2}] = 9 \cdot 10^{c_2 - 1} \\ w &\leftarrow [1.1 \cdot 9 \cdot 10^{c_2 - 1}] = [9.9 \cdot 10^{c_2 - 1}] = 10 \cdot 10^{c_2 - 1} = 10^{c_2} = v \end{aligned}$$

as required. □

Corollary 1. *Given u, v of the form 10^c with $c \in \mathbb{N}$, one can compute the value $w = \text{filter}_-(u, v)$ in four linear operations with floating-point rounding.*

Proof. Observe that $\text{filter}_-(u, v) = v - \text{filter}_+(u, v)$, which can be encoded in four steps by first computing $\text{filter}_+(u, v)$ in three steps. □

4 Pseudo-periodic orbits of non-negative LDS

We shift our focus to proving that model checking is decidable for systems with non-negative matrices. We first establish the behaviour of the system in this section and then complete the proof of Theorem 2 in Section 5. Our main result is that the rounded orbit of an LDS is periodic in the following sense, which we call *pseudo-periodic*.

Definition 4. *A sequence $(x^{(t)})_{i \in \mathbb{N}}$ of d-dimensional vectors of floating-point numbers is called pseudo-periodic if and only if there exists a starting point $N \in$*

$\mathbb{N}$, *period* $T \in \mathbb{N}$ *and growth rates* $\alpha_1, \ldots, \alpha_d \in \mathbb{Z}$ *such that*

$$\forall t \geq N, \forall j \in \{1, \ldots, d\}, (x^{(t+T)})_j = 10^{\alpha_j}(x^{(t)})_j.$$

We say the sequence is effectively *pseudo-periodic if the defining constants* $N, T, \alpha_1, \ldots, \alpha_d$ *can be computed.*

Theorem 3. *Let* (M, x) *be a* d*-dimensional LDS where* M *is non-negative and let* $(x^{(t)})_{t\in\mathbb{N}}$ *be its rounded orbit.*

The rounded orbit $(x^{(t)})_{t\in\mathbb{N}}$ *is effectively pseudo-periodic.*

In order to establish this result, we will find some partitions of the graph associated to M such that each part is effectively pseudo-periodic with the same increasing rate α for every state in the partition.

4.1 Preprocessing periodicity

The core of our approach is to show that, within each SCC of the graph associated to M, the values associated with states are of similar magnitude. This is however only true if the SCC is aperiodic. When a state is in a periodic SCC its value could change drastically depending on which phase the system is in. For example, consider a simple alternation between two states, in which the value is very large in one state and very small in the other; the states will alternate between big and small values.

We "hide" these periodic behaviours by blowing up the system so that each SCC of the new system describes only one of the periodic subsequence and we will subsequently show that the value of each state in an SCC is either zero or of a similar magnitude.

We apply the following construction to our system. Let P be the period, defined as the least common multiple of the length of every simple cycle in the graph. Let Q be the indices of M (*i.e.* the states of the generated automaton). We define new states $Q' = Q \times \{0, \ldots, P-1\}$ by annotating each state in Q with the phase. To avoid cluttering notation we will regularly refer to states in Q' in the form $(q, i + \ell)$ for $\ell \in \mathbb{Z}$, on the understanding that the phase, $i + \ell$, is normalised into $\{0, \ldots, P-1\}$ by taking the residue modulo P if necessary. We define a new matrix M' over the states Q' such that $M'_{(q,i+1),(q',i)} = M_{q,q'}$ for $i \in \{0, \ldots, P-1\}$, and zero otherwise. We initialise a new starting vector $x^{(0)}_{(q,0)} = x^{(0)}_q$ and $x^{(0)}_{(q,i)} = 0$ for $i \in \{1, \ldots, P-1\}$.

Intuitively, at each time step t the vector generated by the original system is equal to the vector of the new system restricted to the states indexed by $i \equiv t \mod P$ and every state with another index is equal to 0.

Let $S \subseteq Q$ be a strongly connected component. In Q' there exists strongly connected components $S'_1, \ldots, S'_k \subseteq Q'$ with $k \leq |S|$ such that $\bigcup_{i=1}^{k} S'_i = S \times \{0, \ldots, P-1\}$. Each set S'_j is periodic, with period P.

Henceforth in the rest of this section we work on the system (M', x') implicitly over states Q' which, by overloading of notation, we rename (M, x) over Q to avoid cluttering notation.

Note that this transformation also requires to marginally complicate the targets. Indeed, consider a set $Y \subseteq \mathbb{R}^Q$. We define the sets Y/i for $i < P$ such that $Y/i = \{y \in \mathbb{R}^{Q'} \mid \exists y' \in Y \ : \ y_{(q,i)} = y'_q \text{ for } q \in Q \text{ and } y'_{(q,j)} = 0 \text{ for } j \neq i\}$. The hitting times of Y, $\mathcal{Z}(Y)$, in the original LDS can then be obtained in the new LDS as the disjoin union: $\bigcup_{i\in\{0,\dots,P-1\}} \mathcal{Z}(Y/i)$. It suffices to characterise the hitting times for each Y/i.

4.2 Pseudo-periodicity within top SCCs

Let us first consider top SCCs, these are SCCs with no incoming edges from states of other SCC, and therefore the value of each variable at each step depends only on the value of states in the same SCC.

Lemma 2. *Let S_j be a strongly connected component of (M, x). Let $S_{j,i} = \{(q, i) \in S_j\}$ be the states associated with S_j from the i-th phase.*

There exists $C \leq Pd^2$, such that, for every i, j, $(M^C)_{S_{j,i}}$ is positive.

Proof. The matrix $(M^P)_{S_{j,i}}$ is non-negative, irreducible (*i.e.*, its graph is strongly connected) and of period 1. As such, $(M^P)_{S_{j,i}}$ is primitive [9] which means that a power C' of this matrix is positive. The theorem follows with $C = PC'$. Moreover, C' is at most $d^2 - 2d + 2$ [25]. □

Our goal is to show that within an SCC, each of the non-zero entries are of a similar magnitude due to the presence of a relatively short path (C) between any two states in the SCC. To do this we introduce the notion of closeness and observe some useful properties.

Definition 5. *We say two numbers $x, x' \in \mathbb{FP}_{10}[p]$ are δ-close, denoted by $x \approx_\delta x'$ if $|\mathsf{exponent}(x) - \mathsf{exponent}(x')| < \delta$. In particular, for every $\delta > 0$, zero is assumed to be δ-close only to itself.*

We extend the notion to vectors $y, y \in \mathbb{FP}_{10}[p]^S$, indexed by $S \subseteq Q$, such that $y \approx_\delta y'$ if all entries of the same phase are δ-close to one another across both y and y', that is, for each phase $i \in \{0, \dots, P-1\}$ and all $(q, i), (q', i) \in S$: $y_{(q,i)} \approx_\delta y'_{(q',i)}$, $y_{(q,i)} \approx_\delta y_{(q',i)}$ and $y'_{(q,i)} \approx_\delta y'_{(q',i)}$.

Proposition 1. *Let $x, x' \in \mathbb{FP}_{10}[p]$ be non-zero floating-point numbers.*

(1) *If $x \approx_\delta x'$ then $10^{-\delta-1} \leq x/x' \leq 10^{\delta+1}$.*
(2) *If $10^{-\delta} \leq x/x' \leq 10^{\delta}$ then $x \approx_{\delta+2} x'$.*
(3) *If $x \approx_\delta x'$ and $x' \approx_\eta x''$ then $x \approx_{\delta+\eta+4} x''$.*

Lemma 3. *Let S_j be a top strongly connected component of (M, x), and let C be as given by Lemma 2.*

There exists $\beta \in \mathbb{N}$ such that for all $(q, i), (q', i) \in S_j$ and every $t \geq C$ then

- *if $t \not\equiv i \mod P$, then $x^{(t)}_{(q,i)} = 0$,*
- *otherwise, $x^{(t)}_{(q,i)} \approx_\beta x^{(t)}_{(q',i)}$.*

Proof. Let $t \in \mathbb{N}$. If $t \not\equiv i \mod P$ then $x^{(t)}_{(q,i)} = 0$ for all $(q,i) \in S_{j,i}$ by construction.

Otherwise, let $m \geq \max_{q,q' \in Q : M_{q,q'} \neq 0} \max\left(M_{q,q'}, (M_{q,q'})^{-1}\right)$ be a constant larger than all values occurring in M and so that $\frac{1}{m}$ is smaller than all non-zero values appearing in M. Let c be the constant from the log bounded property of the rounding function $[\cdot]$ and d be the dimension of M.

Observe that for all $t \in \mathbb{N}$ with $t = i \mod P$ we have

$$\begin{aligned} x^{(t)}_{(q,i)} &= \left[\sum_{(q',i-1)} M_{(q,i),(q',i-1)} x^{(t-1)}_{(q',i-1)} \right] \\ &\geq \frac{1}{c} \sum_{(q',i-1)} M_{(q,i),(q',i-1)} x^{(t-1)}_{(q',i-1)} && \text{(by log bounded)} \\ &\geq \frac{1}{cm} \max_{(q',i-1) \text{ s.t. } M_{(q,i),(q',i-1)} > 0} x^{(t-1)}_{(q',i-1)} && \text{(by defn of } m\text{)} \end{aligned}$$

In particular

$$x^{(t)}_{(q,i)} \geq \frac{1}{cm} x^{(t-1)}_{(q',i-1)} \text{ for all } (q', i-1) \text{ s.t. } M_{(q,i),(q',i-1)} > 0$$

Using induction we obtain:

$$x^{(t+k)}_{(q,i+k)} \geq \frac{1}{(cm)^{k-1}} x^{(t+1)}_{(q',i+1)} \geq \frac{1}{(cm)^k} x^{(t)}_{(q'',i)}$$

for all $(q', i+1), (q'', i)$ such that $M^{k-1}_{(q,i+k),(q',i+1)} > 0$ and $M_{(q',i+1),(q'',i)} > 0$.

In particular, we have $x^{(t+C)}_{(q,i)} \geq \frac{1}{(cm)^C} x^{(t)}_{(q',i)}$ for all q' (since $M^C_{(q,i),(q',i)} > 0$ for all q' by the previous lemma).

On the other hand we have

$$x^{(t+1)}_{(q,i+1)} = \left[\sum_{q' : M_{(q,i+1),(q',i)} > 0} M_{(q,i+1),(q',i)} x^{(t)}_{(q',i)} \right] \leq mcd \max_{(q',i) \in S_j} x^{(t)}_{(q',i)}.$$

By induction we get that $x^{(t+C)}_{(q,i)} \leq (mcd)^C \max_{(q',i) \in S_j} x^{(t)}_{(q',i)}$. Hence, for all $q, q' \in S_j$ we have

$$\frac{1}{(mc)^C} \max_{(q'',i) \in S_j} x^{(t)}_{(q'',i)} \leq x^{(t+C)}_{(q',i)} \quad \text{and} \quad x^{(t+C)}_{(q,i)} \leq (mcd)^C \max_{(q'',i) \in S_j} x^{(t)}_{(q'',i)}.$$

Hence $\frac{x^{(t+C)}_{(q,i)}}{x^{(t+C)}_{(q',i)}} \leq d^C (mc)^{2C}$.

Setting $\gamma = \left\lceil \log_{10} d^C (mc)^{2C} \right\rceil$, we thus have that $10^{-\gamma} x^{(t+C)}_{(q',i)} \leq x^{(t+C)}_{(q,i)} \leq 10^{\gamma} x^{(t+C)}_{(q',i)}$ for all $(q,i), (q',i) \in S_{j,i}$ and $t \in \mathbb{N}$. Then $x^{(t)}_{(q',i)}$ and $x^{(t)}_{(q,i)}$ are $\beta = \gamma + 2$ close by Proposition 1. ◻

Lemma 4. *Let S_j be a top strongly connected component of (M, x). Then the sequence $(x^{(t)}_{S_j})_{t\in\mathbb{N}}$ is effectively pseudo-periodic.*

Proof. Let β and C be as in Lemma 3. Denote $q_1, \dots, q_m$ the states of S_j. We define the sequence $(y^{(t)})_{t\geq C}$ such that for all $t \geq C$ and $q \in S_j$ denoting $(p^{(t)})_q = \mathsf{mantissa}([x^{(t)}_q])$ and $(\alpha^{(t)})_q = \mathsf{exponent}([x^{(t)}_q])$ we have that $y^{(t)} = (p_{q_1}, 0, p_{q_2}, \alpha_{q_2} - \alpha_{q_1}, \dots, p_{q_m}, \alpha_{q_m} - \alpha_{q_1})$. Note that this sequence can only take finitely many values as the mantissas have a precision of p decimals and by Lemma 3, for all $k \leq m$, $\alpha_{q_k} - \alpha_{q_1} \in \{-\beta, \dots, \beta\}$. As a consequence, the sequence $(y^{(t)})_{t\geq C}$ takes the same value multiple times. Let k_1 and k_2 be the two distinct minimal integers such that $y^{(k_1)} = y^{(k_2)}$. Setting $\alpha = \alpha^{(k_2)}_{q_1} - \alpha^{(k_1)}_{q_1}$ We have that $x^{(k_1)} = x^{(k_2)} \cdot 10^{\alpha}$. Since $[\cdot]$ is mantissa-based, one can show by induction that for all $t \geq 0$, $x^{(k_1+t)} = x^{(k_2+t)} \cdot 10^{\alpha}$. Therefore the sequence $(x^{(t)}_{S_j})_{t\in\mathbb{N}}$ is effectively pseudo-periodic with period $T = k_2 - k_1$ and starting point $N = C + k_1$.

Moreover, as the maximum number of different values taken by $(y^{(t)})_{t\geq C}$ is known, we can deduce that both k_1 and $k_2 - k_1$ are smaller than $10^{pm}(2\beta+1)^m+1$. □

Note that the increasing rate is the same for every state of the strongly connected component.

4.3 Pseudo-periodicity within lower SCCs

We consider a strongly connected component S_{me}, which is fed by at least one strongly connected components $F_1, \dots, F_\ell$, $\ell \geq 1$. We let $S_F = F_1 \cup \dots \cup F_\ell$ and assume every F_i is pseudo-periodic.

In this section we show

Theorem 4. *S_{me} is effectively pseudo-periodic and the growth rate of S_{me} is the same for all $q \in S_{me}$.*

We first observe that the difference between values in S_{me} is bounded. This is achieved with a proof similar to the one of Lemma 2 and Lemma 3 (though having to combine considerations of S_{me} and S_F).

Lemma 5. *There exists $\eta, N' \in \mathbb{N}$, such that for all $(q, i), (q', i) \in S_{me}$, all $t \geq N'$ and all $i \in \{0, \dots, P-1\}$ then*

- *if $t \not\equiv i \mod P$, then $x^{(t)}_{(q,i)} = 0$,*
- *otherwise, $x^{(t)}_{(q,i)} \approx_\eta x^{(t)}_{(q',i)}$.*

Definition 6. *We say that $x^{(t)}_q$ is influenced by S_F if*

$$x^{(t)}_q = \left[\sum_{q'\in S_F} M_{q,q'} x^{(t-1)}_{q'} + \sum_{q'\in S_{me}} M_{q,q'} x^{(t-1)}_{q'}\right] \neq \left[\sum_{q'\in S_{me}} M_{q,q'} x^{(t-1)}_{q'}\right]$$

and in particular $x_q^{(t)}$ *is influenced by* $u \in S_F$ *if:*

$$\left[\sum_{q' \in S_F \cup S_{me}} M_{q,q'} x_{q'}^{(t-1)}\right] \neq \left[\sum_{q' \in S_F \cup S_{me} \setminus \{u\}} M_{q,q'} x_{q'}^{(t-1)}\right].$$

We can restrict S_F to the F_i in S_F with the maximum growth rate. Indeed, from some point on, any F_i with non-maximal growth rate is much smaller than the maximal ones, and as by the proof of Lemma 5 the values within S_{me} are close to (or greater than) the maximum value within S_F, this F_i would not influence with any $x_q^{(t)}$ with $q \in S_{me}$. Let N_1 be the point from which we can assume, that the elements of S_F are much larger than any other feeding SCCs and are thus the only ones potentially influencing of S_{me}.

Since each F_i is assumed to be pseudo-periodic, we have that S_F pseudo-periodic. Let T be the period of S_F, N_2 be the starting point and α be the growth rate of every state of S_F (meaning the exponent of every state changes by α every T starting form the N-th step.) Let $N = \max\{N_1, N_2\}$, that is, the point from which we can assume S_F is both pseudo-periodic and dominating non-maximal SCCs feeding S_{me}.

As a direct consequence of having the same growth rate, the non-zero terms within S_F are close:

Proposition 2. *If a sequence of non-zero floating-point vectors* $(v^{(t)})_{t \in \mathbb{N}}$ *is pseudo-periodic with the same growth rate within a set* Q*, then there exists* δ *such that for all* $q, q' \in Q$ *and all* $t \geq N$*,* $v_q^{(t)} \approx_\delta v_{q'}^{(t)}$*.*

Moreover, either S_F does not influence S_{me}, or they are close.

Lemma 6. *There exists* $\beta, N \in \mathbb{N}$ *such that:*
For $t \geq N$ *and* $(q,i) \in S_{me}$*, if* $x_{(q,i)}^{(t)}$ *is influenced by* $(q', i-1) \in S_F$*, then* $x_{(r,i)}^{(t)} \approx_\beta x_{(r',i)}^{(t)}$ *for all* $(r,i), (r',i) \in S_{me} \cup S_F$*.*

We will show Theorem 4 through the following observation:

Observation 1. Observe that S_F either influences S_{me} infinitely many times or finitely many times. We have two cases:

- If S_F influences S_{me} infinitely often, then they are infinitely often β-close by Lemma 6. Then we will observe through a simultaneous version of Lemma 4 that S_{me} is pseudo-periodic.
- If S_F influences S_{me} only finitely often, then clearly from some point on S_{me} behaves like a top SCC, and thus is pseudo-periodic directly by Lemma 4.

It will then remain to show that we can detect which of the two cases applies, and place a bound on the time to detect this, which will effectively reveal the constants of the pseudo-periodic behaviour.

We now present a version of Lemma 4 to observe that if S_F and S_{me} are infinitely often β-close then S_{me} is pseudo-periodic:

Lemma 7. *Suppose* $x^{(t)}_{S_F} \approx_\beta x^{(t)}_{S_{me}}$ *for infinitely many* t*. Then there exists* $t_1 < t_2$*, such that* $x^{(t_1)}_{S_F} \approx_\beta x^{(t_1)}_{S_{me}}$ *and* $x^{(t_2)}_{S_F} \approx_\beta x^{(t_2)}_{S_{me}}$*,* $x^{(t_2)}_{S_F} = 10^\gamma x^{(t_1)}_{S_F}$ *and* $x^{(t_2)}_{S_{me}} = 10^\gamma x^{(t_1)}_{S_{me}}$*. In particular, the sequence* $(x^{(t)}_{S_{me}})_{t\in\mathbb{N}}$ *is pseudo-periodic with period* $(t_2 - t_1)$*, starting from* t_1 *with growth rate of* γ *in every state.*

Proof. At a time t such that $x^{(t)}_{S_F} \approx_\beta x^{(t)}_{S_{me}}$, we denote the vectors $x^{(t)}_{S_F} \in \mathbb{FP}_{10}[p]^{|S_F|}$ and $x^{(t)}_{S_{me}} \in \mathbb{FP}_{10}[p]^{|S_{me}|}$ respectively

$$(m^{(t)}_1 10^{\gamma^{(t)}_1}, m^{(t)}_2 10^{\gamma^{(t)}+\alpha^{(t)}_2}, \ldots, m^{(t)}_{|S_F|} 10^{\gamma^{(t)}+\alpha^{(t)}_{|S_F|}}) \text{ and}$$
$$(n^{(t)}_1 10^{\gamma^{(t)}+\zeta^{(t)}_1}, \ldots, n^{(t)}_{|S_{me}|} 10^{\gamma^{(t)}+\zeta^{(t)}_{|S_{me}|}}),$$

where m_i, n_i are taken from the finite set of mantissa values expressible in p bits, $\gamma^{(t)} \in \mathbb{Z}$ and $\alpha_i, \zeta_i \in \mathbb{Z} \cap [-\beta, \beta]$ denote the offset from $\gamma^{(t)}$.

Let F bound the number of possible values $m_i, n_i, \alpha_i, \zeta_i$ can take on, where $F \leq 10^{p(|S_F|+|S_{me}|)} \cdot (2\beta+1)^{|S_F|+|S_{me}|-1}$. By the pigeonhole principle, after at most $F+1$ times in which $x^{(t)}_{S_F} \approx_\beta x^{(t)}_{S_{me}}$ there must exist two times $t_1 < t_2$ where the values of $m_i, n_i, \alpha_i, \beta_i$'s are equal (although the value of γ could be different), thus $x^{(t_2)}_{S_F \cup S_{me}} = \frac{10^{\gamma^{(t_2)}}}{10^{\gamma^{(t_1)}}} x^{(t_1)}_{S_F \cup S_{me}}$.

Since the rounding function is mantissa-based, the system evolution from $x^{(t_1)}$ is equivalent to the systems evolution from $x^{(t_2)} = 10^\gamma x^{(t_1)}$, where γ is the growth rate, $\gamma^{(t_2)} - \gamma^{(t_1)}$. □

We can in fact decide whether $x^{(t)}_{S_F} \approx_\beta x^{(t)}_{S_{me}}$ for the last time:

Lemma 8. *Let* β, N *be defined as in Lemma 6. If* $t \geq N$ *then it is decidable whether there exists* $t' > t$ *such that* $x^{(t')}_{S_F} \approx_\beta x^{(t')}_{S_{me}}$*.*

Proof Sketch (Full proof available in [19]). If we considered S_{me} in isolation, without the effect of S_F, we know it would be pseudo-periodic. We can simulate one period of S_{me} with and without the effect of S_F and determine if S_F influences S_{me} within one period. If it does then they must be close at this point. If S_F does not influence S_{me} we know that S_{me} will behave pseudo-periodically at least until S_F is close to S_{me} again; having established a growth rate for S_{me}, we can compare the growth rates of S_F and S_{me} to see if S_{me} will ever be close to S_F again in the future. □

Finally to conclude the proof of Theorem 4, we refine Observation 1 to show that the period is bounded and thus the growth rates are computable:

- either S_F is β-close to S_{me} infinitely often, in particular if they become close $F+1$ times then by Lemma 7 it is pseudo-periodic.
- or the system is pseudo-periodic because it behaves like a top-SCC, in which Lemma 4 gives effective computation of the constants.

Which of these occurs is determined by at most $F+1$ applications of Lemma 8.

5 Decidability of model checking

In this section we use the results obtained in the previous section to show that model checking is decidable. We use pseudo-periodicity to show that the characteristic word is eventually periodic, a case for which model checking is decidable.

Theorem 2. *Let (M, x) be a non-negative linear dynamical system, let $Y_1, \ldots, Y_k$ be semialgebraic targets and let ϕ be an MSO formula using predicates over $Y_1, \ldots, Y_k$. It is decidable whether the characteristic word under floating-point rounding satisfies ϕ.*

Consider a semialgebraic target Y, which can be expressed as a Boolean combination of polynomial inequalities over variables representing the dimensions. That is $Y = \{(x_1, \ldots, x_d) \mid \bigwedge_i \bigvee_j P_{ij}(x_1, \ldots, x_n) \triangleright_{ij} 0\}$, where $\triangleright_{ij} \in \{\geq, >, =\}$.

Given a linear dynamical system (M, x) defining the rounded orbit $(x^{(n)})_{n=1}^{\infty}$, recall that $\mathcal{Z}(Y) = \{n \mid x^{(n)} \in Y\}$ are the hitting times of Y. We claim that this set is semi-linear (equivalently eventually periodic) for semialgebraic Y.

Definition 7. *A 1-dimensional linear-set, defined by a base $b \in \mathbb{N}$ and period $p \in \mathbb{N}$, is the set $\{x \mid \exists k \in \mathbb{N} : x = b + k \cdot p\}$. A semi-linear set is the finite union of a finite set $F \subseteq \mathbb{N}$ and linear sets. It can be assumed that each linear-set has the same period. Hence a 1-dimensional semi-linear set X is defined by a finite set $F \subseteq \mathbb{N}$ and integers $m, p, b_1, \ldots, b_m \in \mathbb{N}$ such that $x \in X$ if and only if $x \in F$ or $x = b + k \cdot p$ for some $k \in \mathbb{N}$ and $b \in \{b_1, \ldots, b_m\}$.*

Theorem 5. *Let Y be a semialgebraic target, $\mathcal{Z}(Y)$ is a semi-linear set.*

Theorem 5 essentially completes the proof of Theorem 2. It is almost immediate that the characteristic word is eventually periodic (see the long version [19] for a formal proof) and thus the model-checking problem can be decided by checking $A \cap \overline{B} = \emptyset$, where A is an automaton representing the characteristic word and B encodes the language of ϕ.

It is standard that semi-linear sets are closed under intersection, union, and complementation (see [15] for a nice introduction to semi-linear sets). Thus in order to express the hitting times of $\mathcal{Z}(Y)$ it is sufficient to express the hitting times of $\{(x_1, \ldots, x_d) \mid P(x_1, \ldots, x_n) \geq 0\}$ for a finitely many polynomials P. Conjunction is found by taking the intersection of the hitting times, and disjunction by taking union. The hitting times of $P(x_1, \ldots, x_n) > 0$ can be rewritten as the complement of the hitting times of $-P(x_1, \ldots, x_n) \geq 0$. The hitting times of $P(x_1, \ldots, x_n) = 0$ is the conjunction (intersection) of $P(x_1, \ldots, x_n) \geq 0$ and $-P(x_1, \ldots, x_n) \geq 0$. Thus Theorem 5 is a consequence of the following lemma.

Lemma 9. *Assume $x^{(t)} = (z_1^{(t)}, \ldots, z_d^{(t)})_{i=1}^{\infty}$, is a pseudo-periodic sequence with start point N, period T and growth rates $\alpha_1, \ldots, \alpha_n$ and $P \in \mathbb{Q}[x_1, \cdots, x_d]$ a rational polynomial in d variables.*[9] *Then, $\{i \in \mathbb{N} \mid P(z_1^{(t)}, \cdots, z_d^{(t)}) \geq 0\}$ is a semi-linear set.*

[9] Some variables may be redundant, that is, if the polynomial does not depend on all dimensions of $x^{(t)}$ then some of the variables may not appear in P.

Proof. First, we show that pseudo-periodicity is closed under product. Suppose $x_i^{(N+Tn)} = m_i 10^{\beta_i+\alpha_i \cdot n}$ and $x_j^{(N+Tn)} = m_j 10^{\beta_j+\alpha_j \cdot n}$. Observe that $x_i^{(N+Tn)} \cdot x_j^{(N+Tn)} = m_i \cdot 10^{\beta_i+\alpha_i n} m_j \cdot 10^{\beta_j+\alpha_j n} = m_i m_j \cdot 10^{\beta_i+\beta_j+n(\alpha_i+\alpha_j)}$. We conclude that the vector $(x_i \cdot x_j)^{(t)}$ is pseudo-periodic with growth rate $\alpha_i + \alpha_j$. Observe that the mantissa precision increase by at most 2.

Secondly, we show that if two pseudo-periodic sequences have the same growth rate, then their sum is also pseudo-periodic with the same growth rate. Suppose $x_i^{(N+Tn)} = m_i 10^{\beta_i+\alpha \cdot n}$, and $x_j^{(N+Tn)} = m_j 10^{\beta_j+\alpha \cdot n}$. Observe that $(x_i + x_j)^{(N+Tn)} = m_i 10^{\beta_i+\alpha \cdot n} + m_j 10^{\beta_j+\alpha \cdot n} = (m_i + m_j \cdot 10^{\beta_j-\beta_i}) 10^{\beta_i+\alpha \cdot n}$. Observe that the mantissa precision increased by at most $10^{|\beta_j-\beta_i|}$.

Let $P(x_1, \ldots, x_n) = \sum_{i=1}^{N} c_i Z_i$, where Z_i is a product of $x_1, \ldots, x_n$. Consider each monomial Z_i occurring in P, since produce preserves pseudo-periodicity, we conclude that Z_i is pseudo-periodic. $P^{(t)}$ is thus a linear combination of these pseudo-periodic vectors. Note our prior observation does not immediately imply that $P^{(t)}$ is pseudo-periodic as we required taking the sum of elements with the same growth rate. However, from some point on, we are only interested in those with the maximal growth rate.

Without loss of generality, let $Z_1, \ldots, Z_r$ have the maximum-growth rate, and $Z_{r+1}, \ldots, Z_N$ have strictly smaller growth rate. For every $L \in \mathbb{N}$ there exists $N \in \mathbb{N}$ such that for all $t > N$, $\mathsf{exponent}(Z_1^{(t)}) - \mathsf{exponent}(Z_{r+1}^{(t)}) > L$.

Hence there exists $N \in \mathbb{N}$ such that for all $t > N$ if $\sum_{i=1}^{r} c_i Z_i > 0$ if and only if $\sum_{i=1}^{N} c_i Z_i = \sum_{i=1}^{r} c_i Z_i + \sum_{i=r+1}^{N} c_i Z_i > 0$ because $\left|\sum_{i=r+1}^{N} c_i Z_i\right| < |\sum_{i=1}^{r} c_i Z_i|$ from some point on. Hence $\mathrm{sign}(\sum_{i=1}^{N} c_i Z_i^{(t)}) = \mathrm{sign}(\sum_{i=1}^{r} c_i Z_i^{(t)})$.

Thus we restrict our attention to $\sum_{i=1}^{r} c_i Z_i^{(t)}$. Since each of the Z_i for $i \in \{1, \ldots, r\}$ have the same growth rate, we know that $\sum_{i=1}^{r} c_i Z_i^{(t)}$ is pseudo-periodic. Since $\mathrm{sign}(\sum_{i=1}^{r} c_i Z_i^{(t)})$ does not depend on the exponent, only the periodic mantissa, we have that the sign is periodic. The hitting times for $t \leq N$ can be determined exhaustively and included in the finite set of the semi-linear set. □

Acknowledgements Partially funded by DFG grant 389792660 as part of TRR 248 – CPEC, see perspicuous-computing.science. Joël Ouaknine is also affiliated with Keble College, Oxford as emmy.network Fellow. David Purser was partially supported by the ERC grant INFSYS, agreement no. 950398.

References

1. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive verification of floating-point java programs in key. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Part of ETAPS 2021. Part II. Lecture Notes in Computer Science, vol. 12652, pp. 242–261. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_13

2. Akshay, S., Antonopoulos, T., Ouaknine, J., Worrell, J.: Reachability problems for Markov chains. Inf. Process. Lett. **115**(2), 155–158 (2015). https://doi.org/10.1016/j.ipl.2014.08.013
3. Akshay, S., Bazille, H., Genest, B., Vahanwala, M.: On robustness for the Skolem and Positivity problems. In: Berenbrink, P., Monmege, B. (eds.) 39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022. LIPIcs, vol. 219, pp. 5:1–5:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.STACS.2022.5
4. Almagor, S., Karimov, T., Kelmendi, E., Ouaknine, J., Worrell, J.: Deciding ω-regular properties on linear recurrence sequences. Proc. ACM Program. Lang. **5**(POPL), 1–24 (2021). https://doi.org/10.1145/3434329
5. Baier, C., Funke, F., Jantsch, S., Karimov, T., Lefaucheux, E., Ouaknine, J., Pouly, A., Purser, D., Whiteland, M.A.: Reachability in dynamical systems with rounding. In: 40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020. LIPIcs, vol. 182, pp. 36:1–36:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.FSTTCS.2020.36
6. Baier, C., Funke, F., Jantsch, S., Karimov, T., Lefaucheux, E., Ouaknine, J., Purser, D., Whiteland, M.A., Worrell, J.: Parameter Synthesis for Parametric Probabilistic Dynamical Systems and Prefix-Independent Specifications. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory (CONCUR 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 243, pp. 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.10
7. Becker, H., Panchekha, P., Darulova, E., Tatlock, Z.: Combining tools for optimization and analysis of floating-point computations. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E.P. (eds.) Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018. Lecture Notes in Computer Science, vol. 10951, pp. 355–363. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_21
8. Bilu, Y., Luca, F., Nieuwveld, J., Ouaknine, J., Purser, D., Worrell, J.: Skolem meets Schanuel. In: Szeider, S., Ganian, R., Silva, A. (eds.) 47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022. LIPIcs, vol. 241, pp. 20:1–20:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.MFCS.2022.20
9. Boyle, M.: Notes on the Perron-Frobenius theory of nonnegative matrices (2005)
10. Braverman, M.: Termination of integer linear programs. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference, CAV 2006 Proceedings. Lecture Notes in Computer Science, vol. 4144, pp. 372–385. Springer (2006). https://doi.org/10.1007/11817963_34
11. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: The collected works of J. Richard Büchi, pp. 425–435. Springer (1990)
12. Chonev, V., Ouaknine, J., Worrell, J.: On the complexity of the orbit problem. J. ACM **63**(3), 23:1–23:18 (2016). https://doi.org/10.1145/2857050
13. D'Costa, J., Karimov, T., Majumdar, R., Ouaknine, J., Salamati, M., Soudjani, S., Worrell, J.: The pseudo-Skolem problem is decidable. In: Bonchi, F., Puglisi, S.J. (eds.) 46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021. LIPIcs, vol. 202, pp. 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.MFCS.2021.34

14. D'Costa, J., Karimov, T., Majumdar, R., Ouaknine, J., Salamati, M., Worrell, J.: The pseudo-reachability problem for diagonalisable linear dynamical systems. In: Szeider, S., Ganian, R., Silva, A. (eds.) 47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022. LIPIcs, vol. 241, pp. 40:1–40:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.MFCS.2022.40
15. Haase, C.: A survival guide to Presburger arithmetic. ACM SIGLOG News **5**(3), 67–82 (2018). https://doi.org/10.1145/3242953.3242964
16. Kannan, R., Lipton, R.J.: Polynomial-time algorithm for the orbit problem. J. ACM **33**(4), 808–821 (1986). https://doi.org/10.1145/6490.6496
17. Karimov, T., Kelmendi, E., Ouaknine, J., Worrell, J.: What's decidable about discrete linear dynamical systems? In: Raskin, J., Chatterjee, K., Doyen, L., Majumdar, R. (eds.) Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 13660, pp. 21–38. Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_2
18. Karimov, T., Lefaucheux, E., Ouaknine, J., Purser, D., Varonka, A., Whiteland, M.A., Worrell, J.: What's decidable about linear loops? Proc. ACM Program. Lang. **6**(POPL), 1–25 (2022). https://doi.org/10.1145/3498727
19. Lefaucheux, E., Ouaknine, J., Purser, D., Sharifi, M.: Model checking linear dynamical systems under floating-point rounding. CoRR **abs/2211.04301** (2022). https://doi.org/10.48550/arXiv.2211.04301
20. Lohar, D., Jeangoudoux, C., Sobel, J., Darulova, E., Christakis, M.: A two-phase approach for conditional floating-point verification. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Part of ETAPS 2021. Part II. Lecture Notes in Computer Science, vol. 12652, pp. 43–63. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_3
21. Luca, F., Ouaknine, J., Worrell, J.: Algebraic model checking for discrete linear dynamical systems. In: Bogomolov, S., Parker, D. (eds.) Formal Modeling and Analysis of Timed Systems - 20th International Conference, FORMATS 2022. Lecture Notes in Computer Science, vol. 13465, pp. 3–15. Springer (2022). https://doi.org/10.1007/978-3-031-15839-1_1
22. Maurica, F., Mesnard, F., Payet, E.: Optimal approximation for efficient termination analysis of floating-point loops. In: 2017 1st International Conference on Next Generation Computing Applications (NextComp). pp. 17–22. IEEE (2017)
23. Minsky, M.L.: Computation. Prentice-Hall Englewood Cliffs (1967)
24. Ouaknine, J., Worrell, J.: Positivity problems for low-order linear recurrence sequences. In: Chekuri, C. (ed.) Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014. pp. 366–379. SIAM (2014). https://doi.org/10.1137/1.9781611973402.27
25. Schneider, H.: Wielandt's proof of the exponent inequality for primitive nonnegative matrices. Linear Algebra and its Applications **353**(1), 5–10 (2002)
26. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification, 16th International Conference, CAV 2004 Proceedings. Lecture Notes in Computer Science, vol. 3114, pp. 70–82. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_6
27. Xia, B., Yang, L., Zhan, N., Zhang, Z.: Symbolic decision procedure for termination of linear programs. Formal Aspects Comput. **23**(2), 171–190 (2011). https://doi.org/10.1007/s00165-009-0144-5

Efficient Loop Conditions for Bounded Model Checking Hyperproperties*

Tzu-Han Hsu[1], César Sánchez[2], Sarai Sheinvald[3], and Borzoo Bonakdarpour[1(✉)]

[1] Michigan State University, East Lansing, MI, USA {tzuhan,borzoo}@msu.edu
[2] IMDEA Software Institute, Madrid, Spain cesar.sanchez@imdea.org
[3] Dept. of Software Engineering, Braude College, Karmiel, Israel sarai@braude.ac.il

Abstract. Bounded model checking (BMC) is an effective technique for hunting bugs by incrementally exploring the state space of a system. To reason about infinite traces through a finite structure and to ultimately obtain completeness, BMC incorporates *loop conditions* that revisit previously observed states. This paper focuses on developing loop conditions for BMC of HyperLTL– a temporal logic for hyperproperties that allows expressing important policies for security and consistency in concurrent systems, etc. Loop conditions for HyperLTL are more complicated than for LTL, as different traces may loop inconsistently in unrelated moments. Existing BMC approaches for HyperLTL only considered linear unrollings without any looping capability, which precludes both finding small infinite traces and obtaining a complete technique. We investigate loop conditions for HyperLTL BMC, for HyperLTL formulas that contain up to one quantifier alternation. We first present a general complete automata-based technique which is based on bounds of maximum unrollings. Then, we introduce alternative simulation-based algorithms that allow exploiting short loops effectively, generating SAT queries whose satisfiability guarantees the outcome of the original model checking problem. We also report empirical evaluation of the prototype implementation of our BMC techniques using Z3py.

1 Introduction

Hyperproperties [13] have been getting increasing attention due to their power to reason about important specifications such as information-flow security policies that require reasoning about the interrelation among different execution traces. HyperLTL [12] is an extension of the linear-time temporal logic LTL [31] that allows quantification over traces; hence, capable of describing hyperproperties. For example, the security policy *observational determinism* can be specified as

* This research has been partially supported by the United States NSF SaTC Award 2100989, by the Madrid Regional Gov. Project BLOQUES-CM (S2018/TCS-4339), by Project PRODIGY (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the EU NextGenerationEU/PRTR, and by a research grant from Nomadic Labs and the Tezos Foundation.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 66–84, 2023.
https://doi.org/10.1007/978-3-031-30823-9_4

HyperLTL formula: $\forall\pi.\forall\pi'.(o_\pi \leftrightarrow o_{\pi'})\,\mathcal{W}\,\neg(i_\pi \leftrightarrow i_{\pi'})$, which specifies that for every pair of traces π and π', if they agree on the secret input i, then their public output o must also be observed the same (here '$\mathcal{W}$' denotes the weak until operator).

Several works [14,22] have studied model checking techniques for HyperLTL specifications, which typically reduce this problem to LTL model checking queries of modified systems. More recently, [27] proposed a QBF-based algorithm for the direct application of bounded model checking (BMC) [11] to HyperLTL, and successfully provided a push-button solution to verify or falsify HyperLTL formulas with an arbitrary number of quantifier alternations. However, unlike the classic BMC for LTL, which included the so-called *loop conditions*, the algorithm in [27] is limited to (non-looping) linear exploration of paths. The reason is that extending path exploration to include loops when dealing with multiple paths simultaneously is not straightforward. For example, consider the HyperLTL formula $\varphi_1 = \forall\pi.\exists\pi'.\ \square(a_\pi \rightarrow b_{\pi'})$ and two Kripke structures K_1 and K_2 as follows:

Assume trace π ranges over K_1 and trace π' ranges over K_2. Proving $\langle K_1, K_2\rangle \not\models \varphi_1$ can be achieved by finding a finite counterexample (i.e., path $s_1 s_2 s_3$ from K_1). Now, consider $\varphi_2 = \forall\pi.\exists\pi'.\ \square(a_\pi \leftrightarrow a_{\pi'})$. It is easy to see that $\langle K_1, K_2\rangle \models \varphi_2$. However, to prove $\langle K_1, K_2\rangle \models \varphi_2$, one has to show the absence of counterexamples in infinite paths, which is impossible with model unrolling in finite steps as proposed in [27].

In this paper, we propose efficient loop conditions for BMC of hyperproperties. First, using an automata-based method, we show that lasso-shaped traces are sufficient to prove infinite behaviors of traces within finite exploration. However, this technique requires an unrolling bound that renders it impractical. Instead, our efficient algorithms are based on the notion of *simulation* [32] between two systems. Simulation is an important tool in verification, as it is used for abstraction, and preserves ACTL* properties [6,24]. As opposed to more complex properties such as language containment, simulation is a more local property and is easier to check. The main contribution of this paper is the introduction of practical algorithms that achieve the exploration of infinite paths following a simulation-based approach that is capable of relating the states of multiple models with correct successor relations.

We present two different variants of simulation, $\mathsf{SIM}_{\mathsf{EA}}$ and $\mathsf{SIM}_{\mathsf{AE}}$, allowing to check the satisfaction of $\exists\forall$ and $\forall\exists$ hyperproperties, respectively. These notions circumvent the need to boundlessly unroll traces in both structures and synchronize them. For $\mathsf{SIM}_{\mathsf{AE}}$, in order to resolve non-determinism in the first model, we also present a third variant, where we enhance $\mathsf{SIM}_{\mathsf{AE}}$ by using *prophecy vari-*

ables [1,7]. Prophecy variables allow us to handle cases in which ∀∃ hyperproperties hold despite the lack of a direct simulation. With our simulation-based approach, one can capture infinite behaviors of traces with finite exploration in a simple and concise way. Furthermore, our BMC approach not only model-checks the systems for hyperproperties, but also does so in a way that finds *minimal* witnesses to the simulation (i.e., by partially exploring the existentially quantified model), which we will further demonstrate in our empirical evaluation.

We also design algorithms that generate SAT formulas for each variant (i.e., $\mathsf{SIM}_{\mathsf{EA}}$, $\mathsf{SIM}_{\mathsf{AE}}$, and $\mathsf{SIM}_{\mathsf{AE}}$ with prophecies), where the satisfiability of formulas implies the model checking outcome. We also investigate the practical cases of models with different sizes leading to the eight categories in Table 1. For example, the first row indicates the category of verifying two models of different sizes with the fragment that only allows ∀∃ quantifiers and □ (i.e., *globally* temporal operator); $\forall_{\texttt{small}}\exists_{\texttt{big}}$ means that the first model is relatively smaller than the second model, and the positive outcome ($\models \forall\exists\Box\varphi$) can be proved by our simulation-based technique $\mathsf{SIM}_{\mathsf{AE}}$, while the negative outcome ($\not\models \forall\exists\Box\varphi$) can be easily checked using non-looping unrolling (i.e., [27]). We will show that in certain cases, one can verify a □ formula without exploring the entire state space of the `big` model to achieve efficiency.

Case	φ with □	$\neg\varphi$ with ◇
$\forall_{\texttt{small}}\ \exists_{\texttt{big}}$	$\mathsf{SIM}_{\mathsf{AE}} \rightarrow \models \forall\exists\Box\varphi$	$\mathsf{BMC} \rightarrow \not\models \forall\exists\Box\varphi$
$\forall_{\texttt{big}}\ \exists_{\texttt{small}}$	$\mathsf{SIM}_{\mathsf{AE}} \rightarrow \models \forall\exists\Box\varphi$	$\mathsf{BMC} \rightarrow \not\models \forall\exists\Box\varphi$
$\exists_{\texttt{small}}\ \forall_{\texttt{big}}$	$\mathsf{SIM}_{\mathsf{EA}} \rightarrow \models \exists\forall\Box\varphi$	$\mathsf{BMC} \rightarrow \not\models \exists\forall\Box\varphi$
$\exists_{\texttt{big}}\ \forall_{\texttt{small}}$	$\mathsf{SIM}_{\mathsf{EA}} \rightarrow \models \exists\forall\Box\varphi$	$\mathsf{BMC} \rightarrow \not\models \exists\forall\Box\varphi$

Table 1: Eight categories of HyperLTL formulas with different forms of quantifiers, sizes of models, and different temporal operators.

We have implemented our algorithms[1] using Z3py, the Z3 [15] API in python. We demonstrate the efficiency of our algorithm exploring a subset of the state space for the larger (i.e., `big`) model. We evaluate the applicability and efficiency with cases including conformance checking for distributed protocol synthesis, model translation, and path planning problems. In summary, we make the following contributions: (1) a bounded model checking algorithm for hyperproperties with loop conditions, (2) three different practical algorithms: $\mathsf{SIM}_{\mathsf{EA}}$, $\mathsf{SIM}_{\mathsf{AE}}$, and $\mathsf{SIM}_{\mathsf{AE}}$ with prophecies, and (3) a demonstration of the efficiency and applicability by case studies that cover through all eight different categories of HyperLTL formulas (see Table 1).

Related Work. Hyperproperties were first introduced by Clarkson and Schneider [13]. HyperLTL was introduced as a temporal logic for hyperproperties in [12]. The first algorithms for model checking HyperLTL were introduced in [22] using alternating automata. Automated reasoning about HyperLTL specifications has received attention in many aspects, including static verification [14,20,21,22] and monitoring [2,8,10,18,19,26,33]. This includes tools support, such as MCHy-

[1] Available at: https://github.com/TART-MSU/loop_condition_tacas23

per [22,14] for model checking, EAHyper [17] and MGHyper [16] for satisfiability checking, and RVHyper [18] for runtime monitoring. However, the aforementioned tools are either limited to HyperLTL formulas without quantifier alternations, or requiring additional inputs from the user (e.g., manually added strategies [14]).

Recently, this difficulty of alternating formulas was tackled by the bounded model checker HyperQB [27] using QBF solving. However, HyperQB lacks loop conditions to capture early infinite traces in finite exploration. In this paper, we develop simulation-based algorithms to overcome this limitation. There are alternative approaches to reason about infinite traces, like reasoning about strategies to deal with $\forall\exists$ formulas [14], whose completeness can be obtained by generating a set of prophecy variables [7]. In this work, we capture infinite traces in BMC approach using simulation. We also build an applicable prototype for model-check HyperLTL formulas with models that contain loops.

2 Preliminaries

Kripke structures. A *Kripke structure* K is a tuple $\langle S, S^0, \delta, \mathsf{AP}, L\rangle$, where S is a set of *states*, $S^0 \subseteq S$ is a set of *initial states*, $\delta \subseteq S \times S$ is a total *transition relation*, and $L : S \to 2^{\mathsf{AP}}$ is a *labeling function*, which labels states $s \in S$ with a subset of atomic propositions in AP that hold in s. A *path* of K is an infinite sequence of states $s(0)s(1)\cdots \in S^\omega$, such that $s(0) \in S^0$, and $(s(i), s(i+1)) \in \delta$, for all $i \geq 0$. A *loop* in K is a finite path $s(n)s(n+1)\cdots s(\ell)$, for some $0 \leq n \leq \ell$, such that $(s(i), s(i+1)) \in \delta$, for all $n \leq i < \ell$, and $(s(\ell), s(n)) \in \delta$. Note that $n = \ell$ indicates a *self-loop* on a state. A *trace* of K is a trace $t(0)t(1)t(2)\cdots \in \Sigma^\omega$, such that there exists a path $s(0)s(1)\cdots \in S^\omega$ with $t(i) = L(s(i))$ for all $i \geq 0$. We denote by $\mathit{Traces}(K, s)$ the set of all traces of K with paths that start in state $s \in S$. We use $\mathit{Traces}(K)$ as a shorthand for $\bigcup_{s\in S^0} \mathit{Traces}(K, s)$, and $\mathcal{L}(K)$ as the shorthand for $\mathit{Traces}(K)$.

Simulation relations. Let $K_A = \langle S_A, S_A^0, \delta_A, \mathsf{AP}_A, L_A\rangle$ and $K_B = \langle S_B, S_B^0, \delta_B, \mathsf{AP}_B, L_B\rangle$ be two Kripke structures. A *simulation relation* R from K_A to K_B is a relation $R \subseteq S_A \times S_B$ that meets the following conditions:

1. For every $s_A \in S_A^0$ there exists $s_B \in S_A^0$ such that $(s_A, s_B) \in R$.
2. For every $(s_A, s_B) \in R$, it holds that $L_A(s_A) = L_B(s_B)$.
3. For every $(s_A, s_B) \in R$, for every $(s_A, s'_A) \in \delta_A$, there exists $(s_B, s'_B) \in \delta_B$ such that $(s'_A, s'_B) \in R$.

The Temporal Logic HyperLTL. HyperLTL [12] is an extension of the linear-time temporal logic (LTL) for hyperproperties. The syntax of HyperLTL formulas is defined inductively by the following grammar:

$$\varphi ::= \exists\pi.\varphi \mid \forall\pi.\varphi \mid \phi$$
$$\phi ::= \mathsf{true} \mid a_\pi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi\,\mathcal{U}\ \phi \mid \phi\,\mathcal{R}\ \phi \mid \bigcirc\phi$$

where $a \in \mathsf{AP}$ is an atomic proposition and π is a *trace variable* from an infinite supply of variables $\mathcal{V}$. The Boolean connectives $\neg$, $\vee$, and $\wedge$ have the usual meaning, $\mathcal{U}$ is the temporal *until* operator, $\mathcal{R}$ is the temporal *release* operator,

and $\bigcirc$ is the temporal *next* operator. We also consider other derived Boolean connectives, such as $\rightarrow$ and $\leftrightarrow$, and the derived temporal operators *eventually* $\Diamond\varphi \equiv \mathsf{true}\ \mathcal{U}\ \varphi$ and *globally* $\Box\varphi \equiv \neg\Diamond\neg\varphi$. A formula is *closed* (i.e., a *sentence*) if all trace variables used in the formula are quantified. We assume, without loss of generality, that no trace variable is quantified twice. We use $\mathit{Vars}(\varphi)$ for the set of trace variables used in formula φ.

Semantics. An interpretation $\mathcal{T} = \langle T_\pi \rangle_{\pi \in \mathit{Vars}(\varphi)}$ of a formula φ consists of a tuple of sets of traces, with one set T_π per trace variable π in $\mathit{Vars}(\varphi)$, denoting the set of traces that π ranges over. Note that we allow quantifiers to range over different models, called the *multi-model semantics* [23,27][2]. That is, each set of traces comes from a Kripke structure and we use $\mathcal{K} = \langle K_\pi \rangle_{\pi \in \mathit{Vars}(\varphi)}$ to denote a *family* of Kripke structures, so $T_\pi = \mathit{Traces}(K_\pi)$ is the traces that π can range over, which comes from $K_\pi \in \mathcal{K}$. Abusing notation, we write $\mathcal{T} = \mathit{Traces}(\mathcal{K})$.

The semantics of HyperLTL is defined with respect to a trace assignment, which is a partial map $\Pi \colon \mathit{Vars}(\varphi) \rightharpoonup \Sigma^\omega$. The assignment with the empty domain is denoted by $\Pi_\emptyset$. Given a trace assignment Π, a trace variable π, and a concrete trace $t \in \Sigma^\omega$, we denote by $\Pi[\pi \rightarrow t]$ the assignment that coincides with Π everywhere but at π, which is mapped to trace t. The satisfaction of a HyperLTL formula φ is a binary relation $\models$ that associates a formula to the models $(\mathcal{T}, \Pi, i)$ where $i \in \mathbb{Z}_{\geq 0}$ is a pointer that indicates the current evaluating position. The semantics is defined as follows:

$$
\begin{array}{lll}
(\mathcal{T},\Pi,0) \models \exists \pi.\ \psi & \text{iff} & \text{there is a } t \in T_\pi, \text{ such that } (\mathcal{T}, \Pi[\pi \rightarrow t], 0) \models \psi, \\
(\mathcal{T},\Pi,0) \models \forall \pi.\ \psi & \text{iff} & \text{for all } t \in T_\pi, \text{ such that } (\mathcal{T}, \Pi[\pi \rightarrow t], 0) \models \psi, \\
(\mathcal{T},\Pi,i) \models \mathsf{true} & & \\
(\mathcal{T},\Pi,i) \models a_\pi & \text{iff} & a \in \Pi(\pi)(i), \\
(\mathcal{T},\Pi,i) \models \neg\psi & \text{iff} & (\mathcal{T},\Pi,i) \not\models \psi (\mathcal{T},\Pi,i) \\
(\mathcal{T},\Pi,i) \models \psi_1 \vee \psi_2 & \text{iff} & (\mathcal{T},\Pi,i) \models \psi_1 \text{ or } (\mathcal{T},\Pi,i) \models \psi_2, \\
(\mathcal{T},\Pi,i) \models \psi_1 \wedge \psi_2 & \text{iff} & (\mathcal{T},\Pi,i) \models \psi_1 \text{ and } (\mathcal{T},\Pi,i) \models \psi_2, \\
(\mathcal{T},\Pi,i) \models \bigcirc\psi & \text{iff} & (\mathcal{T},\Pi,i+1) \models \psi, \\
(\mathcal{T},\Pi,i) \models \psi_1\, \mathcal{U}\, \psi_2 & \text{iff} & \text{there is a } j \geq i \text{ for which } (\mathcal{T},\Pi,j) \models \psi_2 \text{ and} \\
 & & \quad \text{for all } k \in [i,j), (\mathcal{T},\Pi,k) \models \psi_1, \\
(\mathcal{T},\Pi,i) \models \psi_1\, \mathcal{R}\, \psi_2 & \text{iff} & \text{either for all } j \geq i,\ (\mathcal{T},\Pi,j) \models \psi_2, \text{ or,} \\
 & & \quad \text{for some } j \geq i, (\mathcal{T},\Pi,j) \models \psi_1 \text{ and} \\
 & & \quad \text{for all } k \in [i,j] : (\mathcal{T},\Pi,k) \models \psi_2.
\end{array}
$$

We say that an interpretation $\mathcal{T}$ satisfies a sentence φ, denoted by $\mathcal{T} \models \varphi$, if $(\mathcal{T}, \Pi_\emptyset, 0) \models \varphi$. We say that a family of Kripke structures $\mathcal{K}$ satisfies a sentence φ, denoted by $\mathcal{K} \models \varphi$, if $\langle \mathit{Traces}(K_\pi) \rangle_{\pi \in \mathit{Vars}(\varphi)} \models \varphi$. When the same Kripke structure K is used for all path variables we write $K \models \varphi$.

Definition 1. A *nondeterministic Büchi automaton* (NBW) is a tuple $A = \langle \Sigma, Q, Q_0, \delta, F \rangle$, where Σ is an *alphabet*, Q is a nonempty finite set of

[2] In terms of the model checking problem, multi-model and (the conventional) single-model semantics where all paths are assigned traces from the same Kripke structure [12] are equivalent (see [23,27]).

states, $Q_0 \subseteq Q$ is a set of *initial states*, $F \subseteq Q$ is a set of *accepting states*, and $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*.

Given an infinite word $w = \sigma_1\sigma_2\cdots$ over Σ, a *run of* A *on* w is an infinite sequence of states $r = (q_0, q_1, \ldots)$, such that $q_0 \in Q_0$, and $(q_{i-1}, \sigma_i, q_i) \in \delta$ for every $i > 0$. The run is *accepting* if r visits some state in F infinitely often. We say that A *accepts* w if there exists an accepting run of A on w. The *language* of A, denoted $\mathcal{L}(A)$, is the set of all infinite words accepted by A. An NBW A is called a *safety* NBW if all of its states are accepting. Every safety LTL formula ψ can be translated into a safety NBW over 2^{AP} such that $\mathcal{L}(A)$ is the set of all traces over AP that satisfy ψ [29].

3 Adaptation of BMC to HyperLTL on Infinite Traces

There are two main obstacles in extending the BMC approach of [27] to handle infinite traces. First, a trace may have an irregular behavior. Second, even traces whose behavior is regular, that is, lasso shaped, are hard to synchronize, since the length of their respective prefixes and lassos need not to be equal. For the latter issue, synchronizing two traces whose prefixes and lassos are of lengths p_1, p_2 and l_1, l_2, respectively, is equivalent to coordinating the same two traces, when defining both their prefixes to be of length $\max\{p_1, p_2\}$, and their lassos to be of length $\mathrm{lcm}\{l_1, l_2\}$, where 'lcm' stands for 'least common multiple'. As for the former challenge, we show that restricting the exploration of traces in the models to only consider lasso traces is sound. That is, considering only lasso-shaped traces is equivalent to considering the entire trace set of the models.

Let $K = \langle S, S^0, \delta, \mathsf{AP}, L\rangle$ be a Kripke structure. A *lasso path* of K is a path $s(0)s(1)\ldots s(\ell)$ such that $(s(\ell), s(n)) \in \delta$ for some $0 \leq n < \ell$. This path induces a *lasso trace* (i.e., a *lasso*) $L(s_0)\ldots L(s_{n-1})\,(L(s_n)\ldots L(s_\ell))^\omega$. Let $\langle K_1, \ldots, K_k\rangle$ be a multi-model, we denote the set of lasso traces of K_i by C_i for all $1 \leq i \leq k$, and we use $\mathcal{L}(C_i)$ as the shorthand for the set of lasso traces of K_i.

Theorem 1. *Let* $\mathcal{K} = \langle K_1, \ldots, K_k\rangle$ *be a multi-model, and let* $\varphi = \mathbb{Q}_1\pi_1.\cdots\mathbb{Q}_k\,\pi_k.\psi$ *be a* HyperLTL *formula, both over* AP*, then* $\mathcal{K} \models \varphi$ *iff* $\langle C_1, \ldots, C_k\rangle \models \varphi$.

Proof. (sketch) For an LTL formula ψ over $\mathsf{AP} \times \{\pi_i\}_{i=1}^k$, we denote the translation of ψ to an NBW over $2^{\mathsf{AP}\times\{\pi_i\}_{i=1}^k}$ by A_ψ [34]. Given $\alpha = \mathbb{Q}_1\pi_1\cdots\mathbb{Q}_k\pi_k$, where $\mathbb{Q}_i \in \{\exists, \forall\}$, we define the satisfaction of A_ψ by $\mathcal{K}$ w.r.t. α, denoted $\mathcal{K} \models (\alpha,\, A_\psi)$, in the natural way: $\exists\pi_i$ corresponds to the existence of a path assigned to π_i in K_i, and dually for $\forall\pi_i$. Then, $\mathcal{K} \models (\alpha, A_\psi)$ iff the various k-assignments of traces of $\mathcal{K}$ to $\{\pi_i\}_{i=1}^k$ according to α are accepted by A_ψ, which holds iff $\mathcal{K} \models \varphi$.

For a model K, we denote by $K \cap_k A_\psi$ the intersection of K and A_ψ w.r.t. $\mathsf{AP} \times \{\pi_k\}$, taking the projection over $\mathsf{AP} \times \{\pi_i\}_{i=1}^{k-1}$. Thus, $\mathcal{L}(K \cap_k A_\psi)$ is the set of all $(k-1)$-words that *an extension* (i.e., $\exists$) by a word in $\mathcal{L}(K)$ to a k-word in $\mathcal{L}(A_\psi)$. Oppositely, $\mathcal{L}(\overline{K \cap_k \overline{A_\psi}})$ is the set of all $(k-1)$-words that *every extension* (i.e., $\forall$) by a k-word in $\mathcal{L}(K)$ is in $\mathcal{L}(A_\psi)$.

We first construct NBWs $A_2, \ldots, A_{k-1}, A_k$, such that for every $1 < i < k$, we have $\langle K_1, \ldots, K_i \rangle \models (\alpha_i, A_{i+1})$ iff $\mathcal{K} \models (\alpha, A_\psi)$, where $\alpha_i = \mathbb{Q}_1\pi_1 \ldots \mathbb{Q}_i\pi_i$.

For $i = k$, if $\mathbb{Q}_k = \exists$, then $A_k = K_k \cap_k A_\psi$; otherwise if $\mathbb{Q}_k = \forall$, $A_k = \overline{K_k \cap_k \overline{A_\psi}}$. For $1 < i < k$, if $\mathbb{Q}_i = \exists$ then $A_i = K_i \cap_i A_{i+1}$; otherwise if $\mathbb{Q}_i = \forall$, $A_i = \overline{K_i \cap_i \overline{A_{i+1}}}$. Then, for every $1 < i < k$, we have $\langle K_1, \ldots, K_i \rangle \models (\alpha_i, A_{i+1})$ iff $\langle K_1, \ldots, K_k \rangle \models \varphi$.

We now prove by induction on k that $\mathcal{K} \models \varphi$ iff $\langle C_1, \ldots C_k \rangle \models \varphi$. For $k = 1$, it holds that $\mathcal{K} \models \varphi$ iff $K_1 \models (\mathbb{Q}_1\pi_1, A_2)$. If $\mathbb{Q}_1 = \forall$, then $K_1 \models (\mathbb{Q}_1\pi_1, A_2)$ iff $K_1 \cap \overline{A_2} = \emptyset$. If $\mathbb{Q}_1 = \exists$, then $K_1 \models (\mathbb{Q}_1\pi_1, A_2)$ iff $K_1 \cap A_2 \neq \emptyset$. In both cases, a lasso witness to the non-emptiness exists. For $1 < i < k$, we prove that $\langle C_1, \ldots, C_i, K_{i+1} \rangle \models (\alpha_{i+1}, A_{i+2})$ iff $\langle C_1, \ldots, C_i, C_{i+1} \rangle \models (\alpha_{i+1}, A_{i+2})$. If $\mathbb{Q}_i = \forall$, then the first direction simply holds because $\mathcal{L}(C_{i+1}) \subseteq \mathcal{L}(K_{i+1})$. For the second direction, every extension of $c_1, c_2, \ldots c_i$ (i.e., lassos in $C_1, C_2, \ldots C_i$) by a path τ in K_{i+1} is in $\mathcal{L}(A_{i+2})$. Indeed, otherwise we can extract a lasso c_{i+1} such that $c_1, c_2, \ldots c_{i+1}$ is in $\overline{\mathcal{L}(A_{i+2})}$, a contradiction. If $\mathbb{Q}_i = \exists$, then $\mathcal{L}(C_{i+1}) \subseteq \mathcal{L}(K_{i+1})$ implies the second direction. For the first direction, we can extract a lasso $c_{i+1} \in \mathcal{L}(C_{i+1})$ such that $\langle c_1, c_2, \ldots c_i, c_{i+1} \rangle \in \mathcal{L}(A_{i+2})$. □

One can use Theorem 1 and the observations above to construct a sound and complete BMC algorithm for both $\forall\exists$ and $\exists\forall$ hyperproperties. Indeed, consider a multi-model $\langle K_1, K_2 \rangle$, and a hyperproperty $\varphi = \forall\pi.\exists\pi'.\ \psi$. Such a BMC algorithm would try and verify $\langle K_1, K_2 \rangle \models \varphi$ directly, or try and prove $\langle K_1, K_2 \rangle \models \neg\varphi$. In both cases, a run may find a short lasso example for the model under $\exists$ (K_2 in the former case and K_1 in the latter), leading to a shorter run. However, in both cases, the model under $\forall$ would have to be explored to the maximal lasso length implicated by Theorem 1, which is doubly-exponential. Therefore, this naive approach would be highly inefficient.

4 Simulation-Based BMC Algorithms for HyperLTL

We now introduce efficient simulation-based BMC algorithms for verifying hyperproperties of the types $\forall\pi.\exists\pi'.\square\mathsf{Pred}$ and $\exists\pi.\forall\pi'.\square\mathsf{Pred}$, where Pred is a *relational predicate* (a predicate over a pair of states). The key observation is that simulation naturally induces the exploration of infinite traces without the need to explicitly unroll the structures, and without needing to synchronize the indices of the symbolic variables in both traces. Moreover, in some cases our algorithms allow to only partially explore the state space of a Kripke structure and give a conclusive answer efficiently.

Let $K_P = \langle S_P, S_P^0, \delta_P, \mathsf{AP}_P, L_P \rangle$ and $K_Q = \langle S_Q, S_Q^0, \delta_Q, \mathsf{AP}_Q, L_Q \rangle$ be two Kripke structures, and consider a hyperproperty of the form $\forall\pi.\exists\pi'.\ \square\mathsf{Pred}$. Suppose that there exists a simulation from K_P to K_Q. Then, every trace in K_P is embodied in K_Q. Indeed, we can show by induction that for every trace $t_p = s_p(1)s_p(2)\ldots$ in K_P, there exists a trace $t_q = s_q(1)s_q(2)\ldots$ in K_Q, such that $s_q(i)$ simulates $s_p(i)$ for every $i \geq 1$; therefore, t_p and t_q are equally labeled. We generalize the labeling constraint in the definition of standard simulation by requiring, given Pred, that if (s_p, s_q) is in the simulation relation, then

$(s_p, s_q) \models \mathsf{Pred}$. We denote this generalized simulation by $\mathsf{SIM_{AE}}$. Following similar considerations, we now have that for every trace t_p in K_P, there exists a trace t_q in K_Q such that $(t_p, t_q) \models \square\mathsf{Pred}$. Therefore, the following result holds:

Lemma 1. *Let K_P and K_Q be Kripke structures, and let $\varphi = \forall\pi.\exists\pi'.\ \square\mathsf{Pred}$ be a HyperLTL formula. If there exists $\mathsf{SIM_{AE}}$ from K_P to K_Q, then $\langle K_P, K_Q\rangle \models \varphi$.*

We now turn to properties of the type $\exists\pi.\forall\pi'.\ \square\mathsf{Pred}$. In this case, we must find a single trace in K_P that matches every trace in K_Q. Notice that $\mathsf{SIM_{AE}}$ (in the other direction) does not suffice, since it is not guaranteed that the same trace in K_P is used to match all traces in K_Q. However, according to Theorem 1, it is guaranteed that if $\langle K_P, K_Q\rangle \models \exists\pi.\forall\pi'.\ \square\mathsf{Pred}$, then there exists such a single lasso trace t_p in K_P as the witness of the satisfaction. We therefore define a second notion of simulation, denoted $\mathsf{SIM_{EA}}$, as follows. Let $t_p = s_p(1)s_p(2)\dots s_p(n)\dots s_p(\ell)$ be a lasso trace in K_P (where $s_p(\ell)$ closes to $s_p(n)$, that is, $(s_p(\ell), s_p(n)) \in \delta_P$). A relation R from t_p to K_Q is considered as a $\mathsf{SIM_{EA}}$ from t_p to K_Q, if the following holds:

1. $(s_p, s_q) \models \mathsf{Pred}$ for every $(s_p, s_q) \in R$.
2. $(s_p(1), s_q) \in R$ for every $s_q \in S_Q^0$.
3. If $(s_p(i), s_q(i)) \in R$, then for every successor $s_q(i+1)$ of $s_q(i)$, it holds that $(s_p(i+1), s_q(i+1)) \in R$ (where $s_p(\ell+1)$ is defined to be $s_p(n)$).

If there exists a lasso trace t_p, then we say that there exists $\mathsf{SIM_{EA}}$ from K_P to K_Q. Notice that the third requirement in fact unrolls K_Q in a way that guarantees that for every trace t_q in K_Q, it holds that $(t_p, t_q) \models \square\mathsf{Pred}$. Therefore, the following result holds:

Lemma 2. *Let K_P and K_Q be Kripke structures, and let $\varphi = \exists\pi.\forall\pi'.\ \square\mathsf{Pred}$. If there exists a $\mathsf{SIM_{EA}}$ from K_P to K_Q, then $\langle K_P, K_Q\rangle \models \varphi$.*

Lemmas 1 and 2 enable sound algorithms for model-checking $\forall\pi.\exists\pi'.\ \square\mathsf{Pred}$ and $\exists\pi.\forall\pi'.\ \square\mathsf{Pred}$ hyperproperties with loop conditions. To check the former, check whether there exists $\mathsf{SIM_{AE}}$ from K_P to K_Q; to check the latter, check for a lasso trace t_p in K_P and $\mathsf{SIM_{EA}}$ from t_p to K_Q. Based on these ideas, we introduce now two SAT-based BMC algorithms.

For $\forall\exists$ hyperproperties, we not only check for the existence of $\mathsf{SIM_{AE}}$, but also iteratively seek a small subset of S_Q that suffices to simulate all states of S_P. While finding $\mathsf{SIM_{AE}}$, as for standard simulation, is polynomial, the problem of finding a simulation with a bounded number of K_Q states is NP-complete (see [28] for details). This allows us to efficiently handle instances in which K_Q is large. Moreover, we introduce in Subsection 4.3 the use of *prophecy variables*, allowing us to overcome cases in which the models satisfy the property but $\mathsf{SIM_{AE}}$ does not exist.

For $\exists\forall$ hyperproperties, we search for $\mathsf{SIM_{EA}}$ by seeking a lasso trace t_p in K_P, whose length increases with every iteration, similarly to standard BMC techniques for LTL. Of course, in our case, t_p must be matched with the states of K_Q in a way that ensures $\mathsf{SIM_{EA}}$. In the worst case, the length of t_p may be

doubly-exponential in the sizes of the systems. However, as our experimental results show, in case of satisfaction the process can terminate much sooner.

We now describe our BMC algorithms and our SAT encodings in detail. First, we fix the unrolling depth of K_P to n and of K_Q to k. To encode states of K_P we allocate a family of Boolean variables $\{x_i\}_{i=1}^{n}$. Similarly, we allocate $\{y_j\}_{j=1}^{k}$ to represent the states of K_Q. Additionally, we encode the simulation relation T by creating $n \times k$ Boolean variables $\{sim_{ij}\}_{i=1,j=1}^{n,k}$ such that sim_{ij} holds if and only if $T(p_i, q_j)$. We now present the three variations of encoding: (1) EA-Simulation ($\mathsf{SIM_{EA}}$), (2) AE-Simulation ($\mathsf{SIM_{AE}}$), and (3) a special variation where we enrich AE-Simulation with prophecies.

4.1 Encodings for EA-Simulation

The goal of this encoding is to find a lasso path t_p in K_P that guarantees that there exists $\mathsf{SIM_{EA}}$ to K_Q. Note that the set of states that t_p uses may be much smaller than the whole of K_P, while the state space of K_Q must be explored exhaustively. We force x_0 be an initial state of K_P and for x_{i+1} to follow x_i for every i we use, but for K_Q we will let the solver fill freely each y_k and add constraints[3] for the full exploration of K_Q.

- **All states are legal states.** The solver must only search legal encodings of states of K_P and K_Q (we use $K_P(x_i)$ to represent the combinations of values that represent a legal state in S_P and similarly $K_Q(y_j)$ for S_Q):

$$\bigwedge_{i=1}^{n} K_P(x_i) \wedge \bigwedge_{j=1}^{k} K_Q(y_j) \tag{1}$$

- **Exhaustive exploration of K_Q.** We require that two different indices y_j and y_r represent two different states in K_Q, so if $k = |K_Q|$, then all states are represented, where $y_j \neq y_r$ captures that some bit distinguishes the states encoded by j and r (note that the validity of states is implied by (1)):

$$\bigwedge_{j \neq r} (K_Q(y_j) \wedge K_Q(y_r)) \rightarrow (y_j \neq y_r) \tag{2}$$

- **The initial S_P^0 state simulates all initial S_Q^0 states.** State x_0 is an initial state of K_P and simulates all initial states of K_Q (we use $I_P(x_0)$ to represent a legal initial state in K_P and $I_Q(y_j)$ for S_Q^0 of K_Q):

$$I_P(x_0) \wedge \Big(\bigwedge_{j=1}^{k} I_Q(y_j) \rightarrow T(x_0, y_j) \Big) \tag{3}$$

[3] An alternative is to fix an enumeration of the states of K_Q and force the assignment of $y_0 \ldots$ according to this enumeration instead of constraining a symbolic encoding, but the explanation of the symbolic algorithm above is simpler.

- **Successors in K_Q are simulated by successors in K_P.** We first introduce the following formula $succ_T(x, x')$ to capture one-step of the simulation, that is, x' follows x and for all y if $T(x, y)$ then x' simulates all successors of y (we use $\delta_Q(y, y')$ to represent that y and y' states are in δ_Q of K_Q, similarly for $(x, x') \in \delta_P$ of K_P we use $\delta_P(x, x')$) :

$$succ_T(x, x') \stackrel{\text{def}}{=} \bigwedge_{y=y_1}^{y_k} T(x, y) \rightarrow \Big(\bigwedge_{y'=y_1}^{y_k} \delta_Q(y, y') \rightarrow T(x', y') \Big)$$

  We can then define that x_{i+1} follows x_i:

$$\bigwedge_{i=1}^{n-1} \big[\delta_P(x_i, x_{i+1}) \wedge succ_T(x_i, x_{i+1}) \big] \tag{4}$$

  And, x_n has a jump-back to a previously seen state:

$$\bigvee_{i=1}^{n} \big[\delta_P(x_n, x_i) \wedge succ_T(x_n, x_i) \big] \tag{5}$$

- **Relational state predicates are fulfilled by simulation.** Everything relating in the simulation fits the relational predicate, defined as a function Pred of two sets of labels (we use $L_Q(y)$ to represent the set of labels on the y-encoded state in K_Q, similarly, $L_P(x)$ for the x-encoded state in K_P):

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} T(x_i, y_j) \rightarrow \mathsf{Pred}(L_P(x_i), L_Q(y_j)) \tag{6}$$

We use $\varphi_{\mathsf{EA}}^{n,k}$ for the SAT formula that results of conjoining (1)-(6) for bounds n and k. If $\varphi_{\mathsf{EA}}^{n,k}$ is satisfiable, then there exists $\mathsf{SIM}_{\mathsf{EA}}$ from K_P to K_Q.

4.2 Encodings for AE-Simulation

Our goal now is to find a set of states $S'_Q \subseteq S_Q$ that is able to simulate all states in K_P. Therefore, as in the previous case, the state space K_P corresponding to the $\forall$ quantifier will be explored exhaustively, and so $n = |K_P|$, while k is the number of states in K_Q, which increases in every iteration. As we have explained, this allows finding a small subset of states in K_Q which suffices to simulate all states of K_P (Note that here we guarantee soundness but not necessarily completeness, which will be further explained in Section 4.3).

- **All states in the simulation are legal states.** Again, every state guessed in the simulation is a legal state from K_P or K_Q:

$$\bigwedge_{i=1}^{n} K_P(x_i) \wedge \bigwedge_{j=1}^{k} K_Q(y_j) \tag{1'}$$

- K_P **is exhaustively explored.** Every two different indices in the states of K_P are different states[4]:

$$\bigwedge_{i \neq r} (K_P(x_i) \wedge K_P(x_r)) \rightarrow (x_i \neq x_r) \tag{2'}$$

- **All initial states in K_P must match with some initial state in K_Q.** Note that, contrary to the $\exists\forall$ case, here the initial state in K_Q may be different for each initial state in K_P:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} I_P(x_i) \rightarrow \big(I_Q(y_j) \wedge T(x_i, y_j)\big) \tag{3'}$$

- **For every pair in the simulation, each successor in K_P must match with some successor in K_Q.** For each (x_i, y_j) in the simulation, every successor state of x_i has a matching successor state of y_j:

$$\bigwedge_{i=1}^{n} \bigwedge_{t=1}^{n} \delta_P(x_i, x_t) \rightarrow \bigwedge_{j=1}^{k} \Big[T(x_i, y_j) \rightarrow \bigvee_{r=1}^{k} \Big(\delta_Q(y_j, y_r) \wedge T(x_t, y_r)\Big)\Big] \tag{4'}$$

- **Relational state predicates are fulfilled.** Similarly, all pairs of states in the simulation should respect the relational Pred:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} T(x_i, y_j) \rightarrow \mathsf{Pred}(L_P(x_i), L_Q(y_j)) \tag{5'}$$

We now use $\varphi_{\mathsf{AE}}^{n,k}$ for the SAT formula that results of conjoining (1′)-(5′) for bounds n and k. If $\varphi_{\mathsf{AE}}^{n,k}$ is satisfiable, then there exists $\mathsf{SIM}_{\mathsf{AE}}$ from K_P to K_Q.

4.3 Encodings for AE-Simulation with Prophecies

The AE-simulation encoding introduced in Section 4.2 is sound but not complete (i.e., the property is satisfied, yet no simulation exists). For example, when the system for the $\forall$ quantifier is non-deterministic, the simulation is required to match immediately the successor of the $\exists$ path without inspecting the future of the $\forall$ path. In this section, we incorporate our encodings with *prophecies* to resolve these kind of cases, which takes us one step towards completeness. We now illustrate with the following example.

Example 1. Consider Kripke structures K_1 and K_2 from Section 1, and HyperLTL formula $\varphi_2 = \forall\pi.\exists\pi'.\ \square(a_\pi \leftrightarrow a_{\pi'})$. It is easy to see that the two models satisfy φ_2, since mapping the sequence of states $(s_1 s_2 s_3)$ to $(q_1 q_2 q_4)$ and $(s_1 s_2 s_4)$ to $(q_1 q_3 q_5)$ guarantees that the matched paths satisfy $\square(a_\pi \leftrightarrow a_{\pi'})$. However, the technique in Section 4.2 cannot differentiate the occurrences of s_2 in the two different cases. □

[4] As in the previous case, we could fix an enumeration of the states of S_P and fix $x_0 x_1 \ldots$ to be the states according to the enumerations.

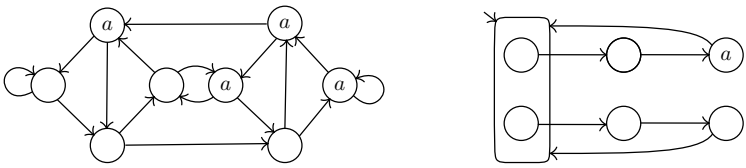


Fig. 1: Prophecy automaton for $\bigcirc\bigcirc a$ (left) and its composition with K_1 (right).

To solve this, we incorporate the notion of *prophecies* to our setting. Prophecies have been proposed as a method to aid in the verification of hyperliveness [14] (see [7] for a systematic method to construct prophecies). For simplicity, we restrict here to prophecies expressed as safety automata. A safety prophecy over AP is a Kripke structure $U = \langle S, S^0, \delta, \mathsf{AP}, L\rangle$, such that $\mathit{Traces}(U) = \mathsf{AP}^\omega$. The product $K \times U$ of a Kripke structure K with a prophecy U preserves the language of K (since the language of U is universal). Recall that in the construction of the product, states $(s, u) \in (K \times U)$ that have incompatible labels are removed. The direct product can be easily processed by repeatedly removing dead states, resulting in a Kripke structure K' whose language is $\mathit{Traces}(K') = \mathit{Traces}(K)$. Note that there may be multiple states in K' that correspond to different states in K for different prophecies. The prophecy-enriched Kripke structure can be directly passed to the method in Section 4.2, so the solver can search for a $\mathsf{SIM_{AE}}$ that takes the value of the prophecy into account.

Example 2. Consider the prophecy automaton shown in Fig. 1 (left), where all states are initial. Note that for every state, either all its successors are labeled with a (or none are), and all successors of its successors are labeled with a (or none are). In other words, this structure encodes the prophecy $\bigcirc\bigcirc a$. The product K_1' of K_1 with the prophecy automaton U for $\bigcirc\bigcirc a$ is shown in Fig. 1 (right). Our method can now show that $\langle K_1', K_2\rangle \models \varphi_2$, since it can distinguish the two copies of s_1 (one satisfies $\bigcirc\bigcirc a$ and is mapped to $(q_1q_2q_4)$, while the other is mapped to $(q_1q_3q_5)$). □

5 Implementation and Experiments

We have implemented our algorithms using the SAT solver `Z3` through its python API `Z3Py` [15]. The SAT formulas introduced in Section 4 are encoded into the two scripts `simEA.py` and `simAE.py`, for finding simulation relations for the $\mathsf{SIM_{EA}}$ and $\mathsf{SIM_{AE}}$ cases, respectively. We evaluate our algorithms with a set of experiments, which includes all forms of quantifiers with different sizes of given models, as presented earlier in Table 1. Our simulation algorithms benefit the most in the cases of the form $\forall_{\texttt{small}}\, \exists_{\texttt{big}}$. When the second model is substantially larger than the first model, $\mathsf{SIM_{AE}}$ is able to prove that a $\forall\exists$ hyperproperty holds by exploring only a subset of the second model. In this section, besides $\forall_{\texttt{small}}\, \exists_{\texttt{big}}$ cases, we also investigate multiple cases on each category in Table 1 to

demonstrate the generality and applicability of our algorithms. All case studies are run on a MacBook Pro with Apple M1 Max chip and 64 GB of memory.

5.1 Case Studies and Empirical Evaluation

Conformance in Scenario-based Programming. In scenario-based programming, scenarios provide a big picture of the desired behaviors of a program, and are often used in the context of program synthesis or code generation. A synthesized program should obey what is specified in the given set of scenarios to be considered *correct*. That is, the program *conforms* with the scenarios. The conformance check between the scenarios and the synthesized program can be specified as a $\forall\exists$-hyperproperty:

$$\varphi_{\mathsf{conf}} = \forall\pi.\exists\pi'.\ \bigwedge_{p\in\mathsf{AP}} \square\ (p_\pi \leftrightarrow p_{\pi'}),$$

where π is over the scenario model and π' is over the synthesized program. That is, for all possible runs in the scenarios, there must exists a run in the program, such that their behaviors always match.

We look into the case of synthesizing an *Alternating Bit Protocol (ABP)* from four given scenarios, inspired by [3]. ABP is a networking protocol that guarantees reliable message transition, when message loss or data duplication are possible. The protocol has two parties: `sender` and `receiver`, which can take three different actions: *send*, *receive*, and *wait*. Each action also specifies which message is currently transmitted: either a *packet* or *acknowledgment* (see [3] for more details). The correctly synthesized protocol should not only have complete functionality but also *include all scenarios*. That is, for every trace that appears in some scenario, there must exist a corresponding trace in the synthesized protocol. By finding $\mathsf{SIM_{AE}}$ between the scenarios and the synthesized protocols, we can prove the conformance specified with φ_{conf}. Note that the scenarios are often much smaller than the actual synthesized protocol, and so this case falls in the $\forall_{\texttt{small}}\ \exists_{\texttt{big}}$ category in Table 1. We consider two variations: a correct and an incorrect ABP (that cannot handle packet loss). Our algorithm successfully identifies a $\mathsf{SIM_{AE}}$ that satisfies φ_{conf} for the correct ABP, and returns UNSAT for the incorrect protocol, since the packet loss scenario cannot be simulated.

Verification of Model Translation. It is often the case that in model translation (e.g., compilation), solely reasoning about the source program does not provide guarantees about the desirable behaviors in the target executable code. Since program verification is expensive compared with repeatedly checking the target, alternative approaches such as *certificate translation* [4] are often preferred. Certificate translation takes inputs of a high-level program (source) with a given specification, and computes a set of verification conditions (certificates) for the low-level executable code (target) to prove that a model translation is safe. However, this technique still requires extra efforts to map the certificates to a target language, and the size of generated certificates might explode quickly

(see [4] for retails). We show that our simulation algorithm can directly show the correctness of a model translation more efficiently by investigating the source and target with the same formula φ_{conf} used for ABP. That is, the specifications from the source runs π are always preserved in some target runs π', which infers a correct model translation. Since translating a model into executable code implies adding extra instructions such as writing to registers, it also falls into the $\forall_{\texttt{small}}\, \exists_{\texttt{big}}$ category in Table 1.

We investigate a program from [4] that performs *matrix multiplication (MM)*. When executed, the C program is translated from high-level code (C) to low-level code RTL (Register Transfer Level), which contains extra steps to read from/write to memories. Specifications are triples of $\langle Pre, annot, Post\rangle$, where *Pre*, and *Post* are assertions and *annot* is a partial function from labels to assertions (see [4] for detailed explanations). The goal is to make sure that the translation does not violate the original verified specification. In our framework, instead of translating the certification, we find a simulation that satisfies φ_{conf}, proving that the translated code also satisfies the specification. We also investigate two variations in this case: a correct translation and an incorrect translation, and our algorithm returns SAT (i.e., finds a correct $\mathsf{SIM}_{\mathsf{AE}}$ simulation) in the former case, and returns UNSAT for the latter case.

Compiler Optimization. Secure compiler optimization aims at preserving input-output behaviors of an original implementation and a target program after applying optimization techniques, including security policies. The conformance between source and target programs guarantees that the optimizing procedure does not introduce vulnerabilities such as information leakage. Furthermore, optimization is often not uniform for the same source, because one might compile the source to multiple different targets with different optimization techniques. As a result, an efficient way to check the behavioral equivalence between the source and target provides a correctness guarantee for the compiler optimization.

```
// Source program S
L1: if (j < arr_size) {
L2:     a := arr[0];
L3:     b := arr[j];
L4: } else {
L5:     a := arr[0];
L6:     b := arr[arr_size - 1];
L7: }
// Target program T
L1: a := arr[0];
L2: if (j < arr_size) {
L3:     b := arr[j];
L4: } else {
L5:
L6:     b := arr[arr_size - 1];
L7: }
```

Fig. 2: The common branch factorization example [30].

Imposing optimization usually results in a smaller program. For instance, *common branch factorization* (CBF) finds common operations in an if-then-else structure, and moves them outside of the conditional so that such operation is only executed once. As a result, for these optimization techniques, checking the conformance of the source and target falls in the $\forall_{\texttt{big}}\, \exists_{\texttt{small}}$ category. That is, given two programs, source (`big`) and target (`small`), we check the following formula:

$$\varphi_{\mathsf{sc}} = \forall\pi.\exists\pi'.\ (\mathsf{in}_{\pi} \leftrightarrow \mathsf{in}_{\pi'}) \rightarrow \square\ (\mathsf{out}_{\pi} \leftrightarrow \mathsf{out}_{\pi'}).$$

In this case study we investigate the strategy CBF using the example in Figure 2 inspired by [30]. We consider two kinds of optimized programs for the strategy, one is the correct optimization, one containing bugs that violates the original behavior due to the optimization. For the correct version, our algorithm successfully discovered a simulation relation between the source and target, and the simulation relation returns a smaller subset of states in the second model (i.e., $|S'_Q| < |S_Q|$). For the incorrect version, we received UNSAT.

Robust Path Planning. In robotic planning, *robustness planning (RP)* refers to a path that is able to consistently complete a mission without being interfered by the uncertainty in the environment (e.g., adversaries). For instance, in the 2-D plane in Fig. 3, an agent is trying to go from the starting point (blue grid) to the goal position (green grid). The plane also contains three adversaries on the three corners other than the starting point (red-framed grids), and the adversaries move trying to catch the agent but can only move in one direction (e.g., clockwise). This is a $\exists_{\texttt{small}}\ \forall_{\texttt{big}}$ setting, since the adversaries may have several ways to cooperate and attempt to catch the agent. We formulate this planning problem as follows:

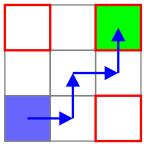


Fig. 3: A robust path.

$$\varphi_{\mathsf{rp}} = \exists\pi.\forall\pi'.\ \square\ (\mathsf{pos}_{\pi} \not\leftrightarrow \mathsf{pos}_{\pi'}).$$

That is, there exists a robust path for the agent to safely reach the goal regardless of all the ways that the adversaries could move. We consider two scenarios, one in which there exists a way for the agent to form a robust path and one does not. Our algorithm successfully returns SAT for case which the agent can form a robust path, and returns UNSAT for which a robust path is impossible to find.

Plan Synthesis. The goal of *plan synthesis (PS)* is to synthesize a single comprehensive plan that can simultaneously satisfy all given small requirements has wide application in planning problems. We take the well-known toy example, *wolf, goat, and cabbage*[5], as a representative case here. The problem is as follows. A farmer needs to cross a river by boat with a wolf, a goat, and a cabbage. However, the farmer can only bring one item with him onto the boat each time. In addition, the wolf would eat the goat, and the goat would eat the cabbage, if they are left unattended. The goal is to find a plan that allows the farmer to successfully cross the river with all three items safely. A plan requires the farmer to go back and forth with the boat with certain possible ways to carry different items, while all small requirements (i.e., the constraints among each item) always satisfied. In this example, the overall plan is a big model while the requirements form a much smaller automaton. Hence, it is a $\exists_{\texttt{big}}\ \forall_{\texttt{small}}$ problem that can be specified with the following formula:

$$\varphi_{\mathsf{ps}} = \exists\pi.\forall\pi'.\ \square\ (\mathsf{action}_{\pi} \not\leftrightarrow \mathsf{violation}_{\pi'}).$$

[5] https://en.wikipedia.org/wiki/Wolf,_goat_and_cabbage_problem

Type	Quants	Cases	$\lvert S_P \rvert$	$\lvert S_Q \rvert$	Z3	Outcome	solve[s]
$\mathsf{SIM_{AE}}$	$\forall_{\mathtt{small}}\ \exists_{\mathtt{big}}$	ABP	11	14	sat	$\lvert S'_Q \rvert$=11	9.37
		$\mathrm{ABP}_{\mathsf{w/\ bug}}$	11	14	unsat	-	9.46
		MM	27	27	sat	$\lvert S'_Q \rvert$=27	67.74
		$\mathrm{MM}_{\mathsf{w/\ bug}}$	27	27	unsat	-	66.85
	$\forall_{\mathtt{big}}\ \exists_{\mathtt{small}}$	CBF	15	9	sat	$\lvert S'_Q \rvert$=8	3.49
		$\mathrm{CBF}_{\mathsf{w/\ bug}}$	15	9	unsat	-	3.51
$\mathsf{SIM_{EA}}$	$\exists_{\mathtt{small}}\ \forall_{\mathtt{big}}$	RP 3^3	8	9	sat	$\lvert S'_P \rvert$=5	1.09
		RP $3^3{}_{\mathsf{no\ sol.}}$	8	9	unsat	-	1.02
	$\exists_{\mathtt{big}}\ \forall_{\mathtt{small}}$	GCW	16	4	sat	$\lvert S'_P \rvert$=8	3.36
		$\mathrm{GCW}_{\mathsf{no\ sol.}}$	16	4	unsat	-	2.27

Table 2: Summary of our case studies. The outcomes with simulation discovered show how our algorithms find a smaller subset for either K_P or K_Q.

5.2 Analysis and Discussion

The summary of our empirical evaluation is presented in Table 2. For the $\forall\exists$ cases, our algorithm successfully finds a set $|S'_Q| < |S_Q|$ that satisfies the properties for the cases ABP and CBF. Note that case MM does not find a small subset, since we manually add extra *paddings* on the first model to align the length of both traces. We note that handling this instance without padding requires asynchornicity— a much more difficult problem, which we leave for future work. For the $\exists\forall$ cases, we are able to find a subset of S_P which forms a single lasso path that can simulate all runs in S_Q for all cases RP and GCW. We emphasize here that previous BMC techniques (i.e., HyperQB) cannot handle most of the cases in Table 2 due to the lack of loop conditions.

6 Conclusion and Future Work

We introduced efficient loop conditions for bounded model checking of fragments of HyperLTL. We proved that considering only lasso-shaped traces is equivalent to considering the entire trace set of the models, and proposed two simulation-based algorithms $\mathsf{SIM_{EA}}$ and $\mathsf{SIM_{AE}}$ to realize infinite reasoning with finite exploration for HyperLTL formulas. To handle non-determinism in the latter case, we combine the models with prophecy automata to provide the (local) simulations with enough information to select the right move for the inner $\exists$ path. Our algorithms are implemented using Z3py. We have evaluated the effectiveness and efficiency with successful verification results for a rich set of input cases, which previous bounded model checking approach would fail to prove.

As for future work, we are working on exploiting general prophecy automata (beyond safety) in order to achieve full generality for the $\forall\exists$ case. The second direction is to handle asynchrony between the models in our algorithm. Even though model checking asynchronous variants of HyperLTL is in general undecidable [25,5,9], we would like to explore semi-algorithms and fragments with decidability properties. Lastly, exploring how to handle infinite-state systems with our framework by applying *abstraction* techniques is also another promising future direction.

References

1. Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
2. Shreya Agrawal and Borzoo Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *Proc. of the 29th IEEE Computer Security Foundations Symp. (CSF'16)*, pages 239–252. IEEE, 2016.
3. Rajeev Alur, Milo Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing finite-state protocols from scenarios and requirements. In *Proc. of the 10th Int'l Haifa Verification Conf. (HVC'14)*, volume 8855 of *LNCS*, pages 75–91. Springer, 2014.
4. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, César Kunz, and Anne Pacalet. Implementing a direct method for certificate translation. In *Proc. of the 11th Int'l Conf. on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *LNCS*, pages 541–560. Springer, 2009.
5. Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In *Proc. of the 33rd Int'l Conf. on Computer Aided Verification (CAV'21), Part I*, volume 12759 of *LNCS*, pages 694–717. Springer, 2021.
6. Saddek Bensalem, Ahmed Bouajjani, Claire Loiseaux, and Joseph Sifakis. Property preserving simulations. In *Proc. of the Fourth Int'l Workshop on Computer Aided Verification (CAV'92)*, volume 663 of *LNCS*, pages 260–273. Springer, 1992.
7. Raven Beutner and Bernd Finkbeiner. Prophecy variables for hyperproperty verification. In *Proc. of 35th IEEE Computer Security Foundations Symp. (CSF'22)*, pages 471–485. IEEE, 2022.
8. Borzoo Bonakdarpour, César Sánchez, and Gerardo Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In *Proc. of the 8th Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18), Part II*, volume 11245 of *LNCS*, pages 8–27. Springer, 2018.
9. Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of HyperLTL. In *Proc. of the 36th Annual ACM/IEEE Symp. on Logic in Computer Science (LICS'21)*, pages 1–13. IEEE, 2021.
10. Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. Rewriting-based runtime verification for alternation-free HyperLTL. In *Proc. of the 23rd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17), Part II*, volume 10206 of *LNCS*, pages 77–93. Springer, 2017.
11. Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design (FMSD)*, 19(1):7–34, 2001.
12. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proc. of the 3rd Int'l Conf. on Principles of Security and Trust (POST'14)*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.
13. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
14. Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In *Proc. of the 31st Int'l Conf. on Computer Aided Verification (CAV'19)*, volume 11561 of *LNCS*, pages 121–139. Springer, 2019.

15. Leonardo M. de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
16. Bernd Finkbeiner, Cristopher Hahn, and Tobias Hans. MGHyper: Checking satisfiability of HyperLTL formulas beyond the $\exists^*\forall^*$ fragment. In *Proc. of the 16th Int'l Symp. on Automated Technology for Verification and Analysis (ATVA'18)*, volume 11138 of *LNCS*, pages 521–527. Springer, 2018.
17. Bernd Finkbeiner, Cristopher Hahn, and Marvin Stenger. EAHyper: Satisfiability, implication, and equivalence checking of hyperproperties. In *Proc. of the 29th Int'l Conf. on Computer Aided Verification (CAV'17), Part II*, volume 10427 of *LNCS*, pages 564–570. Springer, 2017.
18. Bernd Finkbeiner, Cristopher Hahn, Marvin Stenger, and Leander Tentrup. RVHyper: A runtime verification tool for temporal hyperproperties. In *Proc. of the 24th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Part II*, volume 10806 of *LNCS*, pages 194–200. Springer, 2018.
19. Bernd Finkbeiner, Cristopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods in System Design (FMSD)*, 54(3):336–363, 2019.
20. Bernd Finkbeiner, Cristopher Hahn, and Hazem Torfah. Model checking quantitative hyperproperties. In *Proc. of the 30th Int'l Conf. on Computer Aided Verification (CAV'18), Part I*, volume 10981 of *LNCS*, pages 144–163. Springer, 2018.
21. Bernd Finkbeiner, Christian Müller, Helmut Seidl, and Eugene Zalinescu. Verifying security policies in multi-agent workflows with loops. In *Proc. of the 15th ACM Conf. on Computer and Communications Security (CCS'17)*, pages 633–645. ACM, 2017.
22. Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Proc. of the 27th Int'l Conf. on Computer Aided Verification (CAV'15), Part I*, volume 9206 of *LNCS*, pages 30–48. Springer, 2015.
23. Ohad Goudsmid, Orna Grumberg, and Sarai Sheinvald. Compositional model checking for multi-properties. In *Proc. of the 22nd Int'l Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'21)*, volume 12597 of *LNCS*, pages 55–80. Springer, 2021.
24. Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.
25. Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5:1–29, 2021.
26. Cristopher Hahn, Marvin Stenger, and Leander Tentrup. Constraint-based monitoring of hyperproperties. In *Proc. of the 25th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*, volume 11428 of *LNCS*, pages 115–131. Springer, 2019.
27. Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In *Proc. of the 27th Int'l Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'21). Part I*, volume 12651 of *LNCS*, pages 94–112. Springer, 2021.
28. Tzu-Han Hsu, César Sánchez, Sarai Sheinvald, and Borzoo Bonakdarpour. Efficient loop conditions for bounded model checking hyperproperties. *CoRR*, abs/2301.06209, 2023.

29. Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. In *Proc. of the 11th Int'l Conf. on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 172–183. Springer, 1999.
30. Kedar S. Namjoshi and Lucas M. Tabajara. Witnessing secure compilation. In *Proc. of the 21st Int'l Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'20)*, volume 11990 of *LNCS*, pages 1–22. Springer, 2020.
31. Amir Pnueli. The temporal logic of programs. In *Proc. of the 18th Symp. on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE, 1977.
32. Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Proc. of Current Trends in Concurrency, Overviews and Tutorials*, volume 224 of *LNCS*, pages 510–584. Springer, 1985.
33. Sandro Stucki, César Sánchez, Gerardo Schneider, and Borzoo Bonakdarpour. Graybox monitoring of hyperproperties. In *Proc. of the 23rd Int'l Symp. on Formal Methods (FM'19)*, volume 11800 of *LNCS*, pages 406–424. Springer, 2019.
34. Moshe Y. Vardi and Pierre Wolper. Automata theoretic techniques for modal logic of programs. *Journal of Computer and System Sciences*, 32(2):183–221, 1986.

Reconciling Preemption Bounding with DPOR

Iason Marmanis(✉), Michalis Kokologiannakis, and Viktor Vafeiadis

MPI-SWS, Kaiserslautern and Saarbrücken, Germany
{imarmanis,michalis,viktor}@mpi-sws.org

Abstract. There are two major techniques for scaling up stateless model checking: *dynamic partial order reduction* (DPOR), which only explores executions that differ in the ordering of racy accesses, and *preemption bounding*, which only explores executions containing up to k preemptions (preemptive context-switches).

Combining these two techniques is challenging because DPOR-equivalent executions often contain a different number of preemptions, making it incorrect to cut explorations that exceed the preemption bound. To restore completeness, prior work has weakened the DPOR algorithm, which often results in the exploration of many redundant executions.

We propose an alternative approach. Starting from an optimal DPOR algorithm, we achieve completeness by allowing some slack on the preemption-bound of the explored executions. We prove that the required slack does not exceed the number of threads of the program (minus two), and that this upper limit is tight.

1 Introduction

Stateless model checking (SMC) [12] is an effective bug-finding technique for concurrent programs that systematically explores all interleavings of the given input program. As such, it suffers from the state-space explosion problem: the number of possible interleavings of a program grows rapidly with the program size. There are two main approaches to attack this problem in the literature.

Dynamic partial order reduction (DPOR) [11] is based on the idea that permutations of *independent* instructions in an interleaving lead to the same state. DPOR deems such interleavings equivalent and strives to explore only one representative interleaving from each equivalence class.

Preemption bounding (PB, a.k.a. context bounding) [25] is based on the idea that concurrency bugs in practice can be exposed with a small number of preemptions [24]. Leveraging this insight, PB only explores the interleavings that arise with at most k preemptions (for some fixed k), thereby guaranteeing a partial coverage of the state space.

Combining the two approaches is non-trivial. Simply modifying a DPOR algorithm to discard any explored executions that exceed the desired bound k is not complete, as executions with $\leq k$ preemptions are missed. To restore completeness, Coons et al. [10] weaken DPOR by adding extra backtracking points, but such an

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 85–104, 2023.
https://doi.org/10.1007/978-3-031-30823-9_5

approach negates any optimality properties of the underlying DPOR algorithm, and can lead to the (redundant) exploration of multiple equivalent interleavings.

In this paper, we propose a different approach. We adapt a state-of-the-art optimal DPOR algorithm with polynomial memory requirements called TruSt [16] to support preemption-bounded search.

We first observe that the preemption-bound definition of Coons et al. [10] is overly pessimistic for incomplete executions (i.e., executions where at least one thread is enabled) in that an incomplete execution can often be extended to a complete one with a smaller preemption-bound. Updating the definition to be more optimistic, however, does not fully resolve the issue: an intermediate execution that exceeds the bound might still be needed in order to reveal a conflicting instruction that leads to the exploration of the desired execution.

Our solution is to allow the exploration of executions exceeding the bound, as long as they only exceed it by a small amount, which we call *slack*. For programs with $N \geq 2$ threads, we show that a slack value of $N - 2$ suffices to maintain completeness (up to the provided bound). Unlike Coons et al. [10], our approach is optimal in the sense that it does not explore equivalent executions more than once. Although it may explore executions with larger bound than the desired one, we argue that these executions are useful, because they can still reveal bugs.

We have implemented our bounding approach in GenMC [18], a state-of-the-art open-source stateless model checker. We show that for small preemption bounds (and despite the slack), bounded search can perform significantly faster than full search. Moreover, we experimentally confirm the literature observation that small bounds suffice to expose most concurrency bugs. We therefore argue that our combination of preemption bounding and DPOR is useful as a practical testing approach, which also provides certain coverage guarantees.

2 Background

In this section, we recall the basic DPOR approach and how prior work has tried to incorporate preemption-bounded search into it. Subsequently, we review the TruSt algorithm [16], which we later build upon to obtain our results.

2.1 The Basics of Dynamic Partial Order Reduction

DPOR starts by exploring one thread interleaving. In the process, it detects conflicting transitions, i.e., instructions that, if executed in the opposite order, will alter the state of the system. At each state, when an earlier transition t is in conflict with a possible transition t' that can be taken by another thread in this state, DPOR considers the execution where t' is fired before t. To accomplish this, DPOR adds the transition t' to the *backtrack set* of the state immediately before t was fired, to be explored later.

We illustrate DPOR by running it on the following example (Fig. 1).

$$\begin{array}{l|\!|l} (r_x)\ a := x & (w_1)\ y := 1 \\ (r_y)\ b := y & (w_2)\ y := 2 \end{array} \qquad \text{(RR+WW)}$$

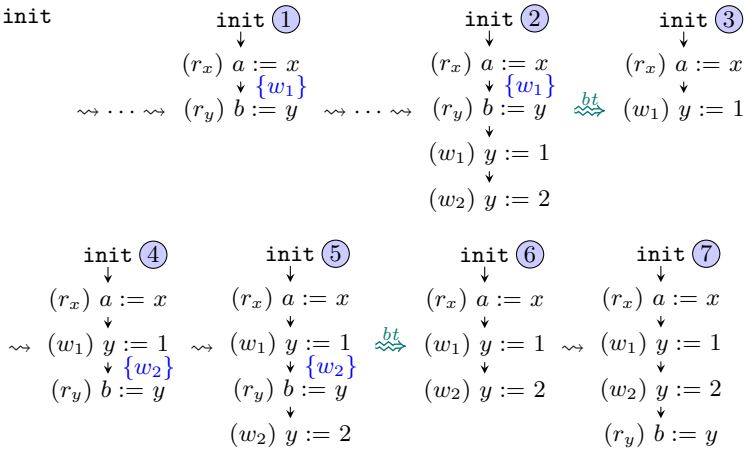


Fig. 1. Left-to-right DPOR exploration of RR+WW

After firing the transitions (r_x) and (r_y) (trace ①), DPOR adds transition (w_1) to the backtrack set of the state after the firing of transition (r_x), since transition (w_1) is in conflict with transition (r_y). When the initial exploration is finished (trace ②), DPOR backtracks to ① and considers the second exploration option, i.e., firing transition (w_1) and thus reaching ③.

Subsequently, DPOR fires (r_y) (trace ④) and notices that this is in conflict with (w_2); it then adds (w_2) as an alternative exploration option for the state before the firing of (r_y) in ④. Again, DPOR finishes with the exploration where the read instruction reads the value 1 (trace ⑤) and backtracks to ③. Now, (w_2) is fired (trace ⑥) and the algorithm continues with the remaining transition, leading to ⑦. DPOR now terminates since there is no other exploration option.

This way, DPOR manages to explores all three equivalence classes (representatives ②, ⑤, ⑦) of the 6 interleavings that correspond to this program.

2.2 Bounded Partial Order Reduction

Preemption bounding (PB) [25] prunes the state space by discarding executions that contain more preemptions than a given constant bound k. A preemption occurs at index i of a sequence of events τ whenever (1) events τ_i and τ_{i+1} originate from different threads and (2) the thread of τ_i remains enabled after τ_i; in particular, τ_i is not the last event of its thread.

Combining DPOR and PB is non-trivial. Specifically, simply pruning from DPOR's exploration space any trace with more than k preemptions is incorrect because their exploration might lead to exploring traces with up to k preemptions.

To see this, consider the run of RR+WW with $k = 0$. DPOR reaches the state where (r_x) is fired and (w_1) is considered as an alternative option in the backtrack

set. Firing transition (w_1) will lead to trace ③, which exceeds the bound, since there is a transition from the second thread present, while the first thread is still enabled. By discarding this state, the execution where $b = 2$ (which is equivalent to ⑦) would never be considered, even though it respects the bound.

To address this issue, Coons et al. [10] conservatively add more backtrack points accounting for such bound-induced dependencies. Concretely, when the two transitions of the first thread are fired (trace ①), Coons et al. [10] adds (w_1) in the backtrack set not only of the state before the firing of (r_y) in ②, as in the unmodified DPOR algorithm, but also of the initial state. Additionally, the initial transition from a state is always picked so that it is from the same thread as of the last fired transition, if possible. As a result, when the state with only (w_1) being fired is reached (due to the additional backtrack point), (w_2) will be fired immediately afterwards, and eventually the interleaving that corresponds to the right-to-left execution of the threads will be explored.

While this solution guarantees that no execution within the bound is lost, it weakens DPOR, i.e., it leads to the exploration of equivalent interleavings that would otherwise not be considered. In RR+WW, for $k > 0$, Coons et al. [10] explore interleavings that only differ in the order of (r_x) and (w_1).

2.3 TruSt: Optimal Dynamic Partial Order Reduction

The basic DPOR algorithm described in § 2.1 does not guarantee optimality, i.e., that only one execution from each equivalent class will be explored. There are several improvements of the basic algorithm, some of which achieve optimality (e.g., [2, 17]). Here, we follow the most recent such improvement, TruSt [16], which achieves optimality with polynomial memory consumption.

TruSt represents program executions as *execution graphs*, a concept that appeared in previous works for DPOR under weak memory models [15, 17]. An execution graph G consists of a set of nodes $G.\mathtt{E}$ (a.k.a. events) representing the individual thread instructions executed, such as read events `R` and write events `W`, and three kinds of directed edges encoding the ordering between events:

- the *program order* $G.\mathtt{po}$, which orders events of the same thread;
- the *coherence order* $G.\mathtt{co}$, which orders writes to the same location; and
- the *reads-from* mapping $G.\mathtt{rf}$, which shows where each read is reading from.

For an execution graph G, we define the following derived relations:

$$G.\mathtt{porf} \triangleq \left(G.\mathtt{po} \cup \left\{\langle G.\mathtt{rf}(r), r\rangle \,\middle|\, r \in G.\mathtt{R}\right\}\right)^{+} \qquad \text{(causality order)}$$

$$G.\mathtt{fr} \triangleq \left\{\langle r, w\rangle \,\middle|\, \langle G.\mathtt{rf}(r), w\rangle \in G.\mathtt{co}\right\} \qquad \text{(reads-before)}$$

The causality order, porf, relates two events if there is a path of program order or read-from dependencies between them, while fr orders a read event before every write that is coherence after the one read by the read.

An execution graph is SC-consistent (sequentially consistent) if there is a total ordering of its events respecting po such that each read event reads from

the immediately preceding same-location write in the total order. Equivalently, a graph is SC-consistent if $\mathtt{porf} \cup \mathtt{co} \cup \mathtt{fr}$ is acyclic.

Execution graphs enable the efficient reversal of many conflicting events. If a write or a read event is in conflict with a previous write event, there is no need to backtrack to the state before the write events is added. Instead, the new event can be directly added in the execution and either read from a co-earlier write in case of a read event, or be placed co-before the conflicting write in case of a write event.

The only reversals where backtracking is necessary are those between a write event and a previously added read event: when a read event is added, it does not have the option to read from a write that has not yet been added. These reversals are referred to as *backward revisits*. To avoid exponential memory consumption, TruSt considers each exploration option eagerly when the new event is added, instead of maintaining backtrack sets for later exploration. In the case of backward revisits, TruSt removes the part of the execution that was added after the read event but is not in the *prefix* of the write event. The prefix of an event is defined as the set of events that precede it in the porf order. This allows the write event to be directly added in the execution graph. Because there is the possibility that many different execution graphs can lead to the same execution after a backward revisit, TruSt only considers the revisit if the events to be removed respect a *maximality condition* which is defined in such a way so that there will always be exactly one such set of deleted events, achieving an optimal exploration.

3 Bounded Optimal DPOR: Obstacles

We discuss the two main obstacles that complicate the application of preemption-bounded search to a DPOR algorithm.

3.1 Pessimistic Bound Definition

The first problem concerns the definition of preemptions for incomplete executions. Recall in the RR+WW example why the naive adaptation of DPOR with preemption bound $k = 0$ (incorrectly) does not generate the execution reading $b = 2$. The partial trace ③ is discarded because it contains at least one preemption according to the definition of Musuvathi et al. [23]. (Both threads are enabled and have executed one instruction each.)

We argue that this trace should be deemed to have no preemptions because of monotonicity. Trace ③ can be extended to a full trace (namely, ⑦) that (is equivalent to one that) does not have any preemptions.

We therefore modify the definition of preemptions as follows. A preemption occurs at index i of an event sequence τ whenever (1) events τ_i and τ_{i+1} originate from different threads and (2) the thread of τ_i remains enabled after τ_i, and has further events in the trace $\tau_{i+1}\tau_{i+2} \ldots \tau_{|\tau|}$. According to our new definition, both

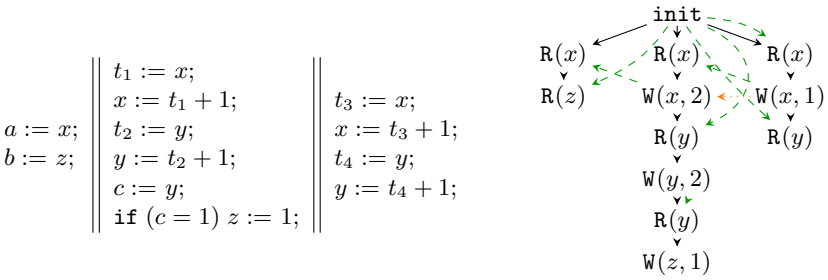


Fig. 2. A program and its intermediate execution that TruSt must explore in order to reach the right-to-left execution.

interleavings that are equivalent with ③ have zero preemptions, because when switching to another thread, the first thread has no further events in the trace.

Our new definition satisfies monotonicity and coincides with the original on complete executions. We note, however, that partial executions with k preemptions cannot always be extended to a complete execution with k preemptions. Consider, for example, trace ④ of RR+WW, which has no preemptions. Firing the only remaining transition leads to trace ⑤, which has one preemption. A DPOR algorithm that employs our definition of preemptions might thus reach states that are *bound-blocked*; the current explored execution respects the bound but there is no final execution reachable from this state that respects the bound. In our experience (see §6), bound-blocked executions do not seem to have a significant effect on the performance of our algorithm.

3.2 Need For Slack

Monotonicity alone is not enough to incorporate bounded search in an algorithm like TruSt, without still forfeiting completeness: some executions that respect the bound might still be lost. Intuitively, since DPOR algorithms operate by detecting conflicting instructions during an interleaving's exploration and reversing the conflict to obtain a new interleaving, it might be the case that for the conflict to be revealed, an execution that exceeds the bound needs to be explored.

We illustrate this point with the example in Fig. 2 where all the variables are initialized to zero. Consider a run of TruSt that always adds the next event from the left-most enabled thread. To reach the final execution that results from executing the threads from right to left, TruSt needs to pass through the execution depicted on the right of Fig. 2 before reaching this final execution. In the next step, the second write of the third thread will be added, which will reveal a conflict with the first read of y of the second thread. The algorithm will then perform a backward revisit, removing the events of the second thread after the first read of y, and change the read's incoming `rf` edge to the new write event. The desired final execution will be reached after the remaining events of the second thread are added again.

It is easy to see that, while the final execution has zero preemptions, the depicted intermediate execution has at least one preemption, and would thus be discarded. This example can in fact be generalized by adding more threads identical to the third one; to reach the final right-to-left execution that has zero preemptions, TruSt must visit an execution that has at least $N-2$ preemptions, where N is the total number of threads. In §4, we show that this is in fact an upper limit; a final execution with k preemptions is always reachable through a sequence of executions that never exceed $k + N - 2$ preemptions. This result directly enables us to incorporate preemption-bounded search into TruSt by allowing some *slack* to the bound.

4 Recovering Completeness via Slack

Our bounded DPOR algorithm, Buster, can be seen in Algorithm 1, where we have highlighted the differences w.r.t. to TruSt [16].

We first discuss some additional notation used in the algorithm. First, each execution graph generated by the algorithm keeps track of the order $<_G$ in which events were added to it. Second, given a graph G and a set of events E, we write $G|_E$ for the restriction of G to E. Third, let $G.\mathsf{cprefix}(e)$ be the causal prefix of an event e in an execution graph G, i.e., the set of all events that causally precede it (including e itself). Formally, $G.\mathsf{cprefix}(e) \triangleq \left\{e' \middle| \langle e', e\rangle \in G.\mathtt{porf}^*\right\}$. Fourth, a subscript $\mathtt{loc}(a)$ restricts a set of events to those that access the same location as event a. Fifth, the function $\mathsf{SetRF}(G, a, w)$ adds an `rf` edge from w to a and $\mathsf{SetCO}(G, w_p, a)$ places a immediately after w_p in `co`. Finally, we define the *traces* of an execution graph as the linearizations of $(G.\mathtt{porf} \cup G.\mathtt{co} \cup G.\mathtt{fr})$ on $G.\mathtt{E}$. We lift the definition of preemptions to an execution graph G: $\mathtt{preemptions}(G)$ is the minimum number of preemptions in the traces of G.

Apart from only exploring SC-consistent executions, Buster eagerly discards executions with more preemptions than the user-provided value k plus the slack (Line 5). If both tests fail, Buster continues by picking an new event to extend the current execution (Line 6). For correctness, we fix $\mathtt{next}_P(G)$ to always return the event that corresponds to the left-most available thread. Depending on the type of the new event, the algorithm proceeds in a different way. We discuss the interesting cases of read and write events.

If the new event a is a read event, Buster simply considers every possible write event as an `rf` option for a (Line 13), and eagerly explores the corresponding execution. If a is a write event, first every `co` placement is considered and explored (Line 15). Afterwards, Buster considers possible backward-revisits; for every read r event that is not in the causal prefix of a, the execution where r reads from a is considered, after deleting the events added after r, that are not in the causal prefix of a (Line 19). To avoid redundant revisits, only when the set of deleted events satisfies a maximality condition (Line 18), is the backward-revisit performed (see [16] for more details).

Algorithm 1 A Bounded DPOR algorithm based on TruSt [16]

```
1: procedure VERIFY(P, k)
2:     VISIT_{P,k}(G_∅)
3: procedure VISIT_{P,k}(G)
4:     if ¬consistent(G) then return
5:     if preemptions(G) > k + N − 2 then return
6:     switch a ← next_P(G) do
7:         case a = ⊥
8:             return "Visited full execution graph G"
9:         case a ∈ error
10:            exit("Visited erroneous execution graph G")
11:        case a ∈ R
12:            for w ∈ G.W_{loc(a)} do
13:                VISIT_{P,k}(SetRF(G, a, w))
14:        case a ∈ W
15:            VISITCOS_{P,k}(G, a)
16:            for r ∈ G.R_{loc(a)} \ G.cprefix(a) do
17:                Deleted ← {e ∈ G.E | r <_G e} \ G.cprefix(a)
18:                if ∀e ∈ Deleted ∪ {r}. ISMAXIMALLYADDED(G, e, a) then
19:                    VISITCOS_{P,k}(SetRF(G|_{G.E\Deleted}, r, a), a)
20:        case _
21:            VISIT_{P,k}(G)
22: procedure VISITCOS_{P,k}(G, a)
23:     for w_p ∈ G.W_{loc(a)} do VISIT_{P,k}(SetCO(G, w_p, a))
```

4.1 Properties of TruSt

We now present some key properties of the TruSt algorithm, i.e., Algorith 1 without Line 5, that are used to prove BUSTER's correctness (Theorem 1).

From TruSt's correctness argument, we know that every SC-consistent execution G_f has exactly one sequence of VISIT$_P$ calls that leads to it. We call the sequence of the corresponding graphs a *production sequence* for G_f.

Given two SC-consistent graphs G and G', we say that G is a *prefix* of G', and write $G \sqsubseteq G'$, if $G'|_{G.\mathtt{E}} = G$. Intuitively, G is a prefix of G' if we can construct G' from G, by adding the missing events in some order for some rf and co.

Let a *maximal step* of an execution G be a execution that results from extending a thread of G by an event e in a *maximal* way, i.e., if $e \in \mathtt{R}$, then e is made to read from the co-latest event and if $e \in \mathtt{W}$, then e is placed at the end of co. We write $G \to G'$ when G' is a maximal step of G, and $G \to_e G'$ when $G \to G'$ and e is the added event. We say that a sequence of maximal steps is non-decreasing when the sequence of the thread identifiers of the added events is non-decreasing. Finally, we write $\mathtt{tid}(e)$ for the thread identifier of an event e.

A key property of TruSt (stated in Prop. 1) is that every execution G in the production sequence of an SC-consistent execution G_f is either a prefix of G_f, or it contains a read event r that does not read from the "correct" write, but there is a prefix $\hat{G}$ of G_f that can by extended to G by a non-decreasing sequence of maximal steps starting with r and not including events of at least one thread to the right of r.

Proposition 1. *Let S be the production sequence of an SC-consistent final execution G_f, and G be an execution in S. Then, either $G \sqsubseteq G_f$ or there exists an execution G_b that is before G in S, a read event $r = \mathsf{next}_P(G_b)$, a thread $t > \mathtt{tid}(r)$ and an execution $\hat{G}$ such that $G_b \sqsubseteq \hat{G} \sqsubseteq G_f|_{G_b.\mathtt{E} \cup G_f.\mathsf{cprefix}(r)}$, $G_f|_{G_f.\mathsf{cprefix}(G_f.\mathtt{rf}(r))} \not\sqsubseteq G$, there is a non-decreasing sequence of maximal steps s.t. $\hat{G} \rightarrow_r \rightarrow^* G$, and $\forall e \in G.\mathtt{E} \setminus \hat{G}.\mathtt{E}.\, \mathtt{tid}(e) \neq t$.*

Intuitively, TruSt tries to construct G_f by exploring an increasing sequence of its prefixes. This is not always possible, because when a read event r is added to G_b, the write event w that it should read from might not yet be present in G_b. In that case, r is made to read from another write and is later revisited by w leading to the execution $G'_b = G_f|_{G_b.\mathtt{E} \cup G_f.\mathsf{cprefix}(r)}$, which is a prefix of G_f. It is possible that additional backward revisit steps may happen between G_b and G'_b. Due to maximality, however, for every intermediate execution G in the production sequence between G_b and G'_b, there will be an execution $G_b \sqsubseteq \hat{G} \sqsubseteq G'_b$ that can be extended to G by a sequence of non-decreasing maximal steps. Execution $\hat{G}$ is exactly the part of G that is not deleted or revisited in a later step in S. Hence, if w is the first write that performed a backward revisit in S after G, then the events of thread $t = \mathtt{tid}(w)$ are already included in $\hat{G}$. Finally, it can be shown that t is to the right of r. The formal proof of this proposition can be found in the extended version of this paper [22].

4.2 Correctness of Slacked Bounding

To see why executions in the production sequence of a graph G_f can have at most $\mathtt{preemptions}(G_f) + N - 2$ preemptions, we start with a definition. A *witness* of a graph G is a trace of G that contains $\mathtt{preemptions}(G)$ preemptions.

Next, we observe that preemptions are monotone w.r.t. execution prefixes. That is, if an execution G requires a certain number of preemptions to be produced, a larger execution $G' \sqsupseteq G$ requires at least that many preemptions.

Lemma 1. *If G, G' are SC-consistent and $G \sqsubseteq G'$, then $\mathtt{preemptions}(G) \leq \mathtt{preemptions}(G')$.*

To prove this, take a witness of G' and restrict to the events of G, thereby obtaining a witness of G. The restriction can only remove preemptions.

Further, we note that the number of preemptions of an execution is unaffected if we extend its last executed thread with a maximal step; if a maximal step adds an event to a different thread, the number is increased by at most one.

Lemma 2. *Let G and G' be SC-consistent executions and $r \in G'.\mathtt{E}$ such that $G \rightarrow_r \rightarrow^* G'$. Then,* $\mathtt{preemptions}(G') \leq \mathtt{preemptions}(G) + S$*, where S is the number of threads that where extended to obtain G' from G.*

Proof. Consider a witness w of G and extend by appending the missing events in the same order they were added in the sequence of maximal steps. Notice that, by construction of the maximal step, the resulting sequencing is a trace of G'. Each time we add an event e in the trace, such that the last event of of the trace was not in the thread of e, we increase the preemption-bound by one: a thread was previously considered as completed, but was now extended with a new event. However, this can only happen S times: the maximal steps keep adding events of the same thread and when another thread is picked, the first is not extended again (the maximal steps are non-decreasing). This gives us a trace of G' with at most $\mathtt{preemptions}(G) + S$ preemptions, which concludes our proof. □

We can now prove that BUSTER is complete, i.e., it visits every full, SC-consistent execution that respects the bound.

Theorem 1. VERIFY(P, k) *visits every full, SC-consistent execution G_f of P with* $\mathtt{preemptions}(G_f) \leq k$.

Proof. Consider a full, SC-consistent execution G_f of P with at most k preemptions. From the completeness of TruSt, we know that a run of Algorithm 1 without the test on Line 5 will visit G_f. It thus suffices to show that for every execution G in the production sequence of G_f has at most $k + N - 2$ preemptions, where N is the number of threads of P. If $G \sqsubseteq G_f$, then from Lemma 1 $\mathtt{preemptions}(G) \leq \mathtt{preemptions}(G_f) \leq k$.

Otherwise, from Prop. 1, there exists an execution G_b that is before G in the production sequence of G_f and an execution $\hat{G}$, such that $G_b \sqsubseteq \hat{G} \sqsubseteq G_f|_{G_b.\mathtt{E} \cup G_f.\mathsf{cprefix}(r)}$, $\mathsf{next}_P(G_b) = r \in \mathtt{R}$, $G_f|_{G_f.\mathsf{cprefix}(G_f.\mathtt{rf}(r))} \not\sqsubseteq G$, $\hat{G} \rightarrow_r \rightarrow^* G$, and no events in $G.\mathtt{E} \setminus \hat{G}.\mathtt{E}$ are in thread t, for some thread t to the right of r.

From the last two properties and Lemma 2 we have $\mathtt{preemptions}(G) \leq k+N-1$ since it is $\mathtt{preemptions}(\hat{G}) \leq \mathtt{preemptions}(G_f)$ ($\hat{G} \sqsubseteq G_f$ and Lemma 1) and at most $N - 1$ threads are extended from $\hat{G}$ to G.

To complete the proof, we will prove that $\mathtt{preemptions}(G) = k + N - 1$ leads to contradiction. The equality implies that $\hat{G}$ had k preemptions and that $N-1$ threads were extended in the maximal steps from $\hat{G}$ to G, and all of them increased the preemptions by one. The sequence of maximal steps from $\hat{G}$ to G is non-decreasing and starts with the thread of r. Since there are at most N threads, $N-1$ are extended, and at least one thread to the right of t is not extended, r is in the leftmost thread.

Let t_r be the leftmost thread, $G'_b \triangleq G_f|_{G_b.\mathtt{E} \cup G_f.\mathsf{cprefix}(r)}$, and $w \triangleq G_f.\mathtt{rf}(r)$. From the proof of TruSt, we can infer that all events of G_b are in the $\mathtt{porf}$-prefix of the last event of t_r. It is $G_f|_{G_f.\mathsf{cprefix}(w)} \not\sqsubseteq G_b$: the opposite, together with $G_b \sqsubseteq \hat{G} \sqsubseteq G$, contradicts $G_f|_{G_f.\mathsf{cprefix}(w)} \not\sqsubseteq G$. Since G_b is in the production sequence of G_f, $G_b \sqsubseteq G_f$, $\mathsf{next}_P(G_b) = r$, and $G_f|_{G_f.\mathsf{cprefix}(w)} \not\sqsubseteq G_b$, TruSt will eventually add the write $w \triangleq G_f.\mathtt{rf}(r)$ and revisit the read r, reaching the

execution $G'_b \sqsubseteq G_f$ that contains all events added before r, i.e., the events of G_b, the events in the porf-prefix of r, and r. Hence, all events in $G'_b.\mathtt{E} \setminus \{r\}$ are in the porf-prefix of r, which implies that any witness of G'_b ends with r.

Since $G'_b \sqsubseteq G_f$, any witness t of G'_b has at most k preemptions. Let G' be the execution G'_b without r, and G'' the unique execution s.t. $\hat{G} \rightarrow_r G''$. Removing the last event r from t gives us a trace t' of G' with at most k preemptions. If t' ends with an event of t_r, then we can restrict t' to the events of $\hat{G}$ and add r at the end, obtaining a trace of G'' with at most k preemptions. Otherwise, t' does not end with an event of t_r, and thus trace t has one more preemption than t, i.e., t' has at most $k-1$ preemptions. Then, we can again restrict t' to the events of $\hat{G}$ and add r a the end, obtaining again a trace of G'' with at most k preemptions. This contradicts our assumption that $\mathtt{preemptions}(\hat{G}) = k$ and all $N-1$ threads that are extended from $\hat{G}$ increase the number of preemptions, since the first thread t_r can be extended without incurring any more preemptions. □

Buster inherits TruSt's optimality, as it only explores a subset of the executions that TruSt does. Here, optimality refers to avoiding redundant work; due to the slack, Verify(P, k) can also visit executions more than k preemptions.

Theorem 2. Verify(P, k) *explores each graph G of a program P at most once.*

5 Implementation

We have implemented Buster on top of the GenMC tool [18], which implements the TruSt algorithm [16]. Since GenMC supports weak memory models and the standard notion of preemption bounding only makes sense for sequential consistency, we enforce SC in our benchmarks by using only SC memory accesses and selecting GenMC's RC11 model [20].

The bulk of our modifications to GenMC concern the checking of whether the preemption-bound of an execution G exceeds a value k. Generally, deciding whether the preemption-bound of a Mazurkiewicz trace exceeds a value is an NP-complete problem [23]. We use an adaptation of the bound computation in Musuvathi et al. [23] to execution graphs, but instead of recursively computing $\mathtt{preemptions}(G)$ (and cache computations across calls to amortize the cost), we recursively compute the predicate $\Phi(G, k) \triangleq \mathtt{preemptions}(G) \leq k$. The benefit of this method is that we can avoid calculating $\mathtt{preemptions}(G)$ exactly when its value exceeds the desired bound. Furthermore, there is no additional state that needs to be stored; Buster remains stateless.

As an optimization, we use as slack (Line 5) the minimum between $N-2$ and the number of threads that have no *deletable* events; an event is not deletable if it is in the porf-prefix of a write that backward revisited. Intuitively, the events that are added in G to reach $\hat{G}$ (Prop. 1) are the events that will later be deleted to eventually reach a graph that is a prefix of the final graph G_f.

Table 1. Buggy benchmarks. An ✗ indicates that an error was found.

	$k = 0$		$k = 1$		$k = 2$		GenMC	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time
account-bad	3 ✗	0.01	3 ✗	0.01	3 ✗	0.01	3 ✗	0.01
bluetooth-driver-bad	1	0.01	3 ✗	0.02	7 ✗	0.02	8 ✗	0.01
circular-buffer-bad	2	0.07	13 ✗	0.49	1 ✗	0.03	1 ✗	0.03
din-phil-sat	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
fsbench-bad	0 ✗	0.93	0 ✗	0.93	0 ✗	0.94	0 ✗	1.01
lazy01-bad	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
queue-bad	20	1.91	56 ✗	27.47	2 ✗	0.18	2 ✗	0.19
reorder-20-bad	◷	◷	◷	◷	◷	◷	10 ✗	0.05
stack-bad	11	0.44	10 ✗	0.35	10 ✗	0.35	10 ✗	0.37
token-ring-bad	12 ✗	0.02	12 ✗	0.02	12 ✗	0.02	12 ✗	0.02
twostage-100-bad	◷	◷	◷	◷	◷	◷	◷	◷
wronglock-bad	5914	164.46	2 ✗	0.02	2 ✗	0.02	2 ✗	0.02
lazy01-unsafe	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
sigma-unsafe	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
singleton-unsafe	5 ✗	0.01	5 ✗	0.01	5 ✗	0.01	5 ✗	0.01
stateful01-1-unsafe	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
triangular-2-unsafe	6	0.04	66	0.40	368	2.06	9069 ✗	29.44
stack-2-unsafe	6	0.06	5 ✗	0.05	5 ✗	0.05	5 ✗	0.05
read-write-lock-2-unsafe	68	0.51	53 ✗	0.25	132 ✗	0.59	276 ✗	0.96
reorder-2	417	0.14	6 ✗	0.01	2 ✗	0.01	2 ✗	0.01

6 Evaluation

To evaluate Buster, we answer the following questions:

§ 6.1 How many preemptions suffice to expose common concurrency bugs? Is Buster effective at finding such concurrency bugs?
§ 6.2 How good is preemption bounding at pruning the search space? Up to what bound does Buster run faster than vanilla DPOR?
§ 6.3 What is the overhead induced by the bound calculation?
§ 6.4 What is the overhead induced by bound-blocked executions?

To that end, we evaluate Buster against GenMC on a diverse set of benchmarks. Unfortunately, we cannot include the approach of Coons et al. [10] in our comparison because their implementation is not available.

We can draw two major conclusions from our evaluation. First, most bugs do manifest with a small number of preemptions (≤ 2), an observation that has been made in the literature before [25, 27]. Second, even though the bound calculation can be fairly expensive expensive, for small bounds Buster outperforms GenMC and can find bugs faster than GenMC.

Experimental Setup We conducted all experiments on a Dell PowerEdge M620 blade system with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz) and 256GB of RAM. We used LLVM 11.0.1 for GenMC and Buster. All reported times are in seconds. We set a timeout limit of 30 minutes.

6.1 Bound and Bug Manifestation

To validate that most bugs require a small number of preemptions, we run Buster and GenMC on three sets of benchmarks:

Table 2. Buggy CD benchmarks. An ✗ indicates that the error was found.

	$k=0$		$k=1$		$k=2$		GenMC	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Exec	Time
dglm-queue-bug(6)	48 ✗	2.55	305 ✗	102.25	810 ✗	272.71	◷	◷
dglm-queue-bug(7)	54 ✗	3.94	404 ✗	209.22	1259 ✗	628.52	◷	◷
dglm-queue-bug(8)	60 ✗	5.88	517 ✗	393.02	1854 ✗	1320.58	◷	◷
ms-queue-bug(6)	84 ✗	7.71	1366 ✗	155.08	9906 ✗	1057.28	◷	◷
ms-queue-bug(7)	103 ✗	12.87	1936 ✗	294.76	◷	◷	◷	◷
ms-queue-bug(8)	124 ✗	20.72	2636 ✗	530.04	◷	◷	◷	◷
bstack(7)	2	0.24	19 ✗	1.26	83 ✗	3.55	◷	◷
bstack(8)	2	0.34	22 ✗	2.06	111 ✗	6.41	◷	◷
bstack(9)	2	0.48	25 ✗	3.23	143 ✗	10.95	◷	◷
msq-bug2(5)	2	0.09	18 ✗	0.48	154 ✗	2.69	37420 ✗	280.64
msq-bug2(6)	2	0.12	22 ✗	0.87	232 ✗	6.29	◷	◷
stack-oe-bug(4)	77	0.64	1086	17.77	375 ✗	9.66	3523 ✗	97.65
stack-oe-bug(5)	92	1.04	1700	38.25	663 ✗	23.61	17032 ✗	763.96
stack-oe-bug(6)	107	1.58	2478	74.83	1076 ✗	50.38	◷	◷
stack-oe-bug(7)	122	2.32	3435	134.89	1638 ✗	97.52	◷	◷

– the unsafe concurrent benchmarks of the SCT suite [27],
– the unsafe benchmarks of the pthread category of SV-COMP [26] included in GenMC's test suite, and
– a set of concurrent data structures (CDs) from GenMC's test suite with randomly induced bugs.

In all cases, we configure Buster to disregard any errors that occur in executions that exceed the bound and are explored due to the slack. We note that this configuration may delay bug finding, since Buster may by chance quickly come across a buggy execution with more than k preemptions (due to slack) before finding any buggy execution with up to k preemptions. Nevertheless, we follow it to ensure that the bugs found arise in executions with up to the desired number of preemptions, so as to be able to validate the claim that bugs manifest in executions with a small number of preemptions.

Table 1 reports our outcomes on the first two classes of benchmarks. As can be seen, Buster was able to find most bugs using a bound of 1. In fact, for most benchmarks, Buster found the bug before exploring a complete execution, hence the "0 ✗" entries in the table. The only benchmarks, where Buster needs a bound greater that 1 are the synthetic benchmarks `triangular`, which needs a bound of 8, as it was specifically designed to make the bug discovery difficult and push model checkers to their limits; `reorder-20` and `twostage-100`, which have a large number of threads (20 and 100, respectively). Buster times out on the latter two benchmarks because the large number of threads put a lot of stress in the bound checking procedure. We note that for `twostage-100`, GenMC also fails to terminate within the time limit.

Table 2 reports our results for our CD benchmarks. For these benchmarks, we have taken CD implementations from the GenMC test suite, and induced bugs into them by randomly dropping a synchronization instruction or replacing a CAS instruction with a normal write or an unconditional exchange instruction, thereby introducing a possible atomicity violation. We then construct medium-

Table 3. Buster and GenMC comparison on safe data structure benchmarks.

	$k=0$		$k=1$		$k=2$		$k=3$		GenMC		Max
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time	k
dglm-queue(6)	2	0.61	12	3.05	62	11.30	162	27.14	924	104.47	7
dglm-queue(7)	2	0.97	14	5.78	86	25.65	266	71.73	3432	570.68	8
ms-queue(6)	2	0.30	18	2.23	128	8.46	513	29.46	18564	321.58	8
ms-queue(7)	2	0.46	21	4.16	177	18.53	840	78.13	◷	◷	
bstack2(8)	2	0.12	16	0.58	114	2.97	408	9.17	12870	159.27	9
bstack2(9)	2	0.15	18	0.88	146	5.08	594	17.75	48620	720.06	8
bstack(5)	2	0.12	20	0.53	92	2.98	310	7.87	4214	88.01	8
bstack(6)	2	0.18	24	0.97	134	6.84	549	21.35	26040	787.64	8
ms-queue(7)	2	0.19	14	1.19	86	5.77	266	16.41	3432	135.85	7
ms-queue(8)	2	0.26	16	1.85	114	10.29	408	33.78	12870	641.64	8
stack-oe(4)	77	0.64	1098	17.62	6208	139.81	23472	641.13	◷	◷	
stack-oe(5)	92	1.06	1713	39.55	11510	377.50	◷	◷	◷	◷	
ms-oe(6)	12	0.27	84	2.93	615	18.82	2039	57.58	10880	218.86	5
ms-oe(7)	14	0.34	100	3.97	800	27.42	2855	91.54	20823	458.09	5
dglm-oe(7)	5	0.20	29	2.14	129	9.27	238	19.53	248	20.88	3
dglm-oe(8)	5	0.23	31	2.62	146	11.77	294	26.33	306	28.50	3
dglm-fifo(7)	26	4.50	128	21.84	128	25.93	128	25.12	128	22.92	1
dglm-fifo(8)	29	6.81	162	35.43	162	42.66	162	41.59	162	37.91	1
ttas-lock2(7)	2	0.12	14	0.48	86	1.89	266	4.57	3432	28.50	7
ttas-lock2(8)	2	0.17	16	0.81	114	3.66	408	10.14	12870	121.94	8
ttas-lock3(4)	21	0.89	195	7.12	1041	29.94	3525	84.55	34650	387.36	5
ttas-lock3(5)	26	2.32	320	23.97	2274	130.62	10494	492.89	◷	◷	

sized clients (with 2-3 threads and up to 12 operations per thread) of these data structures that check for their intended semantics (for example, that a queue has FIFO semantics). In all cases, the induced bugs lead to violations of the assertions in the client programs, and occasionally even to memory errors. Buster can find these bugs easily; a bound of $k = 2$ suffices to expose them. By contrast, GenMC times out for most of these benchmarks, as their state space is enormous.

6.2 Comparison with Plain DPOR on Safe Benchmarks

We have already seen that modulo specially crafted synthetic benchmarks, a small preemption bound is sufficient for finding bugs in practice. Moreover, Buster is pretty good at finding such bugs in concurrent data structures. We now evaluate the application of Buster on a collection of safe benchmarks. For this purpose, we use different variations of the benchmarks of Table 2 (after repairing them so that no assertion is violated), as well as a few locking benchmarks.

Table 3 compares the performance of Buster for small values of k and GenMC. As it can be seen, GenMC struggles with these benchmarks, whereas Buster with $k = 2$ (and often also with $k = 3$) terminates fairly quickly. This is because only a small fraction of the total executions of sizeable benchmarks have few preemptions. Therefore restricting the search to only those executions makes Buster run much faster than GenMC, and guarantees that the program under consideration does not have any common bugs.

In the last column of Table 3 we include the maximum value of k such that Buster terminates faster than GenMC, for the benchmarks that terminate

under GenMC. In most cases Buster is faster than GenMC even for $k > 3$. For the `dglm-fifo` benchmarks Buster is only faster for $k \in \{0, 1\}$, because for these benchmarks a small k suffices to fully explore the state space.

6.3 Bound Calculation Overhead

We now measure the cost of checking that each encountered execution is below the specified bound. As we discussed in §5, checking whether an execution graph's preemption-bound exceeds a value is a NP-complete problem, and thus we expect this calculation to threaten the performance of our tool.

To carefully account for this cost, we compare Buster against the baseline GenMC implementation on benchmarks where preemption bounding does not reduce the number of executions that are explored. In Line 4, we report results on simple CD clients that have only one operation per thread of the Treiber stack [28] and the TTAS lock [13]. The clients are designed so that Buster can explore the full set of program executions with a small bound k. We suffix the name of the benchmarks with the number of writer and reader threads for the Treiber stack and the total number of threads for TTAS.

Column b contains the minimal number of the bound k for which Buster explores the same number of executions as GenMC does. Note that since these benchmarks contain several threads, exploration up to a certain bound (e.g., $k = 0$) does not mean that only executions with k preemptions are visited; due to slack, executions with more preemptions may be visited, and so it is possible for the exploration to cover the entire state space for a smaller bound than intrinsically necessary. In the subsequent columns we report the time overhead (percentage) for bounds $k = b, k = b + 1$, and $k = b + 2$ w.r.t. to GenMC's execution time, which is visible on the last column. The maximum overhead is observed for $k = b$ (the minimal value sufficient to cover the entire state space). This is expected because $k = b$ places the most burden on the calculation of whether the number of preemptions in a given execution are below k. For larger k values, the overhead drops because it is easier to show that the number of preemptions are below the bound; one does not have to calculate the number of preemptions of an execution precisely. Overall, for the Treiber stack benchmark, the overhead introduced by calculating the bounds is fairly low and does not exceed the 23% of the execution time of GenMC. For the plain runs of ttas-lock, the maximal overhead is a bit larger, up to 38%. We note, however, that such overhead only occurs in clients with a large number of threads (7); smaller clients are not affected as much.

6.4 Overhead due to Bound-Blocked Executions

Finally, we measure the overhead caused by bound-blocked executions, by evaluating how often they arise in practice. Specifically, we ran Buster on GenMC's test suite for various preemption-bound values, as well as on the safe CD clients used in § 6.2, and counted the number of such bound-blocked executions.

Table 4. Overhead w.r.t. to GenMC (left) and blocking in benchmarks (right).

Benchmark	b	$k = b$	$k = b + 1$	$k = b + 2$	GenMC
treiber(6,0)	0	10%	6%	4%	30.81
treiber(7,0)	0	23%	12%	5%	529.42
treiber(3,2)	1	6%	5%	5%	2.75
treiber(3,3)	1	7%	6%	5%	31.15
treiber(3,4)	1	13%	8%	6%	332.76
treiber(4,2)	1	9%	7%	5%	47.50
treiber(4,3)	2	10%	7%	5%	777.44
ttas-lock(6)	0	20%	13%	11%	14.52
ttas-lock(7)	0	38%	25%	16%	231.91

# Blocked	# Benchmarks
0	72
1	143
2	45
3	3
4	14
5	4
6	1
8	69
>8	6

For GenMC's test suite, the results are summarized in table 4 (right). We have restricted out attention to the runs with at least 10 executions, so that our results are not skewed by benchmarks that have very few executions. We have also excluded 8 benchmarks from the test suite that use barriers because they are currently not supported by our tool. As it can be seen, bound-blocked executions are rare: most runs lead to one bound-blocked execution, and only 6 lead to more than 8 bound-blocked executions. Bound-blocked executions are on average no more than 6% of the total number of executions explored.

For the CDs clients, bound-blocked executions are even more rare; out of the 22 clients, Buster encounters bound-blocked executions in only 4 of them, for some k. We exclude again from the discussion runs with very few executions. From the remaining runs, only two encounter a considerable number of bound-blocked executions that become negligible as the bound is increased: around 10% for $k = 1$ and less than 1% for $k = 2$

7 Related Work

There is a large body of work that has improved the original DPOR algorithm of Flanagan et al. [11]. Abdulla et al. [2] introduced the first optimal DPOR algorithm, which, however, suffers from possibly exponential memory consumption. Kokologiannakis et al. [16] developed TruSt, which is the first optimal DPOR algorithm that consumes polynomial memory.

Agarwal et al. [6], Chalupa et al. [8], Chatterjee et al. [9], and Huang [14] have extended DPOR for partitions coarser than the one we have focused in this paper, i.e., Mazurkiewicz traces. Abdulla et al. [1, 4, 5] consider DPOR under various weak memory models, while the works of Kokologiannakis et al. [16, 17, 19] provide a DPOR algorithm that is parametric in the choice of the memory model, provided it respects some basic properties.

Qadeer et al. [25] showed the decidability of context-bound verification of concurrent boolean programs. Musuvathi et al. [24] propose *iterative* context bounding, a search algorithm that prioritizes executions with fewer preemptions. Musuvathi et al. [23] combine partial-order reduction with a preemption-bound search, and prove that judging whether the preemption-bound of a Mazurkiewicz trace exceeds a certain value is an NP-complete problem.

To our knowledge, the only attempt to combine DPOR and preemption bounding is by Coons et al. [10], who identify the difficulty of maintaining completeness of the exploration, and resolve it by weakening DPOR.

Abdulla et al. [3] and Atig et al. [7] have extended the notion of preemption bounding to weak memory models. We leave a possible extension of our approach to weak memory models for future work.

Acknowledgments We thank the anonymous reviewers for their valuable feedback. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

8 Data-Availability Statement

All supplementary material is available at [22]. The artifact is also available at [21].

References

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. "Stateless model checking for TSO and PSO". In: *TACAS 2015*. Vol. 9035. LNCS. Berlin, Heidelberg: Springer, 2015, pp. 353–367. DOI: 10.1007/978-3-662-46681-0_28. URL: http://dx.doi.org/10.1007/978-3-662-46681-0_28.

[2] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. "Optimal dynamic partial order reduction". In: *POPL 2014*. New York, NY, USA: ACM, 2014, pp. 373–384. DOI: 10.1145/2535838.2535845. URL: http://doi.acm.org/10.1145/2535838.2535845.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. "Context-Bounded Analysis for POWER". In: *TACAS 2017*. Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 56–74. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_4.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. "Stateless model checking for POWER". In: *CAV 2016*. Vol. 9780. LNCS. Berlin, Heidelberg: Springer, 2016, pp. 134–156. DOI: 10.1007/978-3-319-41540-6_8. URL: https://doi.org/10.1007/978-3-319-41540-6_8.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. "Optimal stateless model checking under the release-acquire semantics". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 135:1–135:29. ISSN: 2475-1421. DOI: 10.1145/3276505. URL: http://doi.acm.org/10.1145/3276505.

[6] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. "Stateless Model Checking Under a Reads-Value-From Equivalence". In: *CAV 2021*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, July 2021, pp. 341–366. ISBN: 978-3-030-81685-8. DOI: 10.1007/978-3-030-81685-8_16.

[7] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. "Context-Bounded Analysis of TSO Systems". In: *FPS 2014*. Ed. by Saddek Bensalem, Yassine Lakhneck, and Axel Legay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 21–38. ISBN: 978-3-642-54848-2. DOI: 10.1007/978-3-642-54848-2_2.

[8] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. "Data-centric dynamic partial order reduction". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 31:1–31:30. ISSN: 2475-1421. DOI: 10.1145/3158119. URL: http://doi.acm.org/10.1145/3158119.

[9] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. "Value-Centric Dynamic Partial Order Reduction". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360550. URL: https://doi.org/10.1145/3360550.

[10] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. "Bounded Partial-Order Reduction". In: *OOPSLA 2013*. Indianapolis, Indiana, USA: ACM, 2013, pp. 833–848. ISBN: 9781450323741. DOI: 10.1145/2509136.2509556. URL: https://doi.org/10.1145/2509136.2509556.

[11] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software". In: *POPL 2005*. New York, NY, USA: ACM, 2005, pp. 110–121. DOI: 10.1145/1040305.1040315. URL: http://doi.acm.org/10.1145/1040305.1040315.

[12] Patrice Godefroid. "Model checking for programming languages using VeriSoft". In: *POPL 1997*. Paris, France: ACM, 1997, pp. 174–186. DOI: 10.1145/263699.263717. URL: http://doi.acm.org/10.1145/263699.263717.

[13] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. 2008.

[14] Jeff Huang. "Stateless model checking concurrent programs with maximal causality reduction". In: *PLDI 2015*. New York, NY, USA: ACM, 2015, pp. 165–174. DOI: 10.1145/2737924.2737975. URL: http://doi.acm.org/10.1145/2737924.2737975.

[15] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. "Effective stateless model checking for C/C++ concurrency". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 17:1–17:32. ISSN: 2475-1421. DOI: 10.1145/3158105. URL: http://doi.acm.org/10.1145/3158105.

[16] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. "Truly stateless, optimal dynamic partial order reduction". In:

Proc. ACM Program. Lang. 6.POPL (Jan. 2022). DOI: 10.1145/3498711. URL: https://doi.org/10.1145/3498711.

[17] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. "Model checking for weakly consistent libraries". In: *PLDI 2019*. New York, NY, USA: ACM, 2019. DOI: 10.1145/3314221.3314609.

[18] Michalis Kokologiannakis and Viktor Vafeiadis. "GenMC: A model checker for weak memory models". In: *CAV 2021*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. LNCS. Springer, 2021, pp. 427–440. DOI: 10.1007/978-3-030-81685-8_20.

[19] Michalis Kokologiannakis and Viktor Vafeiadis. "HMC: Model checking for hardware memory models". In: *ASPLOS 2020*. ASPLOS '20. Lausanne, Switzerland: ACM, 2020, pp. 1157–1171. ISBN: 9781450371025. DOI: 10.1145/3373376.3378480. URL: https://doi.org/10.1145/3373376.3378480.

[20] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. "Repairing sequential consistency in C/C++11". In: *PLDI 2017*. Barcelona, Spain: ACM, 2017, pp. 618–632. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062352. URL: http://doi.acm.org/10.1145/3062341.3062352.

[21] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. "Reconciling Preemption Bounding with DPOR (artifact)". In: (Apr. 2023). DOI: 10.5281/zenodo.7505917.

[22] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. "Reconciling Preemption Bounding with DPOR (supplementary material)". In: (Apr. 2023). URL: https://plv.mpi-sws.org/genmc.

[23] Madalan Musuvathi and Shaz Qadeer. *Partial-Order Reduction for Context-Bounded State Exploration*. Tech. rep. MSR-TR-2007-12. Microsoft Research, 2007. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2007-12.pdf.

[24] Madanlal Musuvathi and Shaz Qadeer. "Iterative Context Bounding for Systematic Testing of Multithreaded Programs". In: *PLDI 2007*. San Diego, California, USA: ACM, 2007, pp. 446–455. ISBN: 9781595936332. DOI: 10.1145/1250734.1250785. URL: https://doi.org/10.1145/1250734.1250785.

[25] Shaz Qadeer and Jakob Rehof. "Context-Bounded Model Checking of Concurrent Software". In: *TACAS 2005*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. LNCS. Springer, 2005, pp. 93–107. DOI: 10.1007/978-3-540-31980-1_7. URL: https://doi.org/10.1007/978-3-540-31980-1%5C_7.

[26] SV-COMP. *Competition on Software Verification (SV-COMP)*. 2019. URL: https://sv-comp.sosy-lab.org/2019/ (visited on 03/27/2019).

[27] Paul Thomson, Alastair F. Donaldson, and Adam Betts. "Concurrency testing using schedule bounding: an empirical study". In: *PPoPP 2014*. ACM, 2014, pp. 15–28. DOI: 10.1145/2555243.2555260. URL: https://doi.org/10.1145/2555243.2555260.

[28] R. Kent Treiber. *Systems Programming: Coping with Parallelism*. Tech. rep. Technical Report RJ5118, IBM, 1986. URL: https://dominoweb.draco.res.ibm.com/58319a2ed2b1078985257003004617ef.html.


Optimal Stateless Model Checking for Causal Consistency

Parosh Abdulla[1], Mohamed Faouzi Atig[1], S. Krishna[2], Ashutosh Gupta[2], and Omkar Tuppe[2](✉)

[1] Uppsala University, Uppsala, Sweden
{parosh,mohamed_faouzi.atig}@it.uu.se
[2] IIT Bombay, Mumbai, India
{krishnas,akg,omkarvtuppe}@cse.iitb.ac.in

Abstract. We present a framework for efficient stateless model checking (SMC) of concurrent programs under three prominent models of causal consistency, `CCv`, `CM`, `CC`. Our approach is based on exploring traces under the program order *po* and the reads from *rf* relations. Our SMC algorithm is provably optimal in the sense that it explores each *po* and *rf* relation exactly once. We have implemented our framework in a tool called Conschecker. Experiments show that Conschecker performs well in detecting anomalies in classical distributed databases benchmarks.

1 Introduction

Traditionally, distributed shared memories ensure that all processes in the system agree on a common order of all operations on memory. Such guarantees are provided by sequential consistency (SC) [33], and by linearizable memory [26]. However, providing these consistency guarantees entails access latencies, making them inefficient for large systems. There is a tradeoff in providing strong consistency guarantees while ensuring low latency and this presents significant efficiency challenges. There is a large body of work which suggests that a systematic weakening of memory consistency can reduce the costs of providing consistency. Weakened consistency guarantees admit more concurrent behaviours than SC or linearizability. To this end, Lamport [32] proposed *causal consistency* which provides an ordering among events in a distributed system in which processes communicate via message passing. This has been adapted [7] to a setting of reads and writes in a shared memory environment. In this setting, the return values of reads must be consistent with causally related reads and writes. As causality only orders events partially, the reading processes can disagree on the relative ordering of concurrent writes. This makes concurrent writer processes independent, reducing the costs of synchronization.

Several efforts have been made to formalize causal consistency [16], [25], [39] [40], [7], [15], [10], [8], [38] and there are many implementations [9], [20], [21] satisfying this criterion as opposed to strong consistency (linearizability).

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 105–125, 2023.
https://doi.org/10.1007/978-3-031-30823-9_6

While strong consistency makes it easier to program than weak ones, they require costly implementations. Weak memories may be easier to implement, but much harder to program. An acceptable medium which has emerged over the years are three important notions in causal consistency, respectively *causal consistency* (CC) [15], [25], *causal convergence* (CCv) [16], [39], [15], [25] and *causal memory* (CM) [7], [39], [15], [25].

The focus of this paper is the verification of shared memory programs under causal consistency. We consider the three variants mentioned above. We propose a stateless model checking (SMC) framework that covers all three variants. SMC is a successful technique for finding concurrency bugs [23]. For a terminating program, SMC systematically explores all process schedulings that are possible during runs of the program. The number of possible schedulings grows exponentially with the execution length in SMC. To counter this and reduce the number of explored executions, the technique of *partial order reduction* [18,22] has been proposed. This has been adapted to SMC as DPOR (dynamic partial order reduction). DPOR was first developed for concurrent programs under SC [1,41]. Recent years have seen DPOR adapted to language induced weak memory models [28,37],[5], as well as hardware-induced relaxed memory models [3,46]. To the best of our knowledge, DPOR algorithms have not been developed for causal consistency models. The goal of this paper is to fill this gap.

DPOR is based on the observation that two executions are equivalent if they induce the same ordering between conflicting events, and hence it is sufficient to consider one such execution from each equivalence class. Under sequential consistency, these equivalence classes are called *Mazurkiewicz* traces [34], while for relaxed memory models, the generalization of these are called *Shasha-Snir* traces [42]. A Shasha-Snir trace characterizes an execution of a concurrent program by the relations (1) *po* program order, which totally orders events of each process, (2), *rf* reads from, which connects each read with the write it reads from, (3) *co* coherence order, which totally orders writes to the same shared variable. DPOR can be optimized further by observing that the assertions to be verified at the end of an execution does not depend on the coherence order of shared variables, and hence it suffices to consider traces over $po - rf$. Based on this observation, the DPOR algorithms for programs under the release-acquire semantics (RA) and SC [5], [4] explores traces with *po*, *rf* and *co* where the *co* edges are added on the fly. The equivalence classes are considered wrt $po - rf$, reducing the number of distinct traces to be analyzed.

Contributions. We propose a DPOR based SMC algorithm for all three consistency models CC, CCv, CM which explores systematically, all the distinct *po*-*rf* traces covering all possible executions of the program. We develop a uniform algorithm for all three models which is sound and complete : that is, all traces explored are consistent wrt the model $X \in \{\texttt{CC}, \texttt{CCv}, \texttt{CM}\}$ under consideration, and all such consistent traces are explored. Moreover, our algorithm is optimal in the sense that, each consistent *po*-*rf* trace is explored exactly once. One of the key challenges during the trace exploration is to maintain the consistency of the traces wrt the model under consideration. We tackle this by defining a *trace se-*

mantics which ensures that the traces generated in each step only contain edges which will be present in any consistent trace. We implement our algorithms in a tool Conschecker which is, to the best of our knowledge, the first of its kind to perform SMC on the three prominent causal consistency models CC, CCv, CM. Conschecker checks for assertion violation of programs under CC, CCv, CM. We evaluate the correctness of our tool on CC, CCv, CM by simulating these models on the memory model simulator Herd [8] and validating our outcomes with theirs. Then we proceed with experimental evaluation on a wide range of benchmarks from distributed databases. We showed that (i) Conschecker correctly detects known consistency bugs [13], [14], [12] and [11] under CCv, CM, CC, (ii) Conschecker correctly detects known assertion violations in applications [19], [27], [12], [36]. We also did a stress test of Conschecker on some SV-COMP benchmarks and parameterized benchmarks which resulted in a large number (6 million) of traces.

Related Work. SMC has been implemented in many tools CHESS [35], Concuerror [17], VeriSoft [24], Nidhugg [3], CDSChecker [37], RCMC[28], GenMC [30], rInspect [46] and Tracer [5]. While most of these work with either Mazurkewicz traces or $po - rf$ traces, [6] proposes a RVF-SMC algorithm where the value read is used to decide equivalence of two runs.

In recent years, there has been much interest in DPOR algorithms : [4] for SC, [30] for the release acquire semantics, [43] for C/C++, and [29] for TSO, PSO and RC11. It is known that CC is weaker than RA, CCv is stronger than RA while CM is incomparable with RA [31]. In conclusion, all the above memory models are different from CC, CCv, CM. Hence we cannot reuse any of the existing DPOR algorithms.

Recent work on causal consistency [15] studies the complexity of checking whether one execution (all executions) of a program under CC, CCv, CM is consistent. They show that checking if an execution is consistent is NP-completeness, while the question of checking if all executions are consistent is undecidable. [11], [12] explore the robustness wrt SC, of transactional programs under CC, CCv, CM. However, none of these papers propose a DPOR algorithm for CC, CCv, CM.

2 Preliminaries

Programs We consider a program $\mathcal{P}$ consisting of a finite set $\mathcal{T}$ of *threads* (*processes*) that share a finite set $\mathbb{X}$ of *(shared) variables*, ranging over a domain $\mathbb{V}$ of *values* that includes a special value 0.

A process has a finite set of local registers that store values from $\mathbb{V}$. Each process runs a deterministic code, built in a standard way from expressions and atomic commands, using standard control flow constructs (sequential composition, selection, and bounded loop constructs). Throughout the paper, we use x, y for shared variables, a, b, c for registers, and e for expressions. *Global statements* are either writes $x := e$ to a shared variable, or reads a $:= x$ from a shared variable. *Local statements* only access and affect the local state of the process and include assignments a $:= e$ to registers, and conditional control flow constructs.

Note that expressions do not contain shared variables, implying that a statement accesses at most one shared variable.

The local state of a process $proc \in \mathcal{T}$ is defined by its program counter and the contents of its registers. A *configuration* of $\mathcal{P}$ is made up of the local states of all the processes. The values of the shared variables are not part of a configuration. A program execution is a sequence of transitions between configurations, starting with the initial configuration γ^{init}. Each transition corresponds to one process performing a local or global statement. A transition between two configurations γ and γ' is of form $\gamma \overset{\ell}{\rightsquigarrow} \gamma'$, where the label ℓ describes the interaction with shared variables. The label ℓ is one of three forms: (i) $\langle proc, \varepsilon \rangle$, indicating a local statement performed by thread *proc*, which updates only the local state of *proc*, (ii) $\langle proc, \mathsf{wt}, x, v \rangle$, indicating a write of the value v to the variable x by the thread *proc*, which also updates the program counter of *proc*, and (iii) $\langle proc, \mathsf{rd}, x, v \rangle$ indicating a read of v from x by the thread *proc* into some register, while also updating the program counter of *proc*. There is no constraint on the values that are used in transitions corresponding to read statements. This will allow some illegal program behaviors, which is sorted by associating runs with so-called *traces*, which represent how reads obtain their values from writes. A causal consistency model $X \in \{\mathtt{CC}, \mathtt{CCv}, \mathtt{CM}\}$ is formulated by imposing restrictions on traces, thereby also restricting the possible runs that are associated with them.

Since local statements are not visible to other threads, we will not represent them explicitly in the transition relation considered in our DPOR algorithm. Instead, we let each transition represent the combined effect of some finite sequence of local statements by a process followed by a global statement by the same process. For configurations γ and γ' and a label ℓ which is either of the form $\langle proc, \mathsf{wt}, x, v \rangle$ or of the form $\langle proc, \mathsf{rd}, x, v \rangle$, we let $\gamma \xrightarrow{\ell} \gamma'$ denote that we can reach γ' from γ by performing a sequence of transitions labeled with $\langle proc, \varepsilon \rangle$ followed by a transition labeled with ℓ. Defining the relation $\rightarrow$ in this manner ensures that we take the effect of local statements into account, while avoiding consideration of interleavings of local statements of different threads in the analysis.

We use $\gamma \rightarrow \gamma'$ to denote that $\gamma \xrightarrow{\ell} \gamma'$ for some ℓ and define $\mathtt{succ}(\gamma) := \{\gamma' \mid \gamma \rightarrow \gamma'\}$, i.e., it is the set of successors of γ wrt. $\rightarrow$. A configuration γ is said to be *terminal* if $\mathtt{succ}(\gamma) = \emptyset$, i.e., no thread can execute a global statement from γ. A *run* ρ from γ is a sequence $\gamma_0 \xrightarrow{\ell_1} \gamma_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} \gamma_n$ such that $\gamma_0 = \gamma$. We say that ρ is *terminated* if γ_n is terminal. We let $Runs(\gamma)$ denote the set of runs from γ.

Events. An event corresponds to a particular execution of a statement in a run of $\mathcal{P}$. A *write event* ev is given by $(id, proc, \mathsf{wt}(x, v))$ where $id \in \mathbb{N}$ is the identifier of the event, *proc* is the process containing the event, $x \in \mathbb{X}$ is a variable, and $v \in \mathbb{V}$ is a value. This event corresponds to a process writing the value v to variable x. Likewise, a *read event* ev is given by $(id, proc, \mathsf{rd}(x))$ where $x \in \mathbb{X}$. This event corresponds to a process reading some value to x. The read event ev does not specify the particular value it reads; this value will be defined in a trace by specifying a write event from which ev fetches its value. For each

variable $x \in \mathbb{X}$, we assume a special write event $\mathsf{init}_x = \mathsf{wt}(x, 0)$ called the *initializer* event for x. This event is not performed by any of the processes in $\mathcal{T}$, and writes the value 0 to x. We define $\mathtt{E}_{init} := \{\mathsf{init}_x \mid x \in \mathbb{X}\}$ as the set of initializer events. If $\mathtt{E}$ is a set of events, we define subsets of $\mathtt{E}$ characterized by particular attributes of its events. For instance, for a variable x, we let $\mathtt{E}^{\mathsf{wt},x}$ denote $\{ev \in \mathtt{E} \mid ev.type = \mathsf{wt} \wedge ev.var = x\}$.

Traces. A *trace* τ is a tuple $\langle \mathtt{E}, po, rf \rangle$, where $\mathtt{E}$ is a set of *events* which includes the set $\mathtt{E}_{init}$ of initializer events, and po (program order), rf (read-from) are binary relations on $\mathtt{E}$ that satisfy:

- ev po ev' if $process(ev) = process(ev')$ and $ev.id < ev'.id$. po totally orders the events of each individual process.
- ev rf ev' if ev is a write event and ev' is a read event on the same variable, which obtains its value from ev.

We can view $\tau = \langle \mathtt{E}, po, rf \rangle$ as a graph whose nodes are $\mathtt{E}$ and whose edges are defined by the relations po, rf. po depicted by red solid edges captures the order in each process while rf edges are depicted as solid blue edges. We define the *empty trace* $\tau_\emptyset := \langle \mathtt{E}_{init}, \emptyset, \emptyset \rangle$, i.e., it contains only the initializer events, and all the relations are empty.

We define when a trace can be associated with a run. Consider a run ρ of form $\gamma_0 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n} \gamma_n$, where $\ell_i = \langle proc_i, \mathsf{t}_i, x_i, v_i \rangle$, and let $\tau = \langle \mathtt{E}, po, rf \rangle$ be a trace. We write $\rho \models \tau$ to denote that the following conditions are satisfied: (i) $\mathtt{E} = \{ev_1, \ldots, ev_n\}$, i.e., each event corresponds exactly to one label in ρ. (ii) If $\ell_i = \langle proc_i, \mathsf{wt}, x_i, v_i \rangle$, then $ev_i = \langle id_i, proc_i, \mathsf{wt}, x_i, v_i \rangle$, and if $\ell_i = \langle proc_i, \mathsf{rd}, x_i, v_i \rangle$, then $ev_i = \langle id_i, proc_i, \mathsf{rd}, x_i \rangle$. An event and its label do the same (write or read) on identical variables, and for writes, they also agree on the written value. (iii) $id_i = |\{j \mid (1 \leq j \leq i) \wedge (proc_j = proc_i)\}|$. $ev.id$ shows how it is ordered relative to the other events of $process(ev)$. (iv) if ev_i rf ev_j then $x_i = x_j$ and $v_i = v_j$. (v) if init_x rf ev_i then $v_i = 0$, i.e., ev_i reads the initial value of x which is 0.

3 Causally Consistent Models

We study three variants [15] of causal consistency : $\mathtt{CC}, \mathtt{CCv}$ and $\mathtt{CM}$. To define the three models formally, we introduce a function that, for each model, extends a given trace uniquely by a set of new edges. Then we define the model by requiring that the extended trace does not contain any cycles. A run of the program satisfies a consistency model X if its associated extended trace has no cycles.

Let CO, called *causality order* represent $(po \cup rf)^+$. Two events e_1, e_2 are *causally related* if either e_1 CO e_2 or e_2 CO e_1.

Causal Consistency $\mathtt{CC}$. We start presenting the weakest notion of causal consistency, $\mathtt{CC}$ [25], [7]. First we give an intuitive description of $\mathtt{CC}$. In $\mathtt{CC}$, events which are not causally related can be executed in different orders in different processes; moreover decisions made about these orders can be revised by each process. To illustrate, consider the program Fig.1(b). The write events $\mathsf{wt}(x, 1), \mathsf{wt}(x, 2)$ are not causally related and hence can be ordered in any way.

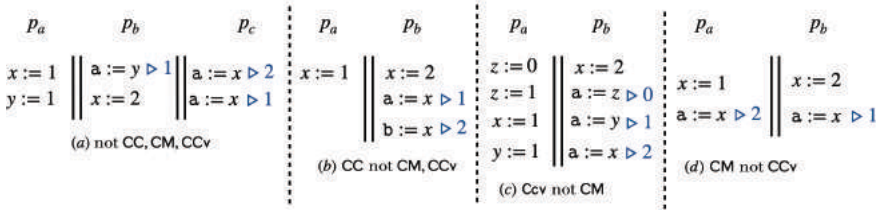


Fig. 1. Programs showing the differences between consistency models. The $\triangleright v$ denotes the expected return value of the read event.

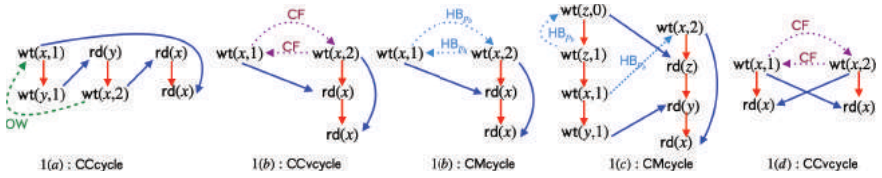


Fig. 2. solid red, blue edges are po,rf, $\mathsf{wt}(x, v)$ and $\mathsf{rd}(x)$ are write, read events.

Note that p_b first orders $x := 1$ after $x := 2$ and reads 1 into a; it then revises this order, and orders $x := 2$ after $x := 1$ and reads 2 into b.

A trace τ does not violate CC as long as there is a causality order which explains the return value of each read event.

To capture traces violating CC, we define a relation OW (for overwrite) on writes to the same variable. For any two writes w_1, w_2 and a read r on a same variable, if w_1 CO w_2 CO r, and w_1 rf r, then w_2 OW w_1. This says that r reads the overwritten write w_1, resulting in a CO $\cup$ OW cycle. We refer to CO $\cup$ OW cycles as CCcycle. We define a function $\mathsf{extend}_{\mathtt{CC}}(\tau)$ which extends a trace $\tau = \langle \mathtt{E}, po, rf \rangle$ by adding all possible OW edges between write events on the same variable. For a trace $\tau = \langle \mathtt{E}, po, rf \rangle$, we say that $\tau \models \mathtt{CC}$ iff $\mathsf{extend}_{\mathtt{CC}}(\tau)$ does not have a CCcycle.

Examples. Program Fig. 1(a) is not CC since there is no causality order which explains the return values of the read events. If we consider any trace (Fig. 2) of the program Fig.1(a), we find that $\mathsf{wt}(y,1)$ rf r where $r = \mathsf{rd}(y)$, $\mathsf{wt}(x,1)$ po $\mathsf{wt}(y,1)$, r po $\mathsf{wt}(x,2)$. Then we get $\mathsf{wt}(x,1)$ CO $\mathsf{wt}(x,2)$, $\mathsf{wt}(x,2)$ CO r' where $r' = \mathsf{rd}(x)$ and $\mathsf{wt}(x,1)$ rf r' giving $\mathsf{wt}(x,2)$ OW $\mathsf{wt}(x,1)$ witnessing CCcycle.

Causal Convergence CCv. Under CCv, we need a total order on all write events per variable. This order, called *arbitration order*, is an abstraction of how conflicts are resolved by all processes to agree upon one ordering among events which are not causally related. Thus, unlike CC, a process cannot revise its ordering of the events which are not causally related, and all processes must follow one ordering. This makes it stronger than CC.

To enforce a total order between all writes, we use a new relation CF called conflict relation on all write events per variable. For all variables $x \in \mathcal{V}$, and writes w_1, w_2 on x and a read $r = \mathsf{rd}(x)$, if w_1 CO r, and w_2 rf r then w_1 CF w_2.

We define a function $\mathsf{extend}_{\mathtt{CCv}}(\tau)$ which extends a trace $\tau = \langle \mathtt{E}, po, rf \rangle$ by adding all possible OW, CF edges between write events on the same variable. Traces violating CCv exhibit a CO ∪ CF ∪ OW cycle in $\mathsf{extend}_{\mathtt{CCv}}(\tau)$, which we refer to as CCvcycle. We say that $\tau \models \mathtt{CCv}$ iff $\mathsf{extend}_{\mathtt{CCv}}(\tau)$ does not contain a CCvcycle.

Examples. For the program Fig.1(b) and any trace τ, $\mathsf{extend}_{\mathtt{CCv}}(\tau)$ has a CCvcycle (see Fig.2) since in any trace, we have $w_1 = \mathsf{wt}(x, 1)$ CO r_2 where $r_2 = \mathsf{rd}(x)$ and w_2 rf r_2 for $w_2 = \mathsf{wt}(x, 2)$ giving w_1 CF w_2. We also have $\mathsf{wt}(x, 2)$ CO r_1 where $r_1 = \mathsf{rd}(x)$ with w_1 rf r_1 giving w_2 CF w_1. Intuitively, we cannot find a total order amongst the writes to justify the reads of 1 and 2.

However, the program Fig.1(c) has a trace τ s.t. $\mathsf{extend}_{\mathtt{CCv}}(\tau)$ does not have CCvcycle. In the corresponding run, we first allow p_a to complete execution, followed by p_b.

Causal Memory CM. The CM model is stronger than CC and incomparable to CCv. Like CC, in CM also, a process can diverge from another one in its ordering of events which are not causally related. However, once a process chooses an ordering of such events, it cannot revise it; this makes it stronger than CC and incomparable to CCv.

A *happened before* relation per process fixes the per process ordering of events. For a read/write event e in a trace, the *Causal Past* of e, $\mathsf{CausalPast}(e) = \{e' \mid e'$ CO $e\}$ is the set of events which are in the causal past of e. For an event e, the happened before relation HB_e [15] is the smallest relation on events which is transitive, and is such that for all events $e_1, e_2 \in \mathsf{CausalPast}(e)$, e_1 CO $e_2 \Rightarrow e_1$ HB_e e_2. In other words, $\mathsf{CO}_{|\mathsf{Ca\ salPast}(e)} \subseteq \mathsf{HB}_e$: HB_e contains all pairs of events obtained by restricting CO to the events in the causal past of e. For any variable x, if we have writes w_1, w_2 on x and a read $r_2 = \mathsf{rd}(x)$ such that
(i) $r_2 = e$ or r_2 po e, w_2 rf r_2, and w_1 HB_e r_2, then w_1 HB_e w_2, and
(ii) if w_1 HB_e w_2 and w_1 rf r_2, then r HB_e w_2.

Let e_p be the po-last event of process p: that is, for all events e in process p, $e = e_p$ or e po e_p. Since $\mathsf{HB}_e \subseteq \mathsf{HB}_{e_p}$ for all events e in process p, HB_{e_p} fixes the ordering among all causally unrelated events for process p. We write HB_p instead of HB_{e_p}.

We define a function $\mathsf{extend}_{\mathtt{CM}}$ which extends a trace $\tau = \langle \mathtt{E}, po, rf \rangle$ by adding all possible OW, HB_p edges for all processes p. Traces violating CM exhibit a OW ∪ HB_p cycle, called a CMcycle in $\mathsf{extend}_{\mathtt{CM}}(\tau)$ for some process p. We say that $\tau \models \mathtt{CM}$ iff $\mathsf{extend}_{\mathtt{CM}}(\tau)$ does not contain a CMcycle. See Figure 3 which motivates conditions (i), (ii) to add HB edges so that $\mathsf{extend}_{\mathtt{CM}}(\tau)$ does not contain CMcycle.

Examples. For the program Fig.1(c) and any trace τ, $\mathsf{extend}_{\mathtt{CM}}(\tau)$ contains CM cycle. Consider the read event $o_{p_b} = \mathsf{rd}(x)$ with $\mathsf{wt}(x, 2)$ rf o_{p_b}. Then $\mathsf{wt}(x, 1)$ po $\mathsf{wt}(y, 1)$ rf $\mathsf{rd}(y)$ po o_{p_b}, that is, $\mathsf{wt}(x, 1)$ CO o_{p_b}. This induces $\mathsf{wt}(x, 1)$ HB_{p_b} o_{p_b}, and $\mathsf{wt}(x, 1)$ HB_{p_b} $\mathsf{wt}(x, 2)$. This results in $\mathsf{wt}(z, 1)$ po $\mathsf{wt}(x, 1)\mathsf{HB}_{p_b}$ $\mathsf{wt}(x, 2)$ po r where $r = \mathsf{rd}(z)$ with $\mathsf{wt}(z, 0)$ rf r. This gives $\mathsf{wt}(z, 1)$ HB_{p_b} $\mathsf{wt}(z, 0)$ resulting in a cycle. However, program Fig.1(d) has a trace τ s.t. $\mathsf{extend}_{\mathtt{CM}}(\tau)$ does not contain CMcycle.

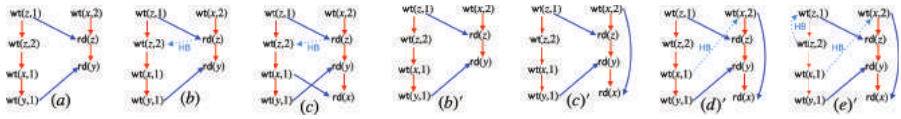


Fig. 3. Start with (a). In (b) we add the HB edge from $\mathsf{rd}(z)$ to $\mathsf{wt}(z,2)$ following condition (ii). Then (c) is obtained on adding $\mathsf{rd}(x), \mathsf{wt}(x,1)$ *rf* $\mathsf{rd}(x)$. In contrast, (b)' does not follow condition (ii). Hence, when the $\mathsf{rd}(x)$ is added in (c)', $\mathsf{wt}(x,2)$ is available to be read. Choosing $\mathsf{wt}(x,2)$ *rf* $\mathsf{rd}(x)$ necessitates adding $\mathsf{wt}(x,1)$ HB $\mathsf{wt}(x,2)$ in (d)' by condition (i). This necessitates adding $\mathsf{wt}(z,2)$ HB $\mathsf{wt}(z,1)$ in (e)' creating CMcycle.

A run ρ satisfies a model $X \in \{\mathtt{CC}, \mathtt{CCv}, \mathtt{CM}\}$ if there exists a trace τ such that $\rho \models \tau$ and $\tau \models X$. Define $\gamma_X := \{\tau_X \mid \exists \rho \in \mathit{Runs}(\gamma).\rho \models \tau_X \wedge \tau_X \models X\}$, the set of traces generated under X from a given configuration γ.

Note. Similar to our characterization of bad traces using cycles, [15] uses bad patterns in *differentiated histories* to capture violations of $\mathtt{CC}, \mathtt{CCv}, \mathtt{CM}$. Differentiated histories are posets labeled with $\mathsf{wt}(x,v)$ and $\mathsf{rd}(x) \triangleright v$ such that no two events $\mathsf{wt}(x,v_1)$ and $\mathsf{wt}(x,v_2)$ have $v_1 = v_2$. Bad patterns are characterized in [15] using the *po* and reads from relations on differentiated histories. Since we work with traces having *po* and *rf*, we do not require differentiated writes.

4 Trace Semantics

To analyse a program $\mathcal{P}$ under a model $X \in \{\mathtt{CC}, \mathtt{CCv}, \mathtt{CM}\}$, all runs of $\mathcal{P}$ must be explored. We do this by exploring the associated traces. In fact, two runs having the same associated traces are equivalent since the assertions to be checked at the end of a run depend only on *po*, *rf*. We begin with the empty trace, and continue exploration by adding enabled read/write events to the traces generated so far. While doing this, we must ensure that the generated traces τ are s.t. $\tau \models X$. We present two efficient operations to add a new read/write event to a trace τ obtaining a trace τ' so that $\mathsf{extend}_X(\tau')$ does not contain a Xcycle. We discuss two notions that are relevant while adding a new read event to a trace.

Readability and Visibility. For all 3 models, readability identifies the write events w from which a newly added read r can fetch its value. Visibility is used to add, in the case of CCv, new CF edges (and in the case of CM, new HB edges) that are implied by the fact that the new read event reads from w. Let $\tau = \langle \mathtt{E}, po, rf \rangle$ be a trace, and $\tau_X = \mathsf{extend}_X(\tau)$. Let τ_X^r denote adding r to τ_X. We define the readable set $\mathtt{readable}(\tau_X^r, r, x)$ for read event r from process p on variable x.

1. For $X = \mathtt{CC}$, $\mathtt{readable}(\tau_X^r, r, x)$ is defined as the set of all write events $w \in \mathtt{E}^{\mathsf{wt},x}$ s.t. there is no write $w' \in \mathtt{E}^{\mathsf{wt},x}$, s.t. w CO w' CO r in τ_X^r.
2. For $X = \mathtt{CCv}$, $\mathtt{readable}(\tau_X^r, r, x)$ is defined as the set of all write events $w \in \mathtt{E}^{\mathsf{wt},x}$ s.t. there is no write $w' \in \mathtt{E}^{\mathsf{wt},x}$ s.t. w $(\mathsf{CO} \cup \mathsf{CF})^+$ w' CO r in τ_X^r.
3. For $X = \mathtt{CM}$, $\mathtt{readable}(\tau_X^r, r, x)$ is defined as the set of all write events $w \in \mathtt{E}^{\mathsf{wt},x}$ s.t. there is no write $w' \in \mathtt{E}^{\mathsf{wt},x}$ s.t. w HB_p w' HB_p r in τ_X^r.

Intuitively, $\texttt{readable}(\tau^r_X, r, x)$ contains all write events which are not hidden in τ^r_X by other writes on x. The newly added read event r can fetch its value from a write in $\texttt{readable}(\tau^r_X, r, x)$. The visible set $\texttt{visible}(\tau^r_X, r, x)$ is defined as the set of events in $\texttt{readable}(\tau^r_X, r, x)$ which can "reach" r in τ^r_X. Let τ^{rw} denote the trace obtained by adding r and w rf r to trace τ.

1. For $X = \texttt{CCv}$, $\texttt{visible}(\tau^r_X, r, x) = \{e \in \texttt{readable}(\tau^r_X, r, x) \mid e \ \mathsf{CO}\ r\}$. The point of $\texttt{visible}(\tau^r_X, r, x)$ is that when the new read r is added, which reads from a write w, $\mathsf{extend}_X(\tau^{rw})$ will not contain Xcycle on adding from each $e \in \texttt{visible}(\tau^r_X, r, x)$ a CF edge to w. Then $\mathsf{extend}_X(\tau^{rw})$ contains $\{(e, w) \mid e \in \texttt{visible}(\tau^r_X, r, x)\}$.
2. For $X = \texttt{CM}$, $\texttt{visible}(\tau^r_X, r, x) = \{e \in \texttt{readable}(\tau^r_X, r, x) \mid e\ \mathsf{HB}_p r\}$ where r is a read in process p. The point of $\texttt{visible}(\tau^r_X, r, x)$ is that when the new read r is added, which reads from a write w, $\mathsf{extend}_X(\tau^{rw})$ will not contain Xcycle on adding from each $e \in \texttt{visible}(\tau^r_X, r, x)$ a HB_p edge to w. Then $\mathsf{extend}_X(\tau^{rw})$ contains $\{(e, w) \mid e \in \texttt{visible}(\tau^r_X, r, x)\}$.

The *trace semantics* for a model $X \in \{\texttt{CC}, \texttt{CCv}, \texttt{CM}\}$ is given as the transition relation $\rightarrow_{X-\texttt{tr}}$, defined as $\tau_X \xrightarrow{\alpha}_{X-\texttt{tr}} \tau'_X$ where $\mathsf{extend}_X(\tau) = \tau_X, \mathsf{extend}_X(\tau') = \tau'_X$. The label α is one of $(read, r, w), (write, w)$ representing respectively, a read r reading from a write w, and a write event w. An important property of $\tau_X \xrightarrow{\alpha}_{X-\texttt{tr}} \tau'_X$ is that if τ_X does not have Xcycle, then τ'_X also does not have Xcycle; in other words, if $\tau \models X$, then $\tau' \models X$. We now describe the transitions $\tau_X \xrightarrow{\alpha}_{X-\texttt{tr}} \tau'_X$ where $\mathsf{extend}_X(\tau) = \tau_X, \mathsf{extend}_X(\tau') = \tau'_X$, $\tau = \langle \texttt{E}, po, rf\rangle, \tau' = \langle \texttt{E}', po', rf'\rangle$. We start from the empty trace τ_0, $\mathsf{extend}_X(\tau_0) = \tau_0$.

- From τ_X, assume that we observe a write w in process p. In this case, the label α is $(write, w)$, and we add w, and a po edge from the po-latest event of process p in τ_X to w obtaining τ'_X.
- From τ_X, assume we observe a read event r on variable x in process p. In this case, the label α is $(read, r, w)$, where w is the write from which r reads. Add the read r, a po edge from the po-latest event of process p in τ_X to r obtaining τ^r_X. Add rf from a $w \in \texttt{readable}(\tau^r_X, r, x)$ to r.
  • When $X = \texttt{CCv}$. Add new CF edges from all $w'' \in \texttt{visible}(\tau^r_X, r, x)$ to w to get τ'_X.
  • When $X = \texttt{CM}$. Add new HB_p edges from all $w'' \in \texttt{visible}(\tau^r_X, r, x)$ to w. Adding these HB_p edges can result in $w_1\ \mathsf{HB}_p\ w_2$ for write events w_1, w_2 on a variable y. If we had $w_1\ rf\ r_1$, $r_1\ po\ r$, then add $r_1\ \mathsf{HB}_p\ w_2$. When we are done adding all such HB_p edges, we obtain τ'_X. (Figure 4(iv)).

Lemma 1. *If* $\tau_X = \mathsf{extend}_X(\tau)$ *with* $\tau \models X$, *and* $\tau_X \xrightarrow{\alpha}_{\texttt{tr}} \tau'_X = \mathsf{extend}_X(\tau')$, *then* $\tau' \models X$ *for* $X \in \{\texttt{CC}, \texttt{CCv}, \texttt{CM}\}$.

Efficiency and Correctness. Each step of $\xrightarrow{\alpha}_{\texttt{tr}}$ is computable in polynomial time . This is based on the fact that readable and visible sets are computable in polynomial time. The correctness of the trace semantics for a model X stems from the fact that it generates only those X-extensions which do not have cycles (Lemma 1). The transitions ensure acyclicity of the resultant extended traces.

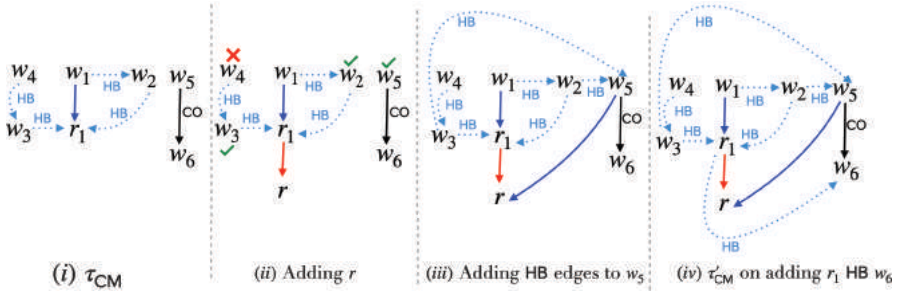


Fig. 4. w_1, w_6 are writes on y, r_1 is a read on y, w_2, w_3, w_4, w_5 are writes on x in $\tau_{\mathtt{CM}}$. Add read r on x to $\tau_{\mathtt{CM}}$. $w_2, w_3, w_5 \in \mathtt{readable}(\tau^r_{\mathtt{CM}}, r, x)$. Choose w_5 *rf* r. Then we add w_2 HB w_5 and w_3 HB w_5. The addition of w_2 HB w_5 results in w_1 HB w_6. Since w_1 *rf* r_1, add the HB edge from r_1 to w_6 to obtain $\tau'_{\mathtt{CM}}$.

Algorithm 1: ExploreTraces(X, τ_X, π)

Input: $X \in \{\mathtt{CC}, \mathtt{CCv}, \mathtt{CM}\}$ is a consistency model, τ_X is an X-extension, π is an observation sequence.

1 **if** $\exists w$ *s.t.* $\tau_X \xrightarrow{\langle write, w\rangle} {}_{X-\mathtt{tr}}\tau'_X$ **then** // handle a write event
2 **let** $w = \mathtt{wt}(x, v)$ and perform $\tau_X \xrightarrow{\langle write, w\rangle} {}_{X-\mathtt{tr}}\tau'_X$ // follow trace semantics write
3 ExploreTraces$(X, \tau'_X, \pi \bullet w)$
4 CreateSchedule$(\tau'_X, \pi \bullet w)$
5 **else if** $\exists r$ *s.t.* $\tau_X \xrightarrow{\langle read, r, -\rangle} {}_{X-\mathtt{tr}}\tau'_X$ **then** // handle a read event
6 $\mathtt{Schedules}(r) \leftarrow \emptyset$; $\mathtt{Swappable}(r) \leftarrow \mathtt{true}$
7 **for** $w, \tau'_X : \tau_X \xrightarrow{\langle read, r, w\rangle} {}_{X-\mathtt{tr}}\tau'_X$ **do** ExploreTraces$(X, \tau'_X, \pi \bullet \langle r, w\rangle)$ // follow trace semantics read
8 **for** $\beta \in \mathtt{Schedules}(r)$ **do** RunSchedule(X, τ_X, π, β)

5 DPOR Algorithm for CC, CCv, CM

We present our DPOR algorithm, which systematically explores, for any terminating program under the consistency models $X \in \{\mathtt{CC}, \mathtt{CCv}, \mathtt{CM}\}$, all traces τ_X wrt X which can be generated by the trace semantics. Enabled write events from any of the processes are added to the trace generated so far, and we proceed with the next event. For a read event r, we add r to the trace, and explore in separate branches, all possible write events w from which r can read from. Each such branch is a sequence of events also called a *schedule*. There may be writes w' which will be added to the trace later in the exploration, from which r can also read. Such writes w' are called *postponed* wrt r; when w' is added to the trace later, the algorithm will have a branch where r can read from w'. In that branch, the algorithm reorders events in the sequence s.t. w' and r exchange places, and all events which are needed for w' to occur are also placed before w' (CreateSchedule). All generated schedules will be executed

Algorithm 2: CreateSchedule(X, τ_X, π)

Input: $X \in \{\texttt{CC}, \texttt{CCv}, \texttt{CM}\}$ is a consistency model, τ_X is an X-extension and π is an explored observation sequence.

```
1  let w be last(π) and x be var(w)
2  for i ← |π| − 1 to 1 do        // look for reads r that have postponed w
3      let r be the element at π[i]
4      if r is a read on x∧¬(r CO w) ∧ Swappable(r) then
5          β ← ϵ; flag = true;
6          for j ← i + 1 to |π| − 1 do    // get all events after r in π and
               precedes w in CO
7              let ev be the element at π[j]
8              if ev CO w then
9                  if r CO ev then
10                     flag = false; break;
11                 else
12                     β ← β • π[j]
13         if flag ∧ ∄β′ ∈ Schedules(r) . β′ ≈ β • w • ⟨r, w⟩ then
14             Schedules(r) ← Schedules(r) ∪ {β • w • ⟨r, w⟩}    // r can read
                   from w
```

Algorithm 3: RunSchedule(X, τ_X, π, β)

Input: $X \in \{\texttt{CC}, \texttt{CCv}, \texttt{CM}\}$ is a consistency model, τ_X is a X-extension, π is an explored-observation sequence, and β is a schedule.

```
1  if β ≠ ϵ then        // explore the sequence of observations one by one
2      let β be α • β′ choose τ′_X : τ_X --α--> X−tr τ′_X     // follow write and read
3      if α = (read, r, w) then Swappable(r) ← false
4      RunSchedule(X, τ′_X, π • α, β′)
5  else ExploreTraces(X, τ_X, π)
```

by RunSchedule. The algorithm is uniform across the models, with the main technical differences being taken care of by the respective trace semantics which guides the exploration of traces in each model.

The ExploreTraces Algorithm. This algorithm takes as input, a consistency model $X \in \{\texttt{CC}, \texttt{CCv}, \texttt{CM}\}$, an X-extension τ_X and an *observation sequence* π. π is a sequence of events of the form $(write, w)$ or $(read, r, w)$. The initial invocation is with the empty trace τ_0 and observation sequence $\pi = \epsilon$. The observation sequence is used to swap read operations with write operations that are *postponed* wrt them. From the initial τ_0, we choose an operation from any of the processes.

If a write operation is enabled, one such is chosen non deterministically from any process, and is added to the trace according to the trace semantics, and also appended to the observation sequence, whereafter ExploreTraces is called recursively to continue the exploration (line 3). After the recursive calls have returned, the algorithm calls CreateSchedule, which finds read operations r in the observation sequence which can read from write operations w if w was

performed before r. For each such read r, CreateSchedule creates a *schedule* for r, an observation sequence that can be explored from the point when r was performed, allowing w to occur before r so that r can read from w. When a read operation r is enabled, the set $\texttt{Schedules}(r)$ is initialized (line 6). This set is updated by CreateSchedule when subsequent writes are explored. We also keep a Boolean flag $\texttt{Swappable}(r)$ for each read event r. This is initialized to true, indicating that r is swappable, that is, subsequent writes can be considered for r. This flag is set to false for read events appearing in a schedule so that they are not swapped, eliminating redundant explorations. For each generated write event w from which r can read, ExploreTraces is called recursively (line 7)to continue the exploration. Once these recursive calls have returned, the set of schedules collected in $\texttt{Schedules}(r)$ for the read r is considered. RunSchedule explores all schedules, where the read fetches its value from the respective write.

The CreateSchedule algorithm. The input to this algorithm is a consistency model X, a trace τ_X wrt X, and an observation sequence π whose last element is a write. The algorithm looks for reads in π for which w is a postponed write. Indeed, this read r and w must be on the same variable, r must be swappable, and r must not precede w wrt CO (line 4). We begin with the closest (from the write w) such read r at position $\pi[j]$. After finding r, a schedule β is created. The schedule consists of all elements following r in π and preceding w wrt CO (line 12). It ends with $w \bullet (r, w)$, allowing r to read from w (line 13). This schedule is added to $\texttt{Schedules}(r)$ if it does not already contain a schedule β' which has the same set of observations : $\texttt{Schedules}(r)$ does not contain $\beta' \approx \beta$.

The RunSchedule Algorithm. The inputs are a consistency model X, a trace τ_X, an observation sequence π and a schedule β. The schedule of observations in β is explored one by one, by recursively calling itself, and updating the trace. The read events in the schedule are not swappable, preventing a redundant exploration for them (schedules where these are swapped with respective writes will be created by CreateSchedule. All proofs and an illustrative example can be found in the extended version of the paper [2].

Theorem 1. *Our DPOR algorithms are sound, complete and optimal.*

Soundness, Optimality and Completeness. The algorithm is sound in the sense that, if we initiate Algorithm 1 from (X, τ_0, ϵ), then, all explored traces τ are s.t. $\tau \models X$. This follows from the fact that the exploration uses the $\rightarrow_{X-\texttt{tr}}$ relation. The algorithm is optimal in the sense that, for any two different recursive calls to Algorithm 1 with arguments (X, τ_X^1, π_1) and (X, τ_X^2, π_2), if τ_X^1, τ_X^2 are extendible, then $\tau_X^1 \neq \tau_X^2$. This follows from (i) for a given read r, each iteration of the for loop in line 7 will correspond to a different write, (ii) in each schedule $\beta \in \texttt{Schedules}(e)$ in line 8 of Algorithm 1, the read event r reads from a write w which is different from all writes it reads from in line 7 (iii) Any two schedules added to $\texttt{Schedules}(e)$ at line 14 of Algorithm 2 will be different. The algorithm explores traces of all terminating runs, and is hence complete.

6 Experimental Evaluation

We describe the implementation of our optimal DPOR algorithm for the causal consistency models CC, CCv, CM as a tool Conschecker, available at[45]. To the best of our knowledge, Conschecker is the first stateless model checking tool for the causal consistency models CC, CCv, CM.

Conschecker. Conschecker extends Nidhugg [3] and works at LLVM IR level accepting a C language program as input. At runtime, Conschecker controls the exploration of the input program until it has explored all the traces using the DPOR algorithm. It can detect user-provided assertion violations by analyzing the generated traces. We conduct all experiments on a Ubuntu 22.04.1 LTS with Intel Core i7-1165G7 and 16 GB RAM. We evaluate Conschecker on the following categories of benchmarks, as seen below.

Experimental Setup. We consider the following categories of benchmarks.

- A set of thousands of litmus tests (sec 6.1) generated from [8]. The main purpose of these experiments is to provide a sanity check of the correctness of Conschecker on all three consistency models.
- A collection (sec 6.2) of concurrent benchmarks taken from the TACAS competition on software verification [44]. These are small programs with 50-100 lines of code used by many tools [4], [5].
- Five applications (sec 6.3) : Voter [19], Twitter clone [27], Fusion ticket [27], two versions of Auction [36], extracted from literature on databases, and verify against assertion violations wrt the three consistency models.
- Classical database benchmarks (sec 6.4) reported in recent papers on consistency models [13], [12] and [14]. We classify these benchmarks SAFE and UNSAFE on all three models depending on whether they witness an assertion violation.
- Eight parameterized programs (sec 6.5) from [5] and [4] to study the scalability of Conschecker when increasing the number of processes, as well as read and write instructions in programs.

6.1 Litmus Benchmarks

We apply Conschecker on a set of 9815 litmus benchmarks generated from [8]. Litmus tests are standard benchmark programs used by many tools running on weak memories. In these litmus tests, the processes execute concurrently, and we validate assertions on the underlying memory model, doing a sanity check for the correctness of Conschecker. We compared the observed outcomes of Conschecker on the litmus tests with expected outcomes generated from [8]. We generated the expected outcomes by simulating the CCv, and CC and CM semantics on [8] for these litmus tests. Out of the 9815 litmus tests, we found no assertion violations in 3810 under CC, CM and 3811 under CCv. Results obtained from Conschecker matched with the expected outcomes. Conschecker took $<$3 mins to execute on all litmus tests across models.

Table 1. Classical benchmarks

Program	CCv	CC	CM
Causality Violation [13]	UNSAFE	UNSAFE	UNSAFE
Causal Violation [14]	SAFE	SAFE	SAFE
Delivery Order [12]	UNSAFE	UNSAFE	SAFE
Long Fork [13]	UNSAFE	UNSAFE	UNSAFE
Lost Update [13]	UNSAFE	UNSAFE	UNSAFE
Message Passing [11]	SAFE	SAFE	SAFE
Conflict violation [14]	UNSAFE	UNSAFE	UNSAFE
Read Atomicity [14]	UNSAFE	UNSAFE	UNSAFE
Repeated Read [14]	UNSAFE	UNSAFE	UNSAFE
Load Buffer	SAFE	SAFE	SAFE
Store Buffer [11]	UNSAFE	UNSAFE	UNSAFE
Write Skew [13]	UNSAFE	UNSAFE	UNSAFE

Table 2. Applications.

App	Time
Vote [19]	1s
Twitter clone [27]	0.09s
FusionTicket [27]	0.75s
Auction [36]	0.11s
Auction-2 [36]	1.17s
Group	0.10s

Table 3. SV-Comp Benchmarks

	CCv		CC		CM	
Program	Traces	Time	Traces	Time	Traces	Time
Lamport	15669	3s	2904225	490s	299028	110s
Szymanski	1023397	131s	1023397	115s	1023397	190s
Peterson	5371	1s	13483	1s	12316	1.5s
Fibonacci	6224342	769s	6224342	695s	6224342	1796s
Dekker	86267	7s	1549862	155s	107698	18s

6.2 SV-COMP Benchmarks

These benchmarks [44] consist of five programs written in C/C++ having 2 processes each, with 50-100 lines of code per process (Table 3). The main challenge in these benchmarks is the large number of traces to be explored. These benchmarks have assertion checks, and under CCv, CM, and CC all these assertions are violated. Conschecker stops exploration as soon as it detects the first assertion violation. To check the efficiency of Conschecker, we removed all assertions and let Conschecker exhaustively explore all *po*-*rf* traces. Since these benchmarks have large number of traces, they serve as a stress test.

6.3 Database Applications

Table 2 reports the performance of Conschecker on a set of programs inspired from five applications extracted from the literature on distributed systems [19], [27], [12], [36]. The applications we considered are

• Voter [19] : This application is derived from a software system used to record votes from a talent show. Users can vote for any of the n contestants from any one of the m sites (processes). The application asserts that users cannot vote from multiple sites and cannot vote for multiple contestants and checks for violations of this. [19] considers 3 sites and 3 users, and we follow suit.

Table 4. Parameterized Benchmarks from [5] and [4]

	CC		CCv		CM	
Program	Traces	Time	Traces	Time	Traces	Time
control-flow(6)	77	0.05s	77	0.05s	77	0.05s
control-flow(8)	273	0.07s	273	0.06s	273	0.10s
control-flow(10)	1045	0.16s	1045	0.12s	1045	0.33s
control-flow(12)	4121	0.60s	4121	0.45s	4121	1.80s
n-writers-a-read(5)	6	0.05s	6	0.05s	6	0.05s
n-writers-a-read(10)	11	0.05s	11	0.05s	11	0.05s
n-writers-a-read(15)	16	0.05s	16	0.05	16	0.05s
n-writers-a-read(20)	21	0.05s	21	0.05	21	0.05s
redundant-co(5)	91	0.07s	91	0.05s	91	0.05s
redundant-co(10)	331	0.09s	331	0.05s	331	0.08s
redundant-co(15)	721	0.11s	721	0.08s	721	0.12a
redundant-co(20)	1261	0.18	1261	0.13s	1261	0.20s
casrot(9)	8579	0.55s	8597	0.77s	8597	2s
casrot(10)	38486	2.50s	38486	3.16s	38486	9s
casrot(11)	182905	14s	182905	16s	182905	49s
floating-read(9)	10	0.05s	10	0.05s	10	0.05s
floating-read(11)	12	0.05s	12	0.05s	12	0.05s
floating-read(13)	14	0.05s	14	0.05s	14	0.05s
lastwrite(9)	9	0.04s	9	0.04s	9	0.04s
lastwrite(11)	11	0.04s	11	0.04	11	0.04s
lastwrite(13)	13	0.04	13	0.04	13	0.04s
lastzero(9)	1536	0.18s	1536	0.20s	1536	0.33s
lastzero(11)	7168	1s	7168	1s	7168	2s
lastzero(13)	32768	5s	32768	5s	32768	12s
readers(9)	512	0.10s	512	0.10s	512	0.18s
readers(11)	2048	0.40s	2048	0.35s	2048	1s
readers(13)	8192	1.5s	8192	1.5s	8192	6s

• Twitter clone [27] : This is based on a twitter like service where each user has some followers. The following assertion is checked : when the user tweets, the tweet ID must be added to the follower's time line exactly once if the user did not remove his tweet. We considered 3 users using 3 processes, each process has 10 tweet IDs and 6 followers.

• Fusion ticket [27] : There is a building having multiple concert rooms (venues). Tickets for venue i are sold by salesperson i who updates in the backend database, the sales for the day. The per venue ticket sale must be updated correctly in the database, so that the concert manager sees the correct total number of tickets sold. A discrepancy in this number is a violation. Each venue is represented by a process, and the communication across processes ensures the total sum is correct. We considered 4 venues and each venue had 10 tickets.

• Auction [36] and Auction-2 [36]: There are n bidders and an auctioneer participating in an auction, modeled using $n+1$ processes. The assertion to be checked

is that the highest bidder must be declared winner. Auction is the buggy version for this application, while Auction-2 is the correct one.
• Group is a synthetic application created by us inspired from whatsapp groups. There is a group with n members, and a new person wants to be added to the group. This person must be added to the group only by one of the existing members. That is, a violation constitutes to adding a person more than once (by one or more members). We check with 6 processes(members).

6.4 Classical Benchmarks

Table 1 consists of classical benchmarks [13], [14], [12] and [11] which test for some assertion violations under the three models. Since the traces generated differ for each model $X \in \{\texttt{CC}, \texttt{CCv}, \texttt{CM}\}$, the violations also differ. For the ones marked SAFE under model $X \in \{\texttt{CC}, \texttt{CCv}, \texttt{CM}\}$, the assertion violation did not occur under any execution, while the unsafe ones reported the violation. We consider twenty such examples. We consider three different versions of each example, varying the number of processes and variables.

For each example, we have three versions by parameterizing the number of processes and instructions. In version 1, we have four processes per program and three to five instructions per process. Version 2 is obtained allowing each process to have seven-ten instructions. Version 3 expands version 2 by allowing each program to have up to five-six processes and up to 15-20 instructions. The number of instructions is increased by introducing fresh variables and having reads/writes on them. Versions 2,3 serve as a stress test for Conschecker as increasing the number of instructions and processes increases the number of of consistent traces. Conschecker took less than $3s$ to finish running all version 1 programs, about 30s to finish running all version 2 programs and about 200s to finish running all version 3 programs.

6.5 Parameterized Benchmarks

Table 4 reports experimental results of Conschecker on 8 parameterized benchmarks. Out of these, in redundant-co(N) (taken from [5]), N is the number of loop iterations per process in a program with 3 processes. In all others, the parameterization is on the number of processes. This set of benchmarks serves to check the scalability of Conschecker. As seen in Table 4, Conschecker scales up to 20 processes (n-writers-a-read) and 13 variables (lastzero).

7 Conclusion

In this paper, we have provided a DPOR algorithm using the $po - rf$ equivalence for three prominent causal consistency models, and also implemented the same in a tool Conschecker. This is the first tool for stateless model checking of causal consistency models. We plan to extend our work by developing a DPOR algorithm for transactional programs under $\texttt{CC}, \texttt{CCv}, \texttt{CM}$ [12]. For these, the extra

complication is the presence of transactions which must be executed atomically without interference in each process. The final notch is to handle snapshot isolation, the strongest among transactional consistency models.

8 Data-Availability Statement

The tool and experimental data for the study are available at the Zenodo repository: [45].

References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. SIGPLAN Not. **49**(1), 373–384 (jan 2014). https://doi.org/10.1145/2578855.2535845, https://doi.org/10.1145/2578855.2535845
2. Abdulla, P., Atig, M.F., S, K., Gupta, A., Tuppe, O.: Optimal Stateless Model Checking for Causal Consistency (Jan 2023). https://doi.org/10.5281/zenodo.7572282, https://doi.org/10.5281/zenodo.7572282
3. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for tso and pso. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 353–367. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360576, https://doi.org/10.1145/3360576
5. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. **2**(OOPSLA) (Oct 2018). https://doi.org/10.1145/3276505, https://doi.org/10.1145/3276505
6. Agarwal, P., Chatterjee, K., Pathak, S., Pavlogiannis, A., Toman, V.: Stateless model checking under a reads-value-from equivalence. In: Silva and Leino [43], pp. 341–366. https://doi.org/10.1007/978-3-030-81685-8_16, https://doi.org/10.1007/978-3-030-81685-8_16
7. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distrib. Comput. **9**(1), 37–49 (Mar 1995). https://doi.org/10.1007/BF01784241, https://doi.org/10.1007/BF01784241
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2) (jul 2014). https://doi.org/10.1145/2627752, https://doi.org/10.1145/2627752
9. Bailis, P., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Bolt-on causal consistency. In: Ross, K.A., Srivastava, D., Papadias, D. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013. pp. 761–772. ACM (2013). https://doi.org/10.1145/2463676.2465279, https://doi.org/10.1145/2463676.2465279
10. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011,

Austin, TX, USA, January 26-28, 2011. pp. 55–66. ACM (2011). https://doi.org/10.1145/1926385.1926394, https://doi.org/10.1145/1926385.1926394
11. Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness between weak transactional consistency models. Programming Languages and Systems **12648**, 87 (2021)
12. Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness Against Transactional Causal Consistency. Logical Methods in Computer Science **Volume 17, Issue 1** (Feb 2021). https://doi.org/10.23638/LMCS-17(1:12)2021, https://lmcs.episciences.org/7149
13. Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: 27th International Conference on Concurrency Theory (CONCUR 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
14. Biswas, R., Enea, C.: On the complexity of checking transactional consistency. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019). https://doi.org/10.1145/3360591, https://doi.org/10.1145/3360591
15. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 626–638. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009888, https://doi.org/10.1145/3009837.3009888
16. Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1-2), 1–150 (2014). https://doi.org/10.1561/2500000011, https://doi.org/10.1561/2500000011
17. Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in erlang programs. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013. pp. 154–163. IEEE Computer Society (2013). https://doi.org/10.1109/ICST.2013.50, https://doi.org/10.1109/ICST.2013.50
18. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. Int. J. Softw. Tools Technol. Transf. **2**(3), 279–287 (1999). https://doi.org/10.1007/s100090050035, https://doi.org/10.1007/s100090050035
19. Difallah, D.E., Pavlo, A., Curino, C., Cudre-Mauroux, P.: Oltp-bench: An extensible testbed for benchmarking relational databases. Proc. VLDB Endow. **7**(4), 277–288 (dec 2013). https://doi.org/10.14778/2732240.2732246, https://doi.org/10.14778/2732240.2732246
20. Du, J., Elnikety, S., Roy, A., Zwaenepoel, W.: Orbe: scalable causal consistency using dependency matrices and physical clocks. In: Lohman, G.M. (ed.) ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013. pp. 11:1–11:14. ACM (2013). https://doi.org/10.1145/2523616.2523628, https://doi.org/10.1145/2523616.2523628
21. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. In: Lazowska, E., Terry, D., Arpaci-Dusseau, R.H., Gehrke, J. (eds.) Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014. pp. 4:1–4:13. ACM (2014). https://doi.org/10.1145/2670979.2670983, https://doi.org/10.1145/2670979.2670983
22. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032. Springer (1996). https://doi.org/10.1007/3-540-60761-7, https://doi.org/10.1007/3-540-60761-7

23. Godefroid, P.: Model checking for programming languages using verisoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 174–186. POPL '97, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/263699.263717, https://doi.org/10.1145/263699.263717
24. Godefroid, P.: Software model checking: The verisoft approach. Form. Methods Syst. Des. **26**(2), 77–101 (Mar 2005). https://doi.org/10.1007/s10703-005-1489-x, https://doi.org/10.1007/s10703-005-1489-x
25. Hamza, J.: Algorithmic Verification of Concurrent and Distributed Data Structures. Ph.D. thesis, Université Paris Diderot (2015)
26. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972, https://doi.org/10.1145/78969.78972
27. Holt, B., Bornholt, J., Zhang, I., Ports, D., Oskin, M., Ceze, L.: Disciplined inconsistency with consistency types. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. p. 279–293. SoCC '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2987550.2987559, https://doi.org/10.1145/2987550.2987559
28. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. **2**(POPL), 17:1–17:32 (2018). https://doi.org/10.1145/3158105, https://doi.org/10.1145/3158105
29. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. **6**(POPL), 1–28 (2022). https://doi.org/10.1145/3498711, https://doi.org/10.1145/3498711
30. Kokologiannakis, M., Vafeiadis, V.: Genmc: A model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 427–440. Springer International Publishing, Cham (2021)
31. Lahav, O.: Verification under causally consistent shared memory. ACM SIGLOG News **6**(2), 43–56 (2019). https://doi.org/10.1145/3326938.3326942, https://doi.org/10.1145/3326938.3326942
32. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563, https://doi.org/10.1145/359545.359563
33. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439, https://doi.org/10.1109/TC.1979.1675439
34. Mazurkiewicz, A.: Trace theory. In: Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency. p. 279–324. Springer-Verlag, Berlin, Heidelberg (1987)
35. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 267–280. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf
36. Nair, S.S., Petri, G., Shapiro, M.: Proving the safety of highly-available distributed objects. In: Müller, P. (ed.) Programming Languages and Systems. pp. 544–571. Springer International Publishing, Cham (2020)

37. Norris, B., Demsky, B.: A practical approach for model checking c/c++11 code. ACM Trans. Program. Lang. Syst. **38**(3) (May 2016). https://doi.org/10.1145/2806886, https://doi.org/10.1145/2806886
38. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 391–407. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_27, https://doi.org/10.1007/978-3-642-03359-9_27
39. Perrin, M., Mostéfaoui, A., Jard, C.: Causal consistency: beyond memory. In: Asenjo, R., Harris, T. (eds.) Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016. pp. 26:1–26:12. ACM (2016). https://doi.org/10.1145/2851141.2851170, https://doi.org/10.1145/2851141.2851170
40. Raynal, M., Schiper, A.: From causal consistency to sequential consistency in shared memory systems. In: Thiagarajan, P.S. (ed.) Foundations of Software Technology and Theoretical Computer Science, 15th Conference, Bangalore, India, December 18-20, 1995, Proceedings. Lecture Notes in Computer Science, vol. 1026, pp. 180–194. Springer (1995). https://doi.org/10.1007/3-540-60692-0_48, https://doi.org/10.1007/3-540-60692-0_48
41. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing. p. 166–182. HVC'06, Springer-Verlag, Berlin, Heidelberg (2006)
42. Shasha, D.E., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. **10**(2), 282–312 (1988). https://doi.org/10.1145/42190.42277, https://doi.org/10.1145/42190.42277
43. Silva, A., Leino, K.R.M. (eds.): Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I, Lecture Notes in Computer Science, vol. 12759. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8, https://doi.org/10.1007/978-3-030-81685-8
44. SV-COMP: Competition on Software Verification. https://sv-comp.sosy-lab.org/2018 (2018), [Online; accessed 2017-11-10]
45. Tuppe, O.: Conschecker (Jan 2023). https://doi.org/10.5281/zenodo.7500551, https://doi.org/10.5281/zenodo.7500551
46. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 250–259. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737956, https://doi.org/10.1145/2737924.2737956

Symbolic Model Checking for TLA+ Made Faster

Rodrigo Otoni[1(✉)], Igor Konnov[2], Jure Kukovec[2], Patrick Eugster[1], and Natasha Sharygina[1]

[1] Università della Svizzera italiana, Lugano, Switzerland
{otonir,eugstp,sharygin}@usi.ch
[2] Informal Systems, Vienna, Austria
{igor,jure}@informal.systems

Abstract. The need to provide formal guarantees about the behaviour of the algorithms underpinning modern distributed systems became evident in recent years. This interest made apparent the complexities involved in applying verification techniques in a distributed setting, with significant effort being made in both academia and industry to aid in this endeavour. Many formalisms have been proposed to tackle the difficulties faced by practitioners, with one that has seen widespread use in industry being TLA$^+$, adopted, for instance, by Amazon Web Services. TLA$^+$ provides engineers with a way of specifying both systems and desired properties, and is supported by a number of verification tools. Despite their extensive use, such tools suffer considerably from lack of scalability. To solve this, we propose a novel encoding of TLA$^+$ into SMT constraints to improve symbolic model checking efficiency. Our insight is the need to provide the SMT solver with structural information about the TLA$^+$ specification encoded, i.e., how data structures and their component elements interact, which we do by relying on the SMT theory of arrays. We implemented our approach by modifying the SMT-based model checker Apalache and evaluated it against comparable tools. Our results show that our approach outperforms existing ones on a number of benchmarks, with an order of magnitude improvement in checking time.

Keywords: Model checking · SMT arrays · Distributed algorithms

1 Introduction

Distributed systems are ubiquitous in the modern world, with many companies directly relying on them to conduct business. Due to this, the ability to ensure that a distributed system is operating correctly is paramount. The search for correctness guarantees led to an influx of interested parties adopting formal verification methodologies in recent years. One of the most famous example of this trend is probably the adoption of TLA$^+$ [17] by Amazon Web Services [19]. TLA$^+$ is a specification language based on the temporal logic of actions (TLA) which allows users to describe the expected behaviour of a system, while abstracting

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 126–144, 2023.
https://doi.org/10.1007/978-3-031-30823-9_7

away implementation details that do not impact high-level properties, e.g., memory management. With TLA$^+$ specifications at hand, Amazon engineers rely on model checking for correctness guarantees of systems such as DynamoDB [23].

Despite recent interest and advances, the verification of distributed systems remains notoriously difficult. This is mainly due to the fact that, given their distributed nature, distributed algorithms' executions admit numerous potential interleavings of steps, with state-spaces generally growing exponentially with the number of participants. In the case of TLA$^+$, a handful of tools are available to aid in verification [14]. TLC [27] is an explicit-state model checker that enumerates all reachable states of the given system. Apalache [13] is a symbolic bounded model checker that uses a satisfiability modulo theories (SMT) encoding of states in order to better tackle the state-space explosion problem. TLAPS [6] is an interactive proof system that enables the proving of properties without the need of exploring the state-space itself. Despite providing the benefit of verifying specifications with infinite state-spaces, and efforts being made towards partial automation [18], TLAPS adoption is still slow, with engineers favouring the push-button automation provided by model checkers.

In this work we focus on symbolic model checking for TLA$^+$, as spearheaded by the SMT encoding which underpins Apalache, but provide insights into SMT-based model checking that may generalise to other contexts. The encoding of TLA$^+$ into SMT done by Apalache removes all structural information present in the encoded specification, with all TLA$^+$ data structures being represented via uninterpreted constants in the generated SMT formula. The information not forwarded to the SMT solver has the potential to significantly improve solving efficiency. We propose an alternative SMT encoding that makes full use of the SMT theory of arrays [8] to encoded the main TLA$^+$ data structures, i.e., sets and functions, with the goal of improving solving performance, which is the determining factor in overall model checking performance.

Concretely, we modify Apalache's abstract reduction system (ARS) to generate constraints in the SMT theory of arrays, while relying on its preprocessing infrastructure, as shown in Figure 1. Apalache rewrites the input specification into the KerA$^+$ verification-friendly fragment of TLA$^+$ [13] and then applies ARS rules to generate the SMT formula to be solved. We implemented our encoding in Apalache and compared it with Apalache's constants encoding and TLC. Our experiments indicate that embedding structural information into the SMT formulas has a significant impact on performance. Our contributions are:

1. Formalisation of a TLA$^+$ encoding into the SMT theory of arrays;
2. Development of a robust open-source implementation of our encoding;
3. Evaluation via checking agreement on three asynchronous protocols.

The paper is structured as follows: background is given in Section 2, the arrays-based encoding and its evaluation are presented in Sections 3 and 4, related work is discussed in Section 5, and our final remarks are made in Section 6.

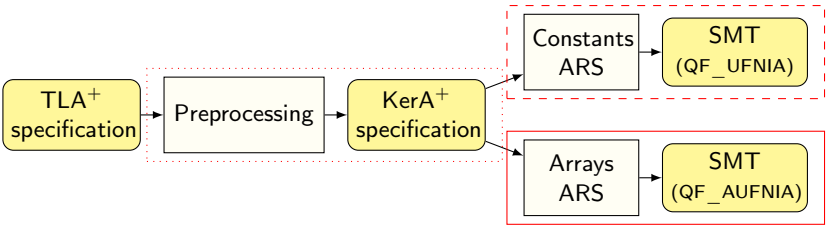


Fig. 1: Overview of the symbolic model checking for TLA$^+$. The dotted box highlights the identification of symbolic transitions from [16] and the rewriting into KerA$^+$. The dashed box highlights the encoding based on uninterpreted constants from [13]. The solid box highlights the arrays-based encoding we propose.

2 Background

In this section we introduce the basics of TLA$^+$, its KerA$^+$ fragment used to represent TLA$^+$'s core, the approach to generate SMT constraints from KerA$^+$ via abstract reduction, and finally the SMT theory of arrays.

2.1 TLA+

We introduce TLA$^+$ via a specification of the asynchronous Byzantine agreement protocol by Bracha and Toueg [5], shown in Figure 2. Here we focus on the most relevant TLA$^+$ constructs, with further details being available in [17].

The first notable aspect of TLA$^+$ is that specifications may be parametrised, e.g., the number of processes and faults may not be fixed. In our example, the keyword CONSTANTS, in line 3, is used to declare its parameters: N, the total number of processes, and T and F, the maximal and actual number of faulty processes. It is important to understand, however, that while a specification may be parametrised, model checking can only be carried out for a specific instance of the protocol at a time, e.g., $N = 4$ and $T = F = 1$. Parameter declarations are followed by variable declarations, by the use of the VARIABLES keyword, in line 4. Variables define the states of the state-machine that the specification describes, with each state being defined by the combination of the values held by each variable. In our example, each state is defined by the values of $sentEcho, sentReady, rcvdEcho, rcvdReady$, and pc.

The remaining TLA$^+$ operators describe state-machine transitions or properties to be checked, and are defined using $\triangleq$. Two operators are of special significance, one that defines the initial-state predicate and one that plays the role of the transition operator. In our example, these operators are $Init$, in line 8, and $Next$, in line 22. Concretely, $Init$ defines the starting point for state-space exploration and $Next$ defines the exploration itself. Transitions are guided by constraints that must hold in both pre-transition states, represented by non-primed variables, and post-transition states, represented by primed variables.

```
1  ──────────────────────── MODULE ABA ────────────────────────
2  EXTENDS Integers, FiniteSets
3  CONSTANTS N, T, F
4  VARIABLES sentEcho, sentReady, rcvdEcho, rcvdReady, pc
5  Corr ≜ 1 .. (N − F)          The set of correct processes
6  Byz ≜ (N − F + 1) .. N       The set of Byzantine processes
7  Proc ≜ 1 .. N                The set of all processes
8  Init ≜ ∧ pc ∈ [Corr → {"V0", "V1"}]
9         ∧ rcvdEcho = [p ∈ Corr ↦ {}] ∧ rcvdReady = [p ∈ Corr ↦ {}]
10        ∧ sentEcho ∈ SUBSET Byz      ∧ sentReady ∈ SUBSET Byz
11 Receive(p, nextEcho, nextReady) ≜ ...    Omited for brevity
12 SendEcho(p, nextEcho, nextReady) ≜ ...   Omited for brevity
13 SendReady(p, nextEcho, nextReady) ≜
14     ∧ pc[p] = "EC"
15     ∧ ∨ Cardinality(nextEcho) ≥ (N + T + 2) ÷ 2
16       ∨ Cardinality(nextReady) ≥ T + 1
17     ∧ pc′ = [pc EXCEPT ![p] = "RD"] ∧ sentReady′ = sentReady ∪ {p}
18     ∧ UNCHANGED sentEcho
19 Decide(p, nextReady) ≜
20     ∧ pc[p] = "RD" ∧ Cardinality(nextReady) ≥ 2 ∗ T + 1
21     ∧ pc′ = [pc EXCEPT ![p] = "AC"] ∧ UNCHANGED ⟨sentEcho, sentReady⟩
22 Next ≜ ∃ p ∈ Corr, nextEcho ∈ SUBSET sentEcho, nextReady ∈ SUBSET sentReady :
23     ∧ Receive(p, nextEcho, nextReady)
24     ∧ ∨ SendEcho(p, nextEcho, nextReady) ∨ SendReady(p, nextEcho, nextReady)
25       ∨ Decide(p, nextReady) ∨ UNCHANGED ⟨pc, sentEcho, sentReady⟩
26 NoDecide ≜ ∀ p ∈ Corr : pc[p] ≠ "AC"   Invariant stating that processes never Decide
27 ──────────────────────────────────────────────────────────────
```

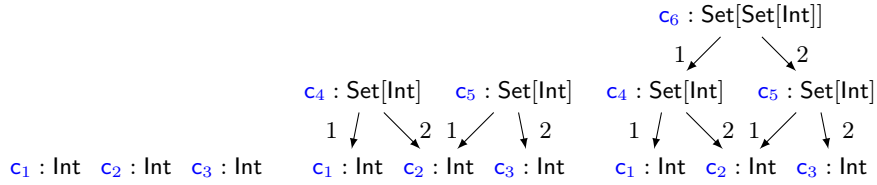
Fig. 2: Example of a TLA$^+$ specification, based on the asynchronous Byzantine agreement protocol by Bracha and Toueg [5]; simplifications made for brevity.

Specifications may optionally define invariants, i.e., properties that should hold in every reachable state. There is no special syntax for invariants, and they are provided by name to model checkers at invocation time. In our example, we have one invariant, *NoDecide*, in line 26. A specification satisfies *NoDecide* if no state reachable from *Init* via any number of *Next* transitions has $pc[p] =$ "*AC*", for some $p \in Corr$. Abstractly, this invariant holds iff *Decide* can never be taken.

2.2 KerA+

TLA$^+$ provides users with a myriad of ways of specifying systems. This richness, although being one its strengths, adds significant difficulty to the generation of SMT constraints. To overcome this challenge, TLA$^+$ specifications are rewritten into a more compact language, KerA$^+$, before being checked. From KerA$^+$, the ARS can generate SMT constraints in a simpler and provably sound way.

The KerA$^+$ language consists of a small subset of TLA$^+$ conjoined with four additional constructs not originating from TLA$^+$, and is able to express almost all TLA$^+$ expressions. It contains constructs for the manipulation of sets,



(a) integers 5, 6, and 7

(b) sets of integers $\{5, 6\}$ and $\{6, 7\}$

(c) set of sets of integers $\{\{5, 6\}, \{6, 7\}\}$

Fig. 3: Illustration of three arenas. The captions describe the modelled elements with the overapproximation $\mathsf{c}_1 = 5$, $\mathsf{c}_2 = 6$, $\mathsf{c}_3 = 7$, $\mathsf{c}_4 = \{5, 6\}$, $\mathsf{c}_5 = \{6, 7\}$, and $\mathsf{c}_6 = \{\{5, 6\}, \{6, 7\}\}$. Note that the concrete value of a cell can be given by any of the possible subtrees having said cell as a root, e.g., for c_6 we have that $\exists\ \mathsf{c}_4 \in \mathcal{P}(\{5, 6\}), \mathsf{c}_5 \in \mathcal{P}(\{6, 7\})\ .\ \mathsf{c}_6 \in \mathcal{P}(\{\mathsf{c}_4, \mathsf{c}_5\})$; $\mathcal{P}$ stands for power set.

functions, records, tuples, and sequences, as well as integer arithmetic operators, Boolean and integer literals, and constants, with all data structures having a bounded size. The semantics of KerA$^+$ derive directly from the TLA$^+$ constructs it uses, with the non-TLA$^+$ based constructs, which help simplify the rewriting system, having simple control semantics. The correctness of the rewriting itself is guaranteed by construction. One example is the rewriting of $S \cup T$ into the set comprehension $\{x \in S : x \in T\}$. Further KerA$^+$ details are available in [13].

2.3 Abstract Reduction System

In order to verify a specification in KerA$^+$ we generate a SMT formula that is equisatisfiable to it. To do so, we use an abstract reduction system (ARS) which iteratively applies reduction rules that transform KerA$^+$ expressions into SMT constraints. The core of the ARS is the *arena*, a graph structure that overapproxiamtes the specification's data structures and guides rule application. The rules collapse KerA$^+$ expressions into *cells*, which represent the symbolic evaluation of these expressions, with the cells then being used as vertices in the arena. The arena edges represent the data structures overapproximation, e.g., a cell representing a set will have directed edges to the cells representing all its potential elements, as illustrated in Figure 3. The reduction process terminates when the initial KerA$^+$ expression e is collapsed into a single cell c, producing a SMT formula Φ in the process, such that $\mathsf{c} \wedge \Phi$ is equisatisfiable to e; equisatisfiability relies on the boundedness of the data structures and is detailed in Section 3.3. The satisfiability of e can then be checked by forwarding $\mathsf{c} \wedge \Phi$ to a SMT solver.

Formally, the ARS is defined as $(\mathcal{S}, \rightsquigarrow)$, with $\mathcal{S}$ being the set of ARS states and $\rightsquigarrow \subseteq \mathcal{S} \times \mathcal{S}$ being the transition relation. A state $(e, \mathcal{A}, \nu, \Phi) \in \mathcal{S}$ is a four-tuple containing a KerA$^+$expression e, an arena $\mathcal{A}$, a binding of names to cells ν, and a first-order formula Φ. ARS states' elements contain a number of cells, which are first-order terms annotated with a type τ. Cells of type Bool and Int are interpreted in SMT as Booleans and integers, while cells of the remaining

types are encoded as uninterpreted constants in the constants encoding; the arrays encoding approach is discussed in Section 3. Cells are referred to via the notation c_{name} or c_{inde} , and they can be seen as both KerA$^+$ constants and first-order terms in SMT. An arena is a directed acyclic graph $\mathcal{A} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ being a finite set of cells and $\mathcal{E} \subseteq \mathcal{V} \times (1..|\mathcal{V}|) \times \mathcal{V}$ being a set of relations between the cells in $\mathcal{V}$. Every relation between cells is represented by an arena edge of form $(\mathsf{c}_a, i, \mathsf{c}_b)$, also written $\mathsf{c}_a \xrightarrow{i} \mathsf{c}_b$, with no duplicates, i.e., for every pair $(\mathsf{c}_{a_1}, i_1, \mathsf{c}_{b_1}), (\mathsf{c}_{a_2}, i_2, \mathsf{c}_{b_2}) \in \mathcal{E}$ we have that $\mathsf{c}_{a_1} = \mathsf{c}_{a_2} \wedge \mathsf{c}_{b_1} \neq \mathsf{c}_{b_2}$ implies $i_1 \neq i_2$, and no gaps in the relation indexes, i.e., for every edge $(\mathsf{c}_a, i, \mathsf{c}_b)$ and index $j \in 1..(i-1)$ we have that $\exists\ \mathsf{c}_c \in \mathcal{V}\ .\ (\mathsf{c}_a, j, \mathsf{c}_c)$. A binding is a partial function from KerA$^+$ variables to $\mathcal{V}$ of $\mathcal{A}$, i.e., a mapping from variables to cells. Finally, Φ is a formula in the SMT fragment supported by the ARS and the target SMT solver, e.g., the quantifier-free uninterpreted functions and non-linear arithmetics (QF_UFNIA) fragment supported by the constants encoding.

A series of n reduction steps has the form $s_0 \rightsquigarrow ... \rightsquigarrow s_n$, with each step generating state s_{i+1} for state s_i, $0 \leq i < n$, by applying a reduction rule. The initial state $s_0 = (e_0, \mathcal{A}_0, \nu_0, \Phi_0)$ has e_0 as the initial KerA$^+$ specification, $\mathcal{A}_0 = (\emptyset, \emptyset)$, ν_0 containing no mappings, and $\Phi_0 = \mathsf{true}$. The reduction steps end upon reaching a state $s_n = (e_n, \mathcal{A}_n, \nu_n, \Phi_n)$, with e_n being a single cell $\mathsf{c} \in \mathcal{V}_n$ and $\mathcal{A}_n = (\mathcal{V}_n, \mathcal{E}_n)$. Below we give two examples of rules.

Integer literal reduction. One of the simplest rules has an integer literal *num* being rewritten into a cell c_{num}. This cell is added to the arena and a constraint equating c_{num} to the literal is conjoined with Φ; we use vertical lines to separate state elements and commas to indicate additions to $\mathcal{A}$ and conjunctions to Φ.

$$\frac{\langle num : \mathsf{Int} \mid \mathcal{A} \mid \nu \mid \Phi \rangle \qquad num \text{ is one of } 0, 1, -1, ...}{\langle \mathsf{c}_{num} \mid \mathcal{A}, \mathsf{c}_{num} : \mathsf{Int} \mid \nu \mid \Phi, \mathsf{c}_{num} = num \rangle} \ (\textsc{Int})$$

The descriptions of rules can be given as inferences, with the premisses above the bar and the resulting state below it. Inferences, although reasonable to express rules such as Int, are not suitable to give the intuition about how more complex rules work. In light of this, we will use a simplified notation moving forward. We inline inferences as $\rightarrowtail$ and omit nonessential information, e.g., propagated values. Below we can see rule Int in this simplified format. Note that only $\mathcal{A}$ and Φ updates are shown, without propagating them, and that ν is omitted.

$$\begin{array}{c} num : \mathsf{Int} \\ num \text{ is one of } 0, 1, -1, ... \end{array} \rightarrowtail \mathsf{c}_{num} \mid \mathsf{c}_{num} : \mathsf{Int} \mid \mathsf{c}_{num} = num \qquad (\texttt{Int})$$

Picking. To pick a cell out of n cells we use an oracle θ, as per rule `FromBasic`. In addition to the FROM ... BY θ expression, this rule requires that all pickable cells are of the same basic type τ, e.g., Int. The resulting state has a new cell c_{pick}, which is equated to one of the n cells if $1 \leq \theta \leq n$ and is unconstrained otherwise. Picking among cells representing data structures, e.g., sets, can be

done via a more general version of rule `FromBasic`, which we omit for brevity.

$$\frac{\mathsf{FROM}\ \mathsf{c}_1, ..., \mathsf{c}_n\ \mathsf{BY}\ \theta : \tau \qquad \tau \text{ is basic and } \mathsf{c}_1 : \tau, ..., \mathsf{c}_n : \tau}{\rightarrowtail \mathsf{c}_{pick} \mid \mathsf{c}_{pick} : \tau \mid \bigwedge_{1 \leq i \leq n} (\theta = i \rightarrow \mathsf{c}_{pick} = \mathsf{c}_i)} \quad (\texttt{FromBasic})$$

2.4 SMT Theory of Arrays

The theory of arrays provides a natural way to encode data structures and is thus a prime candidate as an encoding target for TLA$^+$constructs. Here we present the theory's operators relevant for our work, further details can be found in [8].

Given the set of sorts S, containing one sort s_τ for each type τ in KerA$^+$, an array sort $\mathsf{s}_{\tau_1,\tau_2}$ has the form $\mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}$, with $\mathsf{s}_{\tau_1} \in \mathsf{S}$ being its *index sort* and $\mathsf{s}_{\tau_2} \in \mathsf{S}$ being its *value sort*. Each array sort is supported by two basic operators, $select : (\mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}, \mathsf{s}_{\tau_1}) \rightarrow \mathsf{s}_{\tau_2}$, which handles array access at a given index, and $store : (\mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}, \mathsf{s}_{\tau_1}, \mathsf{s}_{\tau_2}) \rightarrow \mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}$, which updates an array for a given index and value. For brevity, we will write $select(a, i)$ as $a[i]$ in the remainder of the manuscript. Regarding equality between arrays, different interpretations are possible. We use arrays with extensionality [25], which are considered equal if they contain the same values in the same entries. Extensionality is formally defined as $\forall\ a, b : \mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}\ .\ a = b \vee \exists\ i : \mathsf{s}_{\tau_1}\ .\ a[i] \neq b[i]$. For access and update, consistency is ensured by the following property:

$$\forall\ a : \mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2},\ i : \mathsf{s}_{\tau_1},\ j : \mathsf{s}_{\tau_1},\ v : \mathsf{s}_{\tau_2}\ .\ \underbrace{store(a, i, v)[i] = v}_{\text{access consistency}} \wedge \underbrace{(i = j \vee store(a, i, v)[j] = a[j])}_{\text{update consistency}}$$

In addition to *select* and *store*, the theory of arrays can be extended with other operators, two of which are map_f and K_{s_τ}, whose signatures are shown below. The map_f operator applies a n-ary function $f : (\mathsf{s}_{\tau_1}, ..., \mathsf{s}_{\tau_n}) \rightarrow \mathsf{s}_\tau$ to the values stored in each index of its array arguments, producing a new array whose values are the result of the function application, i.e., map_f is the pointwise array extension of f. The K_{s_τ} operator produces a constant array, with all its values being the constant provided as argument. The properties defining the behaviour of these two operators are shown after their signatures.

$$map_f : (\mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_1}, ..., \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_n}) \rightarrow \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_f} \qquad K_{\mathsf{s}_\tau} : \mathsf{s}_{\tau_{const}} \rightarrow \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_{const}}$$

$$\forall\ a_1 : \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_1},\ ...,\ a_n : \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_n},\ i : \mathsf{s}_\tau\ .\ map_f(a_1, ..., a_n)[i] = f(a_1[i], ..., a_n[i])$$

$$\forall\ i : \mathsf{s}_{\tau_1},\ v : \mathsf{s}_{\tau_2}\ .\ K_{\mathsf{s}_{\tau_1}}(v)[i] = v$$

The *select* and *store* operators are part of theory of arrays with extensionality defined in version 2.6 of the SMT-LIB standard [3]. Other operators are provided on a solver-by-solver basis, e.g., Z3 [7] supports both map_f and K_{s_τ}, while CVC5 [2] supports K_{s_τ}; SMT-LIB updates may add them to the standard.

3 Encoding TLA+ using Arrays

Our goal is to encode TLA$^+$ data structures in a structure-preserving way. To do this, we use arrays to represent the main components of TLA$^+$, sets and functions, as SMT constraints. We follow the ARS structure described in Section 2.3, but update the reduction rules handling sets and functions. The remaining TLA$^+$ constructs, e.g., tuples, are represented as per the constants encoding.

The two efficiency benefits of the arrays encoding are the ease of access of data structures and the possibility of using SMT equality. The first benefit can be easily understood by the use of SMT *select*, which allows us to check a stored value by using a single constraint, in contrast to the amount of constraints used in the constants encoding, which is linear in the size of data structures' overapproximation. The second benefit affects the comparison of data structures, which can be done via a single SMT equality for sets and functions in the arrays encoding, since these structures are represented by a single SMT term, while the constants encoding requires a number of constraints that is quadratic in the size of data structures' overapproximation. A summary can be seen in Table 1. We first describe how to encode sets and functions, and then present the correctness argument for the reduction to arrays.

3.1 Encoding TLA+ Sets using Arrays

We use arrays to encode TLA$^+$ sets as characteristic functions, i.e., a set of type τ is represented by an array of sort $\mathsf{s}_\tau \Rightarrow \mathsf{Bool}$. Set membership is encoded by storing true or false on a given array index. The reduction rules used to handle the main set operators are presented below.

Set Enumeration. The simplest way to create a set is to enumerate its elements. Rule Enum reduces an explicit set of cells to a fresh cell c_{set}, whose edges link it to its elements; $\mathsf{c}_{set} \rightarrow \mathsf{c}_1, \ldots, \mathsf{c}_n$ is a shorthand for $\mathsf{c}_{set} \xrightarrow{1} \mathsf{c}_1, ..., \mathsf{c}_{set} \xrightarrow{n} \mathsf{c}_n$. There is no guarantee that the enumerated elements are unique, thus the arena may contain edges to repeated elements.

$$\{\mathsf{c}_1, \ldots, \mathsf{c}_n\} : \mathsf{Set}[\tau] \rightarrowtail \mathsf{c}_{set} \mid \mathsf{c}_{set} : \mathsf{Set}[\tau], \mathsf{c}_{set} \rightarrow \mathsf{c}_1, \ldots, \mathsf{c}_n \mid \mathit{EnumCtr} \quad (\texttt{Enum})$$

The constraints *EnumCtr* added by the arrays encoding create an empty set, by using a constant array with the value false, $\bot$, and updates the array by storing true, $\top$, on the appropriate indexes. The array resulting from the last update, $a^n_{\mathsf{c}_{set}}$, is then equated to c_{set}. Since cells representing repeated elements lead to updates to the same index, we encode standard sets, in contrast the constants encoding, which encodes multisets due to the arena imprecision; multisets lead to multiple constraints being generated to encode membership of a single element.

$$\underbrace{a^0_{\mathsf{c}_{set}} = K_\tau(\bot)}_{\text{empty set}} \wedge \underbrace{\bigwedge_{1 \leq i \leq n} a^i_{\mathsf{c}_{set}} = \mathit{store}(a^{i-1}_{\mathsf{c}_{set}}, \mathsf{c}_i, \top)}_{\text{set updates}} \wedge \underbrace{\mathsf{c}_{set} = a^n_{\mathsf{c}_{set}}}_{\text{cell equality}} \qquad (\mathit{EnumCtr})$$

Although the amount of constraints generated by the arrays encoding to model set enumeration is equal to that of the constants encoding, it has the benefit of generating a defined interpretation for c_{set}, the array $a^n_{\mathsf{c}_{set}}$, which is not present in the constants encoding. This has a significant impact on set membership and cell equality, as described below.

Set Membership. The checking of a membership relation $\mathsf{c}_x \in \mathsf{c}_{set}$, given the presence of the arena edges $\mathsf{c}_{set} \rightarrow \mathsf{c}_1, ..., \mathsf{c}_n$ and $1 \leq x \leq n$, is straightforward. A single fresh cell of Boolean type is introduced and is equated to $\mathsf{c}_{set}[\mathsf{c}_x]$.

Table 1: Amount of constraints generated by each SMT encoding to model the main TLA$^+$ constructs.

Construct	Arrays	Constants
Set enumeration	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Set membership	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Set equality	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$
Set filter	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Set map	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Fun. definition	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Fun. domain	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Fun. equality	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$
Fun. update	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Fun. application	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Cell Equality. The constraints generated by encoding set membership and many other constructs assume that cells can be compared. When this is not directly the case the equalities are cached in preparation. For example, if a set of n tuples c_t of size two is being equated, the constraints $\mathsf{c}_{t_i} = \mathsf{c}_{t_j} \leftrightarrow \mathsf{c}^1_{t_i} = \mathsf{c}^1_{t_j} \wedge \mathsf{c}^2_{t_i} = \mathsf{c}^2_{t_j}$, with $1 \leq i \leq n$ and $1 \leq j \leq n$, are added to Φ; here we use c^1_t and c^2_t to represent the values of the 2-tuple. The need for this caching of equalities only arises when data structures encoded as uninterpreted constants are compared. For the remaining rules we assume that caching was done, if needed, and cells can be compared via direct equality.

Set Filter. In TLA$^+$, the elements of a set S can be filtered by a predicate p via the expression $\{x \in S : p\}$. This expression will create a set F which contains only the elements of S that satisfy p, e.g., $\{x \in \{-1, 0, 1\} : x \geq 0\} = \{0, 1\}$. Rule `Filter` reduces a filter to a new set cell, c_F, whose arena overappoximation contains the elements of S, but whose constraints ensure that only filtered elements are members of F; $p[y/x]$ means that x is replaced by y in p and parentheses indicate the application of another rule, the predicate resolution rule in this case.

$$\begin{aligned}\{x \in \mathsf{c}_S : p\} : \mathsf{Set}[\tau] \text{ and } \mathsf{c}_S &\rightarrow \mathsf{c}_1, \ldots, \mathsf{c}_n \\ &\rightarrowtail \left(p[\mathsf{c}_1/x] : \mathrm{Bool}, \ldots, p[\mathsf{c}_n/x] : \mathrm{Bool} \rightarrowtail \mathsf{c}^p_1, \ldots, \mathsf{c}^p_n\right) \\ &\rightarrowtail \mathsf{c}_F \mid \mathsf{c}_F : \mathsf{Set}[\tau], \mathsf{c}_F \rightarrow \mathsf{c}_1, \ldots, \mathsf{c}_n \mid \mathit{FilterCtr}\end{aligned} \quad (\texttt{Filter})$$

The constraints added use an array $a^0_{\mathsf{c}_F}$ initially unconstrained, i.e., the values mapped by all the indexes of $a^0_{\mathsf{c}_F}$ are unconstrained, as opposed to $a^0_{\mathsf{c}_{set}}$ in *EnumCtr*. The values of $a^0_{\mathsf{c}_F}$ mapped by indexes $\mathsf{c}_1, \ldots, \mathsf{c}_n$ are constrained by $\mathsf{c}^p_1, \ldots, \mathsf{c}^p_n$ via array access, i.e., $a^0_{\mathsf{c}_F}[c_i]$ is asserted to be true or false based on c^p_i, with $1 \leq i \leq n$. We then apply pointwise conjunction to c_S and $a^0_{\mathsf{c}_F}$ via the

map_f SMT operator; we go from a_F^0 to a_F^n to keep the array index in step with the arena overapproximation. Indexes whose values were false in S remain so in F, and indexes whose values were true in S store the filter's predicate evaluation.

$$\underbrace{\bigwedge_{1\leq i\leq n} \mathrm{ite}\left(\mathsf{c}_i^p, a_{\mathsf{c}_F}^0[c_i], \neg a_{\mathsf{c}_F}^0[c_i]\right)}_{\text{predicate-based constraining}} \wedge \underbrace{a_{\mathsf{c}_F}^n = map_\wedge(\mathsf{c}_S, a_{\mathsf{c}_F}^0)}_{\text{pointwise conjunction}} \wedge \underbrace{\mathsf{c}_F = a_{\mathsf{c}_F}^n}_{\text{cell equality}} \quad (FilterCtr)$$

Both encodings generate a linear amount of constraints, since n $p[\mathsf{c}_i/x]$ predicates have to be considered. Unlike with *EnumCtr*, *FilterCtr* does not contain many *store* operations, due to the usage of map_f. This avoids the need to create intermediary arrays, and is not possible in *EnumCtr* due to its constant array.

Set Map. The expression $\{e : x \in S\}$ can be used to construct a set M from a set S, having all the elements of M as $e[y/x]$, with $y \in S$. For example, the expression $\{x \div 5 : x \in \{4, 5, 6\}\}$ yields the set $\{0, 1\}$, with $\div$ denoting standard integer division. To reduce set map we use rule `Map`.

$$\begin{aligned}\{e\colon x \in \mathsf{c}_S\} : \mathsf{Set}[\tau] \text{ and } \mathsf{c}_S{\rightarrow}\mathsf{c}_1, \ldots, \mathsf{c}_n \\ \rightarrowtail \left(e[\mathsf{c}_1/x] : \tau, \ldots, e[\mathsf{c}_n/x] : \tau \rightarrowtail \mathsf{c}_1^e :, \ldots, \mathsf{c}_n^e\right) \\ \rightarrowtail \mathsf{c}_M \mid \mathsf{c}_M : \mathsf{Set}[\tau], \mathsf{c}_M{\rightarrow}\mathsf{c}_1^e, \ldots, \mathsf{c}_n^e \mid MapCtr\end{aligned} \quad (\texttt{Map})$$

The constraints added in rule `Map` are similar to those added in rule `Enum`. The difference between them is that set enumeration precisely defines the elements to be added to the new set cell, while set map is based on an existing set cell, which is a set overapproximation. Due to this, membership in M has to be guarded by membership in S, leading to a linear amount of constraints being generated.

$$\underbrace{a_{\mathsf{c}_M}^0 = K_\tau(\bot)}_{\text{empty set}} \wedge \underbrace{\bigwedge_{1\leq i\leq n} \mathrm{ite}\left(\begin{array}{c}\mathsf{c}_S[\mathsf{c}_i], \\ a_{\mathsf{c}_M}^i = store(a_{\mathsf{c}_M}^{i-1}, \mathsf{c}_i^e, \top), \\ a_{\mathsf{c}_M}^i = a_{\mathsf{c}_M}^{i-1}\end{array}\right)}_{\text{set updates}} \wedge \underbrace{\mathsf{c}_M = a_{\mathsf{c}_M}^n}_{\text{cell equality}} \quad (MapCtr)$$

3.2 Encoding TLA+ Functions using Arrays

We use arrays to encode TLA$^+$ functions directly as functions themselves. To do this, arrays are used in their general format, with a function $f : \mathsf{s}_{\tau_1} \rightarrow \mathsf{s}_{\tau_2}$ being encoded as an array of sort $\mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}$. Since functions with a finite domain can rely on infinite sorts, e.g., the integer numbers, the encoding of each function also includes constraints defining its domain set, by means of the rules described in the previous section; the result of a function application to a value outside its domain is undefined in TLA$^+$. This approach allows us to generate SMT constraints that follow directly from TLA$^+$, making the encoding not only more efficient, but also more natural to describe. In contrast, the constants encoding represents functions explicitly as sets of pairs of form $\{\langle x, f(x)\rangle : x \in \text{DOMAIN}f\}$. Due to this, its function manipulation relies on set manipulation, e.g., function comparison is encoded as set comparison, leading to a quadratic amount of constraints. The reduction rules used to handle functions are presented below.

Function Definition. The definition of a function in TLA$^+$ is an expression of the form $[x \in S \mapsto e]$, which maps every domain value v to the expression $e[v/x]$. This definition is similar to that of set map $\{e : x \in S\}$, and thus generates constraints in a similar fashion to rule `Map`. The main difference is that the evaluations of the expression $e[v/x]$ are stored as array values, rather than array indexes, i.e., function definition uses $store(a, v, e[v/x])$ and set map uses $store(a, e[v/x], \top)$, with v being a value in the function's domain or the set being mapped. Every encoded function has a single argument, with multiple arguments being rewritten as tuples in preprocessing.

Unlike with set cells, a function cell c_F in the arena does not directly point to its values, with the arrays encoding adding two edges to c_F, $\mathsf{c}_F \xrightarrow{1} \mathsf{c}_{F_{dom}}$ and $\mathsf{c}_F \xrightarrow{2} \mathsf{c}_{F_{pairs}}$. Cell $\mathsf{c}_{F_{dom}}$ represents the function's domain and cell $\mathsf{c}_{F_{pairs}}$ represents the set of pairs $\{\langle x, f(x)\rangle : x \in \text{DOMAIN} f\}$. Cell $\mathsf{c}_{F_{pairs}}$, despite being in the arena, has no SMT constraints modelling it in the arrays encoding, with its sole purpose being to help propagate the arena edges of the function's codomain elements.

Function Domain. Accessing a function's domain is trivial in the arrays encoding, since the domain set is generated during function definition. This results in a simple access to the array representing the domain.

Function Update. The update of a TLA$^+$ function f is done by changing the result of applying f to an argument arg, $f[arg]$, to be a given value v, via the expression $[f \text{ EXCEPT! } [arg] = v]$. The update will produce a new function g which is identical f, except that $g[arg] = v$ if $arg \in \text{DOMAIN} f$. The arrays encoding generates a single array update constraint in this case.

Function Application. The application of a function to an argument arg is conceptually simple, but is quite intricate to realize, as can be seen in rule `FunApp`. The arrays encoding uses an oracle to check that c_{arg} is in the domain and to gather the arena edges of c_{res}. The *FunAppCtr* constraints ensure that the oracle chooses the correct index and equates the result cell to an array access on c_F. Note that the value of c_{res} comes directly from the function application expression itself, with the oracle only been needed to gather the arena edges of c_{res}, if $m > 0$, via c^p. The need for an oracle is restricted to functions whose codomain contain structured data, e.g., $f : \mathsf{Int} \to \mathsf{Set}[\mathsf{Int}]$. If this is not the case, e.g., $g : \mathsf{Int} \to \mathsf{Int}$, rule `FunApp` is simplified and *FunAppCtr* becomes $\mathsf{c}_{res} = \mathsf{c}_F[\mathsf{c}_{arg}]$.

$$\begin{aligned}
&\mathsf{c}_F[\mathsf{c}_{arg}] : \tau \text{ and } \mathsf{c}_F \xrightarrow{1} \mathsf{c}_{F_{dom}} \to \mathsf{c}^d_1, \ldots, \mathsf{c}^d_n \text{ and } \mathsf{c}_F \xrightarrow{2} \mathsf{c}_{F_{pairs}} \to \mathsf{c}^p_1, \ldots, \mathsf{c}^p_n \\
&\quad \rightarrowtail \big(\mathsf{FROM}\ \mathsf{c}^p_1, \ldots, \mathsf{c}^p_n\ \mathsf{BY}\ \theta : \langle \tau_{arg}, \tau\rangle \mid \theta : \mathsf{Int} \mid 0 \le \theta \le n \rightarrowtail \mathsf{c}^p\big) \\
&\quad\quad \text{and } \mathsf{c}^p[2] \to \mathsf{c}_1, \ldots, \mathsf{c}_m \\
&\quad \rightarrowtail \mathsf{c}_{res} \mid \mathsf{c}_{res} : \tau, \mathsf{c}_{res} \to \mathsf{c}_1, \ldots, \mathsf{c}_m \mid \mathit{FunAppCtr}
\end{aligned} \tag{FunApp}$$

$$\underbrace{\bigwedge_{1 \le i \le n} \wedge \begin{array}{l}(\theta = i \to \mathsf{c}_{arg} = \mathsf{c}^d_i \wedge \mathsf{c}_{F_{dom}}[\mathsf{c}^d_i]) \\ (\theta = 0 \to \mathsf{c}_{arg} \neq \mathsf{c}^d_i \vee \neg \mathsf{c}_{F_{dom}}[\mathsf{c}^d_i])\end{array}}_{\text{oracle constraining}} \underbrace{\wedge\, \mathsf{c}_{res} = \mathsf{c}_F[\mathsf{c}_{arg}]}_{\text{cell equality}} \tag{FunAppCtr}$$

3.3 Correctness of the Reduction to Arrays

Correctness of the ARS is given by four properties: finiteness of the models, compliance to the target SMT theories, termination of any reduction sequence, and soundness of the reductions. These properties have their correctness sketched for the constants encoding in [13], with detailed proofs present in [26]. Since we rely on the existing ARS and restrict our changes to mainly affect constraint generation, we have the same degree of overapproximation and the correctness arguments made for the constants encoding are in large part valid for the arrays encoding. We present below the definition of a KerA$^+$ model and detail, for each property, how the use of arrays affects the correctness arguments and how they can be adjusted to remain valid.

Models. Every satisfiable KerA$^+$ formula has a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{D}$ is the model domain, consisting of a disjoint union of sets $\mathcal{D}_1, ..., \mathcal{D}_n$, with $\mathcal{D}_i$, $1 \leq i \leq n$, containing the values for type τ_i, and $\mathcal{I}$ is the model interpretation, consisting of assignments of domain values to KerA$^+$ constants. Models are used to access cell values, with the value of a KerA$^+$ expression e in model $\mathcal{M}$ being $[\![e]\!]^{\mathcal{M}}$. In $s_{before} \rightsquigarrow s_{after}$, we go from $\mathcal{M}_{before}$ to $\mathcal{M}_{after}$, with $\mathcal{M}_{after}$ containing the interpretation of additional constants and being thus an extension of $\mathcal{M}_{before}$.

Finiteness. This property states that every interpretation of a KerA$^+$ expression is defined only over finite values. Its proof is derived from the finiteness of the elements being modelled. In the arrays encoding, we potentially use arrays with infinite sorts, e.g., the integers, but all SMT interpretations that can be derived from such arrays are finite, since we encode only finite TLA$^+$ data structures. This guarantees finiteness of all KerA$^+$ models in the arrays encoding.

Theory Compliance. This property states that any sequence of states $s_0 \rightsquigarrow ... \rightsquigarrow s_n$ has the formulas Φ_i, $1 \leq i \leq n$, in the first-order logic fragment containing only quantifier-free expressions over uninterpreted functions and integer arithmetic. Its proof is done by induction on the constraints generated. The constraint Φ_0 is always `true` and is thus trivially compliant. The inductive case is proved by showing that the constraint added by each rule are compliant. The rules in the arrays encoding only add array constraints, in addition to constraints supported by the constants encoding, so theory compliance is straightforward to guarantee.

Termination. This property states that every sequence of ARS reductions is finite, i.e., the reduction process always terminates. Its proof is based on ensuring that every rule r applied to a given state s_{before} yields a state s_{after} with e_{after} being smaller than e_{before}. An expression's length is given based on the length of its sub-expressions. The arrays encoding mainly changes constraint generation, and in the cases where rules are slightly modified they generate resulting expressions of the same size, thus guaranteeing termination.

Soundness. This property is described in Theorem 1. Both e and Φ are KerA$^+$ expressions, but Φ is in the first-order logic fragment supported by SMT solvers. Fundamentally, the ARS is rewriting a formula to forward it to the solver. The soundness proof consists of case analysis of each reduction rule to establish that $e_{before} \wedge \Phi_{before}$ is equisatisfiable to $e_{after} \wedge \Phi_{after}$, no matter the rule applied in $s_{before} \rightsquigarrow s_{after}$. The case analysis, which describes how e_{after} and Φ_{after} can be derived from e_{before} and Φ_{before} for each rule, relies on six invariants of the reduction system. Three invariants, 1, 3, and 4, are encoding independent, and thus are the same as in [13], the remaining three, 2, 5, and 6, are changed due to the new representation of sets and functions. Below we show all six invariants, with the modifications needed to guarantee soundness for the arrays encoding.

Theorem 1. *Let $s_0 \rightsquigarrow ... \rightsquigarrow s_n$ be a sequence of states produced by the ARS, with $s_i = \langle e_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle$ and $1 \leq i \leq n$. Assume that e_0 is a formula, i.e., it has type* Bool*. Then e_0 is satisfiable iff the conjunction $e_n \wedge \Phi_n$ is satisfiable.*

Invariant 1 (type correctness) *In every reachable state $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ of the ARS, the expression e is well typed.*

Invariant 2 (arena membership) *In every reachable state $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ of the ARS, every cell c in either the expression e or the formula Φ is also in $\mathcal{A}$.*

Invariant 3 (model suitability) *Let $s_{before} \rightsquigarrow s_{after}$ be a reachable transition in the ARS, and $\mathcal{M}_{before}$ be a suitable model for s_{before}. An extended model $\mathcal{M}_{after}$ from $\mathcal{M}_{before}$ is suitable for s_{after}.*

Invariant 4 (overapproximation) *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS, and $\mathcal{M}$ be its model. Assume that c_{set} is a set cell in the arena $\mathcal{A}$ and that $\mathsf{c}_{set} \rightarrow \mathsf{c}_1, \ldots, \mathsf{c}_n$ are edges in $\mathcal{A}$, for some $n \geq 0$. Then, it holds that $[\![\mathsf{c}_{set}]\!]^{\mathcal{M}} \subseteq \{[\![\mathsf{c}_1]\!]^{\mathcal{M}}, ..., [\![\mathsf{c}_n]\!]^{\mathcal{M}}\}$.*

Invariant 5 (function domain) *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS. Assume that c_f is a function cell of type $\mathsf{s}_{\tau_1} \rightarrow \mathsf{s}_{\tau_2}$ in the arena $\mathcal{A}$. Then, there is a cell c_{dom} of type $\mathsf{s}_{\mathsf{Set}[\tau_1]}$ such that $\mathsf{c}_f \xrightarrow{1}_{\mathcal{A}} \mathsf{c}_{dom}$.*

Invariant 6 (domain reduction) *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS, and $\mathcal{M}$ be its model. Assume that c_f is a function cell and that $\mathsf{c}_f \xrightarrow{1} \mathsf{c}_{F_{dom}}$ is in the arena $\mathcal{A}$. Then, it follows that $[\![\mathsf{c}_{F_{dom}}]\!]^{\mathcal{M}} = [\![\text{DOMAIN} f]\!]^{\mathcal{M}}$.*

As described in sections 3.1 and 3.2, arrays precisely model TLA$^+$ sets and functions. The handling of sets revolves around membership constraints of form $\mathsf{c}_{set}[\mathsf{c}_i]$, which and can be set to true or false via *store*. Regarding functions, function application and update are trivially equivalent to array access and update. The more elaborate array operators also have a counterpart in TLA$^+$. Constant arrays are equivalent to a function definition for which all range values are the same constant, and array map is equivalent to set map. These equivalences explain how the changes in the arrays encoding do not invalidate the case analysis of the reduction rules used to prove Theorem 1, thus guaranteeing soundness.

4 Evaluation

In order to evaluate the performance impact of the arrays-based encoding, we implemented it in the Apalache model checker, which currently supports the constants encoding. Given a TLA$^+$ specification containing a property P, Apalache is capable of performing bounded model checking up to a length k and, if P is an inductive invariant, it can check if the property holds with an unbounded length. In both modes, Apalache checks if the SMT formula encoding the specification is satisfiable when conjoined with $\neg P$, and if that is the case a counterexample (CEX) in the form of a trace is produced using the arena information and the satisfiable assignment provided by the SMT solver. Our implementation adds new reduction rules to Apalache, which can be enabled via a CLI flag. When enabled, these rules replace the existing ones encoding sets and functions, as described in Section 3. In addition, we also extended Apalache's CEX generation to handle assignments to SMT formulas containing arrays. We use Z3 [7] as our back-end solver. Apalache is open-source and freely available[3].

We performed a number of experiments using Apalache and the explicit-state model checker TLC. For Apalache, we evaluated both its existing constants encoding and two versions of the arrays encoding we propose, called *arrays* and *funArrays*. The *arrays* version encodes both TLA$^+$ sets and functions as arrays, while the *funArrays* version encodes only TLA$^+$ functions as arrays. The purpose of having two versions of our encoding is to evaluate the impact of encoding sets and functions as arrays separately. Our evaluation setup consisted of a machine with 64 AMD EPYC 7452 processors and 256 GB of memory. We first present the benchmarks used and then discuss the results obtained.

4.1 Benchmarks

We consider the TLA$^+$ specifications of three asynchronous protocols as benchmarks. The first benchmark is a specification of the asynchronous Byzantine agreement protocol by Bracha and Toueg [5], showed in a simplified version in Figure 2, to which we refer as *aba*. The second benchmark is a specification of the consensus algorithm with Byzantine faults in one communication step by Dobre and Suri [9], to which we refer as *cab*. The third benchmark is a specification of the asynchronous non-blocking atomic commitment protocol by Guerraoui [12], to which we refer as *nac*. The common use of *aba* and *cba* is in replication scenarios with $N = 3F+1$ replica nodes to tolerate F failures, while the *nac* protocol is typically used for partitioned databases. The specifications are available online[4].

4.2 Results

For each specification we check a variation of the agreement property. The results are shown in Figure 4. We can see that both *arrays* and *funArrays* scale in

[3] Available at https://github.com/informalsystems/apalache
[4] Available at https://github.com/informalsystems/apalache-bench

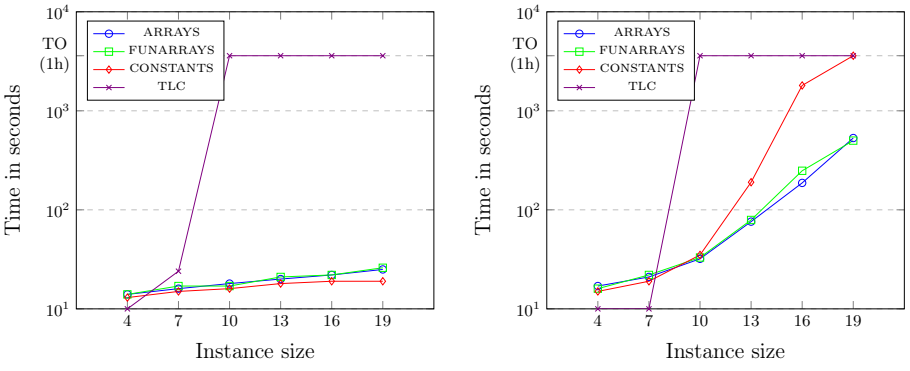
performance better that the constants encoding, with an order of magnitude improvement for some instances. It is also worth pointing out that *arrays* and *funArrays* were able to reach a result before the time limit in 29 and 28 instances, respectively, while the constants encoding was able to do so in only 20 instances. In regards to TLC, it performed worse than the three Apalache encodings in the nontrivial cases, only reaching a result before the time limit in 8 instances.

5 Related Work

An extensive discussion of works related to symbolic model checking for TLA$^+$ can be found in [13]. Here we focus exclusively on closely related publications. The IVy Prover [20] was designed to tackle verification of distributed algorithms with a decidable fragment of relational first-order logic. Some distributed algorithms, such as the one in Figure 2, cannot be directly expressed in this fragment however, due to the use of power sets and set cardinalities. Recent efforts have focused on offering support to reason about set cardinalities [4], but limitations remain. Cut-off based techniques to automatically infer invariants of distributed algorithms in the IVy language, such as relational abstractions of Paxos and two-phase commit, have been recently proposed [10,11]. Similar benchmarks are used in [22] to infer generalized invariants from finite instances of TLA$^+$ and semi-automatically prove invariants with TLAPS. Specifications of fault-tolerant distributed algorithms encoded as threshold automata can be efficiently verified with ByMC [15,24]. The manual rewriting of an algorithm into threshold automata is, however, usually beyond the skills of a typical TLA$^+$ user. The work closest to ours involves the use of SMT arrays to encode EventB and TLA$^+$ specifications in ProB [21]. The focus on ProB aims at handling infinite data structures, in contrast to our choice to work with bounded overapproximations. Reasoning about infinite domains implies the use of quantifiers, which prevents the use of efficient decision procedures available for the decidable fragment of SMT, with this approach been shown to underperform when compared against Apalache in checking the benchmarks from [13]. An important last point to mention is that CVC5 has its own non-standard SMT theory of sets [1]. This theory, however, cannot currently handle nested sets, which is a very commonly used TLA$^+$ construct. It remains as a viable alternative to the SMT theory of arrays for the encoding of flat sets, but whose use implies important restrictions to the input language and, consequentially, to practical application.

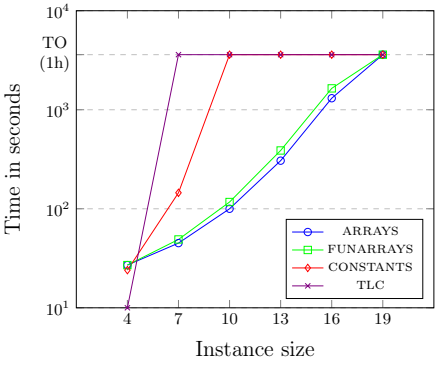
6 Conclusions

We propose an encoding of the main TLA$^+$ constructs into the SMT theory of arrays, with the goal of providing the SMT solver with the structural information it needs to efficiently reach a solution. We implemented our encoding into the Apalache model checker and our evaluation indicates that our arrays-based encoding provides a significant performance improvement when compared against Apalache's existing SMT encoding and the explicit-state model checker TLC.

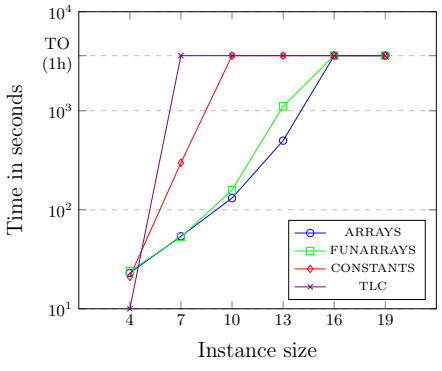


(a) Results for *aba* OK.

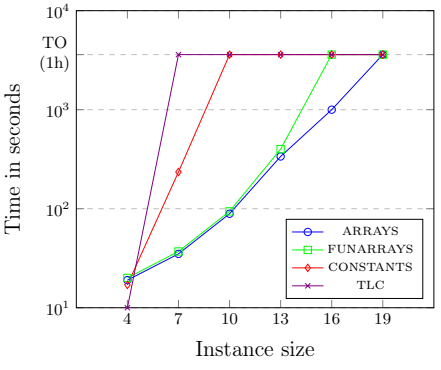
(b) Results for *aba* NotOK.



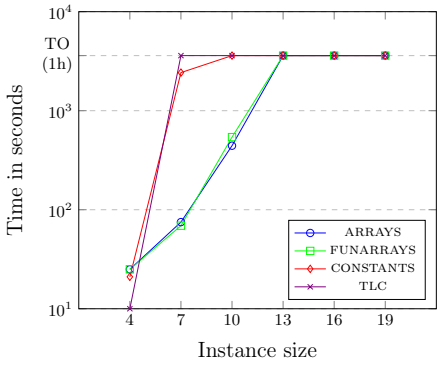
(c) Results for *cab* OK.



(d) Results for *cab* NotOK.



(e) Results for *nac* OK.



(f) Results for *nac* NotOK.

Fig. 4: Time in checking agreement for *aba*, *cab*, and *nac*. Specifications were ran in two configurations, one in which agreement is expected to hold (OK) and one in which it is not (NotOK). Instance size stands for the number of nodes used, and the time is given in seconds in logarithmic scale; Timeout (TO) is 1 hour.

Encoding the remaining TLA$^+$ constructs in a structure-preserving way, be it via SMT arrays or algebraic datatypes, remains an interesting research avenue.

Acknowledgements Rodrigo Otoni and Natasha Sharygina's work was supported by the Swiss National Science Foundation, via grants 200021_197353 and 200021_185031, respectively. Igor Konnov and Jure Kukovec's work was supported by the Interchain Foundation. The authors thank Shon Feder for his kind assistance in preparing the evaluation infrastructure.

References

1. Bansal, K., Reynolds, A., Barrett, C., Tinelli, C.: A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT. In: Proceedings of the 8th International Joint Conference on Automated Reasoning. pp. 82–98 (2016)
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: CVC5: A Versatile and Industrial-Strength SMT Solver. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442 (2022)
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep. (2021), Available at `https://smtlib.cs.uiowa.edu`
4. Berkovits, I., Lazic, M., Losa, G., Padon, O., Shoham, S.: Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In: Proceedings of the 31st International Conference on Computer Aided Verification. pp. 245–266 (2019)
5. Bracha, G., Toueg, S.: Asynchronous Consensus and Broadcast Protocols. Journal of the ACM **32**(4), 824–840 (1985)
6. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA+ Proof System: Building a Heterogeneous Verification Platform. In: Proceedings of the 7th International Colloquium on the Theoretical Aspects of Computing. p. 44 (2010)
7. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008)
8. De Moura, L., Bjørner, N.: Generalized, Efficient Array Decision Procedures. In: Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design. pp. 45–52 (2009)
9. Dobre, D., Suri, N.: One-step Consensus with Zero-Degradation. In: Proceedings of the 36th International Conference on Dependable Systems and Networks. pp. 137–146 (2006)
10. Goel, A., Sakallah, K.: On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In: Proceedings of the 13th NASA Formal Methods International Symposium. p. 131–150 (2021)
11. Goel, A., Sakallah, K.A.: Towards an Automatic Proof of Lamport's Paxos. In: Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design. pp. 112–122 (2021)
12. Guerraoui, R.: On the Hardness of Failure-Sensitive Agreement Problems. Information Processing Letters **79**(2), 99–104 (2001)
13. Konnov, I., Kukovec, J., Tran, T.H.: TLA+ Model Checking Made Symbolic. Proc. ACM Program. Lang. **3**(OOPSLA), 123:1–123:30 (2019)

14. Konnov, I., Kuppe, M., Merz, S.: Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. In: Proceedings of the 11th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. pp. 88–105 (2022)
15. Konnov, I., Lazić, M., Stoilkovska, I., Widder, J.: Tutorial: Parameterized Verification with Byzantine Model Checker. In: Proceedings of the 40th International Conference on Formal Techniques for Distributed Objects, Components, and Systems. pp. 189–207 (2020)
16. Kukovec, J., Tran, T.H., Konnov, I.: Extracting Symbolic Transitions from TLA+ Specifications. Science of Computer Programming **187**, 102361 (2020)
17. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
18. Merz, S., Vanzetto, H.: Encoding TLA+ into unsorted and many-sorted first-order logic. Science of Computer Programming **158**, 3–20 (2018)
19. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses Formal Methods. Communications of the ACM **58**(4), 66–73 (2015)
20. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety Verification by Interactive Generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 614–630 (2016)
21. Schmidt, J., Leuschel, M.: Improving SMT Solver Integrations for the Validation of B and Event-B Models. In: Proceedings of the 26th International Conference on Formal Methods for Industrial Critical Systems. pp. 107–125 (2021)
22. Schultz, W., Dardik, I., Tripakis, S.: Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. In: Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design. pp. 273–283 (2022)
23. Sivasubramanian, S.: Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. p. 729–730 (2012)
24. Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. International Journal on Software Tools for Technology Transfer **24**(1), 33–48 (2022)
25. Stump, A., Barrett, C., Dill, D., Levitt, J.: A Decision Procedure for an Extensional Theory of Arrays. In: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science. pp. 29–37 (2001)
26. Tran, T.H.: Symbolic Verification of TLA+ Specifications with Applications to Distributed Algorithms. Ph.D. thesis, Technische Universität Wien (2023), upcoming thesis
27. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Proceedings of the 10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods. pp. 54–66 (1999)

AutoHyper: Explicit-State Model Checking for HyperLTL

Raven Beutner(✉) and Bernd Finkbeiner

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{raven.beutner,finkbeiner}@cispa.de

Abstract. HyperLTL is a temporal logic that can express hyperproperties, i.e., properties that relate multiple execution traces of a system. Such properties are becoming increasingly important and naturally occur, e.g., in information-flow control, robustness, mutation testing, path planning, and causality checking. Thus far, complete model checking tools for HyperLTL have been limited to alternation-free formulas, i.e., formulas that use only universal or only existential trace quantification. Properties involving quantifier alternations could only be handled in an incomplete way, i.e., the verification might fail even though the property holds. In this paper, we present `AutoHyper`, an explicit-state automata-based model checker that supports full HyperLTL and is complete for properties with arbitrary quantifier alternations. We show that language inclusion checks can be integrated into HyperLTL verification, which allows `AutoHyper` to benefit from a range of existing inclusion-checking tools. We evaluate `AutoHyper` on a broad set of benchmarks drawn from different areas in the literature and compare it with existing (incomplete) methods for HyperLTL verification.

1 Introduction

Hyperproperties [16] are system properties that relate multiple executions of a system. Such properties are of increasing importance as they naturally occur, e.g., in information-flow control [36], robustness [22], linearizability [30,31], path planning [39], mutation testing [27], and causality checking [18]. A prominent logic to express hyperproperties is HyperLTL, which extends linear-time temporal logic (LTL) with explicit trace quantification [15]. HyperLTL can, for instance, express generalized non-interference (GNI) [34], stating that the high-security input of a system does not influence the observable output.

$$\forall \pi.\, \forall \pi'.\, \exists \pi''.\, \square \Big(\bigwedge_{a \in H} a_\pi \leftrightarrow a_{\pi''} \Big) \wedge \square \Big(\bigwedge_{a \in L \cup O} a_{\pi'} \leftrightarrow a_{\pi''} \Big) \qquad \text{(GNI)}$$

Here, H is a set of high-security input, L is a set of low-security inputs, and O is a set of low-security outputs. The formula states that for any traces π, π' there exists a third trace π'' that agrees with the high-security inputs of π and with the low-security inputs and outputs of π'. Any observation made by a low-security attacker is thus compatible with every possible high-security input.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 145–163, 2023.
https://doi.org/10.1007/978-3-031-30823-9_8

We are interested in the model checking (MC) problem of HyperLTL, i.e., whether a given (finite-state) system satisfies a given property. For HyperLTL, the structure of the quantifier prefix directly impacts the complexity of this problem. For alternation-free formulas (i.e., formulas that only use quantifiers of a single type), verification is well understood and is reducible to the verification of a trace property on a self-composition of the system [3]. This reduction has, for example, been implemented in `MCHyper` [29], a tool that can model check (alternation-free) HyperLTL formulas in systems of considerable size (circuits with thousands of latches).

Verification is much more challenging for properties involving quantifier alternations (such as GNI from above). While MC algorithms supporting full HyperLTL exist (see [15,29]), they have not been implemented yet. Instead, over the years, a number of approaches to the verification of such properties in practice have been made: Finkbeiner et al. [29] and D'Argenio et al. [22] manually strengthen properties with quantifier alternation into properties that are alternation-free and can be checked by `MCHyper`. Coenen et al. [19] instantiate existential quantification in a $\forall^*\exists^*$ property (i.e., a property involving an arbitrary number of universal quantifiers followed by an arbitrary number of existential quantifiers, such as GNI) with an explicit (user-provided) strategy, thus reducing to the verification of an alternation-free formula. Alternatively, the strategy that resolves existential quantification can be automatically synthesized [7]. Hsu et al. [31] present a bounded model checking (BMC) approach for HyperLTL that is implemented in `HyperQube`. See Section 4 for more details.

While all these verification tools can verify (or refute) interesting properties, they all suffer from the same fundamental limitation: they are *incomplete*. That is, for all the tools above, we can come up with verification instances where they fail, not because of resource constraints but because of inherent limitations in the underlying verification algorithm. Moreover, such instances are not rare events but are encountered regularly in practice. For example, many of the benchmarks used to evaluate `HyperQube` (by Hsu et al. [31]) do not admit a strategy to resolve existential quantification. Conversely, many of the properties verified by Coenen et al. [19] (such as GNI) cannot be verified using BMC [31].

AutoHyper. In this paper, we present `AutoHyper`, a model checker for HyperLTL. Our tool checks a hyperproperty by iteratively eliminating trace quantification using automata-complementations, thereby reducing verification to the emptiness check of an automaton [29]. Importantly – and different from previous tools for HyperLTL verification such as `MCHyper` [29,19] and `HyperQube` [31] – `AutoHyper` can cope with (and is *complete* for) arbitrary HyperLTL formulas. Model checking using `AutoHyper` does not require manual effort (such as writing an explicit strategy in `MCHyper` [19]), nor does a user need to worry if the given property can even be verified with a given method. `AutoHyper` thus provides a "push-button" model checking experience for HyperLTL.[1]

[1] The name of `AutoHyper` is derived from the fact that it is both **Auto**mata-based and **Auto**matic (i.e., it is complete and does not require any user intervention).

To improve AutoHyper's efficiency, we make the (theoretical) observation that we can often avoid explicit automaton complementation and instead reduce to a language inclusion check on Büchi automata (cf. Proposition 1). On the practical side, this enables AutoHyper to resort to a range of mature language inclusion checkers, including spot [26], RABIT [17], BAIT [25], and FORKLIFT [24].

Evaluation. Using AutoHyper, we extensively study the practical aspects of model checking HyperLTL properties with quantifier alternations. To evaluate the performance of explicit-state model checking, we apply AutoHyper to a broad range of benchmarks taken from the literature and compare it with existing (incomplete) tools. We make the surprising observation that – at least on the currently available benchmarks – explicit-state MC as implemented in AutoHyper performs on-par (and frequently outperforms) symbolic methods such as BMC [31]. Our benchmarks stem from various areas within computer science, so AutoHyper should – thanks to its "push-button" functionality, completeness, and ease of use – be a valuable addition to many areas.

Apart from using AutoHyper as a practical MC tool, we can also use it as a complete baseline to systematically evaluate existing (incomplete) methods. For example, while it is known that replacing existential quantification with a strategy (as done by Coenen et al. [19]) is incomplete, it was, thus far, unknown if this incompleteness occurs frequently or is merely a rare phenomenon. We use AutoHyper to obtain a ground truth and evaluate the strategy-based verification approach in terms of its effectiveness (i.e., how many instances it can verify despite being incomplete) and efficiency.

Structure. The remainder of this paper is structured as follows. In Section 2, we introduce HyperLTL. We recap automata-based verification (which we abbreviate ABV) and our new approach utilizing language inclusion checks in Section 3. We discuss alternative verification approaches for HyperLTL in Section 4. In Section 6, we compare different backend solving techniques and study the complexity of HyperLTL MC with multiple quantifier alternations in practice; In Section 7, we evaluate ABV on a set of benchmarks from the literature and compare with the bounded model checker HyperQube [31]; In Section 8 we use AutoHyper for a detailed analysis of (and comparison with) strategy-based verification [19,7].

2 Preliminaries

We fix a set of atomic propositions AP and define $\Sigma := 2^{AP}$. HyperLTL [15] extends LTL with explicit quantification over traces, thereby lifting it from a logic expressing trace properties to one expressing hyperproperties [16]. Let $\mathcal{V}$ be a set of trace variables. We define HyperLTL formulas by the following grammar:

$$\psi := a_\pi \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc\psi \mid \psi \,\mathcal{U}\, \psi$$
$$\varphi := \exists\pi.\,\varphi \mid \forall\pi.\,\varphi \mid \psi$$

where $\pi \in \mathcal{V}$ and $a \in AP$.

We assume that the formula is closed, i.e., all trace variables that are used in the body are bound by some quantifier. The semantics of HyperLTL is given with respect to a trace assignment $\Pi : \mathcal{V} \rightharpoonup \Sigma^\omega$ mapping trace variables to traces. For $\pi \in \mathcal{V}$ and $t \in \Sigma^\omega$, we write $\Pi[\pi \mapsto t]$ for the trace assignment obtained by updating the value of π to t. Given a set of traces $\mathbb{T} \subseteq \Sigma^\omega$, a trace assignment Π, and $i \in \mathbb{N}$, we define:

$$
\begin{array}{lll}
\Pi, i \models a_\pi & \text{iff} & a \in \Pi(\pi)(i) \\
\Pi, i \models \neg\psi & \text{iff} & \Pi, i \not\models \psi \\
\Pi, i \models \psi_1 \wedge \psi_2 & \text{iff} & \Pi, i \models \psi_1 \text{ and } \Pi, i \models \psi_2 \\
\Pi, i \models \bigcirc \psi & \text{iff} & \Pi, i+1 \models \psi \\
\Pi, i \models \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \exists j \geq i.\, \Pi, j \models \psi_2 \text{ and } \forall i \leq k < j.\, \Pi, k \models \psi_1
\end{array}
$$

$$
\begin{array}{lll}
\Pi \models_{\mathbb{T}} \psi & \text{iff} & \Pi, 0 \models \psi \\
\Pi \models_{\mathbb{T}} \exists \pi.\, \varphi & \text{iff} & \exists t \in \mathbb{T}.\, \Pi[\pi \mapsto t] \models_{\mathbb{T}} \varphi \\
\Pi \models_{\mathbb{T}} \forall \pi.\, \varphi & \text{iff} & \forall t \in \mathbb{T}.\, \Pi[\pi \mapsto t] \models_{\mathbb{T}} \varphi
\end{array}
$$

A *transition system* is a tuple $\mathcal{T} = (S, S_0, \kappa, L)$ where S is a set of states, $S_0 \subseteq S$ is a set of initial states, $\kappa \subseteq S \times S$ is a transition relation, and $L : S \to \Sigma$ is a labeling function. We write $s \xrightarrow{\mathcal{T}} s'$ whenever $(s, s') \in \kappa$. A path is an infinite sequence $s_0 s_1 s_2 \cdots \in S^\omega$, s.t., $s_0 \in S_0$, and $s_i \xrightarrow{\mathcal{T}} s_{i+1}$ for all i. The associated trace is given by $L(s_0)L(s_1)L(s_2)\cdots \in \Sigma^\omega$. We write $\mathit{Traces}(\mathcal{T}) \subseteq \Sigma^\omega$ for the set of all traces generated by $\mathcal{T}$. We say $\mathcal{T}$ satisfies a HyperLTL property φ, written $\mathcal{T} \models \varphi$, if $\emptyset \models_{\mathit{Traces}(\mathcal{T})} \varphi$, where $\emptyset$ denotes the empty trace assignment.

3 Automata-based HyperLTL Model Checking

Given a system $\mathcal{T}$ and HyperLTL property φ, we want to decide whether $\mathcal{T} \models \varphi$. In this section, we recap the automata-based approach to the model checking of HyperLTL [29]. We further show how language inclusion checks can be incorporated into the model checking procedure to make use of a broad collection of mature language inclusion checkers.

3.1 Automata-based Verification

The idea of automata-based verification (ABV) [29] is to iteratively eliminate quantifiers and thus reduce MC to the emptiness check on an automaton. A non-deterministic Büchi automaton (NBA) is a tuple $\mathcal{A} = (Q, Q_0, \delta, F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, and $F \subseteq Q$ is a set of accepting states. We write $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ for the language of $\mathcal{A}$, i.e., all infinite words that have a run that visits states in F infinitely many times (see, e.g., [2]). For traces $t_1, \ldots, t_n \in \Sigma^\omega$, we write $\mathit{zip}(t_1, \ldots, t_n) \in (\Sigma^n)^\omega$ as the pointwise product, i.e., $\mathit{zip}(t_1, \ldots, t_n)(i) := (t_1(i), \ldots, t_n(i))$.

Let $\mathcal{T} = (S, S_0, \kappa, L)$ be a fixed transition system and let $\dot{\varphi}$ be some fixed closed HyperLTL formula (we use the dot to refer to the original formula and use φ, φ' to refer to subformulas of $\dot{\varphi}$). For some subformula φ that contains free trace variables $\pi_1, \dots, \pi_n$, we say an NBA $\mathcal{A}$ over Σ^n is $\mathcal{T}$*-equivalent* to φ, if for all traces $t_1, \dots, t_n$ it holds that $[\pi_1 \mapsto t_1, \dots, \pi_n \mapsto t_n] \models_{Traces(\mathcal{T})} \varphi$ iff $zip(t_1, \dots, t_n) \in \mathcal{L}(\mathcal{A})$. That is, $\mathcal{A}$ accepts exactly the zippings of traces that constitute a satisfying trace assignment for φ.

To check if $\mathcal{T} \models \dot{\varphi}$, we inductively construct an automation $\mathcal{A}_\varphi$ that is $\mathcal{T}$-equivalent to φ for each subformula φ of $\dot{\varphi}$. For the (quantifier-free) LTL body of $\dot{\varphi}$, we can construct this automaton via a standard LTL-to-NBA construction [29,2]. Now consider some subformula $\varphi' = \exists \pi.\varphi$ where φ' contains free trace variables $\pi_1, \dots, \pi_n$ and so φ contains free trace variables $\pi_1, \dots, \pi_n, \pi$. We are given an inductively constructed NBA $\mathcal{A}_\varphi = (Q, Q_0, \delta, F)$ over Σ^{n+1} that is $\mathcal{T}$-equivalent to φ. We define the automaton $\mathcal{A}_{\varphi'}$ over Σ^n as $\mathcal{A}_{\varphi'} := (S \times Q, S_0 \times Q_0, \delta', S \times F)$ where δ' is defined as

$$\delta'\Big((s,q), \langle l_1, \dots, l_n \rangle\Big) := \Big\{ (s', q') \mid s \xrightarrow{\mathcal{T}} s' \ \wedge \ q' \in \delta\big(q, \langle l_1, \dots, l_n, L(s) \rangle\big) \Big\}.$$

Informally, $\mathcal{A}_{\varphi'}$ reads the zippings of traces $t_1, \dots, t_n$ and guesses a trace $t \in Traces(\mathcal{T})$ such that $zip(t_1, \dots, t_n, t) \in \mathcal{L}(\mathcal{A}_\varphi)$. It is easy to see that $\mathcal{A}_{\varphi'}$ is $\mathcal{T}$-equivalent to φ'. To handle universal trace quantification, we consider a formula $\varphi' = \forall \pi.\varphi$ as "$\varphi' = \neg\exists\pi.\neg\varphi$" and combine the construction for existential quantification with an automaton complementation.

Following the inductive construction, we obtain an automaton $\mathcal{A}_{\dot{\varphi}}$ over the singleton alphabet Σ^0 that is $\mathcal{T}$-equivalent to $\dot{\varphi}$. By definition of $\mathcal{T}$-equivalence, $\mathcal{T} \models \dot{\varphi}$ iff $\emptyset \models_{Traces(\mathcal{T})} \dot{\varphi}$ iff $\mathcal{A}_{\dot{\varphi}}$ is non-empty (which we can decide [21]).

3.2 HyperLTL Model Checking by Language Inclusion

The algorithm outlined above requires one complementation for each quantifier alternation in the HyperLTL formula. While we cannot avoid the theoretical cost of this complementation (see [36,15]), we can reduce to a, in practice, more tamable problem: *language inclusion*.

For a system $\mathcal{T}$, and a natural number $n \in \mathbb{N}$ we define $\mathcal{A}^n_{\mathcal{T}}$ as an NBA over Σ^n such that for any traces $t_1, \dots, t_n \in \Sigma^\omega$ we have $zip(t_1, \dots, t_n) \in \mathcal{L}(\mathcal{A}^n_{\mathcal{T}})$ if and only if $t_i \in Traces(\mathcal{T})$ for every $1 \leq i \leq n$. We can construct $\mathcal{A}^n_{\mathcal{T}}$ by building the n-fold self-composition of $\mathcal{T}$ [3] and convert this to an automaton by moving the labels from states to edges and marking all states as accepting. We can now state a formal connection between language inclusion and HyperLTL MC (a proof can be found in the full version [9]):

Proposition 1. *Let $\dot{\varphi} = \forall \pi_1 \dots \forall \pi_n.\varphi$ be a HyperLTL formula (where φ may contain additional trace quantifiers) and let $\mathcal{A}_\varphi$ be an automaton over Σ^n that is $\mathcal{T}$-equivalent to φ. Then $\mathcal{T} \models \dot{\varphi}$ if and only if $\mathcal{L}(\mathcal{A}^n_{\mathcal{T}}) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$.*

We can use Proposition 1 to avoid a complementation for the outermost quantifier alternation. For example, assume $\dot{\varphi} = \forall \pi_1.\forall \pi_2.\exists \pi_3.\psi$ where ψ is quantifier-free. Using the construction from Section 3.1, we obtain an automaton $\mathcal{A}_{\exists \pi_3.\psi}$

that is $\mathcal{T}$-equivalent to $\exists \pi_3.\psi$ (we can construct $\mathcal{A}_{\exists \pi_3.\psi}$ in linear time in the size of $\mathcal{T}$). By Proposition 1, we then have $\mathcal{T} \models \dot{\varphi}$ iff $\mathcal{L}(\mathcal{A}^2_{\mathcal{T}}) \subseteq \mathcal{L}(\mathcal{A}_{\exists \pi_3.\psi})$.

Note that complementation and subsequent emptiness check is a theoretically optimal method to solve the (PSPACE-complete) language inclusion problem. Proposition 1 thus offers no asymptotic advantages over "standard" ABV in Section 3.1. In *practice* constructing an explicit complemented automaton is often unnecessary as the language inclusion or non-inclusion might be witnessed without a complete complementation [26,25,17,24]. This makes Proposition 1 relevant for the present work and the performance of `AutoHyper`.

4 Related Work and HyperLTL Verification Approaches

HyperLTL [15] is the most studied logic for expressing hyperproperties. A range of problems from different areas in computer science can be expressed as HyperLTL MC problems, including (optimal) path panning [39], mutation testing [27], linearizability [31], robustness [22], information-flow control [36], and causality checking [18], to name only a few. Consequently, any model checking tool for HyperLTL is applicable to many disciples within computer science and provides a unified solution to many challenging algorithmic problems. In recent years, different (mostly incomplete) methods for the verification of HyperLTL have been developed. We discuss them below (see the full version [9] for details).

Automata-based Model Checking. Finkbeiner et al. [29] introduce the automata-based model checking approach as presented in Section 3.1. For alternation-free formulas, the algorithms corresponds to the construction of the self-composition of a system [3] and is implemented in the `MCHyper` tool [29]. `MCHyper` can handle systems of significant size (well beyond the reach of explicit-state methods) but is unable to handle any quantifier alternation (the main motivation for `AutoHyper`). `hltl2mc` [15] is a prototype model checker for HyperLTL$_2$ (a fragment of HyperLTL with at most one alternation) built on top of `GOAL` [38]. In contrast to `hltl2mc`, `AutoHyper` supports properties with arbitrarily many quantifier alternations and features automata with symbolic alphabets – which is important to handle large systems with many atomic propositions, cf. Footnote 7.

Strategy-based Verification. Coenen et al. [19] verify $\forall^*\exists^*$ properties by instantiating existential quantification with an explicit strategy. This method – which we refer to as strategy-based verification (SBV) – comes in two flavors: either the strategy is provided by the user or the strategy is synthesized automatically. In the former case, model checking reduces to checking an alternation-free formula and can thus handle large systems, but requires significant user effort (and is thus no "push-button" technique). In the latter case, the method works fully automatically [8,7] but requires an expensive strategy synthesis. SBV is incomplete as the strategy resolving existentially quantified traces only observes finite prefixes of the universally quantified traces. While SBV can be made complete by

adding prophecy variables [7], the automatic synthesis of such prophecies is currently limited to very small systems and properties that are temporally safe [5]. We investigate both the performance and incompleteness of SBV in Section 8.

Bounded Model Checking. Hsu et al. [31] propose a bounded model checking (BMC) procedure for HyperLTL. Similar to BMC for trace properties [11], the system is unfolded up to a fixed depth, and pending obligations beyond that depth are either treated pessimistically (to show the satisfaction of a formula) or optimistically (to show the violation of a formula). While BMC for trace properties reduces to SAT-solving, BMC for hyperproperties naturally reduces to QBF-solving. As usual for bounded methods, BMC for HyperLTL is incomplete. For example, it can never show that a system satisfies a hyperproperty where the LTL body contains an invariant (as, e.g., is the case for GNI).[2] We compare `AutoHyper` and BMC (in the form of `HyperQube` [31]) in Section 7.

5 AutoHyper: Tool Overview

`AutoHyper` is written in `F#` and implements the automata-based verification approach described in Section 3.1 and, if desired by the user, makes use of the language-inclusion-based reduction from Section 3.2. `AutoHyper` uses `spot` [26] for LTL-to-NBA translations and automata complementations. To check language inclusion, `AutoHyper` uses `spot` (which is based on determinization), `RABIT` [17] (which is based on a Ramsey-based approach with heavy use of simulations), `BAIT` [25], and `FORKLIFT` [24] (both based on well-quasiorders). `AutoHyper` is designed such that communication with external automata tools is done via established text-based formats (opposed to proprietary APIs), namely the `HANOI` [1] and `BA` automaton formats. New (or updated) tools that improve on fundamental automata operations, such as complementation and inclusion checks, can thus be integrated easily. Internally we represent automata using symbolic alphabets (similar to `spot`). We store transition formulas as DNFs as this allows for very efficient SAT checks, which are needed during the product construction.

All experiments in this paper were conducted on a Mac Mini with an Intel Core i3 (i3-8100B) and 16GB of memory. We used `spot` version 2.11.1; `RABIT` version 2.4.5; `BAIT` commit 369e1a4; and `FORKLIFT` commit 5d519e3.

Input Formats. `AutoHyper` supports both explicit-state systems (given in a `HANOI`-like [1] input format) and symbolic systems that are internally converted

[2] BMC for trace properties can be made complete by using bounds on the unrolling depth (also called completeness thresholds) [14] and including loop conditions in the encoding [11]. As remarked by Hsu et al. [31], the same is much more challenging for hyperproperties, and no solutions have been proposed. Instead, Hsu et al. [31] propose an alternative unrolling semantics (which they call halting semantics) that can mitigate this incompleteness issue for programs that terminate after a *fixed* number of steps. This is a strong (and often unrealistic) assumption for general reactive systems.

to an explicit-state representation. The support for symbolic systems includes Aiger circuits, symbolic models written in a fragment of the NuSMV input language [13], and a simple boolean programming language [6].

Random Benchmarks. For our evaluation, we use both existing instances from various sources in the literature and randomly generated problems.[3] We generate random transition systems based on the Erdős–Rényi–Gilbert model [28]. Given a size n and a density parameter $p \in [0, 1]$, we generate a graph with n states, where for every two states s, s', there is a transition $s \to s'$ with probability p. To generate a graph with n edges and, in expectation, constant outdegree of k, we can choose $p = \frac{k}{n}$. We further ensure that the system is connected and all states have at least one outgoing edge. We generate random HyperLTL formulas (with a given quantifier prefix) by sampling the LTL matrix using `spot`'s `randltl`.

6 HyperLTL Model Checking Complexity in Practice

Before we turn our attention to benchmarks found in the literature, we compare the different backend inclusion checkers supported by `AutoHyper` by evaluating them on a large set of synthetic (random) benchmarks (in Section 6.1). Moreover, the random generation of benchmarks allows us to peek at formulas with more than one quantifier alternation. The theoretical hardness of model checking properties with multiple alternations has been studied extensively [15,36], and we analyze, for the first time, how these results transfer to practice (in Section 6.2).

6.1 Performance of Inclusion Checkers

As the first set of benchmarks, we compare the different backend inclusion checkers supported by `AutoHyper`. In Figure 1, we depict how many instances can be solved using the inclusion checks of `spot`, `BAIT`, `RABIT`, and `FORKLIFT` within a timeout of 10s and give the median running time used on the instances that could be solved within the timeout. We observe that `spot` clearly outperforms `RABIT`, `BAIT`, and `FORKLIFT` in terms of the percentage of instances that can be checked within 10s.[4] While, in general, `spot` solves the most instances, a manual inspection reveals that there are also instances that can only be solved by `RABIT`

[3] The advantage of randomly generated instances is twofold. First, it allows for the easy generation of a large set of benchmarks. Second, the random generation is parameterized by multiple parameters (such as system size, transition density, formula size, etc.), enabling a comprehensive analysis of the exact impact of different parameters on the model checking complexity in practice.

[4] We remark that `spot` operates on automata with a symbolic alphabet (i.e., transitions are defined as boolean formulas over AP). In contrast, `RABIT`, `BAIT`, and `FORKLIFT` only support explicit alphabets (i.e., automata with one symbol for each element in 2^{AP}).

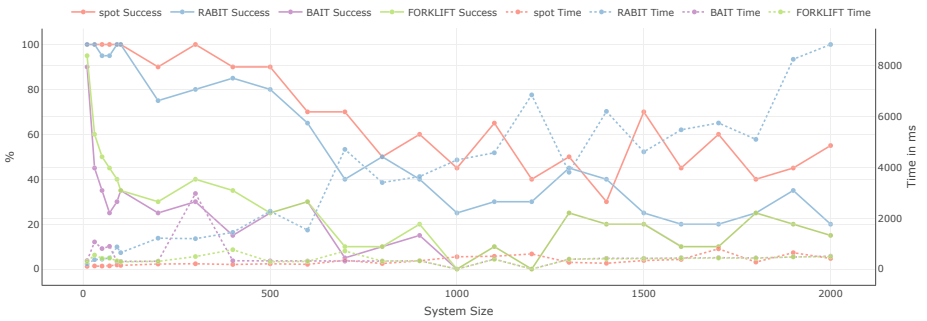


Fig. 1: We evaluate different backend solvers on instances of varying system size with an (on average) constant outdegree of 10 and a fixed property size of 20. We generate 20 samples per system size. We display both the success rate of each solver within a timeout of 10s (on the left axis) and the median running time on the solved instances (on the right axis).

or `BAIT`/`FORKLIFT`. This justifies why `AutoHyper` supports multiple backed inclusion checkers that implement different algorithms and thus excel on different problems (we will confirm this in Section 7). Moreover, our experiments provide evidence that HyperLTL MC is a natural source for challenging language inclusion benchmarks (see the full version [9]).

We remark that we set the timeout of 10s deliberately low to compute (and reproduce) the plots in a reasonable time (computing Figure 1 took about 3.5h). If a user wants to verify a given instance and does not require a result within a few seconds, running the solver for even longer will likely increase the success rate further (see also the evaluation in Section 7).

6.2 Model Checking Beyond ∀*∃*

Using randomly generated benchmarks, we can also peek at the practical complexity of model checking in the presence of multiple quantifier alternations. In *theory*, the model checking complexity of HyperLTL increases by one exponent with each quantifier alternation [15,36]. Using `AutoHyper`, we can, for the first time, investigate the model checking complexity in *practice*.

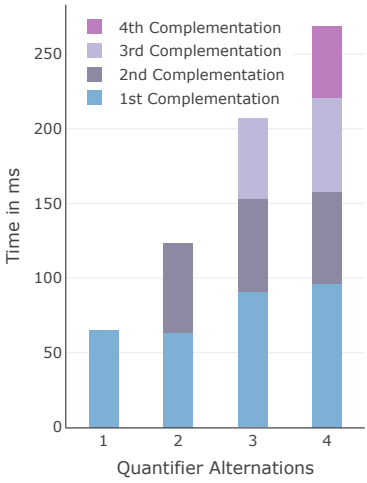


Fig. 2: For properties with a varying number of quantifier alterations, we display the average time spent on the automata complementation during model checking.

We model check randomly generated formulas with 1 to 4 quantifier alternations and visualize the total running time based on the cost of each complementation (using `spot`) in Figure 2 (recall that checking a formula with k alternations

Table 1: We depict the running time of AutoHyper when verifying GNI on the boolean programs taken from [6] and [10]. We give the program, the bitwidth (bw), the size of the intermediate explicit-state representation (Size), and the time taken by each solver. The timeout is set to 60s and indicated by a "-". The property holds in all cases. Times are given in seconds.

Program	bw	Size	t_{spot}	t_{RABIT}	t_{BAIT}	t_{FORKLIFT}
[6].1	1-bit	17	**0.52**	0.59	0.80	0.61
	3-bit	65	**0.56**	0.86	-	22.73
	4-bit	129	**0.99**	5.51	-	-
[6].2	1-bit	55	**0.53**	0.69	-	5.49
[6].3	1-bit	20	**0.52**	0.61	3.05	0.98
	3-bit	80	**0.61**	1.31	-	-
[6].4	1-bit	29	**0.52**	0.56	0.58	0.57
	3-bit	113	**0.67**	1.74	-	-

Program	bw	Size	t_{spot}	t_{RABIT}	t_{BAIT}	t_{FORKLIFT}
[10].1	1-bit	5	0.52	**0.56**	0.58	0.57
[10].2	1-bit	11	**0.51**	0.57	0.72	0.61
	2-bit	27	**0.52**	0.65	35.7	5.43
	4-bit	291	**1.46**	-	-	-
[10].3	1-bit	21	**0.52**	0.60	3.15	1.00
	3-bit	225	-	**45.2**	-	-
[10].4	1-bit	25	**0.52**	0.71	12.8	1.63
	3-bit	193	**0.98**	-	-	-

using ABV requires k automaton complementations). Although the number of quantifier alternations has an undeniable impact on the total running time (the cumulative height of each bar), the increase in runtime is not proportional to the (non-elementary) increase suggested by the theoretical analysis. Different from the theoretical analysis (where the $(k+1)$th complementation is exponentially more expensive than the kth), the cost of each complementation barely increases (or even decreases). This suggests that the $\mathcal{T}$-equivalent automata constructed in each iteration are, in practice, much smaller than indicated by the worst-case theoretical analysis. Verification of properties beyond one alternation is thus less infeasible than the theory suggests (at least on randomly generated test cases).

7 Evaluation on Symbolic Systems

In this section, we challenge AutoHyper with complex model checking problems found in the literature. Our benchmarks stem from a range of sources, including non-interference in boolean programs [6], symmetry in mutual exclusion algorithms [19], non-interference in multi-threaded programs [37], fairness in non-repudiation protocols [32], mutation testing [27], and path planning [39].

7.1 Model Checking GNI on Boolean Programs

We use AutoHyper to verify GNI on a range of boolean programs that process high-security and low-security inputs (taken from [6,10]). Table 1 depicts the runtime results using different backend solvers. We test each program with varying bitwidth and depict the largest bitwidth that can be solved by at least one solver (within a timeout of 60s). We, again, note that spot performs better than

Table 2: We evaluate `HyperQube` and `AutoHyper` on the benchmarks from [31]. We list the system and the property (as given in [31, Table 2]), the quantifier structure (Q^*), the verification result (Res) (✓ indicates that the property holds and ✗ that it is violated), and the total running time of either tool (t). For `HyperQube`, we additionally list the unrolling bound (k) and the unrolling semantics (Sem). For `AutoHyper`, we additionally list the size of the intermediate explicit state space (Size). Times are given in seconds.

				HyperQube [31]			**AutoHyper**	
System	**Spec**	Q^*	**Res**	k	**Sem**	t	**Size**	t
Bakery_3	φ_{S1}	∃∃	✗	7	pes	**1.9**	167	2.3
Bakery_3	φ_{S2}	∀∃	✗	12	pes	**2.0**	167	4.2
Bakery_3	φ_{S3}	∃∀	✗!	20	pes	**2.8**	167	34.6
Bakery_3	φ_{sym1}	∀∃	✗	10	pes	**1.7**	167	16.2
Bakery_3	φ_{sym2}	∀∃	✗	10	pes	**1.6**	167	2.9
Bakery_5	φ_{sym1}	∀∃	✗	10	pes	**17.3**	996	282.1
Bakery_5	φ_{sym2}	∀∃	✗	10	pes	18.2	996	**18.0**
SNARK-bug1	φ_{lin}	∀∃	✗	26	hpes	618.0	4941	**96.1**
$\text{3-Thread}_{correct}$	φ_{NI}	∀∃	✓	10	hopt	1.6	64	**1.3**
$\text{3-Thread}_{incorrect}$	φ_{NI}	∀∃	✗	57	hpes	12.8	368	**7.7**
$NRP : T_{correct}$	φ_{fair}	∃∀	✓	15	hopt	1.3	55	**0.5**
$NRP : T_{incorrect}$	φ_{fair}	∃∀	✓!	15	hopt	1.4	54	**0.8**
Mutant	φ_{mut}	∃∀	✓	8	hopt	1.1	32	**0.8**

other inclusion checkers and, in particular, scales better when the size of the system increases. Note that the number of atomic propositions is 3 in all instances, so `spot`'s support for symbolic alphabets has a negligible impact on the running time. We emphasize that not all instances in Table 1 can be verified using SBV [19,7] without a user-provided fixed lookahead. Likewise, BMC [31] can *never* verify GNI. This provides further evidence why complete model checking tools (of which `AutoHyper` is the first) are necessary.

7.2 Explicit Model Checking of Symbolic Systems

In this section, we evaluate `AutoHyper` on challenging symbolic models (NuSMV models [13]) that were used by Hsu et al. [31] to evaluate `HyperQube`.

The properties we verify cover a wide range of properties. For example, we verify that Lamport's bakery algorithm [33] does not satisfy various symmetry properties (as the algorithm prioritizes processes with a lower ticket ID); We

check linearizability[5] [30] on the SNARK datastructure [23] and identify a previously known bug; And, we generate model-based mutation test cases using the approach proposed by Fellner et al. [27]. Further details on the benchmarks are provided in [31].

We check each instance using both `HyperQube` and `AutoHyper` and depict the results in Table 2.[6] When using `AutoHyper` we always apply `spot`'s inclusion checker.[7] For `HyperQube` we use the unrolling semantics and unrolling depth listed in [31, Table 2]. We observe that for most instances – despite using explicit state methods and thus being complete (cf. Section 7.4) – `AutoHyper` performs on par with `HyperQube`. On instances using Lamport's bakery algorithm, BMC only needs to unroll to very shallow depths, resulting in very efficient solving, whereas `AutoHyper`'s running time is dominated by `spot`'s LTL-to-NBA translation (consuming up to 98% of the total time). Conversely, on the large SNARK example, `AutoHyper` performs significantly better.

7.3 Hyperproperties for Path Planning

As a last set of benchmarks, we use planning problems for robots encoded into HyperLTL as proposed by Wang et al. [39]. For example, the synthesis of a shortest path can be phrased as a $\exists\forall$ property that states that there exists a path to the goal such that all alternative paths to the goal take at least as long. Wang et al. [39] propose a solution to check the resulting HyperLTL property by encoding it in first-order logic, which is then solved by an SMT solver. While not competitive with state-of-the-art planning tools, HyperLTL allows one to express a broad range of problems (shortest path, path robustness, etc.) in a very general way. Hsu et al. [31] observe that the QBF encoding implemented in `HyperQube` outperforms the SMT-based approach by Wang et al. [39]. In this section, we evaluate `AutoHyper` on these planning-hyperproperties and compare it with `HyperQube`[8].

We depict the results in Table 3. It is evident that `AutoHyper` outperforms `HyperQube`, sometimes by orders of magnitude. This is surprising as planning problems (which are essentially reachability problems) on symbolic systems should be advantageous for symbolic methods such as BMC. The large size of the in-

[5] Linearizability asserts that any execution of a concurrent data structure corresponds to a sequential execution, which is naturally expressed as a $\forall\exists$ hyperproperty.

[6] For the two verification instances $(\text{Bakery}_3, \varphi_{S3})$ and $(NRP : T_{incorrect}, \varphi_{fair})$ `HyperQube` provides the wrong verification result. We mark such instances with a "!" to avoid confusion when comparing Table 2 with [31, Table 2]. In particular, the supposedly unfair version of the NRP protocol is, in fact, fair.

[7] The automata use a symbolic alphabet with up to 18 letters. A conversion to an explicit alphabet – as required for `RABIT`, `BAIT`, and `FORKLIFT` – is thus infeasible (this would require 2^{18} symbols).

[8] `AutoHyper` is intended as a model checking tool, i.e., it only checks if a property holds or is violated. However, as we show in the full version [9], we could use the counterexamples returned by the inclusion checker to *synthesize* an actual plan.

Table 3: We evaluate HyperQube and AutoHyper on hyperproperties that encode the existence of a shortest path (φ_{sp}) and robust path (φ_{rp}). We give the specification (Spec), the size of the grid (Grid), and the times taken by HyperQube and AutoHyper (t). For HyperQube, we additionally give the unrolling depth used (k) and the file size of the QBF generated (|QBF|). For AutoHyper, we additionally give the size of the generated explicit state space (Size). Times are given in seconds. The timeout is set to 20 min and indicated by a "-".

		HyperQube [31]			**AutoHyper**	
Spec	**Grid**	k	**|QBF|**	t	**Size**	t
φ_{sp}	10×10	20	8 MB	4.6	146	**0.7**
	20×20	40	26 MB	168.1	188	**1.5**
	40×40	80	-	-	408	**22.7**
	60×60	120	-	-	404	**88.8**
φ_{rp}	10×10	20	13 MB	4.2	266	**0.6**
	20×20	40	84 MB	22.4	572	**0.7**
	40×40	80	419 MB	265.0	1212	**1.6**
	60×60	120	-	-	1852	**3.7**

termediate QBF indicates that a more optimized encoding (perhaps specific to path planning) could improve the performance of BMC on such examples.

7.4 Bounded vs. Explicit-State Model Checking

Bounded model checking has seen remarkable success in the verification of trace properties and frequently scales to systems whose size is well out of scope for explicit-state methods [20]. Similarly, in the context of *alternation-free* hyperproperties, symbolic verification tools such as MCHyper [29] (which internally reduces to the verification of a circuit using ABC [12]) can verify systems that are well beyond the reach of explicit-state methods. In contrast, in the context of model checking for hyperproperties that involve *quantifier alternations*, our findings make a strong case for the use of explicit-state methods (as implemented in AutoHyper):

First, compared to symbolic methods (such as BMC), explicit-state model checking is currently the only method that is *complete*. While BMC was able to verify or refute all properties in Tables 2 and 3, many instances cannot be solved with the current BMC encoding. As a concrete example, BMC can *never* verify formulas whose body contains simple invariants (such as GNI) and can thus not verify any of the instances in Table 1. The most significant advantage of explicit-state MC (as implemented in AutoHyper) is thus that it is both push-button and complete, i.e., it can – at least in theory – verify or refute all properties.

Second, the performance of `AutoHyper` seems to be *on-par* with that of BMC and frequently outperforms it (even by several orders of magnitude, cf. Table 3). We stress that this is despite the fact that for the evaluation of `HyperQube` we already fix an unrolling depth and unrolling semantics, thus creating favorable conditions for `HyperQube`.[9] While BMC for trace properties reduces to SAT solving, BMC of hyperproperties reduces to QBF solving; a problem that is much harder and has seen less support by industry-strength tools. It is, therefore, unclear whether the advance of modern QBF solvers can improve the performance of hyperproperty BMC, to the same degree that the advance of SAT solvers has stimulated the success of BMC for trace properties. Our findings seem to indicate that, at the moment, QBF solving (often) seems inferior to an explicit (automata-based) solving strategy.

8 Evaluating Strategy-based Verification

So far, we have used `AutoHyper` to check hyperproperties on instances arising in the literature. In this last section, we demonstrate that `AutoHyper` also serves as a valuable baseline to evaluate different (possibly incomplete) verification methods. Here we focus on strategy-based verification (SBV), i.e., the idea of automatically synthesizing a strategy that resolves existential quantification in $\forall^*\exists^*$ HyperLTL properties [19,7].

8.1 Effectiveness of Strategy-based Verification

SBV is known to be incomplete [19,7]. However, due to the previous lack of *complete* tools for verifying $\forall^*\exists^*$ properties, a detailed study into how effective SBV is in practice was impossible on a larger scale (i.e., beyond hand-crafted examples). With `AutoHyper`, we can, for the first time, rigorously evaluate SBV. We use the SBV implementation from [7], which synthesizes a strategy for the $\exists$-player by translating the formula to a deterministic parity automaton (DPA) [35] and phrases the synthesizes as a parity game.

We have generated random transition systems and properties of varying sizes and computed a ground truth using `AutoHyper`. We then performed SBV (recall that SBV can never show that a property does not hold and might fail to establish that it does). We find that for our generated instances, the property holds in **61.1%** of the cases, and SBV can verify the property in **60.4%** of the cases. Successful verification with SBV is thus possible in many cases, even without the addition of expensive mechanisms such as prophecies [7]. On the other hand, our results show that random generation produces instances (albeit not many)

[9] In Tables 2 and 3, we perform a single query with a fixed unrolling depth k and semantics, i.e., we already know if we want to show satisfaction or violation and the depth needed to show this (as done in [31]). In a classical BMC loop, we would check for satisfaction and violation with an incrementally increasing unrolling depth and thus perform roughly $2k$ many QBF queries where k is the least bound for which satisfaction or violation can be established (if this bound even exists).

on which SBV fails (so far, examples where SBV fails required careful construction by hand). Reverting to SBV as the default verification strategy is thus not possible, further strengthening the case for complete model checking tools (of which `AutoHyper` is the first).

8.2 Efficiency of Strategy-based Verification

After having analyzed the effectiveness of SBV (i.e., how many instances *can* be verified), we turn our attention to the efficiency of SBV. In theory, (automata-based) model checking of $\forall^*\exists^*$ HyperLTL – as implemented in `AutoHyper` – is `EXPSPACE`-complete in the specification and `PSPACE`-complete in the size of the system [15,36]. Conversely, SBV is 2-`EXPTIME`-complete in the size of the specification but only `PTIME` in the size of the system [19]. Consequently, one would expect that ABV fares better on larger specifications and SBV fares better on larger systems (the more important measure in practice).

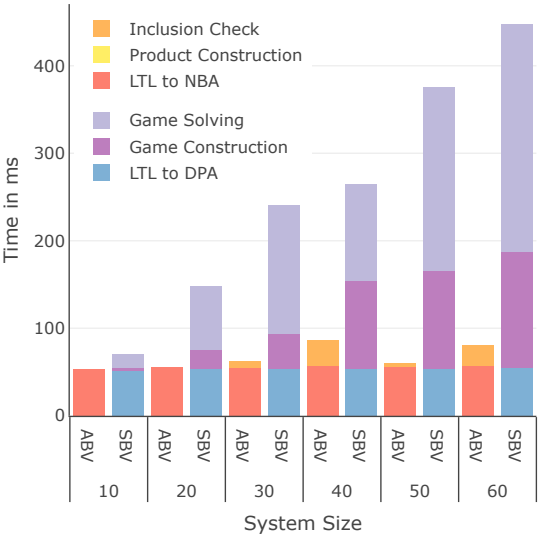


Fig. 3: We compare ABV (`AutoHyper`) and SBV ([7]) on instances of varying system size. We fix the property size to 20. We generate 100 random instances for each size and take the average over the fastest L instances, where L is the minimum number of instances solved within a 5s timeout by both methods.

However, in this section, we show that this does not translate into practice (at least using the current implementation of SBV [7]). We compare the running time of `AutoHyper` (ABV) (using `spot`'s inclusion checker) and SBV. We break the running time into the three main steps for each method. For ABV, this is the LTL-to-NBA translation, the construction of the product automaton, and the inclusion check. For SBV, it is the LTL-to-DPA translation, the construction of the game, and the game-solving.

We depict the average cost for varying system sizes in Figure 3. We observe that SBV performs worse than ABV and, more importantly, scales poorly in the size of the system. This is contrary to the theoretical analysis of ABV and SBV. As the detailed breakdown of the running time suggests, the poor performance is due to the costly construction of the game and the time taken to solve the game. An almost identical picture emerges if we compare ABV in SBV relative

to the property size (we give a plot in the full version [9]). While, in this case, the results match the theory (i.e., SBV scales worse in the size of the specification), we find that the bottleneck for SBV is not the LTL-to-DPA translation (which, in theory, is exponentially more expensive than the LTL-to-NBA translation used in ABV), but, again the construction and solving of the parity game.

We remark that the SBV engine we used [7] is not optimized and always constructs the full (reachable) game graph. The poor performance of SBV can be attributed to the fact that the size of the game does, in the worst case, scale quadratically in the size of the system (when considering $\forall^1\exists^1$ properties). This is amplified in dense systems (i.e., systems with many transitions), as, with increasing transition density, the size of the parity games approaches its worst-case size (see the full version [9]). In contrast, the heavily optimized inclusion checker (in this case `spot`) seems to be able to check inclusion in almost constant time (despite being exponential in theory). This efficiency of mature language inclusion checkers is what enables `AutoHyper` to achieve remarkable performance that often exceeds that of symbolic methods such as BMC (cf. Section 7) and further strengthens the practical impact of Proposition 1.

9 Conclusion

In this paper, we have presented `AutoHyper`, the first complete model checker for HyperLTL with an arbitrary quantifier prefix. We have demonstrated that `AutoHyper` can check many interesting properties involving quantifier alternations and often outperforms symbolic methods such as BMC, sometimes by orders of magnitude. We believe the biggest advantage of `AutoHyper` to be its push-button functionality combined with its completeness: As a user, one does not need to worry whether `AutoHyper` is applicable to a particular property (different from, e.g., SBV or BMC) and does not need to provide hints (e.g., in the form of explicit strategies in SBV).

Apart from evaluating `AutoHyper`'s performance on a range of benchmarks, we have used `AutoHyper` to **(1)** compare various backend language inclusion checkers, **(2)** explore practical verification beyond one quantifier alternation (which is not as infeasible as suggested by the theory), and **(3)** rigorously evaluate the effectiveness and efficiency of strategy-based verification in practice (which, different than suggested by a theoretical analysis, performs worse than automata-based methods).

Acknowledgments. This work was partially supported by the DFG in project 389792660 (Center for Perspicuous Systems, TRR 248), and by by the ERC Grant HYPER (No. 101055412). R. Beutner carried out this work as a member of the Saarbrücken Graduate School of Computer Science.

Data Availability Statement

`AutoHyper` and all experiments are available at [4].

References

1. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Kretínský, J., Müller, D., Parker, D., Strejcek, J.: The Hanoi omega-automata format. In: International Conference on Computer Aided Verification, CAV 2015. LNCS, vol. 9206. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_31
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
3. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6) (2011). https://doi.org/10.1017/S0960129511000193
4. Beutner, R.: AutoHyper: Explicit-state model checking for HyperLTL (2023). https://doi.org/10.5281/zenodo.7309986
5. Beutner, R., Carral, D., Finkbeiner, B., Hofmann, J., Krötzsch, M.: Deciding hyperproperties combined with functional specifications. In: Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2022. ACM (2022). https://doi.org/10.1145/3531130.3533369
6. Beutner, R., Finkbeiner, B.: A temporal logic for strategic hyperproperties. In: International Conference on Concurrency Theory, CONCUR 2021. LIPIcs, vol. 203. Schloss Dagstuhl (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.24
7. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: IEEE Computer Security Foundations Symposium, CSF 2022. IEEE (2022). https://doi.org/10.1109/CSF54842.2022.00030, https://arxiv.org/abs/2206.01797
8. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: International Conference on Computer Aided Verification, CAV 2022. LNCS, vol. 13371. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_17
9. Beutner, R., Finkbeiner, B.: AutoHyper: Explicit-state model checking for HyperLTL. CoRR **abs/2301.11229** (2023). https://doi.org/10.48550/arXiv.2301.11229
10. Beutner, R., Finkbeiner, B.: HyperATL*: A logic for hyperproperties in multi-agent systems. CoRR **abs/2203.07283** (2023). https://doi.org/10.48550/arXiv.2203.07283
11. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS 1999. LNCS, vol. 1579. Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
12. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: International Conference on Computer Aided Verification, CAV 2010. LNCS, vol. 6174. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
13. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: International Conference on Computer Aided Verification, CAV 2002,Copenhagen. LNCS, vol. 2404. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
14. Clarke, E.M., Kroening, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2004. LNCS, vol. 2937. Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_9
15. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference on Principles of Security and Trust, POST 2014. LNCS, vol. 8414. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15

16. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: IEEE Computer Security Foundations Symposium, CSF 2008. IEEE (2008). https://doi.org/10.1109/CSF.2008.7
17. Clemente, L., Mayr, R.: Efficient reduction of nondeterministic automata with application to language inclusion testing. Log. Methods Comput. Sci. **15**(1) (2019). https://doi.org/10.23638/LMCS-15(1:12)2019
18. Coenen, N., Finkbeiner, B., Frenkel, H., Hahn, C., Metzger, N., Siber, J.: Temporal causality in reactive systems. In: International Symposium on Automated Technology for Verification and Analysis, ATVA 2022. LNCS, vol. 13505. Springer (2022). https://doi.org/10.1007/978-3-031-19992-9_13
19. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: International Conference on Computer Aided Verification, CAV 2019. LNCS, vol. 11561. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_7
20. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of bounded model checking at an industrial setting. In: International Conference on Computer Aided Verification, CAV 2001. LNCS, vol. 2102. Springer (2001). https://doi.org/10.1007/3-540-44585-4_43
21. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods Syst. Des. **1**(2/3) (1992). https://doi.org/10.1007/BF00121128
22. D'Argenio, P.R., Barthe, G., Biewer, S., Finkbeiner, B., Hermanns, H.: Is your software on dope? - formal analysis of surreptitiously "enhanced" programs. In: European Symposium on Programming, ESOP 2017. LNCS, vol. 10201. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_4
23. Doherty, S., Detlefs, D., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Jr., G.L.S.: DCAS is not a silver bullet for nonblocking algorithm design. In: Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2004. ACM (2004). https://doi.org/10.1145/1007912.1007945
24. Doveri, K., Ganty, P., Mazzocchi, N.: FORQ-based language inclusion formal testing. In: International Conference on Computer Aided Verification, CAV 2022. LNCS, vol. 13372. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_6
25. Doveri, K., Ganty, P., Parolini, F., Ranzato, F.: Inclusion testing of Büchi automata based on well-quasiorders. In: International Conference on Concurrency Theory, CONCUR 2021. LIPIcs, vol. 203. Schloss Dagstuhl (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.3
26. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: International Conference on Computer Aided Verification, CAV 2022. LNCS, vol. 13372. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9
27. Fellner, A., Befrouei, M.T., Weissenbacher, G.: Mutation testing with hyperproperties. Softw. Syst. Model. **20**(2) (2021). https://doi.org/10.1007/s10270-020-00850-1
28. Fienberg, S.E.: A brief history of statistical models for network analysis and open challenges. Journal of Computational and Graphical Statistics **21**(4) (2012)
29. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: International Conference on Computer Aided Verification, CAV 2015. LNCS, vol. 9206. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_3

30. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990). https://doi.org/10.1145/78969.78972
31. 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. LNCS, vol. 12651. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_6
32. Jamroga, W., Mauw, S., Melissen, M.: Fairness in non-repudiation protocols. In: International Workshop on Security and Trust Management, STM 2011. LNCS, vol. 7170. Springer (2011). https://doi.org/10.1007/978-3-642-29963-6_10
33. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM **17**(8) (1974). https://doi.org/10.1145/361082.361093
34. McCullough, D.: Noninterference and the composability of security properties. In: IEEE Symposium on Security and Privacy, SP 1988. IEEE (1988). https://doi.org/10.1109/SECPRI.1988.8110
35. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3) (2007). https://doi.org/10.2168/LMCS-3(3:5)2007
36. Rabe, M.N.: A temporal logic approach to Information-flow control. Ph.D. thesis, Saarland University (2016)
37. Smith, G., Volpano, D.M.: Secure information flow in a multi-threaded imperative language. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998. ACM (1998). https://doi.org/10.1145/268946.268975
38. Tsai, M., Tsay, Y., Hwang, Y.: GOAL for games, omega-automata, and logics. In: International Conference on Computer Aided Verification, CAV 2013. LNCS, vol. 8044. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_62
39. Wang, Y., Nalluri, S., Pajic, M.: Hyperproperties for robotics: Planning via HyperLTL. In: IEEE International Conference on Robotics and Automation, ICRA 2020. IEEE (2020). https://doi.org/10.1109/ICRA40945.2020.9196874

Machine Learning/Neural Networks

Feature Necessity & Relevancy in ML Classifier Explanations

Xuanxiang Huang[1], Martin C. Cooper[2], Antonio Morgado[3], Jordi Planes[4], and Joao Marques-Silva[5](✉)

[1] University of Toulouse, Toulouse, France xuanxiang.huang@univ-toulouse.fr
[2] Univ. Paul Sabatier, IRIT, Toulouse, France martin.cooper@irit.fr
[3] Universitat de Lleida, Lleida, Spain antonio.morgado@udl.cat
[4] Universitat de Lleida, Lleida, Spain jordi.planes@udl.cat
[5] IRIT, CNRS, Toulouse, France joao.marques-silva@irit.fr

Abstract. Given a machine learning (ML) model and a prediction, explanations can be defined as sets of features which are sufficient for the prediction. In some applications, and besides asking for an explanation, it is also critical to understand whether sensitive features can occur in some explanation, or whether a non-interesting feature must occur in all explanations. This paper starts by relating such queries respectively with the problems of relevancy and necessity in logic-based abduction. The paper then proves membership and hardness results for several families of ML classifiers. Afterwards the paper proposes concrete algorithms for two classes of classifiers. The experimental results confirm the scalability of the proposed algorithms.

Keywords: Formal Explainability · Abduction · Abstraction Refinement.

1 Introduction

The remarkable achievements in machine learning (ML) in recent years [12,32,47] are not matched by a comparable degree of trust. The most promising ML models are inscrutable in their operation. As a direct consequence, the opacity of ML models raises distrust in their use and deployment. Motivated by a critical need for helping human decision makers to grasp the decisions made by ML models, there has been extensive work on explainable AI (XAI). Well-known examples include so-called model agnostic explainers or alternatives based on saliency maps for neural networks [9,50,58,59]. While most XAI approaches do not offer guarantees of rigor, and so can produce explanations that are unsound given the underlying ML model, there have been efforts on developing rigorous XAI approaches over the last few years [40, 54, 63]. Rigorous explainability involves the computation of explanations, but also the ability to answer a wide range of related queries [7, 8, 36].

By building on the relationship between explainability and logic-based abduction [25, 30, 40, 61], this paper analyzes two concrete queries, namely feature

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 167–186, 2023.
https://doi.org/10.1007/978-3-031-30823-9_9

necessity and relevancy. Given an ML classifier, an instance (i.e. point in feature space and associated prediction) and a target feature, the goal of feature necessity is to decide whether the target feature occurs in *all* explanations of the given instance. Under the same assumptions, the goal of feature relevancy is to decide whether a feature occurs in *some* explanation of the given instance. This paper proves a number of complexity results regarding feature necessity and relevancy, focusing on well-known families of classifiers, some of which are widely used in ML. Moreover, the paper proposes novel algorithms for deciding relevancy for two families of classifiers. The experimental results demonstrate the scalability of the proposed algorithms.

The paper is organized as follows. The notation and definitions used throughout are presented in Section 2. The problems of feature necessity and relevancy are studied in Section 3, and example algorithms are proposed in Section 4. Section 5 presents experimental results for a sample of families of classifiers, Section 6 relates our contribution with earlier work and Section 7 concludes the paper.

2 Preliminaries

Complexity classes, propositional logic & quantification. The paper assumes basic knowledge of computational complexity, namely the classes of decision problems P, NP and Σ_2^{P} [6]. The paper also assumes basic knowledge of propositional logic, including the Boolean satisfiability (SAT) problem for propositional logic formulas in conjunctive normal form (CNF), and the use of SAT solvers as oracles for the complexity class NP. The interested reader is referred to textbooks on these topics [6, 13].

2.1 Classification Problems

Throughout the paper, we will consider classifiers as the underlying ML model. Classification problems are defined on a set of features (or attributes) $\mathcal{F} = \{1, \ldots, m\}$ and a set of classes $\mathcal{K} = \{c_1, c_2, \ldots, c_K\}$. Each feature $i \in \mathcal{F}$ takes values from a domain $\mathbb{D}_i$. Domains are categorical or ordinal, and each domain can be defined on boolean, integer/discrete or real values. Feature space is defined as $\mathbb{F} = \mathbb{D}_1 \times \mathbb{D}_2 \times \ldots \times \mathbb{D}_m$. The notation $\mathbf{x} = (x_1, \ldots, x_m)$ denotes an arbitrary point in feature space, where each x_i is a variable taking values from $\mathbb{D}_i$. The set of variables associated with the features is $X = \{x_1, \ldots, x_m\}$. Also the notation $\mathbf{v} = (v_1, \ldots, v_m)$ represents a specific point in feature space, where each v_i is a constant representing one concrete value from $\mathbb{D}_i$. A classifier $\mathbb{C}$ is characterized by a (non-constant) *classification function* κ that maps feature space $\mathbb{F}$ into the set of classes $\mathcal{K}$, i.e. $\kappa : \mathbb{F} \to \mathcal{K}$. An *instance* denotes a pair $(\mathbf{v}, c)$, where $\mathbf{v} \in \mathbb{F}$ and $c \in \mathcal{K}$, with $c = \kappa(\mathbf{v})$.

2.2 Examples of Classifiers

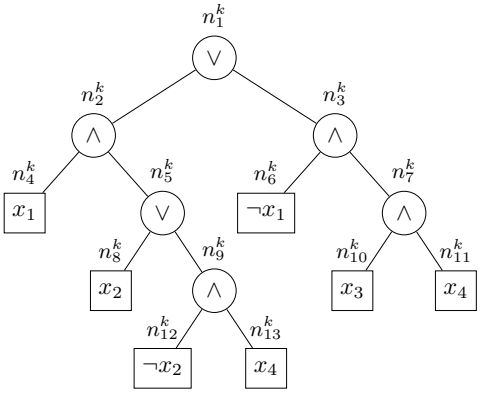
The results presented in the paper apply to a comprehensive range of widely used classifiers [28, 62]. These include, decision trees (DTs) [18, 42], decision graphs (DGs) [44] and diagrams (DDs) [1, 68], decision lists (DLs) [38, 60] and sets (DSs) [19, 41], tree ensembles (TEs) [37], including random forests (RFs) [17, 43] and boosted trees (BTs) [29], neural networks (NNs) [56], naive bayes classifiers (NBCs) [45, 52], classifiers represented with propositional languages, including deterministic decomposable negation normal form (d-DNNFs) [23, 35] and its proper subsets, e.g. sentential decision diagrams (SDDs) [22, 66] and free binary decision diagrams (FBDDs) [23, 31, 68], and also monotonic classifiers. In the rest of the paper, we will analyze some families of classifiers in more detail.

d-DNNF classifiers. Negation normal form (NNF) is a well-known propositional language, where the negation operators are restricted to atoms, or inputs. Any propositional formula can de reduced to NNF in polynomial time. Let the *support* of a node be the set of atoms associated with leaves reachable from the outgoing edges of the node. Decomposable NNF (DNNF) is a restriction of NNF where the children of AND nodes do not share atoms in their support. A DNNF circuit is *deterministic* (referred to as d-DNNF) if any two children of OR nodes cannot both take value 1 for any assignment to the inputs. Restrictions of NNF including DNNF and d-DNNF exhibit important tractability properties [23]. Besides, we briefly introduce FBDDs which is a proper subset of d-DNNFs. An FBDD over a set X of Boolean variables is a rooted, directed acyclic graph comprising two types of nodes: *nonterminal* and *terminal*. A nonterminal node is labeled by a variable $x_i \in X$, and has two outgoing edges, one labeled by 0 and the other by 1. A terminal node is labeled by a 1 or 0, and has no outgoing edges. For a subgraph rooted at a node labeled with a variable x_i, it represents a boolean function f which is defined by the *Shannon expansion*: $f = (x_i \wedge f|_{x_i=1}) \vee (\neg x_i \wedge f|_{x_i=0})$, where $f|_{x_i=1}$ ($f|_{x_i=0}$) denotes the *cofactor* [16] of f with respect to $x_i = 1$ ($x_i = 0$). Moreover, any FBDD is *read-once*, meaning that each variable is tested at most once on any path from the root node to a terminal node.

Monotonic classifiers. Monotonic classifiers find a number of important applications, and have been studied extensively in recent years [26, 48, 65, 70]. Let $\preccurlyeq$ denote a partial order on the set of classes $\mathcal{K}$. For example, we assume $c_1 \preccurlyeq c_2 \preccurlyeq \ldots c_K$. Furthermore, we assume that each domain D_i is ordered such that the value taken by feature i is between a lower bound $\lambda(i)$ and an upper bound $\mu(i)$. Given $\mathbf{v}_1 = (v_{11}, \ldots, v_{1i}, \ldots, v_{1m})$ and $\mathbf{v}_2 = (v_{21}, \ldots, v_{2i}, \ldots, v_{2m})$, we say that $\mathbf{v}_1 \leq \mathbf{v}_2$ if $\forall(i \in \mathcal{F}).(v_{1i} \leq v_{2i})$. Finally, a classifier is monotonic if whenever $\mathbf{v}_1 \leq \mathbf{v}_2$, then $\kappa(\mathbf{v}_1) \preccurlyeq \kappa(\mathbf{v}_2)$.

Running examples. As hinted above, throughout the paper, we will consider two fairly different families of classifiers, namely classifiers represented with d-DNNFs and monotonic classifiers.

Example 1. The first example is the d-DNNF classifier $\mathbb{C}_1$ shown in Fig. 1. It represents the boolean function $(x_1 \wedge (x_2 \vee x_4)) \vee (\neg x_1 \wedge x_3 \wedge x_4)$. The instance considered throughout the paper is $(\mathbf{v}_1, c_1) = ((0, 1, 0, 0), 0)$.



(a) Graphical representation of d-DDNF, i.e. κ_1

$$\mathcal{F}_1 = \{1, 2, 3, 4\}$$
$$\mathbb{D}_{1i} = \{0, 1\}, i = 1, \ldots, 4$$
$$\mathcal{K}_1 = \{0, 1\}$$

(b) Definition of $\mathcal{F}_1, \mathbb{D}_{1i}, \mathcal{K}_1$

IF $x_1 = 1 \wedge x_2 = 1$ THEN 1
ELSE IF $x_1 = 1 \wedge x_4 = 1$ THEN 1
ELSE IF $x_3 = 1 \wedge x_4 = 1$ THEN 1
ELSE 0

(c) Alternative representation of κ_1

Fig. 1: Example of d-DDNF classifier

$$\mathcal{F}_2 = \{1, 2, 3, 4\}$$
$$\mathbb{D}_{2i} = \{0, 1\}, i = 1, \ldots, 4$$
$$\mathcal{K}_2 = \{0, 1\}$$

(a) Definition of $\mathcal{F}_2, \mathbb{D}_{2i}, \mathcal{K}_2$

$$\kappa_2(\mathbf{x}) = \begin{cases} 1 & \text{if } x_1 + x_2 + x_3 \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

(b) Definition of κ_2

Fig. 2: Example of a monotonic classifier

Example 2. The second running example is the monotonic classifier $\mathbb{C}_2$ shown in Fig. 2. The instance that is considered throughout the paper is $(\mathbf{v}_2, c_2) = ((1, 1, 1, 1), 1)$.

2.3 Formal Explainability

Prime implicant (PI) explanations [63] represent a minimal set of literals (relating a feature value x_i and a constant $v_i \in \mathbb{D}_i$) that are logically sufficient for the prediction. PI-explanations are related with logic-based abduction, and so are also referred to as abductive explanations (AXp's) [54]. AXp's offer guarantees of rigor that are not offered by other alternative explanation approaches. More recently, AXp's have been studied in terms of their computational complexity [7, 10]. There is a growing body of recent work on formal explanations [3–5, 14, 15, 24, 27, 33, 51, 54, 67].

Formally, given $\mathbf{v} = (v_1, \ldots, v_m) \in \mathbb{F}$, with $\kappa(\mathbf{v}) = c$, an AXp is any subset-minimal set $\mathcal{X} \subseteq \mathcal{F}$ such that,

$$\mathsf{WAXp}(\mathcal{X}) \quad := \quad \forall(\mathbf{x} \in \mathbb{F}).\ \left[\bigwedge\nolimits_{i \in \mathcal{X}}(x_i = v_i)\right] \rightarrow (\kappa(\mathbf{x}) = c) \tag{1}$$

If a set $\mathcal{X} \subseteq \mathcal{F}$ is not minimal but (1) holds, then $\mathcal{X}$ is referred to as a *weak* AXp. Clearly, the predicate WAXp maps $2^{\mathcal{F}}$ into $\{\bot, \top\}$ (or $\{\mathbf{false}, \mathbf{true}\}$). Given $\mathbf{v} \in \mathbb{F}$, an AXp $\mathcal{X}$ represents an irreducible (or minimal) subset of the features which, if assigned the values dictated by $\mathbf{v}$, are sufficient for the prediction c,

i.e. value changes to the features not in $\mathcal{X}$ will not change the prediction. We can use the definition of the predicate WAXp to formalize the definition of the predicate AXp, also defined on subsets $\mathcal{X}$ of $\mathcal{F}$:

$$\mathsf{AXp}(\mathcal{X}) \quad := \quad \mathsf{WAXp}(\mathcal{X}) \land \forall(\mathcal{X}' \subsetneq \mathcal{X}).\neg\mathsf{WAXp}(\mathcal{X}') \tag{2}$$

The definition of $\mathsf{WAXp}(\mathcal{X})$ ensures that the predicate is *monotone*. Indeed, if $\mathcal{X} \subseteq \mathcal{X}' \subseteq \mathcal{F}$, and if $\mathcal{X}$ is a weak AXp, then $\mathcal{X}'$ is also a weak AXp, as the fixing of more features will not change the prediction. Given the monotonicity of predicate WAXp, the definition of predicate AXp can be simplified as follows, with $\mathcal{X} \subseteq \mathcal{F}$:

$$\mathsf{AXp}(\mathcal{X}) := \mathsf{WAXp}(\mathcal{X}) \land \forall(j \in \mathcal{X}).\neg\mathsf{WAXp}(\mathcal{X} \setminus \{j\}) \tag{3}$$

This simpler but equivalent definition of AXp has important practical significance, in that only a linear number of subsets needs to be checked for, as opposed to exponentially many subsets in (2). As a result, the algorithms that compute one AXp are based on (3) [54].

Example 3. From Example 1, and given the instance $((0,1,0,0),0)$, we can conclude that the prediction will be 0 if features 1 and 3 take value 0, or if features 1 and 4 take value 0. Hence, the AXp's are $\{1,3\}$ and $\{1,4\}$. It is also apparent that the assignment $x_2 = 1$ bears no relevance on the fact that the prediction is 0.

Example 4. From Example 2, we can conclude that any sum of two variables assigned value 1 suffices for the prediction. Hence, given the instance $((1,1,1,1),1)$, the possible AXp's are $\{1,2\}$, $\{1,3\}$, and $\{2,3\}$. Observe that the definition of κ_2 does not depend on feature 4.

Besides abductive explanations, another commonly studied type of explanations are contrastive or counterfactual explanations [8,36,39,55]. As argued in related work [36], the duality between abductive and contrastive explanations implies that for the purpose of the queries studied in this paper, it suffices to study solely abductive explanations.

3 Feature Relevancy & Necessity: Theory

This section investigates the complexity of feature relevancy and necessity[6]. We are interested in membership results, which allow us to devise algorithms for the target problems. We are also interested in hardness results, which serve to confirm that the running time complexities of the proposed algorithms are within reason, given the problem's complexity.

3.1 Defining Necessity, Relevancy & Irrelevancy

Throughout this section, a classifier $\mathbb{C}$ is assumed, with features $\mathcal{F}$, domains $\mathbb{D}_i$, $i \in \mathcal{F}$, classes $\mathcal{K}$, a classification function $\kappa : \mathbb{F} \to \mathcal{K}$, and a concrete instance $(\mathbf{v}, c)$, $\mathbf{v} \in \mathbb{F}, c \in \mathcal{K}$.

[6] For the sake of brevity, we opt to only present sketches of some of the proofs.

Definition 1 (Feature Necessity, Relevancy & Irrelevancy)**.** Let $\mathbb{A}$ denote the set of all AXp's for a classifier given a concrete instance, i.e. $\mathbb{A} = \{\mathcal{X} \subseteq \mathcal{F} \,|\, \mathsf{AXp}(\mathcal{X})\}$, and let $t \in \mathcal{F}$ be a target feature. Then, (i) t is necessary if $t \in \cap_{\mathcal{X} \in \mathbb{A}} \mathcal{X}$; (ii) t is relevant if $t \in \cup_{\mathcal{X} \in \mathbb{A}} \mathcal{X}$; and (iii) t is irrelevant if $t \in \mathcal{F} \setminus \cup_{\mathcal{X} \in \mathbb{A}} \mathcal{X}$.

Throughout the remainder of the paper, the problem of deciding feature necessity is represented by the acronym FNP, and the problem of deciding feature relevancy is represented by the acronym FRP.

Example 5. As shown earlier, for the d-DNNF classifier of Fig. 1, and given the instance $(\mathbf{v}_1, c_1) = ((0, 1, 0, 0), 0)$, there exist two AXp's, i.e. $\{1, 3\}$ and $\{1, 4\}$. Clearly, feature 1 is necessary, and features 1, 3 and 4 are relevant. In contrast, feature 2 is irrelevant.

Example 6. For the monotonic classifier of Fig. 2, and given the instance $(\mathbf{v}_2, c_2) = ((1, 1, 1, 1), 1)$, we have argued earlier that there exist three AXp's, i.e. $\{1, 2\}$, $\{1, 3\}$ and $\{2, 3\}$, which allows us to conclude that features 1, 2 and 3 are relevant, but that feature 4 is irrelevant. In this case, there are no necessary features.

The general complexity of necessity and (ir)relevancy has been studied in the context of logic-based abduction [25, 30, 61]. Recent uses in explainability are briefly overviewed in Section 6.

3.2 Feature Necessity

Proposition 2. If deciding $\mathsf{WAXp}(\mathcal{X})$ is in complexity class $\mathfrak{C}$, then FNP is in the complexity class co-$\mathfrak{C}$.

Given the known polynomial complexity of deciding whether a set is a weak AXp for several families of classifiers [54], we then have the following result:

Corollary 3. For DTs, XpG's[7], NBCs, d-DNNF classifiers and monotonic classifiers, FNP is in P.

3.3 Feature Relevancy: Membership Results

Proposition 4 (Feature Relevancy for DTs [36])**.** FRP for DTs is in P.

Proposition 5. If deciding $\mathsf{WAXp}(\mathcal{X})$ is in P, then FRP is in NP.

The argument above can also be used for proving the following results.

Corollary 6. For XpG's, NBCs, d-DNNF classifiers and monotonic classifiers, FRP is in NP.

Proposition 7. If deciding $\mathsf{WAXp}(\mathcal{X})$ is in NP, then FRP is in Σ_2^{P}.

Corollary 8. For DLs, DSs, RFs, BTs, and NNs, FRP is in Σ_2^{P}.

Additional results. The following result will prove useful in designing algorithms for FRP in practice.

Proposition 9. Let $\mathcal{X} \subseteq \mathcal{F}$, and let $t \in \mathcal{X}$ denote some target feature such that, $\mathsf{WAXp}(\mathcal{X})$ holds and $\mathsf{WAXp}(\mathcal{X} \setminus \{t\})$ does not hold. Then, for any AXp $\mathcal{Z} \subseteq \mathcal{X} \subseteq \mathcal{F}$, it must be the case that $t \in \mathcal{Z}$.

[7] Explanation graphs (XpG's) have been proposed to enable the computation of explanations for decision graphs, and (multi-valued) decision diagrams [36].

3.4 Feature Relevancy: Hardness Results

Proposition 10 (Relevancy for DNF Classifiers [36])**.** Feature relevancy for a DNF classifier is Σ_2^{P}-hard.

Proposition 11. Feature relevancy for monotonic classifiers is NP-hard.

Proof. We say that a CNF is trivially satisfiable if some literal occurs in all clauses. Clearly, SAT restricted to nontrivial CNFs is still NP-complete. Let Φ be a not trivially satisfiable CNF on variables $x_1, \ldots, x_k$. Let $N = 2k$. Let $\tilde{\Phi}$ be identical to Φ except that each occurrence of a negative literal x_i $(1 \leq i \leq k)$ is replaced by x_{i+k}. Thus $\tilde{\Phi}$ is a CNF on N variables each of which occur only positively. Define the boolean classifier κ (on $N+1$ features) by $\kappa(x_0, x_1, \ldots, x_N) = 1$ iff $x_i = x_{i+k} = 1$ for some $i \in \{1, \ldots, k\}$ or $x_0 \land \tilde{\Phi}(x_1, \ldots, x_N) = 1$. To show that Φ is monotonic we need to show that $\mathbf{a} \leq \mathbf{b} \Rightarrow \kappa(\mathbf{a}) \leq \kappa(\mathbf{b})$. This follows by examining the two cases in which $\kappa(\mathbf{a}) = 1$: if $a_i = a_{i+k} \land \mathbf{a} \leq \mathbf{b}$, then $b_i = b_{i+k}$, whereas, if $a_0 \land \tilde{\Phi}(a_1, \ldots, a_N) = 1$ and $\mathbf{a} \leq \mathbf{b}$, then $b_0 \land \tilde{\Phi}(b_1, \ldots, b_N) = 1$ (by positivity of $\tilde{\Phi}$), so in both cases $\kappa(\mathbf{b}) = 1 \geq \kappa(\mathbf{a})$.

Clearly $\kappa(\mathbf{1}_{N+1}) = 1$. There are k obvious AXp's of this prediction, namely $\{i, i+k\}$ $(1 \leq i \leq k)$. These are minimal by the assumption that Φ is not trivially satisfiable. This means that no other AXp contains both i and $i+k$ for any $i \in \{1, \ldots, k\}$. Suppose that $\Phi(\mathbf{u}) = 1$. Let $\mathcal{X}_u$ be $\{0\} \cup \{i \mid 1 \leq i \leq k \land u_i = 1\} \cup \{i+k \mid 1 \leq i \leq k \land u_i = 0\}$. Then $\mathcal{X}_u$ is a weak AXp of the prediction $\kappa(1) = 1$. Furthermore $\mathcal{X}_u$ does not contain any of the AXp's $\{i, i+k\}$. Therefore some subset of $\mathcal{X}$ is an AXp and clearly this subset must contain feature 0. Thus if Φ is satisfiable, then there is an AXp which contains 0.

We now show that the converse also holds. If $\mathcal{X}$ is an AXp of $\kappa(\mathbf{1}_{N+1}) = 1$ containing 0, then it cannot also contain any of the pairs $i, i+k$ $(1 \leq i \leq k)$, otherwise we could delete 0 and still have an AXp. We will show that this implies that we can build a satisfying assignment $\mathbf{u}$ for Φ. Consider first $\mathbf{v} = (v_0, \ldots, v_N)$ defined by $v_i = 1$ if $i \in \mathcal{X}$ $(0 \leq i \leq N)$ and $v_{i+k} = 1$ if neither i nor $i+k$ belongs to $\mathcal{X}$ $(1 \leq i \leq k)$, and $v_i = 0$ otherwise $(1 \leq i \leq N)$. Then $\kappa(\mathbf{v}) = 1$ by definition of an AXp, since $\mathbf{v}$ agrees with the vector 1 on all features in $\mathcal{X}$. We can also note that $v_0 = 1$ since $0 \in \mathcal{X}$. Since $\mathcal{X}$ does not contain i and $i+k$ $(1 \leq i \leq k)$, it follows that $v_i \neq v_{i+k}$. Now let $u_i = 1$ iff $i \in \mathcal{X} \land 1 \leq i \leq k$. It is easy to verify that $\Phi(\mathbf{u}) = \tilde{\Phi}(\mathbf{v}) = \kappa(\mathbf{v}) = 1$.

Thus, determining whether $\kappa(\mathbf{1}_{N+1}) = 1$ has an AXp containing the feature 0 is equivalent to testing the satisfiability of Φ. It follows that FRP is NP-hard for monotonic classifiers by this polynomial reduction from SAT. □

Proposition 12. Relevancy for FBDD classifiers is NP-hard.

Proof. Let ψ be a CNF formula defined on a variable set $X = \{x_1, \ldots, x_m\}$ and with clauses $\{\omega_1, \ldots, \omega_n\}$. We aim to construct an FBDD classifier $\mathcal{G}$ (representing a classification function κ) based on ψ and a target variable in polynomial time, such that: ψ is SAT iff for κ there is an AXp containing this target variable.

For any literal $l_j \in \omega_i$, replace l_j with l_j^i. Let $\psi' = \{\omega_1', \ldots, \omega_n'\}$ denote the resulting CNF formula defined on the new variables $\{x_1^1, \ldots, x_m^1, \ldots x_1^n, \ldots, x_m^n\}$. For each original variable x_j, let I_j^+ and I_j^- denote the indices of clauses con-

taining literal x_j and $\neg x_j$, respectively. So if $i \in I_j^+$, then $x_j^i \in \omega_i'$, if $i \in I_j^-$, then $\neg x_j^i \in \omega_i'$. To build an FBDD D from ψ': 1) build an FBDD D_i for each ω_i'; 2) replace the terminal node 1 of D_i with the root node of D_{i+1}; D is read-once because each variable x_j^i occurs only once in ψ'. Satisfying a literal $x_j^i \in \omega_i'$ means $x_j = 1$, while satisfying a literal $\neg x_j^k \in \omega_k'$ means $x_j = 0$. If both x_j^i and $\neg x_j^k$ are satisfied, then it means we pick inconsistent values for the variable x_j, which is unacceptable. Let us define ϕ to capture inconsistent values for any variable x_j:

$$\phi := \bigvee\nolimits_{1 \le j \le m} \left(\left(\bigvee\nolimits_{i \in I_j^+} x_j^i \right) \wedge \left(\bigvee\nolimits_{k \in I_j^-} \neg x_j^k \right) \right) \tag{4}$$

If $I_j^+ = \emptyset$, then let $\left(\bigvee_{i \in I_j^+} x_j^i\right) = 0$. If $I_j^- = \emptyset$, then let $\left(\bigvee_{k \in I_j^-} \neg x_j^k\right) = 0$. Any true point of ϕ means we pick inconsistent values for some variable x_j, so it represents an unacceptable point of ψ. To avoid such inconsistency, one needs to at least falsify either $\bigvee_{i \in I_j^+} x_j^i$ or $\bigvee_{k \in I_j^-} \neg x_j^k$ for each variable x_j. To build an FBDD G from ϕ: 1) build FBDDs G_j^+ and G_j^- for $\bigvee_{i \in I_j^+} x_j^i$ and $\bigvee_{k \in I_j^-} \neg x_j^k$, respectively; 2) replace the terminal node 1 of G_j^+ with the root node of G_j^-, let G_j denote the resulting FBDD; 3) replace the terminal 0 of G_j with the root node of G_{j+1}; G is read-once because each variable x_j^i occurs only once in ϕ.

Create a root node labeled x_0^0, link its 1-edge to the root of D, 0-edge to the root of G. The resulting graph $\mathcal{G}$ is an FBDD representing $\kappa := (x_0^0 \wedge \psi') \vee (\neg x_0^0 \wedge \phi)$, κ is a boolean classifier defined on $\{x_0^0, x_1^1, \ldots, x_m^n\}$ and x_0^0 is the target variable. The number of nodes of $\mathcal{G}$ is $O(n \times m)$. Let $\mathcal{I} = \{(0,0), (1,1), \ldots (n,m)\}$ denote the set of variable indices, for variable x_j^i, $(i,j) \in \mathcal{I}$.

Pick an instance $\mathbf{v} = \{v_0^0, \ldots, v_j^i, \ldots\}$ satisfying every literal of ψ' (i.e. $v_j^i = 1$ and $v_j^k = 0$ for $x_j^i, \neg x_j^k \in \psi'$) and such that $v_0^0 = 1$, then $\psi'(\mathbf{v}) = 1$, and so $\kappa(\mathbf{v}) = 1$. Suppose $\mathcal{X} \subseteq \mathcal{I}$ is an AXp of $\mathbf{v}$: 1) If $\{(i,j),(k,j)\} \subseteq \mathcal{X}$ for some variable x_j, where $i \in I_j^+$ and $k \in I_j^-$, then for any point $\mathbf{u}$ of κ such that $u_j^i = v_j^i$ for any $(i,j) \in \mathcal{X}$, we have $\kappa(\mathbf{u}) = 1$ and $\phi(\mathbf{u}) = 1$. Moreover, if $\mathbf{u}$ sets $u_0^0 = 1$, then $\kappa(\mathbf{u}) = 1$ implies $\psi'(\mathbf{u}) = 1$, else if $\mathbf{u}$ sets $u_0^0 = 0$, then $\kappa(\mathbf{u}) = 1$ because of $\phi(\mathbf{u}) = 1$. $\kappa(\mathbf{u}) = 1$ regardless the value of u_0^0, so $(0,0) \notin \mathcal{X}$. 2) If $\{(i,j),(k,j)\} \not\subseteq \mathcal{X}$ for any variable x_j, where $i \in I_j^+$ and $k \in I_j^-$, then for some point $\mathbf{u}$ of κ such that $u_j^i = v_j^i$ for any $(i,j) \in \mathcal{X}$, we have $\phi(\mathbf{u}) \neq 1$, in this case $\kappa(\mathbf{u}) = 1$ implies $\psi'(\mathbf{u}) = 1$, besides, any such $\mathbf{u}$ must set $u_0^0 = 1$, so $(0,0) \in \mathcal{X}$.

If case 2) occurs, then ψ is satisfiable. (a satisfying assignment is $x_j = 1$ iff $\exists i \in I_j^+$ s.t. $(i,j) \in \mathcal{X}$). If case 2) never occurs, then ψ is unsatisfiable. It follows that FRP is NP-hard for FBDD classifiers by this polynomial reduction from SAT. □

Corollary 13. Relevancy for d-DNNF classifiers is NP-hard.

4 Feature Relevancy: Example Algorithms

This section details two methods for FRP. One method decides feature relevancy for d-DNNF classifiers, whereas the other method decides feature relevancy for

Table 1: Encoding for deciding whether there is a weak AXp including feature t.

Conditions	Constraints	Fml #
$\mathsf{Leaf}(j), \mathsf{Feat}(j,i), \mathsf{Sat}(\mathsf{Lit}(j), v_i)$	n_j^k	(1.1)
$\mathsf{Leaf}(j), \mathsf{Feat}(j,i), \neg\mathsf{Sat}(\mathsf{Lit}(j), v_i), i = k$	n_j^k	(1.2)
$\mathsf{Leaf}(j), \mathsf{Feat}(j,i), \neg\mathsf{Sat}(\mathsf{Lit}(j), v_i), i \neq k$	$n_j^k \leftrightarrow \neg s_i$	(1.3)
$\mathsf{NonLeaf}(j), \mathsf{Oper}(j) = \vee$	$n_j^k \leftrightarrow \bigvee_{l \in \mathsf{children}(j)} n_l^k$	(1.4)
$\mathsf{NonLeaf}(j), \mathsf{Oper}(j) = \wedge$	$n_j^k \leftrightarrow \bigwedge_{l \in \mathsf{children}(j)} n_l^k$	(1.5)
$\kappa(\mathbf{v}) = 0$	$\neg n_1^0$	(1.6)
$\kappa(\mathbf{v}) = 0$	$s_i \leftrightarrow n_1^i$	(1.7)
	s_t	(1.8)

arbitrary monotonic classifiers. Based on Proposition 2 and Corollary 3, existing algorithm for computing one AXp [35, 36, 52, 53] can be used to decide feature necessity. Hence, there is no need for devising new algorithms. Additionally, the weak AXp returned from the proposed methods (if it exist) can be fed (as a seed) into the algorithms of computing one AXp [35, 53] to extract one AXp in polynomial time.

4.1 Relevancy for d-DNNF Classifiers

This section details a propositional encoding that decides feature relevancy for d-DNNFs. The encoding follows the approach described in the proof of Proposition 9, and comprises two copies ($\mathbb{C}^0$ and $\mathbb{C}^t$) of the same d-DNNF classifier $\mathbb{C}$, $\mathbb{C}^0$ encodes $\mathsf{WAXp}(\mathcal{X})$ (i.e. the prediction of κ remains unchanged), $\mathbb{C}^t$ encodes $\neg\mathsf{WAXp}(\mathcal{X} \setminus \{t\})$ (i.e. the prediction of κ changes). The encoding is polynomial in the size of classifier's representation.

The encoding is applicable to the case $\kappa(\mathbf{x}) = 0$. The case $\kappa(\mathbf{x}) = 1$ can be transformed to $\neg\kappa(\mathbf{x}) = 0$, so we assume both d-DNNF $\mathbb{C}$ and its negation $\neg\mathbb{C}$ are given. To present the constraints included in this encoding, we need to introduce some auxiliary boolean variables and predicates.

1. s_i, $1 \leq i \leq m$. s_i is a selector such that $s_i = 1$ iff feature i is included in a weak AXp candidate $\mathcal{X}$.
2. n_j^k, $1 \leq j \leq |\mathbb{C}|$ and $0 \leq k \leq m$. n_j^k is the indicator of a node j of d-DNNF $\mathbb{C}$ for replica k. The indicator for the root node of k-th replica is n_1^k. Moreover, the semantics of n_j^k is $n_j^k = 1$ iff the sub-d-DNNF rooted at node j in k-th replica is consistent.
3. $\mathsf{Leaf}(j) = 1$ if the node j is a leaf node.
4. $\mathsf{NonLeaf}(j) = 1$ if the node j is a non-leaf node.
5. $\mathsf{Feat}(j, i) = 1$ if the leaf node j is labeled with feature i.
6. $\mathsf{Sat}(\mathsf{Lit}(j), v_i) = 1$ if for leaf node j, the literal on feature i is satisfied by v_i.

The encoding is summarized in Table 1. As literals are d-DNNF leafs, the values of the selector variables only affect the values of the indicator variables of leaf nodes. Constraint (1.1) states that for any leaf node j whose literal is consistent with the given instance, its indicator n_j^k is always consistent regardless of the value of s_i. On the contrary, constraint (1.3) states that for any leaf node j whose literal is inconsistent with the given instance, its indicator n_j^k is consistent iff feature i is not picked, in other words, feature i can take any value. Because replica k ($k > 0$) is used to check the necessity of including feature k in $\mathcal{X}$, we assume the value of the local copy of selector s_k is 0 in replica k. In this case, as defined in constraint (1.2), even though leaf node j labeled feature k has a literal that is inconsistent with the given instance, its indicator n_j^k is consistent. Constraint (1.4) defines the indicator for an arbitrary $\vee$ node j. Constraint (1.5) defines the indicator for an arbitrary $\wedge$ node j. Together, these constraints declare how the consistency is propagated through the entire d-DNNF. Constraint (1.6) states that the prediction of the d-DNNF classifier $\mathbb{C}$ remains 0 since the selected features form a weak AXp. Constraint (1.7) states that if feature i is selected, then removing it will change the prediction of $\mathbb{C}$. Finally, constraint (1.8) indicates that feature t must be included in $\mathcal{X}$.

Example 7. Given the d-DNNF classifier of Fig. 1 and the instance $(\mathbf{v}_1, c_1) = ((0,1,0,0),0)$, suppose that the target feature is 3. We have selectors $\mathbf{s} = \{s_1, s_2, s_3, s_4\}$, and the encoding is as follows:

1. $(n_1^0 \leftrightarrow n_2^0 \vee n_3^0) \wedge (n_2^0 \leftrightarrow n_4^0 \wedge n_5^0) \wedge (n_3^0 \leftrightarrow n_6^0 \wedge n_7^0) \wedge (n_5^0 \leftrightarrow n_8^0 \vee n_9^0) \wedge (n_7^0 \leftrightarrow n_{10}^0 \wedge n_{11}^0) \wedge (n_9^0 \leftrightarrow n_{12}^0 \wedge n_{13}^0) \wedge (n_4^0 \leftrightarrow \neg s_1) \wedge (n_6^0 \leftrightarrow 1) \wedge (n_8^0 \leftrightarrow 1) \wedge (n_{10}^0 \leftrightarrow \neg s_3) \wedge (n_{11}^0 \leftrightarrow \neg s_4) \wedge (n_{12}^0 \leftrightarrow \neg s_2) \wedge (n_{13}^0 \leftrightarrow \neg s_4) \wedge (\neg n_1^0) \wedge (s_3)$
2. $(n_1^3 \leftrightarrow n_2^3 \vee n_3^3) \wedge (n_2^3 \leftrightarrow n_4^3 \wedge n_5^3) \wedge (n_3^3 \leftrightarrow n_6^3 \wedge n_7^3) \wedge (n_5^3 \leftrightarrow n_8^3 \vee n_9^3) \wedge (n_7^3 \leftrightarrow n_{10}^3 \wedge n_{11}^3) \wedge (n_9^3 \leftrightarrow n_{12}^3 \wedge n_{13}^3) \wedge (n_4^3 \leftrightarrow \neg s_1) \wedge (n_6^3 \leftrightarrow 1) \wedge (n_8^3 \leftrightarrow 1) \wedge (n_{10}^3 \leftrightarrow 1) \wedge (n_{11}^3 \leftrightarrow \neg s_4) \wedge (n_{12}^3 \leftrightarrow \neg s_2) \wedge (n_{13}^3 \leftrightarrow \neg s_4) \wedge (s_3 \leftrightarrow n_1^3)$

Given the AXp's listed in Example 3, by solving these formulas we will either obtain $\{1,3\}$ or $\{1,4\}$ as the AXp.

4.2 Relevancy for Monotonic Classifiers

This section describes an algorithm for FRP in the case of monotonic classifiers. No assumption is made regarding the actual implementation of the monotonic classifier.

Abstraction refinement for relevancy. The algorithm proposed in this section iteratively refines an over-approximation (or abstraction) of all the subsets $\mathcal{S}$ of $\mathcal{F}$ such that: i) $\mathcal{S}$ is a weak AXp, and ii) any AXp included in $\mathcal{S}$ also includes the target feature t. Formally, the set of subsets of $\mathcal{F}$ that we are interested in is defined as follows:

$$\mathbb{H} = \{\mathcal{S} \subseteq \mathcal{F} \,|\, \mathsf{WAXp}(\mathcal{S}) \wedge \forall(\mathcal{X} \subseteq \mathcal{S}).\, [\mathsf{AXp}(\mathcal{X}) \rightarrow (t \in \mathcal{X})]\} \tag{5}$$

The proposed algorithm iteratively refines the over-approximation of set $\mathbb{H}$ until one can decide with certainty whether t is included in some AXp. The refinement step involves exploiting counterexamples as these are identified. (The approach is referred to as abstraction refinement FRP, since the use of abstraction refinement

can be related with earlier work (with the same name) in model checking [20].) In practice, it will in general be impractical to manipulate such over-approximation of set $\mathbb{H}$ explicitly. As a result, we use a propositional formula (in fact a CNF formula) $\mathcal{H}$, such that the models of $\mathcal{H}$ encode the subsets of features about which we have yet to decide whether each of those subsets only contains AXp's that include t. (Formula $\mathcal{H}$ is defined on a set of Boolean variables $\{s_1, \ldots, s_m\}$, where each s_i is associated with feature i, and assigning $s_i = 1$ denotes that feature i is included in a given set, as described below.) The algorithm then iteratively refines the over-approximation by filtering out sets of sets that have been shown not to be included in $\mathbb{H}$, i.e. the so-called counterexamples.

Algorithm 1 summarizes the proposed approach[8]. Also, Algorithms 2 and 3 provide supporting functions. (For simplicity, the function calls of Algorithms 2 and 3 show the arguments, but not the parameterizations.) Algorithm 1 iteratively uses an NP oracle (in fact a SAT solver) to pick (or *guess*) a subset $\mathcal{P}$ of $\mathcal{F}$, such that any previously picked set is not repeated. Since we are interested in feature t, we enforce that the picked set must include t. (This step is shown in lines 4 to 7.) Now, the features not in $\mathcal{P}$ are deemed universal, and so we need to account for the range of possible values that these universal features can take. For that, we update lower and upper bounds on the predicted classes. For the features in $\mathcal{P}$ we must use the values dictated by $\mathbf{v}$. (This is shown in lines 8 and 9, and it is sound to do because we have monotonicity of prediction.) If the lower and upper bounds differ, then the picked set is not even a weak AXp, and so we can safely remove it from further consideration. This is achieved by enforcing that at least one of the non-picked elements is picked in the future. (As can be observed $\mathcal{H}$ is updated with a positive clause that captures this constraint, as shown in line 11.) If the lower and upper bounds do not differ (i.e. we picked a weak AXp), and if by allowing t to take any value causes the bounds to differ, then we know that any AXp in $\mathcal{P}$ must include t, and so the algorithm reports $\mathcal{P}$ as a weak AXp that is *guaranteed* to be included in $\mathbb{H}$. (This is shown in line 14.) It should be noted that $\mathcal{P}$ is not necessarily an AXp. However, by Proposition 9, $\mathcal{P}$ is guaranteed to be a weak AXp such that *any* of the AXp's contained in $\mathcal{P}$ *must* include feature t. From [53], we know that we can extract an AXp from a weak AXp in polynomial time, and in this case we are guaranteed to always pick one that includes t. Finally, the last case is when allowing t to take any value does not cause the lower and upper bounds to change. This means we picked a set $\mathcal{P}$ that is a weak AXp, but not all AXp's in $\mathcal{P}$ include the target feature t (again due to Proposition 9). As a result, we must prevent the same weak AXp from being re-picked. This is achieved by requiring that at least one of the picked features not be picked again in the feature set. (This is shown in line 16. As can be observed, $\mathcal{H}$ is updated with a negative clause that captures this constraint.)

As can be concluded from Algorithm 1 and from the discussion above, Proposition 9 is essential to enable us to use at most two classification queries per iter-

[8] Arguments can either represent actual arguments or some parameterization; these are separated by a semi-colon.

Algorithm 1 Deciding feature relevancy for a monotonic classifier

Input: Instance $\mathbf{v}$, Target feature t; Feature Set $\mathcal{F}$, Monotonic Classifier κ

1: **function** DecideRelevant($\mathbf{v}, t; \mathcal{F}, \kappa$)
2: $\quad\mathcal{H} \leftarrow \emptyset$ ▷ $\mathcal{H}$ overapproximates $\mathbb{H}$
3: $\quad$**repeat**
4: $\quad\quad$(outc, $\mathbf{s}$) $\leftarrow$ SAT($\mathcal{H}, s_t$) ▷ Pick candidate weak AXp containing t
5: $\quad\quad$**if** outc = **true then**
6: $\quad\quad\quad\mathcal{P} \leftarrow \{i \in \mathcal{F} \,|\, s_i = 1\}$ ▷ $\mathcal{P}$ is the candidate weak AXp, and $t \in \mathcal{P}$
7: $\quad\quad\quad\mathcal{D} \leftarrow \{i \in \mathcal{F} \,|\, s_i = 0\}$ ▷ $\mathcal{D}$ contains the features not included in $\mathcal{P}$
8: $\quad\quad\quad\mathbf{v}_L \leftarrow (v_{L_1}, \ldots, v_{L_N})$, s.t. $v_{L_i} \leftarrow \text{ITE}(s_i, v_i, \lambda(i))$ ▷ $\mathbf{v}_L$: LB
9: $\quad\quad\quad\mathbf{v}_U \leftarrow (v_{U_1}, \ldots, v_{U_N})$, s.t. $v_{U_i} \leftarrow \text{ITE}(s_i, v_i, \mu(i))$ ▷ $\mathbf{v}_U$: UB
10: $\quad\quad\quad$**if** $\kappa(\mathbf{v}_L) \neq \kappa(\mathbf{v}_U)$ **then** ▷ More than one value possible?
11: $\quad\quad\quad\quad\mathcal{H} \leftarrow \mathcal{H} \cup$ newPosCl($\mathcal{D}, t$) ▷ $\mathcal{P}$ is *not* a weak AXp; block set
12: $\quad\quad\quad$**else** ▷ $\mathcal{P}$ is a weak AXp
13: $\quad\quad\quad\quad$**if** $\kappa(\mathbf{v}_L[v_{L_t} \leftarrow \lambda(t)]) \neq \kappa(\mathbf{v}_U[v_{U_t} \leftarrow \mu(t)])$ **then** ▷ t needed?
14: $\quad\quad\quad\quad\quad$reportWeakAXp($\mathcal{P}$) ▷ t is included in any AXp $\mathcal{X} \subseteq \mathcal{P}$
15: $\quad\quad\quad\quad\quad$**return true**
16: $\quad\quad\quad\quad\mathcal{H} \leftarrow \mathcal{H} \cup$ newNegCl($\mathcal{P}, t$) ▷ t unneeded; block set
17: $\quad$**until** outc = **false**
18: $\quad$**return false** ▷ If $\mathcal{H}$ becomes inconsistent, then *no* AXp contains t

Table 2: Example algorithm execution for $t = 4$

$\mathbf{s}$	$\mathcal{P}$	$\mathcal{D}$	$\kappa(\mathbf{v}_L)$	$\kappa(\mathbf{v}_U)$	Decision	New clause	Line
$(0,0,0,1)$	$\{4\}$	$\{1,2,3\}$	0	1	New pos clause	$(s_1 \vee s_2 \vee s_3)$	11
$(1,0,0,1)$	$\{1,4\}$	$\{2,3\}$	0	1	New pos clause	$(s_2 \vee s_3)$	11
$(1,1,0,1)$	$\{1,2,4\}$	$\{3\}$	1	1	New neg clause	$(\neg s_1 \vee \neg s_2)$	16
$(1,0,1,1)$	$\{1,3,4\}$	$\{2\}$	1	1	New neg clause	$(\neg s_1 \vee \neg s_3)$	16
$(0,1,1,1)$	$\{2,3,4\}$	$\{1\}$	1	1	New pos clause	(s_1)	11
—	—	—	–	–	$\mathcal{H}$ inconsistent	–	17

ation of the algorithm. If we were to use Proposition 5 instead, then the number of classification queries would be significantly larger.

Example 8. We consider the monotonic classifier of Fig. 2, with instance $(\mathbf{v}, c) = ((1,1,1,1), 1)$. Table 2 summarizes a possible execution of the algorithm when $t = 4$. Similarly, Table 3 summarizes a possible execution of the algorithm when $t = 1$. (As with the current implementation, and for both examples, the creation of clauses uses no optimizations.) In general, different executions will be determined by the models returned by the SAT solver.

With respect to the clauses that are added to $\mathcal{H}$ at each step, as shown in Algorithms 2 and 3, one can envision optimizations (shown lines 2 to 7 in both algorithms) that heuristically aim at removing features from the given sets, and so produce shorter (and so logically stronger) clauses. The insight is that any feature, which can be deemed irrelevant for the condition used for constructing

Algorithm 2 Create new pos. clause

Input: Set $\mathcal{D}$, t; κ, $\mathbf{v}_L$, $\mathbf{v}_U$
function newPosCl($\mathcal{D}, t; \kappa, \mathbf{v}_L, \mathbf{v}_U$)
  for all $i \in \mathcal{D}$ **do**
    $(v_{L_i}, v_{U_i}) \leftarrow (v_i, v_i)$
    if $\kappa(\mathbf{v}_L) \neq \kappa(\mathbf{v}_U)$ **then**
      $\mathcal{D} \leftarrow \mathcal{D} \setminus \{i\}$
    else
      $(v_{L_i}, v_{U_i}) \leftarrow (\lambda(i), \mu(i))$
  $\omega \leftarrow (\vee_{i \in \mathcal{D}} s_i)$
  return ω

Algorithm 3 Create new neg. clause

Input: Set $\mathcal{P}$, t; κ, $\mathbf{v}_L$, $\mathbf{v}_U$
function newNegCl($\mathcal{P}, t; \kappa, \mathbf{v}_L, \mathbf{v}_U$)
  for all $i \in \mathcal{P} \setminus \{t\}$ **do**
    $(v_{L_i}, v_{U_i}) \leftarrow (\lambda(i), \mu(i))$
    if $\kappa(\mathbf{v}_L) = \kappa(\mathbf{v}_U)$ **then**
      $\mathcal{P} \leftarrow \mathcal{P} \setminus \{i\}$
    else
      $(v_{L_i}, v_{U_i}) \leftarrow (v_i, v_i)$
  $\omega \leftarrow (\vee_{i \in \mathcal{P} \setminus \{t\}} \neg s_i)$
  return ω

Table 3: Example algorithm execution for $t = 1$

$\mathbf{s}$	$\mathcal{P}$	$\mathcal{D}$	$\kappa(\mathbf{v}_L)$	$\kappa(\mathbf{v}_U)$	Decision	New clause	Line
$(1,0,0,0)$	$\{1\}$	$\{2,3,4\}$	0	1	New pos clause	$(s_2 \vee s_3 \vee s_4)$	11
$(1,1,0,0)$	$\{1,2\}$	$\{3,4\}$	1	1	Weak AXp: $\{1,2\}$	–	14

the clause, can be safely removed from the set. (In practice, our experiments show that the time running the classifier is far larger than the time spent using the NP oracle to guess sets. Thus, we opted to use the simplest approach for constructing the clauses, and so reduce the number of classification queries.)

Given the above discussion, we can conclude that the proposed algorithm is sound, complete and terminating for deciding feature relevancy for monotonic classifiers. (The proof is straightforward, and it is omitted for the sake of brevity.)

Proposition 14. For a monotonic classifier $\mathbb{C}$, defined on set of features $\mathcal{F}$, with κ mapping $\mathbb{F}$ to $\mathcal{K}$, and an instance $(\mathbf{v}, c)$, $\mathbf{v} \in \mathbb{F}$, $c \in \mathcal{K}$, and a target feature $t \in \mathcal{F}$, Algorithm 1 returns a set $\mathcal{P} \subseteq \mathcal{F}$ iff $\mathcal{P}$ is a weak AXp for $(\mathbf{v}, c)$, with the property that any AXp $\mathcal{X} \subseteq \mathcal{P}$ is such that $t \in \mathcal{X}$ (i.e. $\mathcal{P}$ is a witness for the relevancy of t).

5 Experimental Results

This section reports the experimental results on FRP for the d-DNNF and monotonic classifiers. The goal is to show that FRP is practically feasible. We opt not to include experiments for FNP as the complexity of FNP is in P. Besides, to the best of our knowledges, there is no baseline to compare with. The experiments were performed on a MacBook Pro with a 6-Core Intel Core i7 2.6 GHz processor with 16 GByte RAM, running macOS Monterey.

d-DNNF classifiers. For d-DNNFs, we pick its subset SDDs as our target classifier. SDDs support polynomial time negation, so given a SDD $\mathbb{C}$, one can obtain its negation $\neg\mathbb{C}$ efficiently.

Table 4: Solving FRP for SDDs. Sub-Columns Avg. #var and Avg. #cls show, respectively, the average number of variables and clauses in a CNF encoding. Column Runtime reports maximum and average time in seconds for deciding FRP.

Dataset	SDD		%Y	CNF		Runtime (s)	
	#Features	#Nodes		Avg. #var	Avg. #cls	Max	Avg.
Accidents	415	8863	97	26513	78276	56.4	3.5
Audio	272	7224	88	31148	100972	663.1	22.0
DNA	513	8570	91	29155	91288	86.3	11.0
Jester	254	7857	85	35998	121508	362.1	22.7
KDD	306	8109	99	26402	83480	111.2	2.8
Mushrooms	248	7096	91	23874	82112	266.3	15.8
Netflix	292	7039	94	25520	83324	105.7	4.2
NLTCS	183	6661	100	19817	58494	1.4	0.5
Plants	244	6724	97	25356	84782	950.7	20.6
RCV-1	410	9472	90	33438	102500	153.6	11.2
Retail	341	3704	87	10601	28342	1.8	1.1

Monotonic classifiers. For monotonic classifiers, we consider the Deep Lattice Network (DLN) [70] as our target classifier. Since our approach for monotonic classifier is model-agnostic, it could also be used with other approaches for learning monotonic classifiers [48, 69] including Min-Max Network [21, 64] and COMET [65].

Prototype implementation. Prototype implementations of the proposed approaches were implemented in Python [9]. The PySAT toolkit [10] was used for propositional encodings. Besides, PySAT invokes the Glucose 4 [11] SAT solver to pick a weak AXp candidate. SDDs were loaded by using the PySDD [12]package.

Benchmarks & training. For SDDs, we selected 11 datasets from Density Estimation Benchmark Datasets[13]. [34, 46, 49]. 11 datasets were used to learn SDD using LearnSDD [11] (with parameter *maxEdges=20000*). The obtained SDDs were used as binary classifiers. For DLNs, we selected 5 publicly available datasets: *australian* (aus), *breast_cancer* (b.c.), *heart_c*, *nursery* [57] and *pima* [2]. We used the three-layer DLN architecture: Calibrators → Random Ensemble of Lattices → Linear Layer. All calibrators for all models used a fixed number of 20 keypoints. And the size of all lattices was set to 3.

Results for SDDs. For each SDD, 100 test instances were randomly generated. All tested instances have prediction 0. (We didn't pick instances predicted to class 1 as this requires the compilation of a new classifier which may have dif-

[9] https://github.com/XuanxiangHuang/frp-experiment
[10] https://github.com/pysathq/pysat
[11] https://www.labri.fr/perso/lsimon/glucose/
[12] https://github.com/wannesm/PySDD
[13] https://github.com/UCLA-StarAI/Density-Estimation-Datasets

Table 5: Solving FRP for DLN. Column Runtime reports maximum and average time in seconds for deciding FRP. Column SAT Time (resp. $\kappa(\mathbf{v})$ Time) reports maximum and average time in seconds for SAT solver (resp. calling DLN's predict function) to decide FRP. Column SAT Calls (resp. $\kappa(\mathbf{v})$ Calls) reports maximum and average number of calls to the SAT solver (resp. to the DLN's predict function) to decide FRP.

Dataset	%Y	Runtime (s)		SAT Time		SAT Calls		$\kappa(\mathbf{v})$ Time		$\kappa(\mathbf{v})$ Calls		$\frac{\kappa(\mathbf{v})\text{Time}}{\text{Runtime}}$
		Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	
aus	61	40.4	8.31	0.02	0.01	291	65	40.0	8.15	424	98	97.8%
b.c.	45	5.4	1.93	0.00	0.00	53	20	5.3	1.89	78	30	98.0%
heart_c	35	31.5	6.67	0.02	0.00	171	54	31.1	6.52	249	80	97.7%
nursery	45	4.3	1.77	0.00	0.00	31	13	4.3	1.75	73	30	98.6%
pima	74	3.7	1.41	0.00	0.00	33	13	3.7	1.39	47	22	98.4%

ferent size). Besides, for each instance, we randomly picked a feature appearing in the model. Hence for each SDD, we solved 100 queries. Table 4 summarizes the results. It can be observed that the number of nodes of the tested SDD is in the range of 3704 and 9472, and the number of features of tested SDD is in the range of 183 and 513. Besides, the percentage of examples for which the answer is Y (i.e. target feature is in some AXp) ranges from 85% to 100%. Regarding the runtime, the largest running time for solving one query can exceed 15 minutes. But the average running time to solve a query is less than 25 seconds, this highlights the scalability of the proposed encoding.

Results for DLNs. For each DLN, we randomly picked 200 tested instances, and for each tested instance, we randomly pick a feature. Hence for each DLN, we solved 200 queries. Table 5 summarizes the results. The use of a SAT solver has a negligible contribution to the running time. Indeed, for all the examples shown, at least 97% of the running time is spent running the classifier. This should be unsurprising, since the number of the iterations of Algorithm 1 never exceeds a few hundred. (The fraction of a second reported in some cases should be divided by the number of calls to the SAT solver; hence the time spent in each call to the SAT solver is indeed negligible.) As can be observed, the percentage of examples for which the answer is Y (i.e. target feature is in some AXp and the algorithm returns **true**) ranges from 35% to 74%. There is no apparent correlation between the percentage of Y answers and the number of iterations. The large number of queries accounts for the number of times the DLN is queried by Algorithm 1, but it also accounts for the number of times the DLN is queried for extracting an AXp from set $\mathcal{P}$ (i.e. the witness) when the algorithm's answer is **true**. A loose upper bound on the number of queries to the classifier is $4 \times \text{NS} + 2 \times |\mathcal{F}|$, where NS is the number of SAT calls, and $|\mathcal{F}|$ is the number of features. Each iteration of Algorithm 1 can require at most 4 queries to the classifier. After reporting $\mathcal{P}$, at most 2 queries per feature will be required to extract the AXp (see Section 2.3). As can be observed this loose upper bound is respected by the reported results.

6 Related Work

The problems of necessity and relevancy have been studied in logic-based abduction since the early 90s [25, 30, 61]. However, this earlier work did not consider the classes of (classifier) functions that are considered in this paper.

There has been recent work on explainability queries [7, 8, 36]. Some of these queries can be related with feature relevancy and necessity. For example, relevancy and necessity have been studied with respect to a target class [7, 8], in contrast with our approach that studies a concrete instance, and so can be naturally related with earlier work on abduction. Recent work [36] studied feature relevancy under the name feature membership, but neither d-DNNF nor monotonic classifiers were discussed. Moreover, [36] only proved the hardness of deciding feature relevancy for DNF and DT classifiers and did not discuss the feature necessity problem. The results presented in this paper complement this work. Besides, the complexity results of FRP and FNP in this paper also complement the recent work [54] which summarizes the progress of formal explanations. [40] focused on the computation of one arbitrary AXp and one smallest AXp, which is orthogonal to our work. Computing one AXp does not guarantee that either FRP or FNP is decided, since the target feature t may not appear in the computed AXp. [53] studied the computation of one formal explanation and the enumeration of formal explanations in the case study of monotonic classifiers. However, neither FRP or FNP were identified and studied.

7 Conclusions

This paper studies the problems of feature necessity and relevancy in the context of formal explanations of ML classifiers. The paper proves several complexity results, some related with necessity, but most related with relevancy. Furthermore, the paper proposes two different approaches for solving relevancy for two families of classifiers, namely classifiers represented with the d-DNNF propositional language, and monotonic classifiers. The experimental results confirm the practical scalability of the proposed algorithms. Future work will seek to prove hardness results for the families of classifiers for which hardness is yet unknown.

Acknowledgements. This work was supported by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future – PIA3" under Grant agreement no. ANR-19-PI3A-0004, and by the H2020-ICT38 project COALA "Cognitive Assisted agile manufacturing for a Labor force supported by trustworthy Artificial intelligence", and funded by the Spanish Ministry of Science and Innovation (MICIN) under project PID2019-111544GB-C22, and by a María Zambrano fellowship and a Requalification fellowship financed by Ministerio de Universidades of Spain and by European Union – NextGenerationEU.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Transactions on computers **27**(06), 509–516 (1978)
2. Alcalá-Fdez, J., Fernández, A., Luengo, J., Derrac, J., García, S., Sánchez, L., Herrera, F.: Keel data-mining software tool: data set repository, integration of algorithms and experimental analysis framework. Journal of Multiple-Valued Logic & Soft Computing **17** (2011), https://sci2s.ugr.es/keel/dataset.php?cod=21
3. Amgoud, L., Ben-Naim, J.: Axiomatic foundations of explainability. In: IJCAI. pp. 636–642 (2022)
4. Arenas, M., Baez, D., Barceló, P., Pérez, J., Subercaseaux, B.: Foundations of symbolic languages for model interpretability. In: NeurIPS (2021)
5. Arenas, M., Barceló, P., Romero, M., Subercaseaux, B.: On computing probabilistic explanations for decision trees. CoRR **abs/2207.12213** (2022). https://doi.org/10.48550/arXiv.2207.12213, https://doi.org/10.48550/arXiv.2207.12213
6. Arora, S., Barak, B.: Computational Complexity - A Modern Approach. Cambridge University Press (2009), http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264
7. Audemard, G., Bellart, S., Bounia, L., Koriche, F., Lagniez, J., Marquis, P.: On the computational intelligibility of boolean classifiers. In: KR. pp. 74–86 (2021)
8. Audemard, G., Koriche, F., Marquis, P.: On tractable XAI queries based on compiled representations. In: KR. pp. 838–849 (2020)
9. Bach, S., Binder, A., Montavon, G., Klauschen, F., Müller, K.R., Samek, W.: On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. PloS one **10**(7), e0130140 (2015)
10. Barceló, P., Monet, M., Pérez, J., Subercaseaux, B.: Model interpretability through the lens of computational complexity. In: NeurIPS (2020)
11. Bekker, J., Davis, J., Choi, A., Darwiche, A., den Broeck, G.V.: Tractable learning for complex probability queries. In: NeurIPS. pp. 2242–2250 (2015), https://github.com/ML-KULeuven/LearnSDD
12. Bengio, Y., LeCun, Y., Hinton, G.E.: Deep learning for AI. Commun. ACM **64**(7), 58–65 (2021), https://doi.org/10.1145/3448250
13. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336. IOS Press (2021), https://doi.org/10.3233/FAIA336
14. Blanc, G., Lange, J., Tan, L.: Provably efficient, succinct, and precise explanations. In: NeurIPS (2021)
15. Boumazouza, R., Alili, F.C., Mazure, B., Tabia, K.: ASTERYX: A model-Agnostic SaT-basEd appRoach for sYmbolic and score-based eXplanations. In: CIKM. pp. 120–129 (2021)
16. Brayton, R.K., Hachtel, G.D., McMullen, C., Sangiovanni-Vincentelli, A.: Logic minimization algorithms for VLSI synthesis, vol. 2. Springer Science & Business Media (1984)
17. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324, https://doi.org/10.1023/A:1010933404324
18. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth (1984)
19. Clark, P., Boswell, R.: Rule induction with cn2: Some recent improvements. In: European Working Session on Learning. pp. 151–163. Springer (1991)

20. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643, https://doi.org/10.1145/876638.876643
21. Daniels, H., Velikova, M.: Monotone and partially monotone neural networks. IEEE Trans. Neural Networks **21**(6), 906–917 (2010)
22. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: IJCAI. pp. 819–826 (2011)
23. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002). https://doi.org/10.1613/jair.989
24. Darwiche, A., Marquis, P.: On quantifying literals in boolean logic and its applications to explainable AI. J. Artif. Intell. Res. **72**, 285–328 (2021)
25. Eiter, T., Gottlob, G.: The complexity of logic-based abduction. J. ACM **42**(1), 3–42 (1995), https://doi.org/10.1145/200836.200838
26. Fard, M.M., Canini, K.R., Cotter, A., Pfeifer, J., Gupta, M.R.: Fast and flexible monotonic functions with ensembles of lattices. In: NeurIPS. pp. 2919–2927 (2016)
27. Ferreira, J., de Sousa Ribeiro, M., Gonçalves, R., Leite, J.: Looking inside the black-box: Logic-based explanations for neural networks. In: KR. p. 432–442 (2022)
28. Flach, P.A.: Machine Learning - The Art and Science of Algorithms that Make Sense of Data. CUP (2012)
29. Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Annals of statistics pp. 1189–1232 (2001)
30. Friedrich, G., Gottlob, G., Nejdl, W.: Hypothesis classification, abductive diagnosis and therapy. In: ESE. pp. 69–78 (1990)
31. Gergov, J., Meinel, C.: Efficient boolean manipulation with OBDD's can be extended to FBDD's. IEEE Transactions on Computers **43**(10), 1197–1209 (1994). https://doi.org/10.1109/12.324545
32. Goodfellow, I.J., Bengio, Y., Courville, A.C.: Deep Learning. Adaptive computation and machine learning, MIT Press (2016), http://www.deeplearningbook.org/
33. Gorji, N., Rubin, S.: Sufficient reasons for classifier decisions in the presence of domain constraints. In: AAAI (February 2022)
34. Haaren, J.V., Davis, J.: Markov network structure learning: A randomized feature generation approach. In: AAAI (2012)
35. Huang, X., Izza, Y., Ignatiev, A., Cooper, M.C., Asher, N., Marques-Silva, J.: Tractable explanations for d-DNNF classifiers. In: AAAI. pp. 5719–5728 (2022)
36. Huang, X., Izza, Y., Ignatiev, A., Marques-Silva, J.: On efficiently explaining graph-based classifiers. In: KR. pp. 356–367 (2021)
37. Ignatiev, A., Izza, Y., Stuckey, P.J., Marques-Silva, J.: Using MaxSAT for efficient explanations of tree ensembles. In: AAAI. pp. 3776–3785 (2022)
38. Ignatiev, A., Marques-Silva, J.: SAT-based rigorous explanations for decision lists. In: SAT. pp. 251–269 (2021)
39. Ignatiev, A., Narodytska, N., Asher, N., Marques-Silva, J.: From contrastive to abductive explanations and back again. In: AIxIA. pp. 335–355 (2020)
40. Ignatiev, A., Narodytska, N., Marques-Silva, J.: Abduction-based explanations for machine learning models. In: AAAI. pp. 1511–1519 (2019)
41. Ignatiev, A., Pereira, F., Narodytska, N., Marques-Silva, J.: A SAT-based approach to learn explainable decision sets. In: IJCAR. pp. 627–645 (2018)
42. Izza, Y., Ignatiev, A., Marques-Silva, J.: On tackling explanation redundancy in decision trees. J. Artif. Intell. Res. **75**, 261–321 (2022), https://doi.org/10.1613/jair.1.13575

43. Izza, Y., Marques-Silva, J.: On explaining random forests with SAT. In: IJCAI. pp. 2584–2591 (2021)
44. Kohavi, R.: Bottom-up induction of oblivious read-once decision graphs: strengths and limitations. In: AAAI. pp. 613–618 (1994)
45. Kohavi, R., et al.: Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In: Kdd. vol. 96, pp. 202–207 (1996)
46. Larochelle, H., Murray, I.: The neural autoregressive distribution estimator. In: AISTATS. pp. 29–37 (2011)
47. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. nature **521**(7553), 436–444 (2015)
48. Liu, X., Han, X., Zhang, N., Liu, Q.: Certified monotonic neural networks. In: NeurIPS (2020)
49. Lowd, D., Davis, J.: Learning Markov network structure with decision trees. In: ICDM. pp. 334–343 (2010)
50. Lundberg, S.M., Lee, S.: A unified approach to interpreting model predictions. In: NeurIPS. pp. 4765–4774 (2017)
51. Malfa, E.L., Michelmore, R., Zbrzezny, A.M., Paoletti, N., Kwiatkowska, M.: On guaranteed optimal robust explanations for NLP models. In: IJCAI. pp. 2658–2665 (2021)
52. Marques-Silva, J., Gerspacher, T., Cooper, M.C., Ignatiev, A., Narodytska, N.: Explaining naive bayes and other linear classifiers with polynomial time and delay. In: NeurIPS (2020)
53. Marques-Silva, J., Gerspacher, T., Cooper, M.C., Ignatiev, A., Narodytska, N.: Explanations for monotonic classifiers. In: ICML. pp. 7469–7479 (2021)
54. Marques-Silva, J., Ignatiev, A.: Delivering trustworthy AI through formal XAI. In: AAAI. pp. 12342–12350 (2022)
55. Miller, T.: Explanation in artificial intelligence: Insights from the social sciences. Artif. Intell. **267**, 1–38 (2019)
56. Müller, B., Reinhardt, J., Strickland, M.T.: Neural networks: an introduction. Springer Science & Business Media (1995)
57. Olson, R.S., La Cava, W., Orzechowski, P., Urbanowicz, R.J., Moore, J.H.: PMLB: a large benchmark suite for machine learning evaluation and comparison. BioData Mining **10**(1), 36 (2017), https://epistasislab.github.io/pmlb/index.html
58. Ribeiro, M.T., Singh, S., Guestrin, C.: "Why should I trust you?": Explaining the predictions of any classifier. In: KDD. pp. 1135–1144 (2016)
59. Ribeiro, M.T., Singh, S., Guestrin, C.: Anchors: High-precision model-agnostic explanations. In: AAAI. pp. 1527–1535 (2018)
60. Rivest, R.L.: Learning decision lists. Mach. Learn. **2**(3), 229–246 (1987)
61. Selman, B., Levesque, H.J.: Abductive and default reasoning: A computational core. In: AAAI. pp. 343–348 (1990)
62. Shalev-Shwartz, S., Ben-David, S.: Understanding Machine Learning - From Theory to Algorithms. Cambridge University Press (2014)
63. Shih, A., Choi, A., Darwiche, A.: A symbolic approach to explaining bayesian network classifiers. In: IJCAI. pp. 5103–5111 (2018)
64. Sill, J.: Monotonic networks. In: NIPS. pp. 661–667 (1997)
65. Sivaraman, A., Farnadi, G., Millstein, T.D., den Broeck, G.V.: Counterexample-guided learning of monotonic neural networks. In: NeurIPS (2020)
66. Van den Broeck, G., Darwiche, A.: On the role of canonicity in knowledge compilation. In: AAAI. pp. 1641–1648 (2015)

67. Wäldchen, S., MacDonald, J., Hauch, S., Kutyniok, G.: The computational complexity of understanding binary classifier decisions. J. Artif. Intell. Res. **70**, 351–387 (2021), https://doi.org/10.1613/jair.1.12359
68. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM (2000), http://ls2-www.cs.uni-dortmund.de/monographs/bdd/
69. Wehenkel, A., Louppe, G.: Unconstrained monotonic neural networks. In: NeurIPS. pp. 1543–1553 (2019)
70. You, S., Ding, D., Canini, K.R., Pfeifer, J., Gupta, M.R.: Deep lattice networks and partial monotonic functions. In: NeurIPS. pp. 2981–2989 (2017), https://github.com/tensorflow/lattice

Towards Formal XAI: Formally Approximate Minimal Explanations of Neural Networks

Shahaf Bassan and Guy Katz(✉)

The Hebrew University of Jerusalem, Jerusalem, Israel
{shahaf.bassan,g.katz}@mail.huji.ac.il

Abstract. With the rapid growth of machine learning, deep neural networks (DNNs) are now being used in numerous domains. Unfortunately, DNNs are "black-boxes", and cannot be interpreted by humans, which is a substantial concern in safety-critical systems. To mitigate this issue, researchers have begun working on explainable AI (XAI) methods, which can identify a subset of input features that are the cause of a DNN's decision for a given input. Most existing techniques are heuristic, and cannot guarantee the correctness of the explanation provided. In contrast, recent and exciting attempts have shown that formal methods can be used to generate provably correct explanations. Although these methods are sound, the computational complexity of the underlying verification problem limits their scalability; and the explanations they produce might sometimes be overly complex. Here, we propose a novel approach to tackle these limitations. We (i) suggest an efficient, verification-based method for finding *minimal explanations*, which constitute a *provable approximation* of the global, minimum explanation; (ii) show how DNN verification can assist in calculating lower and upper bounds on the optimal explanation; (iii) propose heuristics that significantly improve the scalability of the verification process; and (iv) suggest the use of *bundles*, which allows us to arrive at more succinct and interpretable explanations. Our evaluation shows that our approach significantly outperforms state-of-the-art techniques, and produces explanations that are more useful to humans. We thus regard this work as a step toward leveraging verification technology in producing DNNs that are more reliable and comprehensible.

1 Introduction

Machine learning (ML) is a rapidly growing field with a wide range of applications, including safety-critical, high-risk systems in the fields of health care [18], aviation [38] and autonomous driving [12]. Despite their success, ML models, and especially deep neural networks (DNNs), remain "black-boxes" — they are incomprehensible to humans and are prone to unexpected behaviour and errors. This issue can result in major catastrophes [13,73], and also in poor decision-making due to brittleness or bias [7,24].

In order to render DNNs more comprehensible to humans, researchers have been working on *explainable AI* (*XAI*), where we seek to construct models for

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 187–207, 2023.
https://doi.org/10.1007/978-3-031-30823-9_10

explaining and interpreting the decisions of DNNs [50,55–57]. Work to date has focused on heuristic approaches, which provide explanations, but do not provide guarantees about the correctness or succinctness of these explanations [14,32,44]. Although these approaches are an important step, their limitations might result in skewed results, possibly failing to meet the regulatory guidelines of institutions and organizations such as the European Union, the US government, and the OECD [51]. Thus, producing DNN explanations that are provably accurate remains of utmost importance.

More recently, the formal verification community has proposed approaches for providing formal and rigorous explanations for DNN decision making [27,31, 51,59]. Many of these approaches rely on the recent and rapid developments in DNN verification [1, 8, 9, 39]. These approaches typically produce an *abductive explanation* (also known as a *prime implicant*, or *PI-explanation*) [31, 58, 59]: a minimum subset of input features, which by themselves already determine the classification produced by the DNN, regardless of any other input features. These explanations afford formal guarantees, and can be computed via DNN verification [31].

Abductive explanations are highly useful, but there are two major difficulties in computing them. First, there is the issue of scalability: computing locally minimal explanations might require a polynomial number of costly invocations of the underlying DNN verifier, and computing a globally minimal explanation is even more challenging [10, 31, 48]. The second difficulty is that users may sometimes prefer "high-level" explanations, not based solely on input features, as these may be easier to grasp and interpret compared to "low-level", complex, feature-based explanations.

To tackle the first difficulty, we propose here new approaches for more efficiently producing verification-based abductive explanations. More concretely, we propose a method for *provably approximating* minimum explanations, allowing stakeholders to use slightly larger explanations that can be discovered much more quickly. To accomplish this, we leverage the recently discovered dual relationship between explanations and contrastive examples [30]; and also take advantage of the sensitivity of DNNs to small adversarial perturbations [64], to compute both lower and upper bounds for the minimum explanation. In addition, we propose novel heuristics for significantly expediting the underlying verification process.

In addressing the second difficulty, i.e. the interpretability limitations of "low-level" explanations, we propose to construct explanations in terms of *bundles*, which are sets of related features. We empirically show that using our method to produce bundle explanations can significantly improve the interpretability of the results, and even the scalability of the approach, while still maintaining the soundness of the resulting explanations.

To summarize, our contributions include the following: (i) We are the first to suggest a method that formally produces sound and minimal abductive explanations that *provably approximate* the global-minimum explanation. (ii) Our three suggested novel heuristics expedite the search for minimal abductive explanations, significantly outperforming the state of the art. (iii) We suggest a

novel approach for using bundles to efficiently produce sound and provable explanations that are more interpretable and succinct.

For evaluation purposes, we implemented our approach as a proof-of-concept tool. Although our method can be applied to any ML model, we focused here on DNNs, where the verification process is known to be NP-complete [39], and the scalable generation of explanations is known to be challenging [31, 58]. We used our tool to test the approach on DNNs trained for digit and clothing classification, and also compared it to state-of-the-art approaches [31, 32]. Our results indicate that our approach was successful in quickly producing meaningful explanations, often running 40% faster than existing tools. We believe that these promising results showcase the potential of this line of work.

The rest of the paper is organized as follows. Sec. 2 contains background on DNNs and their verification, as well as on formal, minimal explanations. Sec. 3 covers the main method for calculating approximations of minimum explanations, and Sec. 4 covers methods for improving the efficiency of calculating these approximations. Sec. 5 covers the use of *bundles* in constructing "high-level", provable explanations. Next, we present our evaluation in Sec. 6. Related work is covered in Sec. 7, and we conclude in Sec. 8.

2 Background

DNNs. A deep neural network (DNN) [46] is a directed graph composed of layers of nodes, commonly called *neurons*. In feed-forward NNs the data flows from the first (*input*) layer, through intermediate (*hidden*) layers, and onto an *output* layer. A DNN's output is calculated by assigning values to its input neurons, and then iteratively calculating the values of neurons in subsequent layers. In the case of *classification*, which is the focus of this paper, each output neuron corresponds to a specific *class*, and the output neuron with the highest value corresponds to the class the input is classified to.

Fig. 1 depicts a simple, feed-forward DNN. The input layer includes three neurons, followed by a weighted sum layer, which calculates an affine transformation of values from the input layer. Given the input $V_1 = [1, 1, 1]^T$, the second layers computes the values $V_2 = [6, 9, 11]^T$. Next comes a ReLU layer, which computes the function $\text{ReLU}(x) = \max(0, x)$ for each neuron in the preceding layer, resulting in $V_3 = [6, 9, 11]^T$. The final (output) layer then computes an affine transformation, resulting in $V_4 = [15, -4]^T$. This indicates that input $V_1 = [1, 1, 1]^T$ is classified as the category corresponding to the first output neuron, which is assigned the greater value.

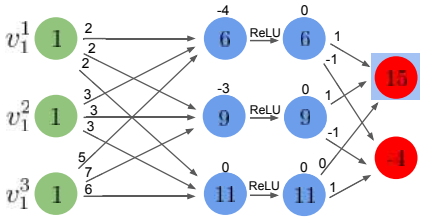


Fig. 1: A simple DNN.

DNN Verification. A DNN verification query is a tuple $\langle P, N, Q \rangle$, where N is a DNN that maps an input vector x to an output vector $y = N(x)$, P is a predicate

on x, and Q is a predicate on y. A DNN verifier needs to decide whether there exists an input x_0 that satisfies $P(x_0) \wedge Q(N(x_0))$ (the `SAT` case) or not (the `UNSAT` case). Typically, P and Q are expressed in the logic of real arithmetic [49]. The DNN verification problem is known to be NP-Complete [39].

Formal Explanations. We focus here on explanations for classification problems, where a model is trained to predict a label for each given input. A classification problem is a tuple $\langle F, D, K, N \rangle$ where (i) $F = \{1, ..., m\}$ denotes the features; (ii) $D = \{D_1, D_2 ..., D_m\}$ denotes the domains of each of the features, i.e. the possible values that each feature can take. The entire feature (input) space is hence $\mathbb{F} = D_1 \times D_2 \times ... \times D_m$; (iii) $K = \{c_1, c_2, ..., c_n\}$ is a set of classes, i.e. the possible labels; and (iv) $N : F \rightarrow K$ is a (non-constant) classification function (in our case, a neural network). A classification instance is the pair (v, c), where $v \in \mathbb{F}$, $c \in K$, and $c = N(v)$. In other words, v is mapped by the neural network N to class c.

Looking at (v, c), we often wish to know why v was classified as c. Informally, an *explanation* is a subset of features $E \subseteq F$, such that assigning these features to the values assigned to them in v already determines that the input will be classified as c, regardless of the remaining features $F \setminus E$. In other words, even if the values that are *not* in the explanation are changed arbitrarily, the classification remains the same. More formally, given input $v = (v_1, ...v_m) \in \mathbb{F}$ with the classification $N(v) = c$, an explanation (sometimes referred to as an *abductive explanation*, or an *AXP*) is a subset of the features $E \subseteq F$, such that:

$$\forall (x \in \mathbb{F}). \quad [\bigwedge_{i \in E} (x_i = v_i) \rightarrow (N(x) = c)] \tag{1}$$

We continue with the running example from Fig. 1. For simplicity, we assume that each input neuron can only be assigned the values 0 or 1. It can be observed that for input $V_1 = [1, 1, 1]^T$, the set $\{v_1^1, v_1^2\}$ is an explanation; indeed, once the first two entries in V_1 are set to 1, the classification remains the same for any value of the third entry (see Fig. 2). We can prove this by encoding a verification query $\langle P, N, Q \rangle = \langle E = v, N, Q_{\neg c} \rangle$, where E is the candidate explanation, and $E = v$ means that we restrict the features in E to their values in v; and $Q_{\neg c}$ implies that the classification is not c. An `UNSAT` result for this query indicates that E is an explanation for instance (v, c).

Clearly, the set of all features constitutes a trivial explanation. However, we are interested in *smaller* explanation subsets, which can provide useful information regarding the decision of the classifier. More precisely, we search for *minimal explanations* and *minimum explanations*. A subset $E \subseteq F$ is a *minimal explanation* (also referred to as a *local-minimal explanation*, or a *subset-minimal explanation*) of instance (v, c) if it is an explanation that ceases to be an explanation if even a single feature is removed from it:

$$\begin{aligned} &(\forall (x \in \mathbb{F}).[\wedge_{i \in E} (x_i = v_i) \rightarrow (N(x) = c)]) \wedge \\ &(\forall (j \in E).[\exists (y \in \mathbb{F}).[\wedge_{i \in E \setminus j} (y_i = v_i) \wedge (N(y) \neq c)]) \end{aligned} \tag{2}$$

Fig. 3 demonstrates that $\{v_1^1, v_1^2\}$ is a minimal explanation in our running example: removing any of its features allows mis-classification.

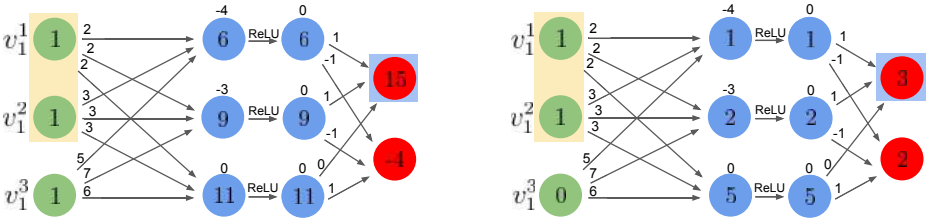


Fig. 2: $\{v_1^1, v_1^2\}$ is an explanation for input $V_1 = [1, 1, 1]^T$.

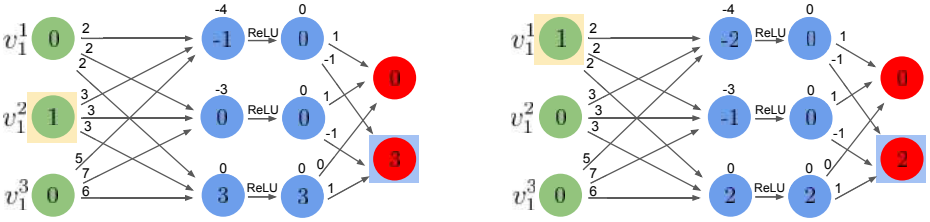


Fig. 3: $\{v_1^1, v_1^2\}$ is a minimal explanation for input $V_1 = [1, 1, 1]^T$.

A *minimum explanation* (sometimes referred to as a *cardinal minimal explanation* or a *PI-explanation*) is defined as a minimal explanation of minimum size; i.e., if E is a minimum explanation, then there does not exist a minimal explanation $E' \neq E$ such that $|E'| < |E|$. Fig. 4 demonstrates that $\{v_1^3\}$ is a minimum explanation for our running example.

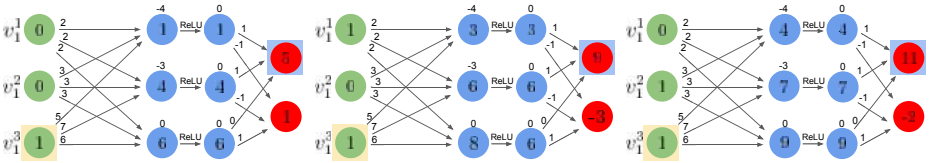


Fig. 4: $\{v_1^3\}$ is a minimum explanation for input $V_1 = [1, 1, 1]^T$.

Contrastive Example. A subset of features $C \subseteq F$ is called a *contrastive example* or a *contrastive explanation (CXP)* if altering the features in C is sufficient to cause the misclassification of a given classification instance (v, c):

$$\exists (x \in \mathbb{F}).[\wedge_{i \in F \setminus C}(x_i = v_i) \wedge (N(x) \neq c)] \tag{3}$$

A contrastive example for our running example is shown in Fig. 5. Notice that the question of whether a set is a contrastive example can be encoded into a verification query $\langle P, N, Q \rangle = \langle (F \setminus C) = v, N, Q_{\neg c} \rangle$, where a `SAT` result indicates that C is a contrastive example. As with explanations, smaller contrastive examples are more valuable than large ones. One useful notion is that of a *contrastive singleton*: a contrastive example of size one. A contrastive singleton could represent a specific pixel in an image, the alteration of which could result in misclassification. Such singletons are leveraged in "one-pixel attacks" [64] (see Fig. 16 in the appendix of the full version of this paper [11]). Contrastive singletons have the following important property:

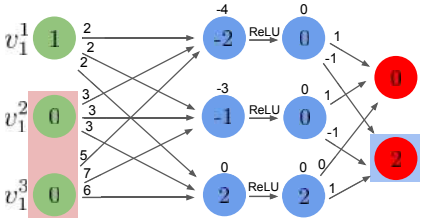


Fig. 5: $\{v_1^2, v_1^3\}$ is a contrastive example for $V_1 = [1, 1, 1]^T$.

Lemma 1. *Every contrastive singleton is contained in all explanations.*

The proof appears in Sec. A of the appendix of the full version of this paper [11]. Lemma 1 implies that each contrastive singleton is contained in all minimal/minimum explanations.

We consider also the notion of a *contrastive pair*, which is a contrastive example of size 2. Clearly, for any pair of features (u, v) where u or v are contrastive singletons, (u, v) is a contrastive pair; however, when we next refer to contrastive pairs, we consider only pairs that *do not* contain any contrastive singletons. Likewise, for every $k > 2$, we can consider contrastive examples of size k, and we exclude from these any contrastive examples of sizes $1, \ldots, k-1$ as subsets.

We state the following theorem, whose proof also appears in Sec. A of the appendix of the full version of this paper [11]:

Lemma 2. *All explanations contain at least one element of every contrastive pair.*

The theorem can be generalized to any $k > 2$; and can be used in showing that the *minimum hitting set (MHS)* of all contrastive examples is exactly the minimum explanation [29, 54] (see Sec. B of the appendix of the full version of this paper [11]). Further, the theorem implies a duality between contrastive examples and explanations [30, 34]: a minimal hitting set of all contrastive examples constitutes a minimal explanation, and a minimal hitting set of all explanations constitutes a minimal contrastive example.

3 Provable Approximations for Minimal Explanations

State-of-the-art approaches for finding minimum explanations exploit the MHS duality between explanations and contrastive examples [31]. The idea is to iteratively compute contrastive examples, and then use their MHS as an under-approximation for the minimum explanation. Finding this MHS is an NP-

complete problem, and is difficult in practice as the number of contrastive examples increases [20]; and although the MHS can be approximated using maximum satisfiability (MaxSAT) or mixed integer linear programming (MILP) solvers [26, 47], existing approaches tackle simpler ML models, such as decision trees [33,36], but face scalability limitations when applied to DNNs [31,58]. Further, enumerating all contrastive examples may in itself take exponential time. Finally, recall that DNN verification is an NP-Complete problem [39]; and so dispatching a verification query to identify each explanation or contrastive example is also very slow, when the feature space is large. Finding *minimal* explanations may be easier [31], but may converge to larger and less meaningful explanations, while still requiring a linear number of calls to the underlying verifier. Our approach, described next, seeks to mitigate these difficulties.

Our overall approach is described in Algorithm 1. It is comprised of two separate threads, intended to be run in parallel. The *upper bounding thread* (T_{UB}) is responsible for computing a minimal explanation. It starts with the entire feature space, and then gradually reduces it, until converging to a minimal explanation. The size of the presently smallest explanation is regarded as an upper bound (UB) for the size of the minimum explanation. Symmetrically, the *lower bounding thread* (T_{LB}) attempts to construct small contrastive sets, used for computing a lower bound (LB) on the size of the minimum explanation. Together, these two bounds allow us to compute the approximation ratio between the minimal explanation that we have discovered and the minimum explanation. For instance, given a minimal explanation of size 7 and a lower bound of size 5, we can deduce that our explanation is at most $\frac{\text{UB}}{\text{LB}} = \frac{7}{5}$ times larger than the minimum. The two threads share global variables that indicate the set of contrastive singletons (Singletons), the set of contrastive pairs (Pairs), the upper and lower bounds (UB, LB), and the set of features that were determined not to participate in the explanation and are "free" to be set to any value (Free). The output of our algorithm is a minimal explanation (F$\setminus$Free), and the approximation ratio ($\frac{\text{UB}}{\text{LB}}$). We next discuss each of the two threads in detail.

Algorithm 1 Minimal Explanation Search

Input N (Neural network), F (features), v (input values), c (class prediction)

1: Singletons, Pairs, Free $\leftarrow \varnothing$, UB $\leftarrow |F|$, LB $\leftarrow 0$ ▷ Global variables
2: Launch thread T_{UB}
3: Launch thread T_{LB}
4: **return** F$\setminus$Free, $\frac{\text{UB}}{\text{LB}}$

The Upper Bounding Thread (T_{UB}). This thread, whose pseudocode appears in Algorithm 2, follows the framework proposed by Ignatiev et al. [31]: it seeks a minimal explanation by starting with the entire feature space, and then iteratively attempting to remove individual features. If removing a feature allows misclassification, we keep it as part of the explanation; otherwise, we remove it

and continue. This process issues a single verification query for each feature, until converging to a minimal explanation (lines 2–8). Although this naıve search is guaranteed to converge to a minimal explanation, it needs not to converge to a *minimum* explanation; and so we apply a more sophisticated ordering scheme, similar to the one proposed by [32], where we use some heuristic model as a way for assigning weights of importance to each input feature. We then check the "least important" input features first, since freeing them has a lower chance of causing a misclassification, and they are consequently more likely to be successfully removed. We then continue iterating over features in ascending order of importance, hopefully producing small explanations.

Algorithm 2 T_{UB}: Upper Bounding Thread

```
1: Use a heuristic model to sort F's features by ascending relevance
2: for each f ∈ F do
3:     Explanation ← F∖Free
4:     if Verify((Explanation∖{f})=v,N,Q_¬c) is UNSAT then
5:         Free ← Free ∪ {f}
6:         UB ← UB – 1
7:     end if
8: end for
```

The Lower Bounding Thread (T_{LB}). The pseudocode for the lower bounding thread (T_{LB}) appears in Algorithm 3. In lines 1–6, the thread searches for contrastive singletons. Neural networks were shown to be very sensitive to adversarial attacks [25] — slight input perturbations that cause misclassification (e.g., the aforementioned one-pixel attack [64]) — and this suggests that contrastive sets, and in particular contrastive singletons, exist in many cases. We observe that identifying contrastive singletons is computationally cheap: by encoding Eq. 3 as a verification query, once for each feature, we can discover all singletons; and in these queries all features but one are fixed, which empirically allows verifiers to dispatch them quickly.

The rest of T_{LB} (lines 9–13) performs a similar process, but with contrastive pairs (which do not contain contrastive singletons as one of their features). We use verification queries to identify all such pairs, and then attempt to find their MHS. We observe that finding the MHS of all contrastive pairs is the 2-MHS problem, which is a reformalization of the *minimum vertex cover* problem (see Sec. B of the appendix of the full version of this paper [11]). Since this is an easier problem than the general MHS problem, solving it with MAX-SAT or MILP often converges quickly. In addition, the minimum vertex cover algorithm has a linear 2-approximating greedy algorithm, which can be used for finding a lower bound in cases of large feature spaces.

More formally, T_{LB} performs an efficient computation of the following bound:

$$\text{LB} = |\text{Singletons}| + |\text{MVC}(\text{Pairs})| \leq \text{MHS}(\text{Cxps}) = E_M \tag{4}$$

Algorithm 3 T_{LB}: Lower Bounding Thread

```
1: for each f ∈ F do                                   ▷ Find all singletons
2:     if Verify((F∖{f}=v,N,Q_¬c) is SAT then
3:         Singletons ← Singletons ∪ {f}
4:         LB ← LB +1
5:     end if
6: end for
7:
8: AllPairs ← Distinct pairs of F∖Singletons
9: for each (a,b) ∈ AllPairs do                        ▷ Find all pairs
10:    if Verify((F∖{a,b}=v,N,Q_¬c) is SAT then
11:        Pairs ←Pairs ∪ {(a, b)}
12:    end if
13: end for
14: LB ← LB + MVC(Pairs)
```

where MVC is the minimum vertex cover, Cxps denotes the set of all contrastive examples, and E_M is the size of the minimum explanation.

It is worth mentioning that this approach can be extended to use contrastive examples of larger sizes ($k = 3, 4, \ldots$), as specified in Sec. C of the appendix of the full version of this paper. The fact that small contrastive examples, such as singletons, exist in large, state-of-the-art DNNs with large inputs [21, 64] suggests that useful approximations exist in large DNNs. In our experiments, we observed that using only singletons and pairs affords good approximations, without incurring overly expensive computations by the underlying verifier.

4 Finding Minimal Explanations Efficiently

Algorithm 1 is the backbone of our approach, but it suffers from limited scalability — particularly, in T_{UB}. As the execution of T_{UB} progresses, and as additional features are "freed", the quickly growing search space slows down the underlying verifier. Here we propose three different methods for expediting this process, by reducing the number of verification queries required.

Method 1: Using Information from T_{LB}. We suggest to leverage the contrastive examples found by T_{LB} to expedite T_{UB}. The process is described in Algorithm 4. In line 3, T_{LB} is queried for the current set of contrastive singletons, which we know must be part of any minimal explanation. These are subtracted from the RemainingFeatures set (features left for T_{UB} to query), and consequently will not be added to the Free set — i.e., they are marked as part of the current explanation. In addition, for any contrastive pair (a, b) found by T_{LB}, either a or b must appear in any minimal explanation; and so, our algorithm skips checking the case where both a and b are removed from F (Line 8). (the method could also be extended to contrastive sets of greater cardinality.)

Algorithm 4 T_{UB} using information from T_{LB}

```
Use a heuristic model to sort F by ascending relevance
RemainingFeatures ← F∖Singletons
for each f ∈ RemainingFeatures do
    Explanation ← F∖Free
    if Verify((Explanation∖{f})=v,N,Q_¬c) is UNSAT then
        Free ← Free ∪ {f}
        UB ← UB – 1
        Delete all features in a pair with f from RemainingFeatures
    end if
end for
```

Method 2: Binary Search. Sorting the features being considered in ascending order of importance can have a significant effect on the size of the explanation found by Algorithm 2. Intuitively, a "perfect" heuristic model would assign the greatest weights to all features in the minimum explanation, and so traversing features in ascending order would first discover all the features that can be removed (`UNSAT` verification queries), followed by all the features that belong in the explanation (`SAT` queries). In this case, a sequential traversal of the features in ascending order is quite wasteful, and it is much better to perform a binary search to find the point where the answer flips from `UNSAT` to `SAT`.

Of course, in practice, the heuristic models are not perfect, leading to potential cases with multiple "flips" from `SAT` to `UNSAT`, and vice versa. Still, if the heuristic is good in practice (which is often the case; see Sec. 6), these flips are scarce. Thus, we propose to perform multiple binary searches, each time identifying one `SAT` query (i.e., a feature added to the explanation). Observe that each time we hit an `UNSAT` query, this indicates that all the queries for features with lower priorities would also yield `UNSAT` — because if "freeing" multiple features cannot change the classification, changing fewer features certainly cannot. Thus, we are guaranteed to find the first `SAT` query in each iteration, and soundness is maintained. This process is described in Algorithm. 6 and in Fig. 14 in the appendix of the full version of this paper [11].

Method 3: Local-Singleton Search. Let N be a DNN, and let x be an input point whose classification we seek to explain. As part of Algorithm 2, T_{UB} iteratively "frees" certain input features, allowing them to take arbitrary values, as it continues to search for features that must be included in the explanation. The increasing number of free features enlarges the search space that the underlying verifier must traverse, thus slowing down verification. We propose to leverage the hypothesis that input points nearby x that are misclassified tend to be clustered; and so, it is beneficial to *fix* the free features to "bad" values, as opposed to letting them take on arbitrary values. We speculate that this will allow the verifier to discover satisfying assignments much more quickly.

This enhancement is shown in Algorithm 5. Given a set Free of features that were previously freed, we fix their values according to some satisfying assignment previously discovered. Thus, the verification of any new feature that we

consider is similar to the case of searching for contrastive singletons, which, as we already know, is fairly fast. See Fig. 15 in the appendix of the full version of this paper [11] for an illustration. The process can be improved further by fixing the freed features to small neighborhoods of the previously discovered satisfying assignment (instead of its exact values), to allow some flexibility while still keeping the query's search space small.

Algorithm 5 T_{UB} using local-singleton search

```
Use a heuristic model to sort F by ascending relevance
RemainingFeatures ← F∖Singletons
for each f ∈ RemainingFeatures do
    Explanation ← F∖Free
    if Verify((Explanation∖{f})=v,N,Q_¬c) is UNSAT then
        Free ← Free ∪ {f}
        UB ← UB – 1
    else
        Extract counter example C
        LocalSingletons ← ∅
        for each f′ ∈ RemainingFeatures do
            if Verify(Explanation∖{f′} = C,N,Q_¬c) is SAT then
                LocalSingletons ← LocalSingletons ∪ {f′}
            end if
        end for
        RemainingFeatures ← RemainingFeatures ∖ LocalSingletons
    end if
end for
```

5 Minimal Bundle Explanations

Fig. 6: Partition input's features into bundles.

So far, we presented methods for generating explanations within a given approximation ratio of the minimum explanation (Sec. 3), and for expediting the computation of these explanations (Sec. 4) — in order to improve the scalability of our explanation generation mechanism. Next, we seek to tackle the second challenge from Sec. 1, namely that these explanations may be too low-level for many users. To address this challenge, we focus on *bundles*, which is a topic well covered in the ML [63] and heuristic XAI literature [50,55] (commonly known as "super-pixels" for computer-vision tasks). Intuitively, bundles are a partitioning of the features into disjoint sets (an illustration appears in Fig. 6). The idea, which we later validate empirically, is that providing explanations in terms of bundles is often easier for humans to comprehend. As an added bonus, using bundles also curtails the search space that the verifier must traverse, expediting the process even further.

Given a feature space $F = \{1, ..., m\}$, a bundle b is just a subset $b \subseteq F$. When dealing with the set of all bundles $B = \{b_1, b_2, ...b_n\}$, we require that they form a partitioning of F, namely $F = \cup b_i$. We define a *bundle explanation* E_B for a classification instance (v, c) as a subset of bundles, $E_B \subseteq B$, such that:

$$\forall(x \in \mathbb{F}).[\wedge_{i \in \cup E_B}(x_i = v_i) \rightarrow (N(x) = c)] \tag{5}$$

The following theorem then connects bundle explanations and explicit, non-bundle explanations:

Theorem 1. *The union of features in a bundle explanation is an explanation.*

The proof directly follows from Eqs. 1 and 5. We note that this definition of bundles implies that features that are not part of the bundle explanation (i.e. features contained in *"free" bundles*) are "free" to be set to any possible value. Another possible alternative for defining bundles could be to allow features in "free" bundles to only change in the same, coordinated manner. We focus here on the former definition, and leave the alternative definition for future work.

Many of the aforementioned results and definitions for explanations can be extended to bundle explanations. In a similar manner to Eq. 5, we can define the notions of minimal and minimum bundle explanations, a contrastive bundle singleton, and contrastive bundle pairs (see Sec. D of the appendix of the full version of this paper [11]). Theorems 1 and 2 can be extended to bundle explanations in a straightforward manner. It then follows that all bundle explanations contain all contrastive singleton bundles, and that all bundle explanations contain at least one bundle of any contrastive bundle pair.

Our method from Secs. 3 and 4 can be similarly performed on bundles rather than on features, and T_{UB} would then be used for calculating a minimal bundle explanation, rather than a minimal explanation. Regarding the aforementioned approximation ratio, we discuss and evaluate two different methods for obtaining it. The first, natural approach is to apply our techniques from Sec. 3 on bundle explanations, thus obtaining a provable approximation for a *minimum bundle explanation*. The upper bound is trivially derived by the size of the bundle explanation found by T_{UB}, whereas the lower bound calculation requires assigning a cost to each bundle, representing the number of features it contains. This is done via a known notion of *minimum hitting sets of bundles (MHSB) [6]* and using minimum *weighted* vertex cover for the approximation of contrastive bundle pairs. This method, which is almost identical to the one mentioned in Sec. 3, is formalized in Sec. D of the appendix of the full version of this paper [11].

The second approach is to calculate an approximation ratio with respect to a regular, non-bundle minimum explanation. The minimal bundle explanation found by T_{UB} is an upper bound on the minimum non-bundle explanation following theorem 5. For computing a lower bound, we can analyze contrastive bundle examples; extract from them contrastive non-bundle examples; and then use the duality property, compute an MHS of these contrastive examples, and derive lower bounds for the size of the minimum explanation. We formalize techniques for performing this calculation in Sec. E of the appendix of the full version of this paper [11].

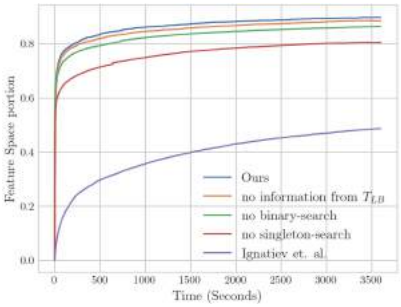
6 Evaluation

Implementation and Setup. For evaluation purposes, we created a proof-of-concept implementation of our approach as a Python framework. Currently, the framework uses the Marabou verification engine [41] as a backend, although other engines may be used. Marabou is a Simplex-based DNN verification framework that is sound and complete [5,39–41,68,69], and which includes support for proof production [35], abstraction [15, 16, 52, 60, 67, 72], and optimization [62]; and has been used in various settings, such as ensemble selection [3], simplification [22, 43] repair [23, 53], and verification of reinforcement-learning based systems [2,4,17]. For sorting features by their relevance, we used the popular XAI method LIME [55]; although again, other heuristics could be used. The MVC was calculated using the classic 2-approximating greedy algorithm. All experiments reported were conducted on x86-64 Gnu/Linux-based machines, using a single Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz core, with a 1-hour timeout.

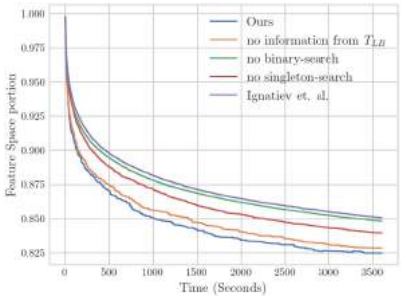
Benchmarks. As benchmarks, we used DNNs trained over the MNIST dataset for handwritten digit recognition [45]. These networks classify 28×28 grayscale images into the digits $0, \ldots, 9$. Additionally, we used DNNs trained over the Fashion-MNIST dataset [71], which classify 28×28 grayscale images into 10 clothing categories ("Dress", "Coat", etc.) For each of these datasets we trained a DNN with the following architecture: (i) an input layer (which corresponds to the image) of size 784; (ii) a fully connected hidden layer with 30 neurons; (iii) another fully connected hidden layer, with 10 neurons; and (iv) a final, softmax layer with 10 neurons, corresponding to the 10 possible output classes. The accuracy of the MNIST DNN was 96.6%, whereas that of the Fashion-MNIST DNN was 87.6%. (We note that we configured LIME to ignore the external border pixels of each input, as these are not part of the actual image.)

In selecting the classification instances to be explained for these networks, we targeted input points where the network was not confident — i.e., where the winning label did not win by a large margin. The motivation for this choice is that explanations are most useful and relevant in cases where the network's decision is unclear, which is reflected in lower confidence scores. Additionally, explanations of instances with lower confidence tend to be larger, facilitating the process of extensive experimentation. We thus selected the 100 inputs from the MNIST and the Fashion-MNIST datasets where the networks demonstrated the lowest confidence scores — i.e., where the difference between the winning output score and the runner-up class score was minimal.

Experiments. Our first goal was to compare our approach to that of Ignatiev et al. [31], which is the current state of the art in verification-based explainability of DNNs. Other approaches consider other ML types, such as decision trees [33,36], or focus on alternative definitions for abductive explanations [42, 70] and are thus not comparable. Because the implementation used in [31] is unavailable, we implemented their approach, using Marabou as the underlying verifier for a fair comparison. In addition, we used the same heuristic model, LIME, for sorting



(a) Average portion of features verified to participate in the explanation.



(b) Average explanation size.

Fig. 7: Our full and ablation-based results, compared to the state of the art for finding minimal explanations on the MNIST dataset.

the input features' relevance. Fig. 7 depicts a comparison of the two approaches, over the MNIST benchmarks. The Fashion-MNIST results were similar, but since the Fashion-MNIST network had lower accuracy it tended to produce larger explanations with lower run-times, resulting in less meaningful evaluations (due to space limitations, these results appear in Fig. 12 in the appendix of the full version of this paper [11]). We compared the approaches according to two criteria: the portion of input features whose participation in the explanation was verified, over time (part (a) of Fig. 7), and the average size of the presently obtained explanation over time, also presented as a fraction of the total number of input features (part (b)). The results indicate that our method significantly improves over the state of the art, verifying the participation of 40.4% additional features, on average, and producing explanations that are 9.7% smaller, on average, at the end of the 1-hour time limit. Furthermore, our method timed out on 10% fewer benchmarks. We regard this as compelling evidence of the potential of our approach to produce more efficient verification-based XAI.

We also looked into comparing our approach to heuristic, non-verification-based approaches, such as LIME itself; but these comparisons did not prove to be meaningful, as the heuristic approaches typically solved benchmarks very quickly, but very often produced incorrect explanations. This matches the findings reported in previous work [14,32].

Next, we set out to evaluate the contribution of each of the components implemented within our framework to overall performance, using an ablation study. Specifically, we ran our framework with each of the components mentioned in Sec. 4, i.e. (i) information exchange between T_{UB} and T_{LB}; (ii) the binary search in T_{UB}; and (iii) local-singleton search, turned off. The results on the MNIST benchmarks appear in Fig. 7; see Fig. 12 in the appendix of the full version of this paper [11] for the Fashion-MNIST results. Our experiments revealed that each of the methods mentioned in Sec. 4 had a favorable impact on both the average portion of features verified, and the average size of the dis-

covered explanation, over time. Fig 7a indicates that the local-singleton search method, used for efficiently proving that features are bound to be *included* in the explanation, was the most significant in reducing the number of features remained for verifying, thus substantially increasing the portion of verified features. Moreover, Fig. 7b indicates that the binary search method, which is used for grouping `UNSAT` queries and proving the *exclusion* of features from the explanation, was the most significant for more efficiently obtaining smaller-sized explanations, over time.

Our second goal was to evaluate the quality of the *minimum* explanation approximation of our method (using the lower/upper bounds) over time. Results are averaged over all benchmarks of the MNIST dataset and are presented in Fig. 8 (similar results on Fashion-MNIST appear in Fig. 13 in the appendix of the full version of this paper [11]). The upper bound represents the average size of the explanation discovered by T_{UB} over time, whereas the lower bound represents the average lower bound discovered by T_{LB} over time. It can be seen that initially, there is a steep increase in the size of the lower bound, as T_{LB} discovered many contrastive singletons. Later, as we begin iterating over contrastive pairs, the verification queries take longer to solve, and progress becomes slower. The average approximation ratio achieved after an hour was 1.61 for MNIST and 1.19 for Fashion-MNIST.

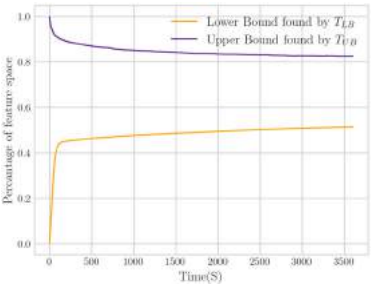
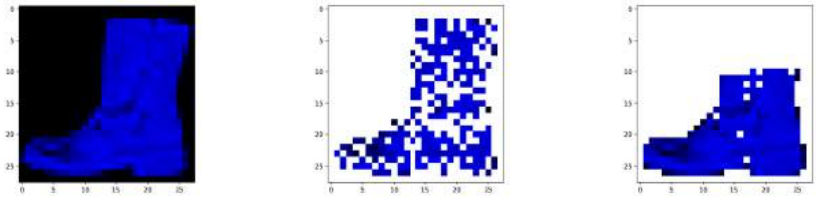


Fig. 8: Average approximation of *minimum* explanation over time.

For our third experiment, we set out to assess the improvements afforded by bundles. We repeated the aforementioned experiments, this time using sets of features representing bundles instead of the features themselves. The segmentation into bundles was performed using the *quickshift* method [65], with LIME again used for assigning relevance to each bundle [55]. We approximate the sizes of the bundle explanations in terms of both the minimum bundle explanation as well as the minimum (non-bundle) explanation (as mentioned in Sec. 5 and in Sec. E of the appendix of the full version of this paper [11]). The bundle configuration showed drastic efficiency improvements, with none of the experiments timing out within the 1-hour time limit, thus improving the portion of timeouts on the MNIST dataset by 84%. The efficiency improvement was obtained at the expense of explanation size, resulting in a decrease of 352% in the approximation ratios obtained for MNIST and 39% for Fashion-MNIST. Nevertheless, when calculating the approximation in terms of the *minimum bundle explanation*, an increase of 12% and 8% was obtained for MNIST and Fashion-MNIST (results are summarized in Table 1 in the appendix of the full version of this paper [11]). For a visual evaluation, we performed the same set of experiments for both bundle and non-bundle implementations, using instances with high confidence rates to obtain smaller-sized explanations that could be more easily interpreted. A



(a) Original Image (b) Explanation (c) Bundle explanation

Fig. 9: Minimal explanations and bundle explanations found by our method on the Fashion-MNIST dataset. White pixels are not part of the explanation.

sample of these results is presented in Fig. 9. Empirically, we observe that the bundle-produced explanations are less complex and more comprehensible.

Overall, we regard our results as compelling evidence that verification-based XAI can soundly produce meaningful explanations, and that our improvements can indeed significantly improve its runtime.

7 Related Work

Our work is another step in the ongoing quest for formal explainability of DNNs, using verification [19, 27, 31, 58]. Related approaches have applied enumeration of contrastive examples [30, 31], which is also an ingredient of our approach. Other approaches focus on producing abductive explanations around an epsilon environment [42, 70]. Similar work has been carried out for decision sets [33], lists [28] and trees [36], where the problem appears to be simpler to solve [36]. Our work here tackles DNNs, which are known to be more difficult to verify [39].

Prior work has also sought to produce approximate explanations, e.g., by using δ-relevant sets [37,66]. This line of work has focused on probabilistic methods for generating explanations, which jeopardizes soundness. There has also been extensive work in heuristic XAI [50, 55, 56, 61], but here, too, the produced explanations are not guaranteed to be correct.

8 Conclusion

Although DNNs are becoming crucial components of safety-critical systems, they remain "black-boxes", and cannot be interpreted by humans. Our work seeks to mitigate this concern, by providing formally correct explanations for the choices that a DNN makes. Since discovering the minimum explanations is difficult, we focus on approximate explanations, and suggest multiple techniques for expediting our approach — thus significantly improving over the current state of the art. In addition, we propose to use bundles to efficiently produce more meaningful explanations. Moving forward, we plan to leverage lightweight DNN verification

techniques for improving the scalability of our approach [49], as well as extend it to support additional DNN architectures.

References

1. M. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, pages 197–210, 2019.
2. G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Verifying Learning-Based Robotic Navigation Systems, 2022. Technical Report. https://arxiv.org/abs/2205.13536.
3. G. Amir, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 27–37, 2022.
4. G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.
5. G. Amir, H. Wu, C. Barrett, and G. Katz. An SMT-Based Approach for Verifying Binarized Neural Networks. In *Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 203–222, 2021.
6. E. Angel, E. Bampis, and L. Gourvès. On the Minimum Hitting Set of Bundles Problem. *Theoretical Computer Science*, 410(45):4534–4542, 2009.
7. J. Angwin, J. Larson, S. Mattu, and L. Kirchner. Machine Bias. *Ethics of Data and Analytics*, pages 254–264, 2016.
8. G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.
9. T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks And its Security Applications. In *Proc. 26th ACM Conf. on Computer and Communication Security (CCS)*, pages 1249–1264, 2019.
10. P. Barceló, M. Monet, J. Pérez, and B. Subercaseaux. Model interpretability through the lens of computational complexity. *Advances in neural information processing systems*, 33:15487–15498, 2020.
11. S. Bassan and G. Katz. Towards Formally Approximate Minimal Explanations of Neural Networks, 2022. Technical Report. https://arxiv.org/abs/2210.13915.
12. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. http://arxiv.org/abs/1604.07316.
13. CACM. A Case Against Mission-Critical Applications of Machine Learning. *Communications of the ACM*, 62(8):9–9, 2019.
14. O.-M. Camburu, E. Giunchiglia, J. Foerster, T. Lukasiewicz, and P. Blunsom. Can I Trust the Explainer? Verifying Post-Hoc Explanatory Methods, 2019. Technical Report. http://arxiv.org/abs/1910.02065.
15. Y. Elboher, E. Cohen, and G. Katz. Neural Network Verification using Residual Reasoning. In *Proc. 20th Int. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 173–189, 2022.
16. Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 43–65, 2020.

17. T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 305–318, 2021.
18. A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean. A Guide to Deep Learning in Healthcare. *Nature Medicine*, 25(1):24–29, 2019.
19. T. Fel, M. Ducoffe, D. Vigouroux, R. Cadène, M. Capelle, C. Nicodème, and T. Serre. Don't Lie to Me! Robust and Efficient Explainability with Verified Perturbation Analysis, 2022. Technical Report. http://arXivpreprintarXiv:2202.07728.
20. A. Gainer-Dewar and P. Vera-Licona. The Minimal Hitting Set Generation Problem: Algorithms and Computation. *SIAM Journal on Discrete Mathematics*, 31(1):63–100, 2017.
21. S. Garg and G. Ramakrishnan. BAE: Bert-Based Adversarial Examples for Text Classification, 2020. Technical Report. https://arxiv.org/abs/2004.01970.
22. S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.
23. B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In *Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 260–278, 2020.
24. I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. http://arxiv.org/abs/1412.6572.
25. S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel. Adversarial Attacks on Neural Network Policies, 2017. Technical Report. http://arxiv.org/abs/1702.02284.
26. IBM. The CPLEX Optimizer, 2018.
27. A. Ignatiev. Towards Trustable Explainable AI. In *Proc. 29th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 5154–5158, 2020.
28. A. Ignatiev and J. Marques-Silva. SAT-Based Rigorous Explanations for Decision Lists. In *Proc. 24th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, pages 251–269, 2021.
29. A. Ignatiev, A. Morgado, and J. Marques-Silva. Propositional Abduction with Implicit Hitting Sets, 2016. Technical Report. http://arxiv.org/abs/1604.08229.
30. A. Ignatiev, N. Narodytska, N. Asher, and J. Marques-Silva. From Contrastive to Abductive Explanations and Back Again. In *Proc. 19th Int. Conf. of the Italian Association for Artificial Intelligence (AIxIA)*, pages 335–355, 2020.
31. A. Ignatiev, N. Narodytska, and J. Marques-Silva. Abduction-Based Explanations for Machine Learning Models. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, pages 1511–1519, 2019.
32. A. Ignatiev, N. Narodytska, and J. Marques-Silva. On Validating, Repairing and Refining Heuristic Ml Explanations, 2019. Technical Report. http://arxiv.org/abs/1907.02509.
33. A. Ignatiev, F. Pereira, N. Narodytska, and J. Marques-Silva. A SAT-Based Approach to Learn Explainable Decision Sets. In *Proc. 9th Int. Joint Conf. on Automated Reasoning (IJCAR)*, pages 627–645, 2018.
34. A. Ignatiev, A. Previti, M. Liffiton, and J. Marques-Silva. Smallest MUS Extraction with Minimal Hitting Set Dualization. In *Proc. 21st Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 173–182, 2015.

35. O. Isac, C. Barrett, M. Zhang, and G. Katz. Neural Network Verification with Proof Production. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 38–48, 2022.
36. Y. Izza, A. Ignatiev, and J. Marques-Silva. On Explaining Decision Trees, 2020. Technical Report. http://arxiv.org/abs/2010.11034.
37. Y. Izza, A. Ignatiev, N. Narodytska, M. Cooper, and J. Marques-Silva. Efficient Explanations with Relevant Sets, 2021. Technical Report. http://arxiv.org/abs/2106.00546.
38. K. Julian, M. Kochenderfer, and M. Owen. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.
39. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
40. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021.
41. G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.
42. E. La Malfa, A. Zbrzezny, R. Michelmore, N. Paoletti, and M. Kwiatkowska. On Guaranteed Optimal Robust Explanations for NLP Models, 2021. Technical Report. https://arxiv.org/abs/2105.03640.
43. O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 183–192, 2021.
44. H. Lakkaraju and O. Bastani. "How do I Fool You?" Manipulating User Trust via Misleading Black Box Explanations. In *Proc. AAAI/ACM Conf. on AI, Ethics, and Society (AIES)*, pages 79–85, 2020.
45. Y. LeCun. The MNIST Database of Handwritten Digits, 1998. http://yann.lecun.com/exdb/mnist/.
46. Y. LeCun, Y. Bengio, and G. Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.
47. C. Li and F. Manya. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, pages 903–927. IOS Press, 2021.
48. P. Liberatore. Redundancy in logic i: Cnf propositional formulae. *Artificial Intelligence*, 163(2):203–232, 2005.
49. C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. Algorithms for Verifying Deep Neural Networks, 2020. Technical Report. http://arxiv.org/abs/1903.06758.
50. S. M. Lundberg and S.-I. Lee. A Unified Approach to Interpreting Model Predictions. In *Proc. 31st Conf. on Neural Information Processing Systems (NeurIPS)*, 2017.
51. J. Marques-Silva and A. Ignatiev. Delivering Trustworthy AI through formal XAI. In *Proc. 36th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 3806–3814, 2022.
52. M. Ostrovsky, C. Barrett, and G. Katz. An Abstraction-Refinement Approach to Verifying Convolutional Neural Networks. In *Proc. 20th. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2022.

53. I. Refaeli and G. Katz. Minimal Multi-Layer Modifications of Deep Neural Networks. In *Proc. 5th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS)*, 2022.
54. R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.
55. M. Ribeiro, S. Singh, and C. Guestrin. "Why should I Trust You?" Explaining the Predictions of any Classifier. In *Proc. 22nd Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1135–1144, 2016.
56. M. Ribeiro, S. Singh, and C. Guestrin. Anchors: High-Precision Model-Agnostic Explanations. In *Proc. 32nd AAAI Conf. on Artificial Intelligence (AAAI)*, 2018.
57. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-Cam: Visual Explanations from Deep Networks via Gradient-Based Localization. In *Proc. 20th IEEE Int. Conf. on Computer Vision (ICCV)*, pages 618–626, 2017.
58. W. Shi, A. Shih, A. Darwiche, and A. Choi. On Tractable Representations of Binary Neural Networks, 2020. Technical Report. http://arxiv.org/abs/2004.02082.
59. A. Shih, A. Choi, and A. Darwiche. A Symbolic Approach to Explaining Bayesian Network Classifiers, 2018. Technical Report. http://arxiv.org/abs/1805.03364.
60. G. Singh, T. Gehr, M. Puschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.
61. D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg. Smoothgrad: Removing Noise by Adding Noise, 2017. Technical Report. http://arxiv.org/abs/1706.03825.
62. C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU Networks. *Journal of Machine Learning*, pages 1–28, 2021.
63. D. Stutz, A. Hermans, and B. Leibe. Superpixels: An Evaluation of the State-of-the-Art. *Computer Vision and Image Understanding*, 166:1–27, 2018.
64. J. Su, D. Vargas, and K. Sakurai. One Pixel Attack for Fooling Deep Neural Networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019.
65. A. Vedaldi and S. Soatto. Quick Shift and Kernel Methods for Mode Seeking. In *Proc. 10th European Conf. on Computer Vision (ECCV)*, pages 705–718, 2008.
66. S. Waeldchen, J. Macdonald, S. Hauch, and G. Kutyniok. The Computational Complexity of Understanding Binary Classifier Decisions. *Journal of Artificial Intelligence Research*, 70:351–387, 2021.
67. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, 2018.
68. H. Wu, A. Ozdemir, A. Zeljić, A. Irfan, K. Julian, D. Gopinath, S. Fouladi, G. Katz, C. Păsăreanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 128–137, 2020.
69. H. Wu, A. Zeljić, G. Katz, and C. Barrett. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In *Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 143–163, 2022.
70. M. Wu, H. Wu, and C. Barrett. VeriX: Towards Verified Explainability of Deep Neural Networks, 2022. Technical Report. https://arxiv.org/abs/2212.01051.
71. H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNist: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017. Technical Report. http://arxiv.org/abs/1708.07747.

72. T. Zelazny, H. Wu, C. Barrett, and G. Katz. On Reducing Over-Approximation Errors for Neural Network Verification. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 17–26, 2022.
73. Z. Zhou and L. Sun. Metamorphic Testing of Driverless Cars. *Communications of the ACM*, 62(3):61–67, 2019.

OccRob: Efficient SMT-Based Occlusion Robustness Verification of Deep Neural Networks

Xingwu Guo [1], Ziwei Zhou [1], Yueling Zhang [1], Guy Katz [2], Min Zhang [1(✉)]

[1] Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
zhangmin@sei.ecnu.edu.cn
[2] The Hebrew University of Jerusalem, Jerusalem, Isarel

Abstract. Occlusion is a prevalent and easily realizable semantic perturbation to deep neural networks (DNNs). It can fool a DNN into misclassifying an input image by occluding some segments, possibly resulting in severe errors. Therefore, DNNs planted in safety-critical systems should be verified to be robust against occlusions prior to deployment. However, most existing robustness verification approaches for DNNs are focused on non-semantic perturbations and are not suited to the occlusion case. In this paper, we propose the first efficient, SMT-based approach for formally verifying the occlusion robustness of DNNs. We formulate the occlusion robustness verification problem and prove it is NP-complete. Then, we devise a novel approach for encoding occlusions as a part of neural networks and introduce two acceleration techniques so that the extended neural networks can be efficiently verified using off-the-shelf, SMT-based neural network verification tools. We implement our approach in a prototype called OccRob and extensively evaluate its performance on benchmark datasets with various occlusion variants. The experimental results demonstrate our approach's effectiveness and efficiency in verifying DNNs' robustness against various occlusions, and its ability to generate counterexamples when these DNNs are not robust.

1 Introduction

Deep neural networks (DNNs) are computer-trained *programs* that can implement hard-to-formally-specify tasks. They have repeatedly demonstrated their potential in enabling artificial intelligence in various domains, such as face recognition [6] and autonomous driving [27]. They are increasingly being incorporated into safety-critical applications with interactive environments. To ensure the security and reliability of these applications, DNNs must be highly dependable against adversarial and environmental perturbations. This dependability property is known as *robustness* and is attracting a considerable amount of research efforts from both academia and industry, aimed at ensuring robustness via different technologies such as adversarial training [13,28], testing [40,33], and formal verification [34,10,5].

Occlusion is a prevalent kind of perturbation, which may cause DNNs to misclassify an image by occluding some segment thereof [38,25,8]. For instance, a "turn left" traffic sign may be misclassified as "go straight" after it is occluded by a tape, probably resulting in traffic accidents. A similar situation may occur in face recognition, where many well-trained neural networks fail to recognize faces correctly when they are partially occluded, such as when glasses are worn[37]. A neural network is called *robust against occlusions*

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 208–226, 2023.
https://doi.org/10.1007/978-3-031-30823-9_11

if small occlusions do not alter its classification results. Generally, we wish a DNN to be robust against occlusions that appear negligible to humans.

It is challenging to verify whether a DNN is robust or not on an input image if the image is occluded. On the one hand, the verification problem is non-convex due to the non-linear activation functions in DNNs. It is NP-complete even when dealing with common, fully connected feed-forward neural networks (FNNs) [20]. On the other hand, unlike existing perturbations, occlusions are challenging to encode using L_p norms. Most existing robustness verification approaches assume that perturbations need to be defined by L_p norms and then apply approximations and abstract interpretation techniques [34,10,5] as part of the verification process. The semantic effect of occlusions partially alters the values of some neighboring pixels from large to small or in the inverse direction, e.g., 255 to 0, when a black occlusion occludes a white pixel. Therefore, existing techniques for perturbations in L_p norms are not suited to occlusion perturbations.

SMT-based approaches have been shown to be an efficient approach to DNN verification [20]. They are both sound and complete, in that they always return definite results and produce counterexamples in non-robust cases. We show that, although it is straightforward to encode the occlusion robustness verification problem into SMT formulas, solving the constraints generated by this naïve encoding is experimentally beyond the reach of state-of-the-art SMT solvers, due to the inclusion of a large number of the piece-wise ReLU activation functions. Consequently, such a straightforward encoding approach cannot scale to large networks.

In this paper, we systematically study the occlusion robustness verification problem of DNNs. We first formalize and prove that the problem is NP-complete for ReLU-based FNNs. Then, we propose a novel approach for encoding various occlusions and neural networks together to generate new equivalent networks that can be efficiently verified using off-the-shelf SMT-based robustness verification tools such as Marabou [21]. In our encoding approach, although additional neurons and layers are introduced for encoding occlusions, the number is reasonably small and independent of the networks to be verified. The efficiency improvement of our approach comes from the fact that our approach significantly reduces the number of constraints introduced while encoding the occlusion and leverages the backend verification tool's optimization against the neural network structure. Furthermore, we introduce two acceleration techniques, namely input-space splitting to reduce the search space of a single verification, which can significantly improve verification efficiency, and label sorting to help verification terminates earlier. We implement a tool called OccRob with Marabou as the backend verification tool. To our knowledge, this is the first work on formally verifying the occlusion robustness of deep neural networks.

To demonstrate the effectiveness and efficiency of OccRob, we evaluate it on six representative FNNs trained on two benchmark datasets. The empirical results show that our approach is effective and efficient in verifying various types of occlusions with respect to the occlusion position, size, and occluding pixel value.

Contributions. We make the following three major contributions: (i) we propose a novel approach for encoding occlusion perturbations, by which we can leverage *off-the-shelf* SMT-based robustness verification tools to verify the robustness of neural networks

against various occlusion perturbations; (ii) we prove the verification problem of the occlusion robustness is NP-complete and introduce two acceleration techniques, i.e., label sorting and input space splitting, to improve the efficiency of verification further; and (iii) we implement a tool called OccRob and conduct experiments extensively on a collection of benchmarks to demonstrate its effectiveness and efficiency.

Paper Organization. Sec. 2 introduces preliminaries. Sec. 3 formulates the occlusion robustness verification problem and studies its complexity. Sec. 4 presents our encoding approach and acceleration techniques for the verification. Sec. 5 shows the experimental results. Sec. 6 discusses related work, and Sec. 7 concludes the paper.

We omit the complete proofs and experimental results due to the page limit. Please refer to the technical report [15] for more details.

2 Preliminaries

2.1 Deep Neural Networks and the Robustness

As shown in Fig. 1, a deep neural network consists of multiple layers. The neurons on the input layer take input values, which are computed and propagated through the hidden layers and then output by the output layer. The neurons on each layer are connected to those on the predecessor and successor layers. We only consider fully connected, feedforward networks (FNNs) [11].

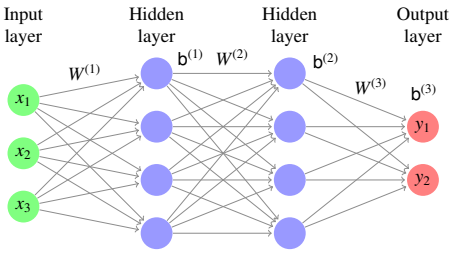


Fig. 1: A fully-connected feed-forward neural network (FNN).

Given a λ-layer neural network, let $W^{(i)}$ be the weight matrix between the $(i-1)$-th and i-th layers, and $\mathsf{b}^{(i)}$ be the biases of the corresponding neurons, where $i = 1, 2, \ldots, \lambda$. The network implements a function $F : \mathbb{R}^u \to \mathbb{R}^r$ that is recursively defined by:

$$z^{(0)} = x$$
$$z^{(i)} = \sigma(W^{(i)} \cdot z^{(i-1)} + \mathsf{b}^{(i)}),\ \ for\ i = 1, \ldots, \lambda - 1 \qquad \text{(Layer Function)}$$
$$F(x) = W^{(\lambda)} \cdot z^{(\lambda-1)} + \mathsf{b}^{(\lambda)} \qquad \text{(Network Function)}$$

where $\sigma(\cdot)$ is called an *activation function* and $z^{(i)}$ denotes the result of neurons at the i-th layer.

For example, Fig. 1 shows a 3-layer neural network with three input neurons and two output neurons, namely, $\lambda = 3$, $u = 3$ and $r = 2$.

For the sake of simplicity, we use $\Phi_F(x) = arg\ max_{\ell \in L} F(x)$ to denote the label ℓ such that the probability $F_\ell(x)$ of classifying x to ℓ is larger than those to other labels, where L represents the set of labels. The activation function σ usually can be a piece-wise Rectified Linear Unit (ReLU), $\sigma(x) = max(x, 0)$), or S-shape functions like Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$, Tanh $\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, or Arctan $\sigma(x) = tan^{-1}(x)$. In this work, we focus on the networks that only contain ReLU activation functions, which are widely adopted in real-world applications.

Fig. 2: Two multiform and uniform occlusions to traffic signs causing mis-classifications.

A neural network is called *robust* if small perturbations to its inputs do not alter the classification result [39]. Specifically, given a network F, an input x_0 and a set Ω of perturbed inputs of x_0, F is called locally robust with respect to x_0 and Ω if F classifies all the perturbed inputs in Ω to the same label as it does x_0.

Definition 1 (Local Robustness [17]). *A neural network $F : \mathbb{R}^u \to \mathbb{R}^r$ is called locally robust with respect to an input x_0 and a set Ω of perturbed inputs of x if $\forall x \in \Omega, \Phi_F(x) = \Phi_F(x_0)$ holds.*

Usually, the set Ω of perturbed inputs is defined by an ℓ_p-norm ball around x_0 with a radius of ϵ, i.e., $\mathbb{B}_p(x_0, \epsilon) := \{x \mid \|x - x_0\|_p \leq \epsilon\}$ [17,2].

2.2 Occlusion Perturbation

In the context of image classification networks, occlusion is a kind of perturbation that blocks the pixels in certain areas before the image is fed into the network. Existing studies showed that the classification accuracy of neural networks could be significantly decreased when the input objects are artificially occluded [23,44].

Occlusions can have various occlusion shapes, sizes, colors, and positions. The shapes can be square, rectangle, triangle, or irregular shape. The size is measured by the number of occluded pixels. The occlusion color specifies the colors occluded pixels can take. The coloring of an occlusion can be either uniform, where all occluded pixels share the same color, or multiform, where these colors can vary in the range of $[-\epsilon, \epsilon]$, where ϵ specifies the threshold between an occluded pixel's value and its original value.

Prior studies [8,3] showed that both the uniform and multiform occlusions could cause misclassification to neural networks. Fig. 2 shows two examples of multiform and uniform occlusions, respectively. The traffic sign for "70km/h speed limit" in Fig. 2(a) is misclassified to "30km/h" by adding a 5×5 multiform occlusion. Fig. 2(d) shows another sign, with different light conditions, where a 3×3 uniform occlusion (in Fig. 2(c)) causes the sign to be misclassified to "30km/h".

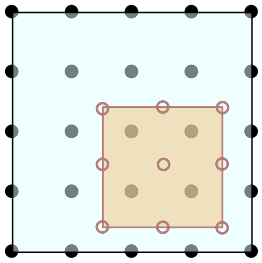


Fig. 3: An example occlusion on a 5×5 image at real number position.

The occlusion position is another aspect of defining occlusions. An occlusion can be placed precisely on the pixels of an image, or between a pixel and its neighbors. Fig. 3 shows an example, where the dots represent image

pixels and the circles are the occluding pixels that will substitute the occluded ones. We say that an occlusion pixel $\vartheta_{i',j'}$ at location (i', j') surrounds an image pixel $p_{i,j}$ at location (i, j) if and only if $|i - i'| < 1$ and $|j - j'| < 1$. Note that i', j' are real numbers, representing the location where the occlusion pixel o is placed on the image. An image pixel can be occluded by the substitute occlusion pixels if the occlusion pixels surround the image pixel.

There are at most four surrounding occlusion pixels for each image pixel, as shown in Fig. 3. Let $\mathbb{I}_p$ be the set of the locations where the surrounding occlusion pixels of p are placed. After the occlusion, the value of pixel $p_{i,j}$ is altered to the new one denoted by $p'_{i,j}$, which can be computed by interpolation [19,22] such as next neighbour interpolation or Bi-linear interpolation based on occlusion pixels in $\mathbb{I}_p$. Besides that, we use a method based on L_1-distance to calculate how much a pixel is occluded. Since the L_1-distance of two adjacent pixels is 1, a surrounding occlusion pixel should not affect the image pixel if their L_1-distance is greater than 1. The formula $max(0, (|1 - i' + i|) + (1 - j' + j) - 1)$ indicates how much an image pixel at (i, j) is occluded by an occlusion pixel at (i', j'). For instance, occlusion pixel at $(i', j') = (0.9, 0.9)$ has no effect to image pixel $(i, j) = (0, 0)$ since their L_1-distance is larger than 1. Therefore, the occlusion factor $s_{i,j}$ for pixel p at (i, j) can be calculated based on all surrounding occlusion pixels in $\mathbb{I}_p$ as:

$$s_{i,j} = max(0, \sum_{i'_0, j' \in \mathbb{I}_p} (|1 - j + j'|) + \sum_{i', j'_0 \in \mathbb{I}_p} (|1 - i' + i|) - 1) \tag{1}$$

where (i'_0, j'_0) is the first element of $\mathbb{I}_p$. Notably, s is 1 for completely occluded pixel and 0 for the pixel that is not occluded, otherwise s has a value between $(0, 1)$. Also, it is a special case for Equation 1 when (i', j') are integers, where s can be reduced to 0 or 1.

3 The Occlusion Robustness Verification Problem

Let $\mathbb{R}^{m \times n}$ be the set of images whose height is m and width is n. We use $\mathbb{N}_{1,m}$ (*resp.* $\mathbb{N}_{1,n}$) to denote the set of all the natural numbers ranging from 1 to m (*resp.* n). A coloring function $\zeta : \mathbb{R}^{m \times n} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a mapping of each pixel of an image to its corresponding color value. Given an image $x \in \mathbb{R}^{m \times n}$, $\zeta(x, i, j)$ defines the value to color the pixel of x at (i, j).

Definition 2 (Occlusion function). *Given a coloring function ζ and an occlusion ϑ of size $w \times h$, the occlusion function is defined as function $\gamma_{\zeta,w \times h} : \mathbb{R}^{m \times n} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{m \times n}$ such that $x' = \gamma_{\zeta,w \times h}(x, a, b)$ if for all $i \in \mathbb{N}_{1,n}$ and $j \in \mathbb{N}_{1,m}$, there is,*

$$x'_{i,j} = x_{i,j} - s_{i,j} \times (x_{i,j} - \zeta(x, i, j)), \tag{2}$$

$$where, \zeta(x, i, j) = \frac{\sum_{(i',j') \in \mathbb{I}_{x_{i,j}}} \vartheta_{i',j'} \sqrt{(i - i')^2 + (j - j')^2}}{\sum_{(i',j') \in \mathbb{I}_{x_{i,j}}} \sqrt{(i - i')^2 + (j - j')^2}}. \tag{3}$$

s in Equation 2 is the occlusion factor for pixel at (i, j) as mentioned in Sec. 2.2. Note that when i', j' are integers, Equation 2 can be reduced to $x_{i,j} = \vartheta_{i,j}$, which represents that $x_{i,j}$ is completely occluded by the occlusion. In other words, the integer case is a

special case of the real number case. Also, when pixel at (i, j) is not occluded, since $s_{i,j} = 0$. In this case, Equation 2 can be reduced to $x'_{i,j} = x_{i,j}$.

Interpolation is handled by ζ showed in Equation 3. It shows the standard form for the color of the new $x'_{i,j}$. A unique color value is specified for all the pixels in the occluded area for a uniform occlusion. Therefore, ζ in Equation 3 can be reduced to $\zeta(x, i, j) = \mu$ for some $\mu \in [0, 1]$. The coloring function in a multiform occlusion is defined as $\zeta(x, i, j) = x_{i,j} + \Delta_p$ with $\Delta_p \in [-\epsilon, \epsilon]$, where $\epsilon \in \mathbb{R}$ defines the threshold that a pixel can be altered.

Definition 3 (Local occlusion robustness). *Given a DNN $F : \mathbb{R}^{m\times n} \to \mathbb{R}^r$, an occlusion function $\gamma_{\zeta,w\times h} : \mathbb{R}^{m\times n} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{m\times n}$ with respect to coloring function ζ and occlusion size $w \times h$, and an input image x, F is called local occlusion robust on x with $\gamma_{\zeta,w\times h}$ if $\Phi_F(x) = \Phi_F(\gamma_{\zeta,w\times h}(x, a, b))$ holds for all $1 \le a \le n$ and $1 \le b \le m$.*

Intuitively, Definition 3 means that F is robust on x against the occlusions of $\gamma_{\zeta,w\times h}$, if on any occluded image of x by the occlusion function $\gamma_{\zeta,w\times h}$, F always returns the same classification result as on the original image x. Depending on the coloring function ζ, the definition applies to various occlusions concerning shapes, colors, sizes, and positions. We can also extend the above definition to the global occlusion robustness if F is robust on all images concerning $\gamma_{\zeta,w\times h}$.

We prove that even for the case of uniform occlusion, a special case of the multiform one, the local occlusion robustness verification problem is NP-complete on the ReLU-based neural networks.

4 SMT-Based Occlusion Robustness Verification

4.1 A Naïve SMT Encoding Method

The verification problem of FNNs' local occlusion robustness can be straightforwardly encoded into an SMT problem. In Definition 3, we assume that x is classified by Φ to the label ℓ_q, i.e., $\Phi(x) = \ell_q$, for a label $\ell_q \in L$. To prove F is robust on x after x is occluded by occlusion ϑ with size $w \times h$, it suffices to prove that F classifies every occluded image $x' = \gamma_{\zeta,w\times h}(a, b)$ to ℓ_q for all $1 \le a \le n$ and $1 \le b \le m$. This is equivalent to proving that the following constraints are not satisfiable:

$$1 \le a \le n, 1 \le b \le m, \tag{4}$$

$$\bigwedge_{i\in\mathbb{N}_{1,n}, j\in\mathbb{N}_{1,m}} \Big(((a-1 < i < a+w+1) \wedge (b-1 < j < b+h+1) \wedge x'_{i,j} = \gamma_{\zeta,w\times h}(x, a, b)_{i,j}) \vee ((i \ge a+w+1) \vee (i \le a-1) \vee (j \ge b+h+1) \vee (j \le b-1)) \wedge x'_{i,j} = x_{i,j})\Big), \tag{5}$$

$$\bigvee_{l\in\mathbb{N}_{1,q-1}\cup\mathbb{N}_{q+1,r}} F(x')_l \ge F(x')_q. \tag{6}$$

The conjuncts in Eq. 5 define that x' is an occluded instance of x, and the disjuncts in Eq. 6 indicate that, when satisfiable, there exists some label ℓ_i which has a higher probability than ℓ_q to be classified to. Namely, the occlusion robustness of F on x is falsified, with x' being a witness of the non-robustness. Note that this naive encoding

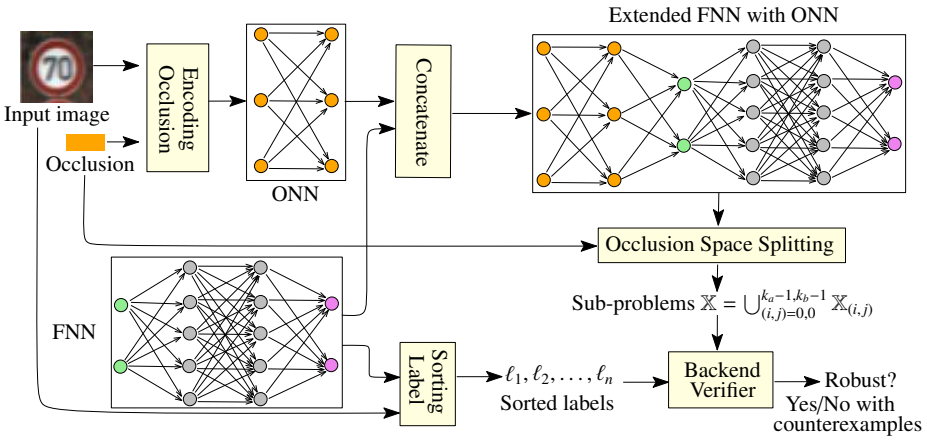


Fig. 4: The workflow of encoding and verifying FNN's robustness against occlusions.

considers the occlusion position's real number cases since function γ implicitly includes the interpolation.

Although the above encoding is straightforward, solving the encoded constraints is experimentally beyond the reach of general-purpose existing SMT solvers due to the piece-wise linear ReLU activation functions in the definition of F in the constraints of Eq. 6, and the large search space $m \times n \times (2\epsilon)^{w \times h}$ (see Experiment II in Sec. 5).

4.2 Our Encoding Approach

An Overview of the Approach. To improve efficiency, we propose a novel approach for encoding occlusion perturbations into four layers of neurons and concatenating the original network to these so-called *occlusion layers*, constituting a new neural network which can be efficiently verified using state-of-the-art, SMT-based verifiers.

Fig. 4 shows the overview of our approach. Given an input image and an occlusion, we first construct a 3-hidden-layer occlusion neural network (ONN) and then concatenate it to the original FNN by connecting the ONN's output layer to the FNN's input layer. The combined network represents all possible occluded inputs and their classification results. The robustness of the constructed network can be verified using the existing SMT-based neural network verifiers.

We introduce two acceleration techniques to speed up the verification further. First, we divide the occlusion space into several smaller, orthogonal spaces, and verify a finite set of sub-problems on the smaller spaces. Second, we employ the eager falsification technique [14] to sort the labels according to their probabilities of being misclassified to. The one with a larger probability is verified earlier by the backend tools. Whenever a counterexample is returned, an occluded image is found such that its classification result differs from the original one. If all sub-problems are verified and no counterexamples are found, the network is verified robust on the input image against the provided occlusion.

Encoding Occlusions as Neural Networks. Given a coloring function ζ, an occlusion size $w \times h$ and an input image x of size $m \times n$, we construct a neural network $O : \mathbb{R}^{4+ct} \rightarrow \mathbb{R}^{m \times n}$ to encode all the possible occluded images of x, where $c = 1$ if x is a grey image

and $c = 3$ if x is an RGB image, $t = 0$ for the uniform occlusion and $t = w \times h$ for the multiform one.

Fig. 5 shows the neural network architecture for encoding occlusions. We divide it into a fundamental part and an additional part. The former encodes the occlusion position and the uniform occlusion color. The additional part is needed only by the multiform occlusion to encode the coloring function. Without loss of generality, we assume that the input layer takes the vector (a, w, b, h, ζ), where (a, b) is the top-left coordinate of occlusion area in x. The coloring function ζ is admitted by other $c \times t$ neurons in the input layer when the occlusion is multiform.

(1) Encoding occlusion positions. We explain the weights and biases that are defined in the neural network to encode the occlusion position. On the connections between the input layer and the first hidden layer, the weights in matrices $W_{1,1}$, $W_{1,2}$ and $W_{1,3}$ are 1, -1 and -1, respectively. Note that we hide all the edges whose weights are 0 in the figure for clarity. The biases in $\overline{b}_{1,1}$ are $(-1, -2, \ldots, -m)$ for the first m neurons on the first hidden layer. Those in $\overline{b}_{1,2}$ are $(2, 3, \ldots, m + 1)$. The weights in $W_{1,4}$, $W_{1,5}$, $W_{1,6}$ and the biases in $\overline{b}_{1,3}$ and $\overline{b}_{1,4}$ are defined in the same way. We omit the details due to the page limitation.

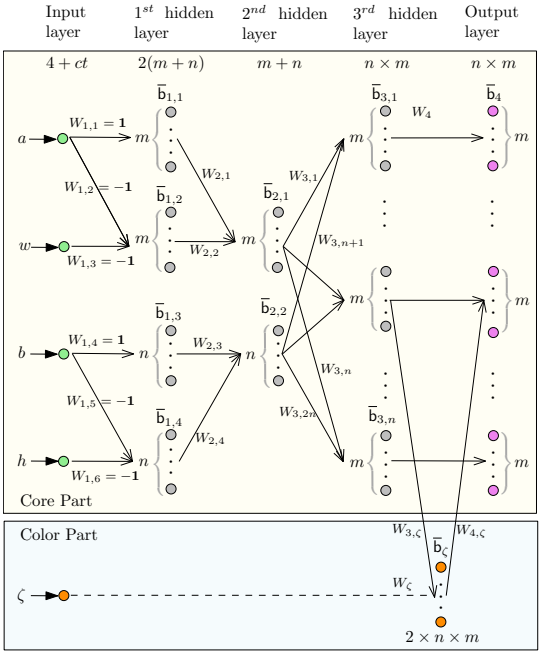


Fig. 5: An occlusion neural network for the occlusions on an image x with ζ and $w \times h$.

For the second layer, the diagonals of weight matrices $W_{2,1}$ to $W_{2,4}$ are set to -1, and the rest of their entries are 0. The biases in $\overline{b}_{2,1}$ and $\overline{b}_{2,2}$ are 1. After the propagation to the second hidden layer, a pixel at position (i, j) in the image x is occluded if and only if both the outputs of the i^{th} neuron in the first m neurons and the j^{th} neuron in the remaining n neurons on the second hidden layer are 1.

The third hidden layer represents the occlusion status of each pixel in the original image x. $2n$ weight matrices connect the second layer and the $n \times m$ neurons of the third layer. For example, we consider the weights in $W_{3,i}$ and $W_{3,n+i}$ which connect the i^{th} group of m neurons in the third layer to the second layer. The size of $W_{3,i}$ is $m \times m$, and the weights in the i^{th} row are 1 while the rest is 0. The size of $W_{3,n+i}$ is $m \times n$. The weights on its diagonal are set to 1, while the rest are set to 0. All the biases in $\overline{b}_{3,1}$ to $\overline{b}_{3,n}$ are -1. The output of the third layer indicates the occlusion status of all the pixels. If a pixel at (i, j) is occluded, then the output of the $(i \times m + j)^{th}$ neuron in the third layer is 1, and otherwise, 0.

(2) Encoding Coloring Functions. We consider the uniform and multiform coloring functions separately for verification efficiency, although the former is a special case of the latter. We first consider the general multiform case. In the multiform case, we introduce $2 \times n \times m$ extra neurons in the third hidden layer, as shown in the bottom part of Fig. 5. These neurons can be combined with the third layer, but it would be more clear to separate them. The weight matrix $W_{3,\zeta}$ connects the third layer to these neurons, with its first half of diagonal set to 1, and the second half set to -1. This helps retain the sign of the input ζ during propagation. The weight matrix W_ζ connects the input ζ to these neurons, whose diagonal are 1, and the biases $\overline{\mathsf{b}}_\zeta$ are -1. These neurons work just like the third layer, except that they not only represent the occlusion status of pixels, but also preserve the input ζ. If a pixel at (i, j) is occluded and ζ has a positive value, then the $(i \times m + j)^{th}$ output in the first half of them is ζ. The $(i \times m + j)^{th}$ output in the second half is ζ when ζ has a negative value. Otherwise, the output is 0. In the uniform case, it can be encoded together with input images, and we thus explain it in the following paragraph.

(3) Encoding Input Images. In the fourth layer, we use W_4 to denote the weight matrix connecting the third layer. W_4 is used to encode pixel values of the input image x and the coloring function ζ of occlusions. In the uniform case, the weight $\mathsf{w}(i, i)$ in the diagonal of W_4 is $\mathsf{w}(i, i) = \zeta_i - x_i$ and the biases $\overline{\mathsf{b}}_4 = \mathbf{x}$ where $\mathbf{x}$ is the flattened vector of the original input image. In the multiform case, the weight matrix $W_{4,\zeta}$ connects the neurons in the bottom part that preserves information of input ζ to the fourth layer. The first half of $W_{4,\zeta}$ is identical to W_4, and the second half of $W_{4,\zeta}$ has its diagonal set to -1. It provides the value of the coloring function ζ with any sign for each occluded pixel. The output of the j^{th} neuron in the i^{th} group of the fourth layer is the raw pixel value plus ζ if the pixel at (i, j) is occluded; otherwise, it is the raw pixel value of p.

An Illustrative Example. We show an example of constructing the occlusion network on a 2×2, single-channel image in Fig. 6. In this example, we assume that the input image is $x = [0.4, 0.6, 0.55, 0.72]$ and the occlusion applied to x has a size of 1×1, which means $w = 1$ and $h = 1$. For uniform occlusion, the coloring function ζ has a fixed value of 0, and for multiform case, the threshold ϵ that a pixel can be altered is 0.1.

We suppose the occlusion is applied at position $(1, 2)$, which means $a = 1$ and $b = 2$ for the input of occlusion network. In the forward propagation, we calculate the output of the first layer by $a \times W_{1,1} + \overline{\mathsf{b}}_{1,1}$ and $a \times W_{1,2} + b \times W_{1,3} + \overline{\mathsf{b}}_{1,2}$ and can get $(0, 0, 0, 1)$ for the first four neurons. Following the same process, we get the output of the second 4 neurons, $(1, 0, 0, 0)$. After propagation to the second layer, it outputs $(1, 0), (0, 1)$ based on $W_{2,1}$, $W_{2,2}$ and $\overline{\mathsf{b}}_2$, representing the second column and the first

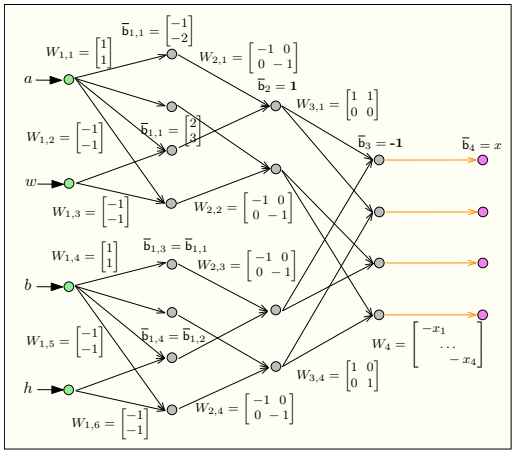


Fig. 6: An example of encoding a one-pixel uniform occlusion as a neural network.

row of x is under occlusion. Likely, the third layer outputs $(0, 1, 0, 0)$ based on its weight matrices and biases, representing that the second pixel in the first row is occluded. After propagation to the fourth layer, the occlusion network outputs an occluded image $x' = [0.4, 0, 0.55, 0.72]$ based on W_4 and $\overline{\mathsf{b}}_4$. It is identical to the expected occluded image, where the second pixel is occluded, and other pixels stay unchanged. Suppose we change a to some real number, for instance, 1.5. After the same propagation, we will get an output of $(0, 0.5, 0, 0.5)$ in the third layer, representing that the neurons in the second column are affected by the occlusion by a factor of 0.5. The fourth layer then outputs $[0.4, 0.3, 0.55, 0.36]$, which is the corresponding occluded image x'.

In the multiform case, as mentioned at the first, we suppose the threshold $\epsilon = 0.1$, and keep all other settings. Then after the same propagation to the third layer, the third layer would output $(0, 1, 0, 0)$, representing that the second pixel is occluded. Those extra neurons then output $(0, 0.1, 0, 0, 0, 0, 0, 0)$ where the second neuron in the first half is 0.1 and 0 for the remaining. This indicates both that the second pixel in the first row is occluded, and has an epsilon of 0.1. After propagation to the fourth layer, the occlusion network outputs $x' = [0.4, 0.7, 0.55, 0.72]$ based on its W_4 and $\overline{\mathsf{b}}_4$. As expected, the second pixel is occluded and increases by 0.1, and other pixels stay unchanged. For the case of a negative ϵ of -0.1, the extra neurons output $(0, 0, 0, 0, 0, 0.1, 0, 0)$. Note that the second neuron in the second half is 0.1 and the remaining are 0, which helps retain the sign of -0.1. The fourth layer then outputs $[0.4, 0.5, 0.55, 0.72]$, which is the expected occluded image where the second pixel decreases by 0.1.

4.3 The Correctness of the Encoding

Given an input image x, a rectangle occlusion of size $w \times h$, and a coloring function ζ, let O be the corresponding occlusion neural network constructed in the approach above. Let F be the FNN to verify. We concatenate O to F by connecting O's output layer to F's input layer. The combined network implements the composed function $F \circ O$. The problem of verifying the occlusion robustness of F on the input image x is reduced to a regular robustness verification problem of $F \circ O$.

Theorem 1 (Correctness). *An FNN F is robust on the input image x with respect to a rectangle occlusion in the size of $w \times h$ and a coloring function ζ if and only if $\Phi_{F \circ O}((a, w, b, h, \zeta)) = \Phi_F(x)$ for all $1 \leq a \leq n$ and $1 \leq b \leq m$.*

Theorem 1 means that all the occluded images from x are classified by F to the same label as x, which implies the correctness of our proposed encoding approach. To prove Theorem 1, it suffices to show that the encoded occlusion neural network represents all the possible occluded images. In other words, when being perceived as a function, the network outputs the same occluded image as the occlusion function for the same occlusion coordinate (a, b), as formalized in the following lemma.

Lemma 1. *Given an occlusion function* $\gamma_{\zeta,w\times h} : \mathbb{R}^{m\times n} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{m\times n}$ *and an input image* x*, let* $O_{\gamma,x} : \mathbb{R}^{4+ct} \to \mathbb{R}^{m\times n}$ *be the corresponding occlusion neural network. There is* $\gamma_{\zeta,w\times h}(x, a, b) = O_{\gamma,x}(a, w, b, h, \zeta)$ *for all* $1 \le a \le n$ *and* $1 \le b \le m$*.*

Proof (Sketch). It suffices to prove $\gamma_{\zeta,w\times h}(x, a, b)_{i,j} = O_{\gamma,x}(a, w, b, h, \zeta)_{i,j}$ for all $i \in \mathbb{N}_{1,n}$ and $j \in \mathbb{N}_{1,m}$. By Definition 2, we consider the following two cases:

Case 1: When a pixel p *at position* (i, j) *is fully occluded, we have* $\gamma_{\zeta,w\times h}(x, a, b)_{i,j} = \zeta(x, i, j)$*. We need to prove that* $O_{\gamma,x}(a, w, b, h, \zeta)_{i,j} = \zeta(x, i, j)$*.*

Suppose p is covered by an arbitrary uniform occlusion with size of $w_0 \times h_0$ at position (a_0, b_0). We can observe that for that pixel p, $i > a_0 \wedge i < a_0 + w_0 - 1$ and $j > b_0 \wedge j < b_0 + h_0 - 1$ hold since p is covered by the occlusion.

We show the output of $O_{\gamma,x}(a, w, b, h, \zeta)_{i,j}$ by inspecting the $(i * n + j)^{th}$ output of the occlusion network after propagation, starting from inspecting the output of the i^{th} and $(i + m)^{th}$ neurons of the first layer. According to the network structure discussed in Sec. 4.2, we can tell that the i^{th} neuron in the first layer is 0 only when $i > a_0$, the same property holds for the $(i + m)^{th}$ neuron when $i < a_0 + w_0 - 1$. Therefore, the output for the i^{th} and $(i + m)^{th}$ neurons of the first layer is 0, which leads to the i^{th} neuron in the first part of the second layer has output of value 1. Through the similar process, we can get that the value of $z_j^{(2)}$ in the second part of the second layer is also 1.

The $(i \times n + j)^{th}$ neuron in the third layer is based on the i^{th} neuron and j^{th} neuron of the second layer that we just discussed. Therefore, the output of that neuron, $z_{i\times n+j}^{(3)}$, is 1. For uniform occlusion, suppose the coloring function ζ has a fixed value μ_0. By propagating the output $z_{i\times n+j}^{(3)}$ to the fourth layer, which is calculated as $W_4 \times z^{(3)} + \overline{b}_4$, the $(i \times n + j)^{th}$ output of the fourth layer is $1 \times (\mu_0 - p_{i,j}) + p_{i,j} = \mu_0$. Likely, for multiform occlusion, ζ indicates the threshold ϵ_0 that a pixel can change. The $(i \times n + j)^{th}$ extra neuron outputs ϵ_0 , then the corresponding neuron in the fourth layer outputs $p_{i,j} + \epsilon_0$.

This output of $O_{\gamma,x}(a, w, b, h, \zeta)_{i,j}$ is identical to $\gamma_{\zeta,w\times h}(x, a, b)_{i,j}$, the expected pixel value at position (i, j), which also indicates that the color is correctly encoded.

Case 2: When a pixel p *at position* (i, j) *is not occluded, we have* $\gamma_{\zeta,w\times h}(x, a, b)_{i,j} = x_{i,j}$*. Then, we need to prove that* $O_{\gamma,x}(a, w, b, h, \zeta)_{i,j} = x_{i,j}$*.*

In this case, we can observe that $i < a_0 \vee i \ge a_0 + w_0$ and $j < b_0 \vee j \ge b_0 + h_0$ hold for pixel p. Then We can tell that the corresponding neuron in the third layer outputs 0 and the output of the $(i * n + j)^{th}$ neuron in the fourth layer is the origin pixel value of p following the similar process discussed in case 1.

For the occlusion with real number position, some more cases need to be discussed, but the proof has a very similar sketch as the normal occlusion with integer position. We leverage the equality of $a \times b = exp(log(a) + log(b))$ and add it to the propagation between the third layer and those extra neurons only when the occlusion is at real number positions in the multiform case. And we use $ReLU(a + b - 1)$ as an alternative to logarithms and exponents in implementation since Marabou does not support such operations. Due to the page limit, please refer to [15] for the details of the full proof.

Theorem 1 can be directly derived from Lemma 1 and Definition 3 by substituting $\gamma_{\zeta,w\times h}(x, a, b)$ for $O_{\gamma,x}(a, w, b, h, \zeta)$ in the definition.

4.4 Verification Acceleration Techniques

Existing SMT-based neural network verification tools can directly verify the composed neural network. The number of ReLU activation functions in the network is the primary factor in determining the verification time cost by the backend tools. In the occlusion part, the number of ReLU nodes is independent of the scale of the original networks to be verified. Therefore, our approach's scalability relies only on the underlying tools.

To further improve the verification efficiency, we integrate two algorithmic acceleration techniques by dividing the verification problem into small independent sub-problems that can be solved separately.

Occlusion Space Splitting. We observed that verifying the composed neural network with a large input space can significantly degrade the efficiency of backend verifiers. Even for small FNNs with only tens of ReLUs, the verifiers may run out of time due to the large occlusion space for searching. For instance, the complexity of Reluplex [20] can be derived from the original SMT method of Simplex [32]. It has a complexity of $\Omega(v \times m \times n)$, where m and n represent the number of constraints and variables, and v represents the number of pivots operated in the Simplex method. In the worst case, v can grow exponentially. Reduction in the search space can reduce the number of pivot operations, therefore significantly improving verification efficiency.

Based on the above observation, we can divide $[1, m]$ (*resp.* $[1, n]$) into $k_m \in \mathbb{Z}^+$ (*resp.* $k_n \in \mathbb{Z}^+$) intervals $[m_0, m_1], \ldots, [m_{k_m-1}, m_{k_m}]$ (*resp.* $[n_0, n_1], \ldots, [n_{k_n-1}, n_{k_n}]$) and verify the problem on the Cartesian product of the two sets of intervals.

$$\begin{aligned} &\forall x' \in \mathbb{X}.\Phi(x') = \Phi(x) \equiv \textstyle\bigwedge_{(i,j)=(0,0)}^{(k_m-1,k_n-1)} \forall x' \in \mathbb{X}_{(i,j)}.\Phi(x') = \Phi(x), \text{ where} \\ &\mathbb{X} = \textstyle\bigcup_{(i,j)=(0,0)}^{(k_m-1,k_n-1)} \mathbb{X}_{(i,j)} = \bigcup_{(i,j)=(0,0)}^{(k_m-1,k_n-1)} \{\gamma_{\zeta,w\times h}(x,a,b) | m_i \le a \le m_{i+1}, n_j \le b \le n_{j+1}\}. \end{aligned} \tag{7}$$

In this way, we split the occlusion space into $k_m \times k_n$ sub-spaces. It is equivalent to prove $\forall x' \in \mathbb{X}.\Phi(x')$ for all $\mathbb{X}_{(i,j)}$ with $0 \le i < k_m$ and $0 \le j < k_n$, without losing the soundness and completeness. We call each verification instance a *query*, which can be solved more efficiently than the one on the whole occlusion space by backend verifiers. Furthermore, another advantage of occlusion space splitting is that these divided queries can be solved in parallel by leveraging multi-threaded computing.

Eager Falsification by Label Sorting. Another *Divide & Conquer* approach for acceleration is to divide the verification problem into independent sub-problems by the classification labels in L, as defined below:

$$\forall x' \in \mathbb{X}.\Phi(x') = \Phi(x) \equiv \forall x' \in \mathbb{X}. \textstyle\bigwedge_{\ell' \in L} \Phi(x) = \ell' \vee \Phi(x') \neq \ell'. \tag{8}$$

The dual problem to disprove the robustness can be solved to find some label ℓ' such that $\Phi(x) \neq \ell' \wedge \Phi(x') = \ell'$. We can first solve those that have higher probabilities of being non-robust. Once a sub-problem is proved non-robust, the verification terminates, with no need to solve the remainder. Such approach is called *eager falsification* [14]. Based on this methodology, we sort the sub-problems in a descent order according to the probabilities at which the original image is classified to the corresponding labels by the neural network. A higher probability implies that the image is more likely to be classified to the corresponding label. Heuristically, there is a higher probability of finding

Table 1: Occlusion verification results on two medium FNNs trained on MNIST and GTSRB in different occlusion sizes 2×2 and 5×5 and occlusion radius ϵ.

		Medium FNN (600 ReLUs) on MNIST					Medium FNN (343 ReLUs) on GTSRB				
Size	ϵ	- / +	T_+	T_-	T_{build}	TO(%)	- / +	T_+	T_-	T_{build}	TO(%)
2×2	0.05	**2** / 28	120.01	11.98	0.068	0.00	**8** / 13	103.64	24.18	0.089	0.00
	0.10	**3** / 27	121.37	19.18	0.067	0.00	**8** / 13	108.62	22.57	0.088	0.00
	0.20	**4** / 26	122.12	39.57	0.067	0.00	**10** / 11	113.7	23.17	0.084	0.00
	0.30	**6** / 24	165.98	45.6	0.086	0.00	**11** / 10	117.97	26.41	0.089	0.00
	0.40	**7** / 23	183.65	47.32	0.098	4.75	**14** / 7	115.49	31.53	0.096	0.14
5×5	0.05	**5** / 25	123.45	49.04	0.065	0.00	**9** / 12	123.99	26.02	0.101	0.00
	0.10	**6** / 24	124.13	44.09	0.073	0.00	**12** / 9	127.65	26.96	0.01	0.00
	0.20	**10** / 20	179.89	52.51	0.073	3.26	**16** / 5	126.98	27.22	0.102	0.00
	0.30	**14** / 16	284.67	65.98	0.076	5.45	**18** / 3	146.68	29.11	0.100	0.04
	0.40	**22** / 8	339.78	97.28	0.074	7.33	**19** / 2	169.17	26.52	0.103	0.09

* - / +: the numbers of non-robust and robust cases; T_+ (*resp.* T_-): average verification time in robust (*resp.* non-robust) cases; T_{build}: the building time of occlusion neural networks; TO (%): the percentage of runtime-out cases among all the queries.

an occlusion such that the occluded image is misclassified to that label. We sequence the queries into backend verifiers until all are verified, or a non-robust case is reported. Our experimental results will show that this approach can achieve up to 8 and 24 times speedup in the robust and non-robust cases, respectively.

5 Implementation and Evaluation

We implemented our approach in a Python tool called OccRob, using the PyTorch framework. As a backend tool, we chose the Marabou [21] state-of-the-art, SMT-based DNN verifier. We evaluated our proposed approach extensively on a suite of benchmark datasets, including MNIST [24] and GTSRB [16]. The size of the networks trained on the datasets for verification is measured by the number of ReLUs, ranging from 70 to 1300. All the experiments are conducted on a workstation equipped with a 32-core AMD Ryzen Threadripper CPU @ 3.7GHz and 128 GB RAM and Ubuntu 18.04. We set a timeout threshold of 60 seconds for a single verification task. All code and experimental data, including the models and verification scripts can be accessed at https://github.com/MakiseGuo/OccRob.

We evaluate our proposed method concerning efficiency and scalability in the occlusion robustness verification of ReLU-based FNNs. Our goals are threefold:

1. To demonstrate the effectiveness of the proposed approach for the robustness verification against various types of occlusion perturbations.
2. To evaluate the efficiency improvement of the proposed approach, compared with the naive SMT-based method.
3. To demonstrate the effectiveness of the acceleration techniques in efficiency improvement.

Experiment I: Effectiveness. We first evaluate the effectiveness of OccRob in robustness verification against various types of occlusions of different sizes and color ranges. Table 1

Fig. 7: Occlusive adversarial examples automatically generated for non-robust images.

shows the verification results and time costs against multiform occlusions on two medium FNNs trained on MNIST and GTSRB. We consider two occlusion sizes, 2×2 and 5×5, respectively. The occluding color range is from 0.05 to 0.40. In each verification task, we selected the first 30 images from each of the two datasets and verified the network's robustness around them, under corresponding occlusion settings. As expected, larger occlusion sizes and occluding color ranges imply more non-robust cases. One can see that OccRob can almost always verify and falsify each input image, except for a few time-outs. The robust cases cost more time than the non-robust cases, but all can be finished in a few minutes. Note that the time overhead for building occlusion neural networks is almost negligible, compared with the verification time. The effectiveness against uniform occlusions is shown in the following experiment.

Fig. 7 shows several occlusive adversarial examples that are generated by OccRob under different occlusion settings. These occlusions do not alter the semantics of the original images and should be classified to the same results as those non-occluded ones. However, they are misclassified to other results.

Experiment II: Efficiency improvement over the naive encoding method. We compare the efficiency of OccRob with that of a naive SMT encoding approach on verifying uniform occlusions since the naive encoding approach cannot handle verification against multiform occlusions. We apply the same acceleration techniques, such as parallelization and a variant of input space splitting, to the naive approach, which otherwise times out for almost all verification tasks even on the smallest model.

Table 2 shows the average verification time on six FNNs of different sizes against uniform occlusions. We can observe that OccRob affords a significant improvement in efficiency, up to 30 times higher than the naive approach. It can always finish before the preset time threshold, while the naive method fails to verify the two large networks

under the same time threshold. The timeout proportion of two medium networks is over 70%. While the small network on MNIST only has an 8% of timeout proportion with the naive method, OccRob barely timeouts on every network.

Table 2: Performance comparison between OccRob (OR) and the naive (NAI) methods on MNIST and GTSRB under different occlusion sizes.

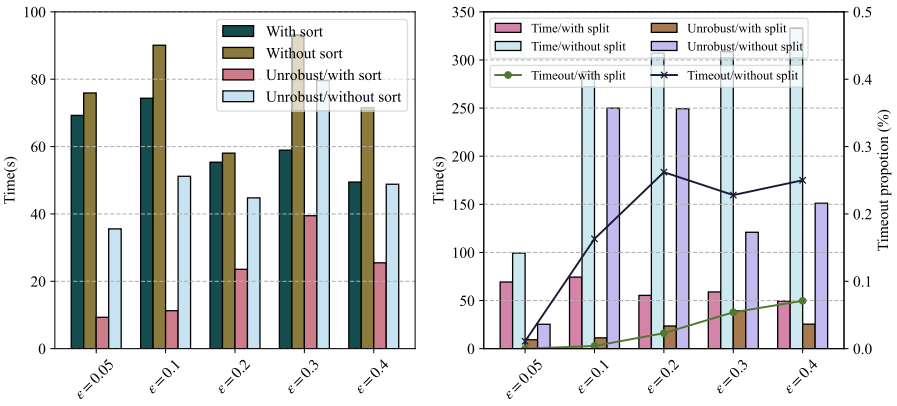
	MNIST						GTSRB					
FNNs	Small FNN		Medium FNN		Large FNN		Small FNN		Medium FNN		Large FNN	
Size	OR	NAI	OR	NAI	OR	NAI	OR	NAI	OR	NAI	OR	NAI
1×1	46.44	63.12	110.18	759.93	206.50	TO	29.76	472.23	69.28	989.08	173.62	TO
2×2	49.62	165.53	98.60	832.98	199.17	TO	21.04	340.89	42.16	680.81	103.42	TO
3×3	51.23	298.59	111.14	863.74	205.67	TO	11.93	169.35	32.00	499.31	81.17	TO
4×4	44.78	256.22	115.99	886.73	225.02	TO	8.90	141.85	31.24	419.62	106.41	TO
5×5	48.96	270.23	113.01	803.40	264.79	TO	6.11	190.81	27.97	418.56	118.99	TO
6×6	47.81	318.28	127.98	642.01	288.18	TO	7.49	213.35	21.70	282.04	60.02	TO
7×7	34.99	357.78	124.47	589.41	222.65	TO	6.02	153.81	31.96	404.18	62.60	TO
8×8	36.05	324.34	129.27	469.24	215.53	TO	5.99	123.07	28.44	250.97	54.37	TO
9×9	34.58	224.01	141.54	375.97	219.61	TO	6.42	102.39	31.30	160.84	59.87	TO
10×10	28.98	178.44	78.89	398.01	182.36	TO	6.61	127.20	28.59	153.96	40.69	TO

Experiment III: Effectiveness of the integrated acceleration techniques. We finally evaluate the effectiveness of the two acceleration techniques integrated with the tool. We evaluate each technique separately by excluding it from OccRob and comparing the verification time of OccRob and the corresponding excluded versions. Fig. 8 shows the experimental results of verifying the medium FNN trained on GTSRB against multiform occlusions by the tools. Fig. 8 (a) shows that label sorting can improve efficiency in both robust and non-robust cases. In particular, the improvement is more significant in the non-robust case, with up to 5 times speedup in the experiment. That is because solving each query is faster than solving all simultaneously, and further OccRob immediately stops dispatching queries once a counterexample is found in the non-robust case. Fig. 8 (b) shows that occlusion space splitting can also significantly improve the efficiency, with up to 8 and 24 times speedups in the robust and non-robust cases, respectively. In addition, Fig. 8 (b) also shows a significant reduction in the number of time-outs.

6 Related Work

Robustness verification of neural networks has been extensively studied recently, aiming at devising efficient methods for verifying neural networks' robustness against various types of perturbations and adversarial attacks. We classify those methods into two categories according to the type of perturbations, which can be semantic or non-semantic. Semantic perturbation has an interpretable meaning, such as occlusions and geometric transformations like rotation, while non-semantic perturbation means that noises perturb inputs with no particular meanings.

Non-semantic perturbations are usually represented as L_p norms, which define the ranges in which an input can be altered. Some robustness verification approaches for



(a) Comparison of label sorting (b) Comparison of input splitting

Fig. 8: Efficiency evaluation results of the two acceleration techniques.

non-semantic perturbations are both sound and complete by leveraging SMT [20,1] and MILP (mixed integer linear programming) [36] techniques, while some sacrifice the completeness for better scalability by over-approximation [29,2,7], abstract interpretation [34,10,5], interval analysis by symbolic propagation [43,42,26], etc.

In contrast to a large number of works on non-semantic robustness verification, there are only a few studies on the semantic case. Because semantic perturbations are beyond the range of L_p norms [9], those abstraction-based approaches cannot be directly applied to verifying semantic perturbations. Mohapatra et al. [30] proposed to verify neural networks against semantic perturbations by encoding them into neural networks. Their encoding approach is general to a family of semantic perturbations such as brightness and contrast changes and rotations. Their approach for verifying occlusions is restricted to uniform occlusions at integer locations. Sallami et al.[31] proposed an interval-based method to verify the robustness against the occlusion perturbation problem under the same restriction. Singh et al. [35] proposed a new abstract domain to encode both non-semantic and semantic perturbations such as rotations. Chiang et al. [4] called occlusions *adversarial patches* and proposed a certifiable defense by extending interval bound propagation (IBP) [12]. Compared with these existing verification approaches for semantic perturbations, our SMT-based approach is both sound and complete, and meanwhile, it supports a larger class of occlusion perturbations.

7 Conclusion and Future Work

We introduced an SMT-based approach for verifying the robustness of deep neural networks against various types of occlusions. An efficient encoding method was proposed to represent occlusions using neural networks, by which we reduced the occlusion robustness verification problem to a regular robustness verification problem of neural networks and leveraged *off-the-shelf* SMT-based verifiers for the verification. We implemented a resulting prototype OccRob and intensively evaluated its effectiveness and efficiency on a series of neural networks trained on the public benchmarks, including MNIST and GTSRB. Moreover, as the scalability of DNN verification engines continues to improve, our approach, which uses them as blackbox backends, will also become more scalable.

As our occlusion encoding approach is independent of target neural networks, we believe it can be easily extended to other complex network structures, such as convolutional and recurrent ones, which only depend on the backend verifiers. It would also be interesting to investigate how the generated adversarial examples could be used for neural network repairing [41,18] to train more robust networks.

Acknowledgments

This work has been supported by National Key Research Program (2020AAA0107800), NSFC-ISF Joint Program (62161146001, 3420/21) and NSFC projects (61872146, 61872144), Shanghai Science and Technology Commission (20DZ1100300), Shanghai Trusted Industry Internet Software Collaborative Innovation Center and “Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (Grant No. 22510750100).

References

1. Amir, G., Wu, H., Barrett, C., Katz, G.: An smt-based approach for verifying binarized neural networks. In: TACAS'21. pp. 203–222. Springer (2021)
2. Boopathy, A., Weng, T.W., Chen, P.Y., Liu, S., Daniel, L.: Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In: AAAI'19. vol. 33, pp. 3240–3247 (2019)
3. Brown, T.B., Mané, D., Roy, A., Abadi, M., Gilmer, J.: Adversarial patch. arXiv preprint arXiv:1712.09665 (2017)
4. Chiang, P.y., Ni, R., Abdelkader, A., Zhu, C., Studer, C., Goldstein, T.: Certified defenses for adversarial patches. arXiv preprint arXiv:2003.06693 (2020)
5. Cohen, J., Rosenfeld, E., Kolter, Z.: Certified adversarial robustness via randomized smoothing. In: ICML'19. pp. 1310–1320. PMLR (2019)
6. Coşkun, M., Uçar, A., Yildirim, Ö., Demir, Y.: Face recognition based on convolutional neural network. In: MEES'17. pp. 376–379. IEEE (2017)
7. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: CAV'20. pp. 43–65. Springer (2020)
8. Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., Prakash, A., Kohno, T., Song, D.: Robust physical-world attacks on deep learning visual classification. In: CVPR'18. pp. 1625–1634 (2018)
9. Fischer, M., Baader, M., Vechev, M.: Certified defense to image transformations via randomized smoothing. NeurIPS'20 **33**, 8404–8417 (2020)
10. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: Safety and robustness certification of neural networks with abstract interpretation. In: S&P'18. pp. 3–18. IEEE (2018)
11. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning, pp. 168–196. MIT Press (2016), http://www.deeplearningbook.org
12. Gowal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Arandjelovic, R., Mann, T., Kohli, P.: On the effectiveness of interval bound propagation for training verifiably robust models. arXiv preprint arXiv:1810.12715 (2018)
13. Gowal, S., Dvijotham, K.D., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Arandjelovic, R., Mann, T., Kohli, P.: Scalable verified training for provably robust image classification. In: ICCV'19. pp. 4842–4851 (2019)

14. Guo, X., Wan, W., Zhang, Z., Zhang, M., Song, F., Wen, X.: Eager falsification for accelerating robustness verification of deep neural networks. In: ISSRE'21. pp. 345–356. IEEE (2021)
15. Guo, X., Zhou, Z., Zhang, Y., Katz, G., Zhang, M.: OccRob: Efficient smt-based occlusion robustness verification of deep neural networks. arXiv preprint (2023)
16. Houben, S., Stallkamp, J., Salmen, J., Schlipsing, M., Igel, C.: Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In: IJCNN'13 (2013)
17. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: CAV'17. pp. 3–29. Springer (2017)
18. Islam, M.J., Pan, R., Nguyen, G., Rajan, H.: Repairing deep neural networks: Fix patterns and challenges. In: ICSE'20. pp. 1135–1146. IEEE (2020)
19. Jaderberg, M., Simonyan, K., Zisserman, A., et al.: Spatial transformer networks. In: Advances in neural information processing systems. pp. 2017–2025. PMLR (2015)
20. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: CAV'17. pp. 97–117. Springer (2017)
21. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al.: The Marabou framework for verification and analysis of deep neural networks. In: CAV'19. pp. 443–452. Springer (2019)
22. Kirkland, E.J.: Bilinear Interpolation, pp. 261–263. Springer US, Boston, MA (2010)
23. Kortylewski, A., Liu, Q., Wang, A., Sun, Y., Yuille, A.: Compositional convolutional neural networks: A robust and interpretable model for object recognition under occlusion. International Journal of Computer Vision **129**(3), 736–760 (2021)
24. LeCun, Y., Cortes, C.: MNIST handwritten digit database (2010), http://yann.lecun.com/exdb/mnist/
25. Lengyel, H., Remeli, V., Szalay, Z.: Easily deployed stickers could disrupt traffic sign recognition. Perner's Contacts **19**(Special Issue 2), 156–163 (2019)
26. Li, J., Liu, J., Yang, P., Chen, L., Huang, X., Zhang, L.: Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In: SAS'19. pp. 296–319. Springer (2019)
27. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971 (2015)
28. Lyu, Z., Guo, M., Wu, T., Xu, G., Zhang, K., Lin, D.: Towards evaluating and training verifiably robust neural networks. In: CVPR'21. pp. 4308–4317 (2021)
29. Lyu, Z., Ko, C.Y., Kong, Z., Wong, N., Lin, D., Daniel, L.: Fastened crown: Tightened neural network robustness certificates. In: AAAI'20. vol. 34, pp. 5037–5044 (2020)
30. Mohapatra, J., Weng, T.W., Chen, P.Y., Liu, S., Daniel, L.: Towards verifying robustness of neural networks against a family of semantic perturbations. In: ICCV'20. pp. 244–252 (2020)
31. Mziou Sallami, M., Ibn Khedher, M., Trabelsi, A., Kerboua-Benlarbi, S., Bettebghor, D.: Safety and robustness of deep neural networks object recognition under generic attacks. In: ICONIP'19. pp. 274–286. Springer (2019)
32. Nelder, J.A., Mead, R.: A simplex method for function minimization. The computer journal **7**(4), 308–313 (1965)
33. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: SOSP'17. pp. 1–18 (2017)
34. Raghunathan, A., Steinhardt, J., Liang, P.: Certified defenses against adversarial examples. arXiv preprint arXiv:1801.09344 (2018)
35. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
36. Singh, G., Gehr, T., Püschel, M., Vechev, M.: Robustness certification with refinement. In: ICLR'19 (2019)
37. Song, L., Gong, D., Li, Z., Liu, C., Liu, W.: Occlusion robust face recognition based on mask learning with pairwise differential siamese network. In: ICCV'19. pp. 773–782 (2019)

38. Su, J., Vargas, D.V., Sakurai, K.: One pixel attack for fooling deep neural networks. IEEE Transactions on Evolutionary Computation **23**(5), 828–841 (2019)
39. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
40. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: ICSE'18. pp. 303–314 (2018)
41. Usman, M., Gopinath, D., Sun, Y., Noller, Y., Păsăreanu, C.S.: Nn repair: Constraint-based repair of neural network classifiers. In: CAV'21. pp. 3–25. Springer (2021)
42. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: NeurIPS'18. vol. 31. Curran Associates, Inc. (2018)
43. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: USENIX Security'18. pp. 1599–1614 (2018)
44. Zhu, H., Tang, P., Park, J., Park, S., Yuille, A.: Robustness of object recognition under extreme occlusion in humans and computational models. arXiv preprint arXiv:1905.04598 (2019)

Neural Network-Guided Synthesis of Recursive List Functions

Naoki Kobayashi(✉) and Minchao Wu

The University of Tokyo, Tokyo, Japan
koba@is.s.u-tokyo.ac.jp

Abstract. Kobayashi et al. have recently proposed NeuGuS, a framework of neural-network-guided synthesis of logical formulas or simple program fragments, where a neural network is first trained based on sample data, and then a logical formula over integers is constructed by using the weights and biases of the trained network as hints. The previous method was, however, restricted the class of formulas of quantifier-free linear integer arithmetic. In this paper, we propose a NeuGuS method for the synthesis of recursive predicates over lists definable by using the left fold function. To this end, we design and train a special-purpose recurrent neural network (RNN), and use the weights of the trained RNN to synthesize a recursive predicate. We have implemented the proposed method and conducted preliminary experiments to confirm the effectiveness of the method.

1 Introduction

Kobayashi et al. [12] have recently proposed a framework called *Neural-Network-Guided Synthesis (*NeuGuS*)* for the synthesis of quantifier-free logical expressions over integer variables, which may also be viewed as simple program expressions over integer variables. Given sample data (also called *training data* below, which consist of positive/negative samples and implication constraints [6] such as "if d_1 is a positive sample, so is d_2, but it is unknown whether d_1 is indeed a positive sample), NeuGuS first trains a feed-forward neural network with respect to the sample data, and then constructs a logical expression on integers (more precisely, a Boolean combination of inequalities on integer variables) by using the weights and biases of the neural network as hints. The main characteristic of NeuGuS is its *gray*-box use of neural networks. NeuGuS first trains a neural network, but instead of directly using the trained network as a classifier, it tries to construct a simple logical expression by using the trained network as a hint. Advantages of the gray-box approach over the white-box approach of using the network itself as a classifier include: (i) if successful, a simple classifier is obtained that is easier to understand (for human beings) and verify (for computers), and (ii) we need not worry too much about overfitting; even if the trained network is overfit to the given sample data, we may still be able to extract useful information such as features important for the classification, and use them to construct a simple classifier. Kobayashi et al. [12,13] have applied the proposed framework to automated program verification, where NeuGuS is

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 227–245, 2023.
https://doi.org/10.1007/978-3-031-30823-9_12

used to find program invariants, and also to program synthesis where, given a program sketch containing holes called *oracles*, NeuGuS is used to find program expressions to fill the holes.

In this paper, we extend NeuGuS to enable the synthesis of *recursive* predicates over Booleans, integers, and lists of Booleans, and lists of integers from positive/negative samples and implication constraints. For example, in the case of the synthesis of a sortedness predicate, the extended NeuGuS (henceforth, simply called NeuGuSR) takes as inputs sample data like:

$$\texttt{sorted}([1;3;4]) \quad \texttt{sorted}([2;5;6;7]) \quad \neg\texttt{sorted}([3;1;4]) \quad \neg\texttt{sorted}([5;2;7;6])$$
$$\texttt{sorted}([1;3;5]) \Rightarrow \texttt{sorted}([1;3;5;6]) \quad \cdots .$$

Here, $\texttt{sorted}([1;3;5]) \Rightarrow \texttt{sorted}([1;3;5;6])$ means that if $\texttt{sorted}([1;3;5])$ is true, so is $\texttt{sorted}([1;3;5;6])$. The goal of the synthesis is to construct a recursive program that satisfies the constraints specified by the sample data. In the case of the above example, we aim to construct a program (written in OCaml language: https://ocaml.org/) like:

```
let sorted l =
  let rec sorted_aux l b r =
    match l with [] -> b
                 | x::l' -> sorted_aux l' (b && r <= x) x
  in sorted_aux l true 0
```

Here, the Boolean argument `b` of the auxiliary function `sorted_aux` denotes whether the elements of the list read so far are sorted (in the ascending order), and the integer argument `r` keeps the last element read (which is initially set to 0; hence, the function `sorted` judges the sortedness of a list consisting of non-negative integers), to compare it with the next element. The recursive programs constructed with our method are restricted to those definable by using the left fold function. Note that the function `sorted` above can be expressed as $\mathit{foldl}\ (\lambda(b,r).\lambda x.(b \wedge r \leq x, x))\ (\texttt{true}, 0)$ using the left fold function foldl.[1]

To synthesize recursive predicates, we first train a recurrent neural network (RNN), and construct a recursive program like the one above by using, as hints, the weights of the RNN and information about the executions of the RNN for the training data. We have designed a special-purpose RNN for that purpose, with the synthesis of recursive programs in mind. Figure 1 shows the overall structure of our RNN. The RNN has two kinds of inputs: Boolean lists and integer lists (where their elements are read one by one), and a Boolean output. The inputs and output correspond to those of the function to be synthesized, which takes m Boolean lists and n integer lists as arguments, and returns a Boolean value. Here, we assume that the lists are of equal length, by replicating integer arguments and padding short lists with dummy elements if necessary. For

[1] In fact, the program above is written so that it matches the computation of the left fold function. Otherwise, `sorted_aux` could alternatively be defined so that it returns `false` immediately when $r > x$ holds.

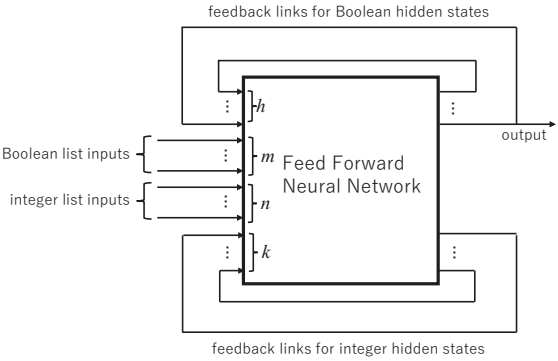


Fig. 1. The overall structure of the special-purpose RNN

example, if the argument of the function to be synthesized is $([1;2;3],0,[1;0])$, then the input for RNN will be $([1;2;3],[0;0;0],[1;0;-1])$. The Boolean values `true` and `false` are respectively represented as 1 and -1. The RNN has also two kinds of hidden states: Booleans and integers. The Boolean hidden states are actually represented as numerical values, but they are constrained to range over $[-1,1]$ by using the hyperbolic tangent function **tanh** as the activation function for those values inside the feed-forward network. The details of the feed-forward network will be discussed later.

After training the RNN, by using (i) the weights and biases of each link/node and (ii) the the input/output behavior of the trained feed-forward network as hints, we construct a function:

$$step : \mathbf{B}^m \times \mathbf{Z}^n \times \mathbf{B}^h \times \mathbf{Z}^k \to \mathbf{B}^h \times \mathbf{Z}^k,$$

which takes the current input (consisting of m Booleans and n integers) and the current values of Boolean and integer hidden states, and returns the next hidden states. Here, $\mathbf{Z}$ and $\mathbf{B}$ are the types of integers and Booleans respectively. We then construct the whole program as the one that "folds" the input lists by using the *step* function, where the base-case values correspond to the initial values of the hidden states; more details are discussed in later sections. Finally, we check whether the synthesized program conforms to the sample data and if so, output the program; otherwise we retrain the RNN and retry the program synthesis.

We have implemented a program synthesis tool based on the above idea. We have confirmed through experiments that the tool worked reasonably well; our tool could successfully synthesize the sortedness predicate above, as well as other non-trivial predicates, including the binary predicate `avge`(ℓ, n), which means that the average value of the elements in the list ℓ is no less than n.

The rest of this paper is structured as follows. Section 2 defines the program synthesis problem considered in this paper. Section 3 introduces our special-purpose RNN. Section 4 explains how to synthesize a program from a trained

RNN. Section 5 reports an implementation and experimental results. Section 6 discusses related work and Section 7 concludes the paper.

2 The Synthesis Problem

This section defines the problem of program synthesis considered in this paper. We write $\mathbf{B}$ and $\mathbf{Z}$ for the sets of Booleans and integers respectively. For a set S, we write S^* for the set of sequences consisting of elements of S, and $S_1 \times \cdots \times S_k$ for the set of tuples of the form $(v_1, \ldots, v_k)$ with $v_i \in S_i$ for each i. We sometimes call an element of S^* a *list*, based on the terminology used in programming languages, and write $[a_1; \cdots; a_n]$ instead of $a_1 \cdots a_n$.

We assume a finite set of variables called *predicate variables*. A *signature* maps each predicate variable to its domain of the form $T_1 \times \cdots \times T_k$, where $T_i \in \{\mathbf{B}, \mathbf{Z}, \mathbf{B}^*, \mathbf{Z}^*\}$. For example, for a signature $\mathcal{K}$ and a predicate variable p, $\mathcal{K}(p) = \mathbf{Z}^* \times \mathbf{Z}$ means that p is a binary predicate that takes an integer list and an integer as arguments.

For a signature $\mathcal{K}$, we write $\mathbf{Atoms}_{\mathcal{K}}$ for the set of pairs (p, v) where $v \in \mathcal{K}(p)$; we often write $p(v)$ for (p, v). An *implication constraint* is a formula of the form $a_1 \wedge \cdots \wedge a_k \Rightarrow b_1 \vee \cdots \vee b_\ell$, where $a_1, \ldots, a_k, b_1, \ldots, b_\ell \in \mathbf{Atoms}_{\mathcal{K}}$. Let Θ be an interpretation for predicate variables, i.e., a map that assigns a predicate $P \subseteq \mathcal{K}(p)$ to each predicate $p \in dom(\mathcal{K})$. We write $\Theta \models p(v)$ if $v \in \Theta(p)$. We write $\Theta \models a_1 \wedge \cdots \wedge a_k \Rightarrow b_1 \vee \cdots \vee b_\ell$ and say that Θ satisfies the implication constraint $a_1 \wedge \cdots \wedge a_k \Rightarrow b_1 \vee \cdots \vee b_\ell$, when $\Theta \models b_j$ for some $j \in \{1, \ldots, \ell\}$ if $\Theta \models a_i$ for every $i \in \{1, \ldots, k\}$.

The *synthesis problem* considered in this paper is the problem of, given a signature $\mathcal{K}$ and a set of implication constraints as input, finding (a description of) a predicate assignment Θ that satisfies all the implication constraints. As a description of the predicate assigned to each predicate variable, we consider the class of functions f defined by programs of the following form:

$$
\begin{array}{l}
\textbf{let } f(\widetilde{x} : T_1 \times \cdots \times T_n) = \\
\quad \textbf{let rec } g(y, \widetilde{r}) = \textbf{match } y \textbf{ with} \\
\qquad \texttt{[]} \texttt{ -> } r_1 \\
\qquad \texttt{|} \ (u_1, \ldots, u_n) :: y' \texttt{ -> } \textbf{let } \widetilde{r'} = step(u_1, \ldots, u_n, \widetilde{r}) \textbf{ in } g(y', \widetilde{r'}) \\
\quad \textbf{in } g(ezip_{T_1 \times \cdots \times T_n}(\widetilde{x}), \widetilde{d}).
\end{array}
$$

Here, $\widetilde{x}$ denotes a sequence $x_1, \ldots, x_k$, and $\widetilde{d}$ denotes a sequence of default integer or Boolean values $d_1, \ldots, d_\ell$, where d_i is `true` or 0; we write $d_{\mathbf{B}}$ for `true` and $d_{\mathbf{Z}}$ for 0.[2] The function *ezip* is an extended "zip" function, which maps a tuple

[2] The use of fixed default values slightly restricts the class of functions. In fact, the value of $f([\,])$ is restricted to `true`. To remove the restriction, it suffices to either (i) allow $\widetilde{d}$ to take other values and make them also learnable, or (ii) replace r_1 with $h(r_1)$ and make the Boolean function h also learnable.

consisting of lists, integers, and Booleans to a list of tuples. It is defined by:

$$ezip_{T_1\times\cdots\times T_n}(v_1,\ldots,v_n) =$$
$$\begin{cases} [\,] & \text{if every } v_i \text{ is } [\,], \text{ an integer, or a Boolean} \\ (ehd_{T_1}(v_1),\ldots,ehd_{T_1}(v_n)) :: (etl_{T_1}(v_1),\ldots,etl_{T_n}(v_n)) & \text{otherwise} \end{cases}$$
$$ehd_{\mathbf{Z}^*}([\,]) = -1 \quad ehd_{\mathbf{Z}^*}(n :: v) = n \quad ehd_{\mathbf{B}^*}([\,]) = \texttt{false} \quad ehd_{\mathbf{B}^*}(b :: v) = b$$
$$ehd_{\mathbf{Z}}(n) = n \quad ehd_{\mathbf{B}}(b) = b \quad etl_{\mathbf{Z}}(n) = n \quad etl_{\mathbf{B}}(b) = b$$
$$etl_{\mathbf{Z}^*}([\,]) = [\,] \quad etl_{\mathbf{Z}^*}(n :: v) = v \quad etl_{\mathbf{B}^*}([\,]) = [\,] \quad etl_{\mathbf{B}^*}(b :: v) = v.$$

For example, $ezip_{\mathbf{Z}^*\times\mathbf{Z}^*\times\mathbf{Z}}([1;2;3],[2;3],1) = [(1,2,1);(2,3,1);(3,-1,1)]$. The function *step* is the main target of the synthesis. It should be a function on integers and Booleans, consisting of (i) Boolean operations, (ii) affine expressions of the form $c_0 + c_1x_1 + \cdots + c_kx_k$ and (iii) inequalities of the form $e \le 0$, where e is an affine expression. The function g above can also be expressed as

$$\lambda\widetilde{x}.\#_1(foldl\ step'\ (\widetilde{d})\ (ezip_{T_1\times\cdots\times T_n}(\widetilde{x}))),$$

where *foldl* is the left fold function, *step*′ is the curried version of *step*, and $\#_1$ denotes the projection of a tuple to its first element.

In the case of the sortedness predicate discussed in Section 1, $T_1 = \mathbf{Z}^*$ with $k = 1$, the length $|\widetilde{z}|$ of the auxiliary parameters of g is 2, and $step : \mathbf{Z}\times\mathbf{B}\times\mathbf{Z} \to \mathbf{B}\times\mathbf{Z}$ is given by $step(u, r_b, r_i) = (r_b \wedge (r_i \le u), u)$.

For the predicate `avge` mentioned in Section 1, $T_1 = \mathbf{Z}^*$ and $T_2 = \mathbf{Z}$ with $k = 2$, and $step : \mathbf{Z}\times\mathbf{Z}\times\mathbf{B}\times\mathbf{Z} \to \mathbf{B}\times\mathbf{Z}$ is given by $step(u_1, u_2, r_b, r_i) = (r_i + u_1 - u_2 \ge 0, r_i + u_1 - u_2)$. Here, during the computation of $avge(\ell, m)$, the parameter z accumulates the sum of $\ell_i - m$ (where ℓ_i is the i-th element of ℓ). Whether the average of the elements of ℓ is no less than m can be determined by checking whether the final value of z is no less than 0.

Our synthesis problem subsumes the problem of learning automata (which is obtained as a special case, where the signature consists of a single predicate p: $(\mathbf{B}^*)^m$ and $step$:$\mathbf{B}^{m+n} \to \mathbf{B}^n$; input symbols and states are encoded as elements of $\mathbf{B}^m$ and $\mathbf{B}^n$ respectively) and also that of symbolic automatic relations [19]. In fact, the automatic synthesis of symbolic automatic relations was one of the motivations behind the present paper, as explained below.

The motivations for the synthesis problem above come from automated program verification and synthesis. For automated program verification, we have CHC-based program verification [1] in mind, where various program verification problems are reduced to the satisfiability problem for Constrained Horn Clauses (CHCs). For programs using lists, the CHCs obtained by the reduction involve predicates over lists, but the current CHC solvers [10,14,2] are not very good at solving such CHCs. A solver for the synthesis problem above can be used as an important component in a CHC solver [2,4] based on the ICE-learning framework [6], to synthesize a candidate solution for CHCs involving lists. Another application is the oracle-based programming mentioned in Section 1, whose goal is to synthesize code fragments to fill the holes of a given program pattern. By solving the synthesis problem above, we can automatically synthesize code

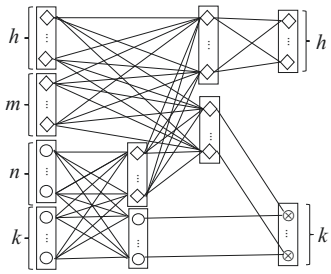


Fig. 2. The feed-forward network inside the RNN

fragments that involve recursive computation over lists. The roles of implication constraints in those applications are explained in [2,13].

In both of the applications above, the validity of a synthesized program is determined based on the whole verification or synthesis goal (in the case of verification, a synthesized predicate over lists is valid if it is indeed a solution for the CHC satisfiability problem). Thus, in the actual applications, the synthesis problem defined above needs to be repeatedly solved with the set of sample data being gradually expanded, until the end goal of program verification or synthesis is achieved.

3 The Design and Training of the RNN

This section describes the design of our special-purpose recursive neural network (RNN) tailored for our synthesis problem, and how to train it.

3.1 The Architecture of the RNN

The overall structure of the RNN is as already depicted in Figure 1. The structure of the feed-forward (FF) network inside the RNN is shown in Figure 2. The network consists of four layers of nodes, where the first layer (the leftmost one) consists of input nodes of the FF network, which hold the input values and hidden state values of the whole RNN, and the fourth layer (the rightmost one) consists of output nodes of the FF network, which hold the next states of the RNN. The nodes of the diamond shape take values in the range $[-1, 1]$ (either by the assumption on inputs or by the use of **tanh** as the activation function), and those of the circle shape take arbitrary floating point numbers. The value of each diamond-shaped node is computed by $\tanh(b + w_1x_1 + \cdots + w_kx_k)$ and that of each circle node is computed by $b + w_1x_1 + \cdots + w_kx_k$, where the bias b and the weight w_i vary for each node and link. Each $\otimes$ node in the fourth layer has exactly two inputs x and y, and outputs $\frac{x+1}{2}y$, where x is the output of the diamond-shaped node.

The part of the FF network to compute the diamond-shaped nodes in the fourth layer is analogous to the network in the previous NeuGuS framework [12] for the synthesis of logical formulas. Each diamond-shaped node in the second layer, whose output is $\tanh(b+w_1x_1+\cdots+w_kx_k)$, is intended to recognize linear inequalities of the form $c_0+c_1x_1+\cdots c_kx_k \geq d$ where $|d|$ is a small integer, and $c_i/c_0 = w_i/b$. The idea is that the value of the node $\tanh(b+w_1x_1+\cdots+w_kx_k) = \tanh((b/c_0)\cdot(c_0+c_1x_1+\cdots+c_kx_k))$ is close to -1 or 1 when both $|b/c_0|$ and $|c_0+c_1x_1+\cdots+c_kx_k|$ are large, so that the node carries only information about whether $c_0+c_1x_1+\cdots c_kx_k \geq d$ holds for each d such that $|d|$ is small. The diamond-shaped nodes in the third and fourth layers are intended to compute the Boolean combinations of those linear inequalities and Boolean inputs/hidden states.

The rest of the FF network, for computing the $\otimes$-nodes in the fourth layer, is intended to compute conditional expressions of the form

$$\textbf{if } b \textbf{ then } c_0+c_1x_1+\cdots+c_kx_k \textbf{ else } 0,$$

where b is a logical combination of linear inequalities and Boolean inputs/hidden states. Each circle node in the second layer compute the part $c_0+c_1x_1+\cdots+c_kx_k$, each node in the lower group of the third layer computes the Boolean value b, and each $\otimes$-node emulates the conditional expression. The idea is that the Boolean value b is actually represented as a value in $[-1,1]$ where values close to -1 and 1 are respectively interpreted as `false` and `true`. Thus, $\frac{b+1}{2}(c_0+c_1x_1+\cdots+c_kx_k)$ is close to $c_0+c_1x_1+\cdots+c_kx_k$ when b represents `true`, and it is close to 0 when b represents `false`. Note that the general conditional **if** b **then** e_1 **else** e_2 can be expressed by (**if** b **then** e_1 **else** 0)+(**if** $\neg b$ **then** e_2 **else** 0) $= \frac{b+1}{2}e_1+\frac{-b+1}{2}e_2$, which can be computed in the next cycle if we have hidden states that correspond to **if** b **then** e_1 **else** 0 and **if** $\neg b$ **then** e_2 **else** 0.

Remark 1. As explained above, the internal structure of our RNN is specialized for the purpose of solving our synthesis problem, and quite different from other popular RNNs. The $\otimes$-node is a reminiscent of a multiplicative gate of LSTM [9], but its main role is to emulate a conditional expression, rather than to address the problems of conventional RNNs such as the gradient vanishing problem. In fact, we do not expect that our RNN scales for very long lists. Fortunately, however, training data with short lists would often suffice for our synthesis problem.

3.2 Training the RNN

Let $\mathbf{R}$ be the set of real numbers and $g \in [-1,1]^{h+m}\times\mathbf{R}^{n+k} \to [-1,1]^h\times\mathbf{R}^k$ be the function computed by the FF network. The function $f \in ([-1,1]^m\times\mathbf{R}^n)^* \to [-1,1]$ computed by the whole RNN is defined by: $f(\ell) = f'(\widetilde{1},\ell,\widetilde{0})$, where:

$$f'(b_1,\ldots,b_h,[\,],\widetilde{0}) = b_1 \quad f'(\widetilde{b},x::\ell',\widetilde{z}) = f'(\widetilde{b}',\ell',\widetilde{z}') \text{ where } (\widetilde{b}',\widetilde{z}') = g(\widetilde{b},x,\widetilde{z}).$$

Here, $f' \in [-1,1]^h\times([-1,1]^m\times\mathbf{R}^n)^*\times\mathbf{R}^k \to [-1,1]$.

For an atom $p(\widetilde{v}, \widetilde{w})$ with $\widetilde{v} \in \mathbf{B}^m$ and $\widetilde{w} \in \mathbf{Z}^n$, we write $O_{p(\widetilde{v},\widetilde{w})}$ for $f(\widetilde{v}^\dagger, \widetilde{w})$, where $\texttt{true}^\dagger = 1$ and $\texttt{false}^\dagger = -1$. For an implication constraint $a_1 \wedge \cdots \wedge a_k \Rightarrow b_1 \vee \cdots \vee b_\ell$, we define the loss $loss_{a_1\wedge\cdots\wedge a_k \Rightarrow b_1\vee\cdots\vee b_\ell}$ for the implication constraint by:[3]

$$loss_{a_1\wedge\cdots\wedge a_k \Rightarrow b_1\vee\cdots\vee b_\ell} := \prod_{i\in\{1,\ldots,k\}} \left(\frac{1+O_{a_i}}{2}\right)^2 \cdot \prod_{j\in\{1,\ldots,\ell\}} \left(\frac{1-O_{b_j}}{2}\right)^2.$$

Note that $loss_{a_1\wedge\cdots\wedge a_k \Rightarrow b_1\vee\cdots\vee b_\ell}$ is 0 just if one of the a_i's is $\texttt{false}$ or one of the b_j's is $\texttt{true}$, which matches the meaning of the implication constraint. For a set $C = \{\gamma_1, \ldots, \gamma_p\}$ of implication constraints, the overall loss is defined by: $loss_C := \sum_{i\in\{1,\ldots,p\}} loss_{\gamma_i}$. Using the loss function above, we train the RNN with a gradient descent method.

Adjusting the loss function. The diamond-shaped nodes in Figure 2 are intended to hold Boolean values (which correspond to 1 and -1), but those nodes in the actual RNN trained by using the above loss function may take values close to 0, which cannot be interpreted as $\texttt{true}$ or $\texttt{false}$. That is problematic during the program synthesis, because the behavior of the RNN may deviate too much from that of an ordinary program to be synthesized. To remedy the problem, we also use a modified version of the loss function, obtained by replacing O_a in the basic loss function above with $O'_a := O_a \cdot \prod_i \frac{1+\lambda v_i^2}{1+\lambda}$ where $\lambda \geq 0$ (note that the modified loss function coincides with the basic loss function when $\lambda = 0$), and v_i is the value of a diamond-shaped node in the second or fourth layer of the FF network in Figure 2. This penalizes the use of "non-Boolean values" in diamond-shaped nodes. Note that if v_i cannot be interpreted as $\texttt{true}$ or $\texttt{false}$, i.e., if $|v_i|$ is close to 0, then $\frac{1+\lambda v_i^2}{1+\lambda}$ is much smaller than 1; thus, $|O'_a|$ would also be much smaller than 1, causing a large loss.

4 Synthesis Based on the Trained RNN

This section discusses how to construct the function *step* in Section 2, by using the trained RNN as a hint. From the trained RNN and its runs for training data, we gather and use the following information.

- The weight and bias of each link and node in the FF network.
- A collection of the inputs given to the FF network and the corresponding outputs of each node.

The output of the function *step* consists of Booleans and integers. We first discuss how to construct the integer part. The integer part of *step* corresponds

[3] This loss function is different from the one used in [12]. The difference is partly due to the encoding of Boolean values; Kobayashi et al. [12] used 0 for $\texttt{false}$ while we use -1. Another difference is the use of log vs squared loss. We preferred the latter for simplicity, but more experiments are necessary to tune the shape of the loss function.

to the $\otimes$-nodes of the FF-network in Figure 2, whose values are computed by a function of the form:

$$I(\widetilde{r}_{1,\ldots,m}, \widetilde{v}_{1,\ldots,m}, \widetilde{u}_{1,\ldots,n}, \widetilde{s}_{1,\ldots,k}) := \\ B(\widetilde{r}_{1,\ldots,m}, \widetilde{v}_{1,\ldots,m}, \widetilde{u}_{1,\ldots,n}, \widetilde{s}_{1,\ldots,k}) \otimes (b_0 + \textstyle\sum_{i\in\{1,\ldots,n\}} w_i u_i + \sum_{j\in\{1,\ldots,k\}} w'_j s_j),$$

where $\widetilde{r}$, $\widetilde{v}$, $\widetilde{u}$, and $\widetilde{s}$ respectively represent the hidden Boolean states, Boolean inputs, integer inputs, and hidden integer states; the function B is the output of a node in the lower half in the third layer in Figure 2; the part $b_0 + \cdots$ is the output of a circle node in the second layer; and $x \otimes y = \frac{x+1}{2} y$ as defined before.

Since the value of I is $b_0 + \sum_{i\in\{1,\ldots,n\}} w_i x_i + \sum_{j\in\{1,\ldots,k\}} w'_j y_j$ if the value of B is 1, and 0 if the value of B is -1, one may be tempted to construct the corresponding program expression as:

$$\textbf{if } \varphi_B \textbf{ then } b_0 + \textstyle\sum_{i\in\{1,\ldots,n\}} w_i u_i + \sum_{j\in\{1,\ldots,k\}} w'_j s_j \textbf{ else } 0,$$

where φ_B is a Boolean expression corresponding to B. That is problematic, however, because we wish to construct an integer program expression, but the weights and bias (w_i, w'_j, b_0) may be arbitrary floating point numbers. We thus re-scale the coefficients w_i, w'_j, and b_0 as follows. We first pick integers $c_0, c_1, \ldots, c_n$ and a real number r so that $rb_0, rw_1, \ldots, rw_n$ are close to $c_0, c_1, \ldots, c_n$. For w'_j, we just pick an integer c'_j close to w'_j, and prepare the integer expression:

$$\textbf{if } \varphi_B \textbf{ then } c_0 + \textstyle\sum_{i\in\{1,\ldots,n\}} c_i u_i + \sum_{j\in\{1,\ldots,k\}} c'_j s_j \textbf{ else } 0,$$

and use it as the integer-part of the function *step*.

Before constructing Boolean expressions (including φ_B), we adjust (i) the hidden integer states in the run history of RNNs for training data and (ii) the weights for the hidden integer nodes accordingly, to reflect the re-scaling of the coefficients for computing hidden integer states. We multiply (i) with r, and divide (ii) by r. To see the need for the adjustment, let us recall the *step* function for the sortedness:

$$step(u, r_b, r_i) = (r_b \wedge (r_i \leq u), u).$$

The RNN may actually learn the following function:

$$step(u, r_b, r_i) = (r_b \wedge (2r_i \leq u), 0.5u).$$

Suppose we have re-scaled $0.5u$ to u, to make the coefficient an integer. That would increase the value of the hidden integer state by a factor of 2, so that the coefficient of r_i in the inequality $2r_i \leq u$ should be decreased by half, to obtain $r_i \leq u$. We can thus obtain

$$step(u, r_b, r_i) = (r_b \wedge (r_i \leq u), u)$$

correctly.

Table 1. The value of each node of the FF-network for [2; 3; 5].

Before re-scaling.				After re-scaling.			
1st layer	2nd layer	3rd layer	4th layer	1st layer	2nd layer	3rd layer	4th layer
1.000	0.996	0.936	0.999	1.000	0.996	0.936	0.999
2.000	0.997	0.994		2.000	0.997	0.994	
0.000	-0.999	0.975	2.231	0.000	-0.999	0.975	1.978
	0.992	-0.969			0.992	-0.969	
	2.235	0.998			1.982	0.998	
0.999	1.000	0.936	0.999	0.999	1.000	0.936	0.999
3.000	0.977	0.992		3.000	0.977	0.992	
2.231	-1.000	0.967	3.256	1.978	-1.000	0.967	2.888
	0.924	-0.969			0.924	-0.969	
	3.262	0.998			2.893	0.998	
0.999	1.000	0.936	0.999	0.999	1.000	0.936	0.999
5.000	0.998	0.994		5.000	0.998	0.994	
3.256	-1.000	0.975	5.463	2.888	-1.000	0.975	4.844
	0.995	-0.969			0.995	-0.969	
	5.472	0.998			4.853	0.998	

Example 1. As a concrete example, consider the synthesis of a sortedness predicate *sorted*, which takes a list ℓ and returns whether ℓ is sorted in the ascending order. We set $h = n = k = 1$, and $m = 0$. The numbers of hidden nodes in the upper-half of the second layer and those in the upper-half of the third layer were both set to 4. We have trained the network by using 200 positive samples (like $\Rightarrow$ *sorted*([2; 3; 5])) and 94 negative samples (like *sorted*([9; 8]) $\Rightarrow$). After the training, we re-ran the RNN for the training data, and collected the value of each node of the FF-network. For example, for the data [2; 3; 5], we obtained the information shown on the left-hand side of Table 1. Here, the first group (separated by the horizontal line), shows the values of the nodes for the first element 2 of the list, and the second group shows those for the second element 3. We also look at the weights and biases of the FF-network to synthesize the target function *step*.

By inspecting the weights and bias for the the circle node in the second layer, we can find that the function computed by the node is: $-0.023+1.128u-0.045s$, where u and s respective denote the values of the integer input and the hidden integer state. The ratio between the constant and the coefficient of u is about $0:1$, and the co-efficient of s is close to 0. Thus, we set the integer expression to compute the next hidden integer state to **if** φ_B **then** u **else** 0, where the condition φ_B is yet to be synthesized.

The replacement of $-0.023 + 1.128u - 0.045s$ with u results in the decrease of the value of the hidden integer state by a factor of $1/1.128$, as shown on the right-hand side of Table 1. The weights for the nodes in the second layer are also accordingly re-scaled. □

It remains to construct Boolean expressions, consisting of linear inequalities on integer variables and Boolean variables. That can be achieved in a manner similar to [12]; we have, however, adopted the following procedure, which utilizes information about the value of each node in the FF network. In contrast, Kobayashi et al.'s method [12] uses only the weights and biases, in addition to the input and output for each training data; they did not utilize the values of internal nodes for each training data.

We synthesize linear inequalities corresponding to the diamond-shaped nodes in the second layer as follows. Let

$$\tanh(b_0 + w_1 u_1 + \cdots + w_n u_n + w_{n+1} s_1 + \cdots + w_{n+k} s_k)$$

be the value computed by a diamond-shaped node in the second layer (where we assume that the weights $w_{n+1}, \ldots, w_{n+k}$ have already been re-scaled). Let $c_0, c_1, \ldots, c_{n+k}$ be integers whose ratios are close to those of $b_0, w_1, \ldots, w_{n+k}$. Then we set the corresponding inequality to

$$c_0 + c_1 u_1 + \cdots + c_n u_n + c_{n+1} s_1 + \cdots + c_{n+k} s_k > e,$$

where $e \in \{-1, 0, 1\}$ is chosen so that the truth value of the inequality best-matches the actual input-output behavior of the node for training data; recall the discussion in Section 3.1.

Next, we construct Boolean functions corresponding to the diamond-shape nodes in the fourth layer and the lower-half of the third layer in Figure 2. This is performed by first constructing the truth tables for those functions based on the runs of the RNN for the training data, and then using a method for Boolean decision tree construction [7],[4] where Boolean variables and the inequalities synthesized above are used as qualifiers (i.e., atomic predicates that constitute Boolean functions). Those qualifiers are prioritized based on the weights for the nodes in the third and fourth layers. The synthesized functions may not completely match the truth tables if appropriate inequalities have not been found in the previous step. Even so, we proceed to the next step to construct the *step* function and test it; recall that in our *gray-box* use of the neural network, the internal behavior of the synthesized program need not completely match that of the RNN.

Example 2. Recall Example 1. The next step is to synthesize linear inequalities from the (re-scaled) weights of the nodes in the second layer. After the re-scaling of weights, the functions computed by the diamond-shaped nodes are:

$$\tanh(1.396 + 0.876u + 1.182s) \quad \tanh(1.066 + 1.084u - 1.052s) \quad \cdots$$

Based on the ratios between the constant and coefficients, we synthesize linear inequalities of the form:

$$4 + 3u + 4s > e_1 \qquad 1 + u - s > e_2 \qquad -6 - 4u - 3s > e_3 \qquad u - s > e_4.$$

[4] Kobayashi et al. [12] suggested using the Quine-McClusky method for this purpose, but we prefer the Boolean decision tree construction for two reasons. First, the Quine-McClusky method would not scale when the dimension is large. Second, we wish to give priorities to some qualifiers as explained below.

We then check the re-scaled trace information (such as the one in Table 1, but including the trace information for all the training data), we choose appropriate values for each e_i. In the present case, we obtain:

$$4 + 3u + 4s > 0 \qquad 1 + u - s > -1 \qquad -7 - 3u - 4s > 0 \qquad u - s > -1.$$

It remains to synthesize Boolean functions. To this end, for each diamond-shaped node in the fourth layer and in the lower-half of the third layer, we construct a truth table, where the inputs are Boolean values obtained by discretizing the values of the diamond-shaped nodes in the first and second layers. For example, from Table 2, we obtain the following truth table for the diamond-shaped node in the fourth layer. The duplicated rows can be removed before the synthesis of a logical function.

input					output
I_0	I_1	I_2	I_3	I_4	O
`true`	`true`	`true`	`false`	`true`	`true`
`true`	`true`	`true`	`false`	`true`	`true`
...	...	...	...	...	...

Here, I_0 corresponds to the value of the hidden Boolean node, and I_1–I_4 correspond to the diamond-shaped nodes in the second layer, which represent inequality constraints extracted above. We interpret values close to 1 (say, those greater than 0.5) as `true`, and those close to -1 (say, those less than -0.5) as `false`, ignoring the other values.

Once a truth table has been constructed, we can apply a classical method to synthesize a logical function that conforms to the truth table. In our implementation, we have employed a technique for Boolean decision tree construction; instead of computing the entropy [7], however, we have prioritized Boolean inputs (I_0–I_4, in the above case) based on the weights for the nodes in the third and fourth layer, which indicate which Boolean inputs affected the output node.

Suppose that the logical function $O = I_0 \wedge I_4$ has been synthesized in the above example. Suppose also that the constant function `true` has been synthesized for the diamond-shaped node in the third layer. Since I_0 corresponds to the hidden Boolean state, and I_4 corresponds to the inequality $u - s > -1$ (which is equivalent to $s \leq u$), we obtain

$$\mathit{step}(u, r_b, s) = (r_b \wedge (s \leq u), \textbf{if } \texttt{true} \textbf{ then } u \textbf{ else } 0)$$

as the step function. □

By combining the procedures above, we can construct the function *step*. After constructing the *step* function, we test the synthesized recursive function against training data, and check whether the outputs of the synthesized function satisfy all the implication constraints. If some constraints are not satisfied, we re-train the RNN and repeat the synthesis procedure above. To avoid the re-training of the RNN from scratch, however, we first fix the part for computing the hidden integer states. This is because the process of re-scaling the parameters for the

hidden integer states as explained above is costly and error-prone. Upon repeated failures, however, we reset all the parameters of the RNN and re-train it from scratch.

5 Implementation and Experiments

We have implemented a tool called NeuGuSR for the synthesis of recursive predicates based on the method described above in OCaml using the machine learning framework ocaml-torch (https://github.com/LaurentMazare/ocaml-torch), which is an OCaml interface for the PyTorch library. Our tool is available at https://github.com/naokikob/neugusR. This section describes the experiments we conducted that confirm the effectiveness of our approaches.

All the experiments below were conducted on a laptop computer with Intel(R) Core(TM) i5-8265U CPU (1.60GHz) and 8 GB memory. Training was done using only CPU.

5.1 Dataset and predicates

We have prepared 11 recursive predicates over integer lists and integers for synthesis. Examples include predicates such as $\texttt{max}(l, n)$ which says the largest element of l is n, $\texttt{sumle}(l_1, l_2)$ which says the sum of l_1 is less than or equal to the sum of l_2, and predicates $\texttt{sorted}(l)$ and $\texttt{avge}(l, n)$ as already described in Section 1.

For experiments, we consider positive constraints (of the form $\Rightarrow a_k$ where $a_k \in \mathbf{Atoms}_{\mathcal{K}}$ and $\mathcal{K}$ is the corresponding signature of the predicate), negative constraints (of the form $b_k \Rightarrow$), as well as general implication constraints as defined in Section 2.

For each problem (predicate), we performed 3 runs to see if the solver was able to synthesize a program that matches all the training examples. We set the time limit of each run to 1200 seconds. In each run, the neural network is trained for 30000 steps by default. At each step, all the training examples of the predicate were used to optimize the neural network. In each run, the training was terminated early if the accuracy reached 100% on the training examples and the loss was less than a threshold, which in the current setting is 10^{-6}.

If the accuracy did not reach 100% within 30000 steps or there are constraints not satisfied by the synthesized program, the training was set to restart with fresh parameters except for the weights of the hidden integer states. If there are three consecutive failures of convergence, however, we reset all the parameters and restart training from scratch.

We used the Adam optimizer [11] for training with the default setting of ocaml-torch ($\beta_1 = 0.9$, $\beta_2 = 0.999$ without weight decay), and the learning rate was 0.001. Learned parameters are not shared between different problems.

5.2 Evaluation

The specification of RNN used for each problem is as follows. For all the predicates other than `updown` and `max`, we used 4 nodes for the second layer of the RNN, 8 nodes for the third layer of the RNN, and 1 node each for the integer hidden state and the boolean hidden state. For `updown` and `max`, we used 2 nodes for the boolean hidden states and 16 nodes for the third layer. For `max`, we also used 8 nodes for the second layer instead of 4.

We report the performance of our tool NeuGuSR with respect to the following metrics.

- #retry: the total number of retries. For each run, up to 10 retries were allowed within the time limit. There can be 3×10 retries for each problem in total in the worst case.
- #success: the number of runs in which a program that correctly classifies the positive and negative examples was constructed.
- time: the average execution time per run. The execution time includes the whole process for training the RNN and synthesizing/testing a program, though it was dominated by the time for training the RNN.

Table 2 shows the performance of NeuGuSR for each predicate. It can be seen that NeuGuSR was able to solve all the problems consistently, with the only exception of `max` which failed once due to a timeout. The small number of retries triggered during the synthesis of each predicate suggests that our approach is effective. Our RNN was able to classify the positive and negative examples very well, because otherwise multiple restarts of training would have been forced even before entering the extraction phase. Our extraction procedure was also reasonably accurate — while errors could occur, they were quickly fixed within a few retries (3 on average as can be seen in Table 2).

Table 2. Performance on the predicates to be synthesized.

Predicate	# retry	# success (out of 3)	time (s)
`sorted`(l)	0	3	171.2
`sortedrev`(l)	1	3	217.5
`stairge`(l)	5	3	560.6
`allge`(l, n)	1	3	272.3
`allle`(l, n)	1	3	355.1
`somege`(l, n)	1	3	376.2
`avge`(l, n)	8	3	571.2
`listle`(l_1, l_2)	0	3	214.4
`sumle`(l_1, l_2)	2	3	241.4
`updown`(l)	1	3	226.1
`max`(l, n)	7	2	557.6

The predicate `max` is the only predicate that involves equality among the 11 predicates, which probably explains why it is the most difficult one. The

fact that `max` can be synthesized was more of a surprise which demonstrated the generality of our approach to some extent. While our framework was not designed specifically to handle equalities, the neural network, if lucky, might still be able to find clever ways to express equalities using inequalities. This is one of the reasons we specified 8 nodes for the second layer when dealing with `max` — the more inequalities we have, the more likely a combination of them happens to express certain equality.

Remark 2. We could not find any previous tool that can be directly compared with ours. A possible alternative approach to our synthesis problem would be to prepare a template for the step function, generate constraints on parameters in the template, and use an SMT solver to solve them.

6 Related Work

As already mentioned, the present work may be considered an extension of Kobayashi et al.'s NeuGuS framework [12], where feed-forward neural networks are used as gray-boxes to synthesize formulas of quantifier-free linear integer arithmetic. We have significantly expanded the scope of NeuGuS, by enabling the synthesis of recursive predicates on lists; to that end, we have employed special-purpose recursive neural networks.

Our work has been partially motivated by Shimoda et al.'s work on an extension of symbolic automata called symbolic automatic relations (SARs) [19]. They introduced SARs to express recursive predicates on lists, and used them to express loop invariants on lists (more precisely, to express candidate solutions for the CHC satisfiability problem [1]) for automated verification of list-manipulating programs. They left it to future work how to automatically infer SARs from positive, negative, and implication constraints. Our work fills that gap, since the class of programs synthesized in our framework corresponds to their SARs (more precisely, Σ_1^{sar}-formulas [19]). Further refinement and optimizations would be, however, required for our tool to be effectively used in that context.

Our work is also related to neural network-based approaches to the synthesis of finite automata [16,21]. Our method deals with a much wider class of programs involving integers and integer lists. Also, the problem setting is slightly different; Weiss et al.'s method [21] takes a trained RNN as the ground truth, and aims to construct an automaton whose behavior matches that of the RNN. In contrast, in our approach, we allow the behavior of the synthesized program and that of the RNN to be different for inputs other than those given as training data. This is because in the NeuGuS framework, the trained RNN is supposed to be used just as a hint, and does not necessarily provide the ground truth. The ground truth is determined from the whole verification or synthesis goal [12,13], as discussed at the end of Section 2. In the context of program verification, the synthesized predicate is used as a candidate program invariant, and it is checked whether it is indeed an inductive invariant; if not, then new training data are added and

NeuGuS should be repeated. In the context of oracle-based program synthesis, the synthesized function is used as a component of the whole program, and then it is checked whether the whole program satisfies a specification; if not, then new training data for the function are generated and NeuGuS should be repeated. Recently, the above line of work has also been further extended to infer weighted automata [22,15] and context-free grammars [23], which are incompatible with the class of programs synthesized by our method.

There have been studies of other approaches to program synthesis based on neural networks, most notably, those based on transformers [3,18,17]. Both the problem settings and approaches (the ways in which neural networks are used) are quite different between those studies and our work. Our goal is to synthesize programs from positive/negative/implication constraints (where those constraints are added as necessary in the whole loop of program verification or synthesis), and it is not clear to us how to effectively apply transformers-based approaches to program synthesis for that purpose. Whilst the transformers-based approaches can in principle be used for our program synthesis problem, huge training data (which consist of pairs of positive/negative/implication constraints and a program that satisfies the constraints) would be required and they might not work well for the synthesis of unseen programs. Other neural network-based approaches include that of AlphaTensor [5], which used deep reinforcement learning to discover new matrix multiplication algorithms.

The synthesis of predicates from positive/negative samples (but without implication constraints) is an instance of the well-studied problem of *programming by examples* (PBE). PBE has been successful especially in the synthesis of string-to-string functions in DSL [8], and machine learning has also been recently applied [20]. To our knowledge, however, the synthesis of recursive functions has not been much studied in that context.

7 Conclusion

We have proposed a novel approach to automated synthesis of recursive predicates on lists, as an extension of Kobayashi et al.'s neural-network-guided synthesis (NeuGuS) [12]. We have designed a special-purpose recursive neural network and devised a method to synthesize a recursive predicate by using the trained network as a hint. We have implemented a synthesis tool based on the method and confirmed that the tool works reasonably well for various examples. We plan to further refine the tool and deploy it in the context of automated verification of list-manipulating programs [19] and oracle-based program synthesis [13]. We also plan to extend the method to enable the synthesis of a larger class of recursive programs, including more general list-processing programs that go beyond the "fold" functions, and tree-processing programs.

Acknowledgments

We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP20H05703.

References

1. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. LNCS, vol. 9300, pp. 24–51. Springer (2015)
2. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. J. Autom. Reason. **64**(7), 1393–1418 (2020), https://doi.org/10.1007/s10817-020-09571-y
3. Chen, M., et al.: Evaluating large language models trained on code. CoRR **abs/2107.03374** (2021), https://arxiv.org/abs/2107.03374
4. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 131:1–131:25 (2018), https://doi.org/10.1145/3276501
5. Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., Ruiz, F.J.R., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., Kohli, P.: Discovering faster matrix multiplication algorithms with reinforcement learning. Nat. **610**(7930), 47–53 (2022). https://doi.org/10.1038/s41586-022-05172-4, https://doi.org/10.1038/s41586-022-05172-4
6. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Proceedings of CAV 2014. LNCS, vol. 8559, pp. 69–87. Springer (2014)
7. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016. pp. 499–512. ACM (2016), https://doi.org/10.1145/2837614.2837664
8. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 317–330. ACM (2011). https://doi.org/10.1145/1926385.1926423
9. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735, https://doi.org/10.1162/neco.1997.9.8.1735
10. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–7 (2018)
11. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: 3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1412.6980
12. Kobayashi, N., Sekiyama, T., Sato, I., Unno, H.: Toward neural-network-guided program synthesis and verification. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12913, pp. 236–260. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_12
13. Kobayashi, N., Sekiyama, T., Sato, I., Unno, H.: Toward neural-network-guided program synthesis and verification. CoRR **abs/2103.09414** (2021), https://arxiv.org/abs/2103.09414
14. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Des. **48**(3), 175–205 (2016), https://doi.org/10.1007/s10703-016-0249-4

15. Okudono, T., Waga, M., Sekiyama, T., Hasuo, I.: Weighted automata extraction from recurrent neural networks via regression on state spaces. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020. pp. 5306–5314. AAAI Press (2020), https://ojs.aaai.org/index.php/AAAI/article/view/5977
16. Omlin, C.W., Giles, C.L.: Extraction of rules from discrete-time recurrent neural networks. Neural Networks **9**(1), 41–52 (1996). https://doi.org/10.1016/0893-6080(95)00086-0
17. Poesia, G., Polozov, A., Le, V., Tiwari, A., Soares, G., Meek, C., Gulwani, S.: Synchromesh: Reliable code generation from pre-trained language models. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net (2022), https://openreview.net/forum?id=KmtVD97J43e
18. Roper, J.: Transformer-based program synthesis for low-data environments. CoRR **abs/2205.09246** (2022). https://doi.org/10.48550/arXiv.2205.09246
19. Shimoda, T., Kobayashi, N., Sakayori, K., Sato, R.: Symbolic automatic relations and their applications to SMT and CHC solving. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12913, pp. 405–428. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_20
20. Verbruggen, G., Le, V., Gulwani, S.: Semantic programming by example with pre-trained models. Proc. ACM Program. Lang. **5**(OOPSLA), 1–25 (2021). https://doi.org/10.1145/3485477
21. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5244–5253. PMLR (2018), http://proceedings.mlr.press/v80/weiss18a.html
22. Weiss, G., Goldberg, Y., Yahav, E.: Learning deterministic weighted automata with queries and counterexamples. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 8558–8569 (2019), https://proceedings.neurips.cc/paper/2019/hash/d3f93e7766e8e1b7ef66dfdd9a8be93b-Abstract.html
23. Yellin, D.M., Weiss, G.: Synthesizing context-free grammars from recurrent neural networks. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12651, pp. 351–369. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_19

Automata

Modular Mix-and-Match Complementation of Büchi Automata

Vojtěch Havlena[1(✉)], Ondřej Lengál[1(✉)], Yong Li[2,3(✉)], Barbora Šmahlíková[1(✉)], and Andrea Turrini[3,4(✉)]

[1] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
ihavlena@fit.vut.cz, lengal@vut.cz, xsmahl00@vut.cz
[2] Department of Computer Science, University of Liverpool, Liverpool, UK
liyong@liverpool.ac.uk
[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, People's Republic of China
turrini@ios.ac.cn
[4] Institute of Intelligent Software, Guangzhou, Guangzhou, People's Republic of China

Abstract. Complementation of nondeterministic Büchi automata (BAs) is an important problem in automata theory with numerous applications in formal verification, such as termination analysis of programs, model checking, or in decision procedures of some logics. We build on ideas from a recent work on BA determinization by Li *et al.* and propose a new modular algorithm for BA complementation. Our algorithm allows to combine several BA complementation procedures together, with one procedure for a subset of the BA's strongly connected components (SCCs). In this way, one can exploit the structure of particular SCCs (such as when they are inherently weak or deterministic) and use more efficient specialized algorithms, regardless of the structure of the whole BA. We give a general framework into which partial complementation procedures can be plugged in, and its instantiation with several algorithms. The framework can, in general, produce a complement with an Emerson-Lei acceptance condition, which can often be more compact. Using the algorithm, we were able to establish an exponentially better new upper bound of $\mathcal{O}(4^n)$ for complementation of the recently introduced class of elevator automata. We implemented the algorithm in a prototype and performed a comprehensive set of experiments on a large set of benchmarks, showing that our framework complements well the state of the art and that it can serve as a basis for future efficient BA complementation and inclusion checking algorithms.

1 Introduction

Nondeterministic Büchi automata (BAs) [8] are an elegant and conceptually simple framework to model infinite behaviors of systems and the properties they are expected to satisfy. BAs are widely used in many important verification tasks, such as termination analysis of programs [30], model checking [54], or as the underlying formal model of decision procedures for some logics (such as S1S [8] or a fragment of the first-order logic over Sturmian words [31]). Many of these applications require to perform *complementation* of BAs: For instance, in termination analysis of programs within Ultimate Automizer [30], complementation is used to keep track of the set of paths whose termination still needs to be proved. On the other hand, in model checking[5] and decision

[5] Here, we consider model checking w.r.t. a specification given in some more expressive logic, such as S1S [8], QPTL [50], or HyperLTL [12], rather than LTL [44], where negation is simple.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 249–270, 2023.
https://doi.org/10.1007/978-3-031-30823-9_13

procedures of logics, complement is usually used to implement negation and quantifier alternation. Complementation is often the most difficult automata operation performed here; its worst-case state complexity is $O((0.76n)^n)$ [48,2] (which is tight [55]).

In these applications, efficiency of the complementation often determines the overall efficiency (or even feasibility) of the top-level application. For instance, the success of Ultimate Automizer in the Termination category of the International Competition on Software Verification (SV-COMP) [51] is to a large degree due to an efficient BA complementation algorithm [6,11] tailored for BAs with a special structure that it often encounters (as of the time of writing, it has won 6 gold medals in the years 2017–2022 and two silver medals in 2015 and 2016). The special structure in this case are the so-called *semi-deterministic BAs* (SDBAs), BAs consisting of two parts: (i) an initial part without accepting states/transitions and (ii) a deterministic part containing accepting states/transitions that cannot transition into the first part.

Complementation of SDBAs using one from the family of the so-called NCSB algorithms [6,5,11,28] has the worst-case complexity $O(4^n)$ (and usually also works much better in practice than general BA complementation procedures). Similarly, there are efficient complementation procedures for other subclasses of BAs, e.g., (i) *deterministic BAs* (DBAs) can be complemented into BAs with $2n$ states [35] (or into co-Büchi automata with $n+1$ states) or (ii) *inherently weak BAs* (BAs where in each *strongly connected component* (SCC), either all cycles are accepting or all cycles are rejecting) can be complemented into DBAs with $O(3^n)$ states using the Miyano-Hayashi algorithm [42].

For a long time, there has been no efficient algorithm for complementation of BAs that are highly structured but do not fall into one of the categories above, e.g., BAs containing inherently weak, deterministic, and some nondeterministic SCCs. For such BAs, one needed to use a general complementation algorithm with the $O((0.76n)^n)$ (or worse) complexity. To the best of our knowledge, only recently has there appeared works that exploit the structure of BAs to obtain a more efficient complementation algorithm: (i) The work of Havlena *et al.* [29], who introduce the class of *elevator automata* (BAs with an arbitrary mixture of inherently weak and deterministic SCCs) and give a $O(16^n)$ algorithm for them. (ii) The work of Li *et al.* [37], who propose a BA determinization procedure (into a deterministic Emerson-Lei automaton) that is based on decomposing the input BA into SCCs and using a different determinization procedure for different types of SCCs (inherently weak, deterministic, general) in a synchronous construction.

In this paper, we propose a new BA complementation algorithm inspired by [37], where we exploit the fact that complementation is, in a sense, more relaxed than determinization. In particular, we present a *framework* where one can plug-in different partial complementation procedures fine-tuned for SCCs with a specific structure. The procedures work only with the given SCCs, to some degree *independently* (thus reducing the potential state space explosion) from the rest of the BA. Our top-level algorithm then orchestrates runs of the different procedures in a *synchronous* manner (or completely independently in the so-called *postponed* strategy), obtaining a resulting automaton with potentially a more general acceptance condition (in general an Emerson-Lei condition), which can help keeping the result small. If the procedures satisfy given correctness requirements, our framework guarantees that its instantiation will also be correct. We also propose its optimizations by, e.g., using round-robin to decrease the amount of nondeterminism, using a shared breakpoint to reduce the size and the number of colours for certain class of partial algorithms, and generalize simulation-based pruning of macrostates.

We provide a detailed description of partial complementation procedures for inherently weak, deterministic, and initial deterministic SCCs, which we use to obtain a *new* exponentially better upper bound of $O(4^n)$ for the class of elevator automata (i.e., the same upper bound as for its strict subclass of SDBAs). Furthermore, we also provide two partial procedures for general SCCs based on determinization (from [37]) and the rank-based construction. Using a prototype implementation, we then show our algorithm complements well existing approaches and significantly improves the state of the art.

2 Preliminaries

We fix a finite non-empty alphabet Σ and the first infinite ordinal ω. An (infinite) word w is a function $w\colon \omega \to \Sigma$ where the i-th symbol is denoted as w_i. Sometimes, we represent w as an infinite sequence $w = w_0 w_1 \dots$ We denote the set of all infinite words over Σ as Σ^ω; an *ω-language* is a subset of Σ^ω.

Emerson-Lei Acceptance Conditions. Given a set $\Gamma = \{0, \dots, k-1\}$ of k *colours* (often depicted as 0, 1, etc.), we define the set of *Emerson-Lei acceptance conditions* $\mathbb{EL}(\Gamma)$ as the set of formulae constructed according to the following grammar:

$$\alpha ::= \mathsf{Inf}(c) \mid \mathsf{Fin}(c) \mid (\alpha \wedge \alpha) \mid (\alpha \vee \alpha) \tag{1}$$

for $c \in \Gamma$. The *satisfaction* relation $\models$ for a set of colours $M \subseteq \Gamma$ and condition α is defined inductively as follows (for $c \in \Gamma$):

$$M \models \mathsf{Fin}(c) \text{ iff } c \notin M, \qquad M \models \alpha_1 \vee \alpha_2 \text{ iff } M \models \alpha_1 \text{ or } M \models \alpha_2,$$
$$M \models \mathsf{Inf}(c) \text{ iff } c \in M, \qquad M \models \alpha_1 \wedge \alpha_2 \text{ iff } M \models \alpha_1 \text{ and } M \models \alpha_2.$$

Emerson-Lei Automata. A (nondeterministic transition-based[6]) *Emerson-Lei automaton* (TELA) over Σ is a tuple $\mathcal{A} = (Q, \delta, I, \Gamma, \mathsf{p}, \mathsf{Acc})$, where Q is a finite set of *states*, $\delta \subseteq Q \times \Sigma \times Q$ is a set of *transitions*[7], $I \subseteq Q$ is the set of *initial* states, Γ is the set of *colours*, $\mathsf{p}\colon \delta \to 2^\Gamma$ is a *colouring function* of transitions, and $\mathsf{Acc} \in \mathbb{EL}(\Gamma)$. We use $p \xrightarrow{a} q$ to denote that $(p, a, q) \in \delta$ and sometimes also treat δ as a function $\delta\colon Q \times \Sigma \to 2^Q$. Moreover, we extend δ to sets of states $P \subseteq Q$ as $\delta(P, a) = \bigcup_{p \in P} \delta(p, a)$. We use $\mathcal{A}[q]$ for $q \in Q$ to denote the automaton $\mathcal{A}[q] = (Q, \delta, \{q\}, \Gamma, \mathsf{p}, \mathsf{Acc})$, i.e., the TELA obtained from $\mathcal{A}$ by setting q as the only initial state. $\mathcal{A}$ is called *deterministic* if $|I| \leq 1$ and $|\delta(q, a)| \leq 1$ for each $q \in Q$ and $a \in \Sigma$. If $\Gamma = \{0\}$ and $\mathsf{Acc} = \mathsf{Inf}(0)$, we call $\mathcal{A}$ a *Büchi automaton* (BA) and denote it as $\mathcal{A} = (Q, \delta, I, F)$ where F is the set of all transitions coloured by 0, i.e., $F = \mathsf{p}^{-1}(\{0\})$. For a BA, we use $\delta_F(p, a) = \{q \in \delta(p, a) \mid \mathsf{p}(p \xrightarrow{a} q) = \{0\}\}$ (and extend the notation to sets of states as for δ). A BA $\mathcal{A} = (Q, \delta, I, F)$ is called *semi-deterministic* (SDBA) if for every accepting transition $(p \xrightarrow{a} q) \in F$, the reachable part of $\mathcal{A}[q]$ is deterministic.

A *run* of $\mathcal{A}$ from $q \in Q$ on an input word w is an infinite sequence $\rho\colon \omega \to Q$ that starts in q and respects δ, i.e., $\rho_0 = q$ and $\forall i \geq 0\colon \rho_i \xrightarrow{w_i} \rho_{i+1} \in \delta$. Let $\inf_\delta(\rho) \subseteq \delta$ denote the set of transitions occurring in ρ infinitely often and $\inf_\Gamma(\rho) = \bigcup\{\mathsf{p}(x) \mid x \in$

[6] We only consider transition-based acceptance in order to avoid cluttering the paper by always dealing with accepting states *and* accepting transitions. Extending our approach to state/transition-based (or just state-based) automata is straightforward.

[7] Note that some authors use a more general definition of TELAs with $\delta \subseteq Q \times \Sigma \times 2^\Gamma \times Q$; we only use them as the output of our algorithm, where the simpler definition suffices.

$\inf_\delta(\rho)\}$ be the set of infinitely often occurring colours. A run ρ is *accepting* in $\mathcal{A}$ iff $\inf_\Gamma(\rho) \models \mathsf{Acc}$ and the *language* of $\mathcal{A}$, denoted as $\mathcal{L}(\mathcal{A})$, is defined as the set of words $w \in \Sigma^\omega$ for which there exists an accepting run in $\mathcal{A}$ starting with some state in I.

Consider a BA $\mathcal{A} = (Q, \delta, I, F)$. For a set of states $S \subseteq Q$ we use $\mathcal{A}_S$ to denote the copy of $\mathcal{A}$ where accepting transitions only occur between states from S, i.e., the BA $\mathcal{A}_S = (Q, \delta, I, F \cap \delta|_S)$ where $\delta|_S = \{p \xrightarrow{a} q \in \delta \mid p, q \in S\}$. We say that a non-empty set of states $C \subseteq Q$ is a *strongly connected component* (SCC) if every pair of states of C can reach each other and C is a maximal such set. An SCC of $\mathcal{A}$ is *trivial* if it consists of a single state that does not contain a self-loop and *non-trivial* otherwise. An SCC C is *accepting* if it contains at least one accepting transition and *inherently weak* iff either (i) every cycle in C contains a transition from F or (ii) no cycle in C contains any transitions from F. An SCC C is *deterministic* iff the BA $(C, \delta|_C, \{q\}, \emptyset)$ for any $q \in C$ is deterministic. We denote inherently weak components as IWCs, accepting deterministic components that are not inherently weak as DACs (deterministic accepting), and the remaining accepting components as NACs (nondeterministic accepting). A BA $\mathcal{A}$ is called an *elevator automaton* if it contains no NAC.

We assume that $\mathcal{A}$ contains no accepting transition outside its SCCs (no run can cycle over such transitions). We use δ_{SCC} to denote the restriction of δ to transitions that do not leave their SCCs, formally, $\delta_{\mathrm{SCC}} = \{p \xrightarrow{a} q \in \delta \mid p \text{ and } q \text{ are in the same SCC}\}$. A *partition block* $P \subseteq Q$ of $\mathcal{A}$ is a nonempty union of its accepting SCCs, and a *partitioning* of $\mathcal{A}$ is a sequence $P_1, \ldots, P_n$ of pairwise disjoint partition blocks of $\mathcal{A}$ that contains all accepting SCCs of $\mathcal{A}$. Given a P_i, let $\mathcal{A}_{P_i}$ be the BA obtained from $\mathcal{A}$ by removing colours from transitions outside P_i. The following fact serves as the basis of our decomposition-based complementation procedure.

Fact 1. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{P_1}) \cup \ldots \cup \mathcal{L}(\mathcal{A}_{P_n})$

The complement (automaton) of a BA $\mathcal{A}$ is a TELA that accepts the complement language $\Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$ of $\mathcal{L}(\mathcal{A})$. In the paper, we call a state and a run of a complement automaton a *macrostate* and a *macrorun*, respectively.

3 A Modular Complementation Algorithm

In a nutshell, the main idea of our BA complementation algorithm is to first decompose a BA $\mathcal{A}$ into several partition blocks according to their properties, and then perform complementation for each of the partition blocks (potentially using a different algorithm) independently, using either a *synchronous* construction, synchronizing the complementation algorithms for all partition blocks in each step, or a *postponed* construction, which complements the partition blocks independently and combines the partial results using automata product construction. The decomposition of $\mathcal{A}$ into partition blocks can either be trivial—i.e., with one block for each accepting SCC—, or more elaborate, e.g., a partitioning where one partition block contains all accepting IWCs, another contains all DACs, and each NAC is given its own partition block. In this way, one can avoid running a general complementation algorithm for unrestricted BAs with the state complexity upper bound $O((0.76n)^n)$ and, instead, apply the most suitable complementation procedure for each of the partition blocks. This comes with three main advantages:

1. The complementation algorithm for each partition block can be selected differently in order to exploit the properties of the block. For instance, for partition blocks

with IWCs, one can use complementation based on the breakpoint (the so-called Miyano-Hayashi) construction [42] with $O(3^n)$ macrostates (cf. Sec. 4.1), while for partition blocks with only DACs, one can use an algorithm with the state complexity $O(4^n)$ based on an adaptation of the NCSB construction [6,5,11,28] for SDBAs (cf. Sec. 4.2). For NACs, one can choose between, e.g., rank- [34,21,48,10,24,29] or determinization-based [46,43,45] algorithms, depending on the properties of the NACs (cf. Sec. 6).

2. The different complementation algorithms can focus only on the respective blocks and do not need to consider other parts of the BA. This is advantageous, e.g., for rank-based algorithms, which can use this restriction to obtain tighter bounds on the considered ranks (even tighter than using the refinement in [29]).
3. The obtained automaton can be more compact due to the use of a more general acceptance condition than Büchi [47]—in general, it can be a conjunction of any $\mathbb{EL}$ conditions (one condition for each partition block), depending on the output of the complementation procedures; this can allow a more compact encoding of the produced automaton allowed by using a mixture of conditions. E.g., a deterministic BA can be complemented with constant extra generated states when using a co-Büchi condition rather than a linear number of generated states for a Büchi condition (see Sec. 5.1).

Those partial complementation algorithms then need to be orchestrated by a top-level algorithm to produce the complement of $\mathcal{A}$.

One might regard our algorithm as an optimization of an approach that would for each partition block P obtain a BA $\mathcal{A}_P$, complement $\mathcal{A}_P$ using the selected algorithm, and perform the intersection of all obtained $\mathcal{A}_P$'s (which would, however, not be able to get the upper bound for elevator automata that we give in Sec. 4.3). Indeed, we also implemented the mentioned procedure (called the *postponed* approach, described in Sec. 5.2) and compared it to our main procedure (called the *synchronous* approach).

3.1 Basic Synchronous Algorithm

In this section, we describe the basic *synchronous* top-level algorithm. Then, in Sec. 4, we provide its instantiation for elevator automata and give a new upper bound for their complementation; in Sec. 5, we discuss several optimizations of the algorithm; and in Sec. 6, we give a generalization for unrestricted BAs. Let us fix a BA $\mathcal{A} = (Q, \delta, I, F)$ and, w.l.o.g., assume that $\mathcal{A}$ is *complete*, i.e., $|I| > 0$ and all states $q \in Q$ have an outgoing transition over all symbols $a \in \Sigma$.

The synchronous algorithm works with partial complementation algorithms for BA's partition blocks. Each such algorithm $\mathtt{Alg}$ is provided with a structural condition $\varphi_{\mathtt{Alg}}$ characterizing partition blocks it can complement. For a BA $\mathcal{B}$, we use the notation $\mathcal{B} \models \varphi$ to denote that $\mathcal{B}$ satisfies the condition φ. We say that $\mathtt{Alg}$ is a *partial complementation algorithm for a partition block* P if $\mathcal{A}_P \models \varphi_{\mathtt{Alg}}$. We distinguish between $\mathtt{Alg}$, a general algorithm able to complement a partition block of a given type, and $\mathtt{Alg}_P$, its instantiation for the partition block P. Each instance $\mathtt{Alg}_P$ is required to provide the following:

- $\mathtt{T}^{\mathtt{Alg}_P}$ — the type of the macrostates produced by the algorithm;
- $\mathtt{Colours}^{\mathtt{Alg}_P} = \{0, \ldots, k^{\mathtt{Alg}_P} - 1\}$ — the set of used colours;
- $\mathtt{Init}^{\mathtt{Alg}_P} \in 2^{\mathtt{T}^{\mathtt{Alg}_P}}$ — the set of initial macrostates;
- $\mathtt{Succ}^{\mathtt{Alg}_P} : (2^Q \times \mathtt{T}^{\mathtt{Alg}_P} \times \Sigma) \to 2^{\mathtt{T}^{\mathtt{Alg}_P} \times \mathtt{Colours}^{\mathtt{Alg}_P}}$ — a function returning the successors of a macrostate such that $\mathtt{Succ}^{\mathtt{Alg}_P}(H, M, a) = \{(M_1, \alpha_1), \ldots, (M_k, \alpha_k)\}$, where H is the set of all states of $\mathcal{A}$ reached over the same word, M is the $\mathtt{Alg}_P$'s

macrostate for the given partition block, a is the input symbol, and each (M_i, α_i) is a pair (*macrostate*, *set of colours*) such that M_i is a successor of M over a w.r.t. H and α_i is a set of colours on the edge from M to M_i (H helps to keep track of *new* runs coming into the partition block); and

- $\mathtt{Acc}^{\mathtt{Alg}_P} \in \mathbb{EL}(\mathtt{Colours}^{\mathtt{Alg}_P})$ — the acceptance condition.

Let $P_1, \ldots, P_n$ be a partitioning of $\mathcal{A}$ (w.l.o.g., we assume that $n > 0$), and $\mathtt{Alg}^1, \ldots, \mathtt{Alg}^n$ be a sequence of algorithms such that $\mathtt{Alg}^i$ is a partial complementation algorithm for P_i. Furthermore, let us define the following auxiliary *renumbering* function λ as $\lambda(c, j) = c + \sum_{i=1}^{j-1} |\mathtt{Colours}^{\mathtt{Alg}^i_{P_i}}|$, which is used to make the colours and acceptance conditions from the partial complementation algorithms disjoint. We also lift λ to sets of colours in the natural way, and also to $\mathbb{EL}$ conditions such that $\lambda(\varphi, j)$ has the same structure as φ but each atom $\mathsf{Inf}(c)$ is substituted with the atom $\mathsf{Inf}(\lambda(c, j))$ (and likewise for Fin atoms). The synchronous complementation algorithm then produces the TELA $\mathtt{ModCompl}\,(\mathtt{Alg}^1_{P_1}, \ldots, \mathtt{Alg}^n_{P_n}, \mathcal{A}) = (Q^C, \delta^C, I^C, \Gamma^C, \mathsf{p}^C, \mathsf{Acc}^C)$ with components defined as follows (we use $[S_i]_{i=1}^n$ to abbreviate $S_1 \times \cdots \times S_n$):

- $Q^C = 2^Q \times [\mathtt{T}^{\mathtt{Alg}^i_{P_i}}]_{i=1}^n$,
- $I^C = \{I\} \times [\mathtt{Init}^{\mathtt{Alg}^i_{P_i}}]_{i=1}^n$,
- $\Gamma^C = \{0, \ldots, \lambda(k^{\mathtt{Alg}^n_{P_n}} - 1, n)\}$,
- $\mathsf{Acc}^C = \bigwedge_{i=1}^n \lambda(\mathtt{Acc}^{\mathtt{Alg}^i_{P_i}}, i)$,[8] and
- δ^C and p^C are defined such that if

$$((M'_1, \alpha_1), \ldots, (M'_n, \alpha_n)) \in [\mathtt{Succ}^{\mathtt{Alg}^i_{P_i}}(H, M_i, a)]_{i=1}^n,$$

then δ^C contains the transition $t\colon (H, M_1, \ldots, M_n) \xrightarrow{a} (\delta(H, a), M'_1, \ldots, M'_n)$, coloured by $\mathsf{p}^C(t) = \bigcup\{\lambda(\alpha_i, i) \mid 1 \leq i \leq n\}$, and δ^C is the smallest such a set.

In order for $\mathtt{ModCompl}$ to be correct, the partial complementation algorithms need to satisfy certain properties, which we discuss below.

For a structural condition φ and a BA $\mathcal{B} = (Q, \delta, I, F)$, we define $\mathcal{B} \models_P \varphi$ iff $\mathcal{B} \models \varphi$, P is a partition block of $\mathcal{B}$, and $\mathcal{B}$ contains no accepting transitions outside P. We can now provide the correctness condition on $\mathtt{Alg}$.

Definition 1. *We say that* $\mathtt{Alg}$ *is* correct *if for each BA* $\mathcal{B}$ *and partition block* P *such that* $\mathcal{B} \models_P \varphi_{\mathtt{Alg}}$ *it holds that* $\mathcal{L}(\mathtt{ModCompl}\,(\mathtt{Alg}_P, \mathcal{B})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$.

The correctness of the synchronous algorithm (provided that each partial complementation algorithm is correct) is then established by Theorem 1.

Theorem 1. *Let* $\mathcal{A}$ *be a BA,* $P_1, \ldots, P_n$ *be a partitioning of* $\mathcal{A}$*, and* $\mathtt{Alg}^1, \ldots, \mathtt{Alg}^n$ *be a sequence of partial complementation algorithms such that* $\mathtt{Alg}^i$ *is* correct *for* P_i*. Then, we have* $\mathcal{L}(\mathtt{ModCompl}\,(\mathtt{Alg}^1_{P_1}, \ldots, \mathtt{Alg}^n_{P_n}, \mathcal{A})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.

4 Modular Complementation of Elevator Automata

In this section, we first give partial algorithms to complement partition blocks with only accepting IWCs (Sec. 4.1) and partition blocks with only DACs (Sec. 4.2). Then, in Sec. 4.3, we show that using our algorithm, the upper bound on the size of the complement of elevator BAs is in $\mathcal{O}(4^n)$, which is *exponentially better* than the known upper bound $\mathcal{O}(16^n)$ established in [29].

[8] If we drop the condition that $\mathcal{A}$ is complete, we also need to add an *accepting sink state* (representing the case for $H = \emptyset$) with self-loops over all symbols marked by a new colour (S), and enrich Acc^C with $\ldots \vee \mathsf{Inf}(\text{(S)})$.

4.1 Complementation of Inherently Weak Accepting Components

First, we introduce a partial algorithm MH with the condition $\varphi_{\texttt{MH}}$ specifying that all SCCs in the partition block P are *accepting* IWCs. Let P be a partition block of $\mathcal{A}$ such that $\mathcal{A}_P \models \varphi_{\texttt{MH}}$. Our proposed approach makes use of the Miyano-Hayashi construction [42]. Since in accepting IWCs, all runs are accepting, the idea of the construction is to accept words such that all runs over the words eventually leave P.

Therefore, we use a pair (C, B) of sets of states as a macrostate for complementing P. Intuitively, we use C to denote the set of all runs of $\mathcal{A}$ that are in P (C for "*check*"). The set $B \subseteq C$ represents the runs being inspected whether they leave P at some point (B for "*breakpoint*"). Initially, we let $C = I \cap P$ and also sample into breakpoint all runs in P, i.e., set $B = C$. Along reading an ω-word w, if all runs that have entered P eventually leave P, i.e., B becomes empty infinitely often, the complement language of P should contain w (when B becomes empty, we sample B with all runs from the current C). We formalize $\texttt{MH}_P$ as a partial procedure in the framework from Sec. 3.1 as follows:

- $\texttt{T}^{\texttt{MH}_P} = 2^P \times 2^P$, $\quad \texttt{Colours}^{\texttt{MH}_P} = \{\textcircled{0}\}$, $\quad \texttt{Init}^{\texttt{MH}_P} = \{(I \cap P, I \cap P)\}$,
- $\texttt{Acc}^{\texttt{MH}_P} = \mathsf{Inf}(\textcircled{0})$, and $\quad \texttt{Succ}^{\texttt{MH}_P}(H, (C, B), a) = \{((C', B'), \alpha)\}$ where
  - $C' = \delta(H, a) \cap P$,
  - $B' = \begin{cases} C' & \text{if } B^\star = \emptyset \text{ for } B^\star = \delta(B, a) \cap C', \\ B^\star & \text{otherwise, and} \end{cases}$
  - $\alpha = \begin{cases} \{\textcircled{0}\} & \text{if } B^\star = \emptyset \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$

We can see that checking whether w is accepted by the complement of P reduces to check whether B has been cleared infinitely often. Since every time when B becomes empty, we emit the colour $\textcircled{0}$, we have that w is not accepted by $\mathcal{A}$ within P if and only if $\textcircled{0}$ occurs infinitely often. Note that the transition function $\texttt{Succ}^{\texttt{MH}_P}$ is deterministic, i.e., there is exactly one successor.

Lemma 1. *The partial algorithm* MH *is correct.*

4.2 Complementation of Deterministic Accepting Components

In this section, we give a partial algorithm CSB with the condition $\varphi_{\texttt{CSB}}$ specifying that a partition block P consists of *DACs*. Let P be a partition block of $\mathcal{A}$ such that $\mathcal{A}_P \models \varphi_{\texttt{CSB}}$. Our approach is based on the NCSB family of algorithms [6,11,5,28] for complementing SDBAs, in particular the NCSB-MaxRank construction [28]. The algorithm utilizes the fact that runs in DACs are deterministic, i.e., they do not branch into new runs. Therefore, one can check that a run is non-accepting if there is a time point from which the run does not see accepting transitions any more. We call such a run that does not see accepting transitions any more *safe*. Then, an ω-word w is not accepted in P iff all runs over w in P either (i) leave P or (ii) eventually become safe.

For checking point (i), we can use a similar technique as in algorithm MH, i.e., use a pair (C, B). Moreover, to be able to check point (ii), we also use the set S that contains runs that are supposed to be *safe*, resulting in macrostates of the form (C, S, B)[9]. To make sure that all runs are deterministic, we will use δ_{SCC} instead of δ when computing the successors of S and B since there may be nondeterministic jumps between different DACs in P; we will not miss any run in P since if a run moves between DACs of P, it

[9] In contrast to MH, here we use $C \cup S$ rather than C to keep track of all runs in P.

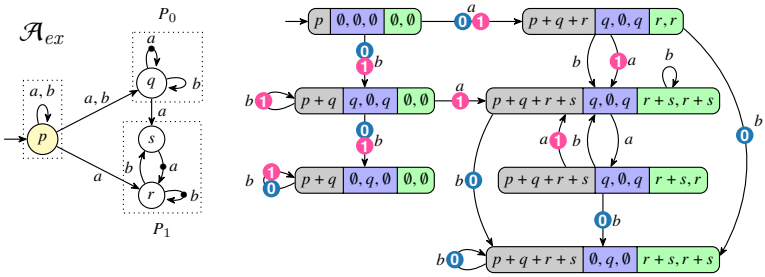


Fig. 1: Left: BA $\mathcal{A}_{ex}$ (dots represent accepting transitions). Right: the outcome of ModCompl $(\mathtt{CSB}_{P_0}, \mathtt{MH}_{P_1}, \mathcal{A}_{ex})$ with Acc: $\mathsf{Inf}(\textcircled{0}) \wedge \mathsf{Inf}(\textcircled{1})$. States are given as $(H, (C_0, S_0, B_0), (C_1, B_1))$; to avoid too many braces, sets are given as sums.

can be seen as the run leaving P and a new run entering P. Since a run eventually stays in one SCC, this guarantees that the run will not be missed.

We formalize $\mathtt{CSB}_P$ in the top-level framework as follows:

- $\mathtt{T}^{\mathtt{CSB}_P} = 2^P \times 2^P \times 2^P$, $\mathtt{Init}^{\mathtt{CSB}_P} = \{(I \cap P, \emptyset, I \cap P)\}$,
- $\mathtt{Colours}^{\mathtt{CSB}_P} = \{\textcircled{0}\}$, $\mathtt{Acc}^{\mathtt{CSB}_P} = \mathsf{Inf}(\textcircled{0})$, and
- $\mathtt{Succ}^{\mathtt{CSB}_P}(H, (C, S, B), a) = U$ such that
  - if $\delta_F(S, a) \neq \emptyset$, then $U = \emptyset$ (Runs in S must be *safe*),
  - otherwise U contains $((C', S', B'), c)$ where
    - $S' = \delta_{\mathrm{SCC}}(S, a) \cap P$, $C' = (\delta(H, a) \cap P) \setminus S'$,
    - $B' = \begin{cases} C' & \text{if } B^\star = \emptyset \text{ for } B^\star = \delta_{\mathrm{SCC}}(B, a), \\ B^\star & \text{otherwise, and} \end{cases}$
    - $c = \begin{cases} \{\textcircled{0}\} & \text{if } B^\star = \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$

    Moreover, in the case $\delta_F(B, a) = \emptyset$, then U also contains $((C'', S'', C''), \{\textcircled{0}\})$ where $S'' = S' \cup B'$ and $C'' = C' \setminus S''$.

Intuitively, when $\delta_F(B, a) \cap \delta_{\mathrm{SCC}}(B, a) = \emptyset$, we make the following guess: (i) either the runs in B all become safe (we move them to S) or (ii) there might be some unsafe runs (we keep them in B). Since the runs in B are deterministic, the number of tracked runs in B will not increase. Moreover, if all runs in B are eventually safe, we are guaranteed to move all of them to S at the right time point, e.g., the maximal time point where all runs are safe since the number of runs is finite.

As mentioned above, w is not accepted within P iff all runs over w either (i) leave P or (ii) become safe. In the context of the presented algorithm, this corresponds to (i) B becoming empty infinitely often and (ii) $\delta_F(S, a)$ never seeing an accepting transition. Then we only need to check if there exists an infinite sequence of macrostates $\hat{\rho} = (C_0, S_0, B_0) \ldots$ that emits $\textcircled{0}$ infinitely often.

Lemma 2. *The partial algorithm* CSB *is correct.*

It is worth noting that when the given partition block P contains all DACs of $\mathcal{A}$, we can still use the construction above, while the construction in [28] only works on SDBAs.

Example 1. In Fig. 1, we give an example of the run of our algorithm on the BA $\mathcal{A}_{ex}$. The BA contains three SCCs, one of them (the one containing p) non-accepting (therefore,

it does not need to occur in any partition block). The partition block P_0 contains a single DAC, so we can use algorithm CSB, and the partition block P_1 contains a single accepting IWC, so we can use MH. The resulting ModCompl $(\mathtt{CSB}_{P_0}, \mathtt{MH}_{P_1}, \mathcal{A}_{ex})$ uses two colours, ⓪ from CSB and ① from MH. The acceptance condition is $\mathsf{Inf}(⓪) \wedge \mathsf{Inf}(①)$. □

4.3 Upper-bound for Elevator Automata Complementation

We now give an upper bound on the size of the complement generated by our algorithm for elevator automata, which significantly improves the best previously known upper bound of $O(16^n)$ [29] to $O(4^n)$, the same as for SDBAs, which are a strict subclass of elevator automata [6] (we note that this upper bound cannot be obtained by a determinization-based algorithm, since determinization of SDBAs is in $\Omega(n!)$ [17,40]).

Theorem 2. *Let $\mathcal{A}$ be an elevator automaton with n states. Then there exists a BA with $O(4^n)$ states accepting the complement of $\mathcal{L}(\mathcal{A})$.*

Proof (Sketch). Let Q_W be all states in accepting IWCs, Q_D be all states in DACs, and Q_N be the remaining states, i.e., $Q = Q_W \uplus Q_D \uplus Q_N$. We make two partition blocks: $P_0 = Q_W$ and $P_1 = Q_D$ and use MH and CSB respectively as the partial algorithms, with macrostates of the form $(H, (C_0, B_0), (C_1, S_1, B_1))$. For each state $q_N \in Q_N$, there are two options: either $q_N \notin H$ or $q_N \in H$. For each state $q_W \in Q_W$, there are three options: (i) $q_W \notin C_0$, (ii) $q_W \in C_0 \setminus B_0$, or (iii) $q_W \in C_0 \cap B_0$. Finally, for each $q_D \in Q_D$, there are four options: (i) $q_D \notin C_1 \cup S_1$, (ii) $q_D \in S_1$, (iii) $q_D \in C_1 \setminus B_1$, or (iv) $q_D \in C_1 \cap B_1$. Therefore, the total number of macrostates is $2 \cdot 2^{|Q_N|} \cdot 3^{|Q_W|} \cdot 4^{|Q_D|} \in O(4^n)$ where the initial factor 2 is due to degeneralization from two to one colour (the two colours can actually be avoided by using our shared breakpoint optimization from Sec. 5.4). □

5 Optimizations of the Modular Construction

In this section, we propose optimizations of the basic modular algorithm. In Sec. 5.1, we give a partial algorithm to complement initial partition blocks with DACs. Further, in Sec. 5.2, we propose the postponed construction allowing to use automata reduction on intermediate results. In Sec. 5.3, we propose the round-robin algorithm alleviating the problem with the explosion of the size of the Cartesian product of partial successors. In Sec. 5.4, we provide an optimization for partial algorithms that are based on the breakpoint construction, and, finally, in Sec. 5.5, we show how to employ simulation to decrease the size of macrostates in the synchronous construction.

5.1 Complementation of Initial Deterministic Partition Blocks

Our first optimization is an algorithm CoB for a subclass of partition blocks containing DACs. In particular, the condition $\varphi_{\mathtt{CoB}}$ specifies that the partition block P is deterministic and can be reached only deterministically in $\mathcal{A}$ (i.e., $\mathcal{A}_P$ after removing redundant states is deterministic). Then, we say that P is an *initial deterministic* partition block. The algorithm is based on complementation of deterministic BAs into co-Büchi automata.

The algorithm $\mathtt{CoB}_P$ is formalized below:

- $\mathtt{T}^{\mathtt{CoB}_P} = P \cup \{\emptyset\}$, $\mathtt{Init}^{\mathtt{CoB}_P} = I \cap P$, $\mathtt{Colours}^{\mathtt{CoB}_P} = \{⓪\}$, $\mathtt{Acc}^{\mathtt{CoB}_P} = \mathsf{Fin}(⓪)$,

– $\texttt{Succ}^{\texttt{CoB}_P}(H, q, a) = \{(q', \alpha)\}$ where
  - $q' = \begin{cases} r & \text{if } \delta(H, a) \cap P = \{r\} \text{ and} \\ \emptyset & \text{otherwise,} \end{cases}$
  - $\alpha = \begin{cases} \{\textcircled{0}\} & \text{if } q \xrightarrow{a} q' \in F \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$

Intuitively, all runs reach P deterministically, which means that over a word w, at most one run can reach P (so $|\texttt{Init}^{\texttt{CoB}_P}| = 1$). Thus, we have $|\delta(H, w_j) \cap P| = 1$ for some $j \geq 0$ if there is a run over w to P, corresponding to $\delta(H, a) \cap P = \{r\}$ in the construction. To check whether w is not accepted in P, we only need to check whether the run from $r \in P$ over w visits accepting transitions only finitely often. We give an example of complementation of a BA containing an initial deterministic partition block in [27].

Lemma 3. *The partial algorithm* CoB *is correct.*

5.2 Postponed Construction

The modular synchronous construction from Sec. 3.1 utilizes the assumption that in the simultaneous construction of successors for each partition block over a, if one partial macrostate M_i does not have a successor over a, then there will be no successor of the $(H, M_1, \ldots, M_n)$ macrostate in δ^C as well. This is useful, e.g., for inclusion testing, where it is not necessary to generate the whole complement. On the other hand, if we need to generate the whole automaton, a drawback of the proposed modular construction is that each partial complementation algorithm itself may generate a lot of useless states. In this section, we propose the *postponed construction*, which complements the partition blocks (with their surrounding) independently and later combines the intermediate results to obtain the complement automaton for $\mathcal{A}$. The main advantage of the postponed construction is that one can apply automata reduction (e.g., based on removing useless states or using simulation [13,18,1,9]) to decrease the size of the intermediate automata.

In the postponed construction, we use product-based BA intersection operation (i.e., for two TELAs $\mathcal{B}_1$ and $\mathcal{B}_2$, a product automaton $\mathcal{B}_1 \cap \mathcal{B}_2$ satisfying $\mathcal{L}(\mathcal{B}_1 \cap \mathcal{B}_2) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$[10]). Further, we employ a function Red performing some language-preserving reduction of an input TELA. Then, the postponed construction for an elevator automaton $\mathcal{A}$ with a partitioning $P_1, \ldots, P_n$ and a sequence $\texttt{Alg}^1, \ldots, \texttt{Alg}^n$ where $\texttt{Alg}^i$ is a partial complementation algorithm for P_i, is defined as follows:

$$\text{PostpCompl}\,(\texttt{Alg}^1_{P_1}, \ldots, \texttt{Alg}^n_{P_n}, \mathcal{A}) = \bigcap_{i=1}^{n} \texttt{Red}\left(\text{ModCompl}\,(\texttt{Alg}^i_{P_i}, \mathcal{A}_{P_i})\right). \qquad (2)$$

The correctness of the construction is then summarized by the following theorem.

Theorem 3. *Let $\mathcal{A}$ be a BA, $P_1, \ldots, P_n$ be a partitioning of $\mathcal{A}$, and $\texttt{Alg}^1, \ldots, \texttt{Alg}^n$ be a sequence of partial complementation algorithms such that $\texttt{Alg}^i$ is* correct *for P_i. Then, $\mathcal{L}(\mathit{PostpCompl}\,(\texttt{Alg}^1_{P_1}, \ldots, \texttt{Alg}^n_{P_n}, \mathcal{A})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

5.3 Round-Robin Algorithm

The proposed basic synchronous approach from Sec. 3.1 may suffer from the combinatorial explosion because the successors of a macrostate are given by the Cartesian product of all successors of the partial macrostates. To alleviate this explosion, we propose

[10] Alternatively, one might also avoid the product and generate linear-sized *alternating* TELA, but working with those is usually much harder and not used in practice.

a *round-robin* top-level algorithm. Intuitively, the round-robin algorithm actively tracks runs in only one partial complementation algorithm at a time (while other algorithms stay passive). The algorithm periodically changes the active algorithm to avoid starvation (the decision to leave the active state is, however, fully directed by the partial complementation algorithm). This can alleviate an explosion in the number of successors for algorithms that generate more than one successor (e.g., for rank-based algorithms where one needs to make a nondeterministic choice of decreasing ranks of states in order to be able to accept [34,21,48,10,24,29]; such a choice needs to be made only in the active phase while in the passive phase, the construction just needs to make sure that the run is consistent with the given ranking, which can be done deterministically).

The round-robin algorithm works on the level of *partial complementation round-robin algorithms*. Each instance of the partial algorithm provides *passive types* to represent partial macrostates that are passive and *active types* to represent currently active partial macrostates. In contrast to the basic partial complementation algorithms from Sec. 3.1, which provide only a single successor function, the round-robin partial algorithms provide several variants of them. In particular, `SuccPass` returns (passive) successors of a passive partial macrostate, `Lift` gives all possible active counterparts of a passive macrostate, and `SuccAct` returns successors of an active partial macrostate. If `SuccAct` returns a partial macrostate of the passive type, the round-robin algorithm promotes the next partial algorithm to be the active one. For instance, in the round-robin version of `CSB`, the passive type does not contain the breakpoint and only checks that safe runs stay safe, so it is deterministic. Due to space limitations, we give a formal definition and more details about the round-robin algorithm in [27].

5.4 Shared Breakpoint

The partial complementation algorithms `CSB` and `MH` (and later `RNK` defined in Sec. 6) use a breakpoint to check whether the runs under inspection are accepting or not. As an optimization, we consider merging of breakpoints of several algorithms and keeping only a single breakpoint for all supported algorithms. The top-level algorithm then needs to manage only one breakpoint and emit a colour only if this sole breakpoint becomes empty. This may lead to a smaller number of generated macrostates since we synchronize the breakpoint sampling among several algorithms. The second benefit is that this allows us to generate fewer colours (in the case of elevator automata complemented using algorithms `CSB` and `MH`, we get only one colour).

5.5 Simulation Pruning

Our construction can be further optimized by a simulation (or other compatible) relation for pruning macrostates.[11] A simulation is, broadly speaking, a relation $\preccurlyeq \subseteq Q \times Q$ implying language inclusion of states, i.e., $\forall p, q \in Q\colon p \preccurlyeq q \implies \mathcal{L}(\mathcal{A}[p]) \subseteq \mathcal{L}(\mathcal{A}[q])$. Intuitively, our optimization allows to remove a state p from a macrostate M if there is also a state q in M such that (i) $p \preccurlyeq q$, (ii) p is not reachable from q, and (iii) p is smaller than q in an arbitrary total order over Q (this serves as a tie-breaker for

[11] This optimization can be seen as a generalization of the simulation-based pruning techniques that appeared, e.g., in [41,28] in the context of concrete determinization/complementation procedures. Here, we generalize the technique to all procedures that are based on run tracking.

simulation-equivalent mutually unreachable states). The reason why p can be removed is that its behaviour can be completely mimicked by q. In our construction, we can then, roughly speaking, replace each call to the functions $\delta(U, a)$ and $\delta_F(U, a)$, for a set of states U, by $pr(\delta(U, a))$ and $pr(\delta_F(U, a))$ respectively in each partial complementation algorithm, as well as in the top-level algorithm, where $pr(S)$ is obtained from S by pruning all eligible states. The details are provided in [27].

6 Modular Complementation of Non-Elevator Automata

A non-elevator automaton $\mathcal{A}$ contains at least one NAC, besides possibly other IWCs or DACs. To complement $\mathcal{A}$ in a modular way, we apply the techniques seen in Sec. 4 to its DACs and IWCs, while for its NACs we resort to a general complementation algorithm `Alg`. In theory, rank- [34], slice- [32], Ramsey- [50], subset-tuple- [2], and determinization- [46] based complementation algorithms adapted to work on a single partition block instead of the whole automaton are all valid instantiations of `Alg`. Below, we give a high-level description of two such algorithms: rank- and determinization-based.

Rank-based partial complementation algorithm. Working on each NAC independently benefits the complementation algorithm even if the input BA contains only NACs. For instance, in rank-based algorithms [34,21,48,33,10,24,29], the fact whether all runs of $\mathcal{A}$ over a given ω-word w are non-accepting is determined by *ranks* of states, given by the so-called *ranking functions*. A ranking function is a (partial) function from Q to ω. The main idea of rank-based algorithms is the following: (i) every run is initially nondeterministically assigned a rank, (ii) ranks can only decrease along a run, (iii) ranks need to be even every time a run visits an accepting transition, and (iv) the complement automaton accepts iff all runs eventually get trapped in odd ranks[12]. In the standard rank-based procedure, the initial assignment of ranks to states in (i) is a function $Q \rightharpoonup \{0, \ldots, 2n-1\}$ for $n = |Q|$. Using our framework, we can, however, significantly restrict the considered ranks in a partition block P to only $P \rightharpoonup \{0, \ldots, 2m-1\}$ for $m = |P|$ (here, it makes sense to use partition blocks consisting of single SCCs). One can further reduce the considered ranks using the techniques introduced in, e.g., [24,29].

In order to adapt the rank-based construction as a partial complementation algorithm `RNK` in our framework, we need to extend the ranking functions by a fresh "box state" $\blacksquare$ representing states outside the partition block. The ranking function then uses $\blacksquare$ to represent ranks of runs newly coming into the partition block. The box-extension also requires to change the transition in a way that $\blacksquare$ always represents reachable states from the outside. We provide the details of the construction, which includes the MaxRank optimization from [24], in [27].

Determinization-based partial complementation algorithm. In [52,29] we can see that determinization-based complementation is also a good instantiation of `Alg` in practice, so, we also consider the standard Safra-Piterman determinization [46,43,45] as a choice of `Alg` for complementing NACs. Determinization-based algorithms use a layered subset construction to organize all runs over an ω-word w. The idea is to identify a subset $S \subseteq H$ of reachable states that occur infinitely often along reading w such that between every two occurrences of S, we have that (i) every state in the second occurrence of S can be reached

[12] Since we focus on intuition here, we use runs rather than the directed acyclic graphs of runs.

Table 1: Statistics for our experiments. The column **unsolved** classifies unsolved instances by the form *timeouts : out of memory : other failures*. For the cases of VBS we provide just the number of unsolved cases. The columns **states** and **runtime** provide *mean : median* of the number of states and runtime, respectively.

tool	solved	unsolved	states	runtime
Kofola$_S$	39,738	89 : 10 : 0	76 : 3	0.32 : 0.03
Kofola$_P$	39,750	76 : 11 : 0	86 : 3	0.41 : 0.03
VBS$_+$	39,834	3	78 : 3	0.05 : 0.01
VBS$_-$	39,834	3	96 : 3	0.05 : 0.01

tool	solved	unsolved	states	runtime
COLA	39,814	21 : 0 : 2	80 : 3	0.17 : 0.02
Ranker	38,837	61 : 939 : 0	45 : 4	3.31 : 0.01
Seminator	39,026	238 : 573 : 0	247 : 3	1.98 : 0.03
Spot	39,827	8 : 0 : 2	160 : 4	0.08 : 0.02

by a state in the first occurrence of S and (ii) every state in the second occurrence is reached by a state in the first occurrence while seeing an accepting transition. According to König's lemma, there must then be an accepting run of $\mathcal{A}$ over w.

The construction initially maintains only one set H: the set of reachable states. Since S as defined does not necessarily need to be H, every time there are runs visiting accepting transitions, we create a new subset C for those runs and remember which subset C is coming from. This way, we actually organize the current states of all runs into a tree structure and do subset construction in parallel for the sets in each tree node. If we find a tree node whose labelled subset, say S', is equal to the union of states in its children, we know the set S' satisfies the condition above and we remove all its child nodes and emit a good event. If such good event happens infinitely often, it means that S' also occurs infinitely often. So in complementation, we only need to make sure those good events only happen for finitely many times. Working on each NAC separately also benefits the determinization-based approach since the number of possible trees will be less with smaller number of reachable states. Following the idea of [37], to adapt for the construction as the partial complementation algorithm, we put all the newly coming runs from other partition blocks in a newly created node without a parent node. In this way, we actually maintain a forest of trees for the partial complementation construction. We denote the determinization-based construction as `DET`; cf. [37] for details.

7 Experimental Evaluation

To evaluate the proposed approach, we implemented it in a prototype tool Kofola [25] (written in C++) built on top of Spot [16] and compared it against COLA [37], Ranker [28] (v. 2), Seminator [5] (v. 2.0), and Spot [15,16] (v. 2.10.6), which are the state of the art in BA complementation [29,28,37]. Due to space restrictions, we give results for only two instantiations of our framework: Kofola$_S$ and Kofola$_P$. Both instantiations use `MH` for IWCs, `CSB` for DACs, and `DET` for NACs. The partitioning selection algorithm merges all IWCs into one partition block, all DACs into one partition block, and keeps all NACs separate. Simulation-based pruning from Sec. 5.5 is turned on, and round-robin from Sec. 5.3 is turned off (since the selected algorithms are quite deterministic). Kofola$_S$ employs the *synchronous* and Kofola$_P$ employs the *postponed* strategy. We also consider the Virtual Best Solver (VBS), i.e., a virtual tool that would choose the best solver for each single benchmark among all tools (VBS$_+$) and among all tools except both versions of Kofola (VBS$_-$). We ran our experiments on an Ubuntu 20.04.4 LTS system running on a desktop machine with 16 GiB RAM and an

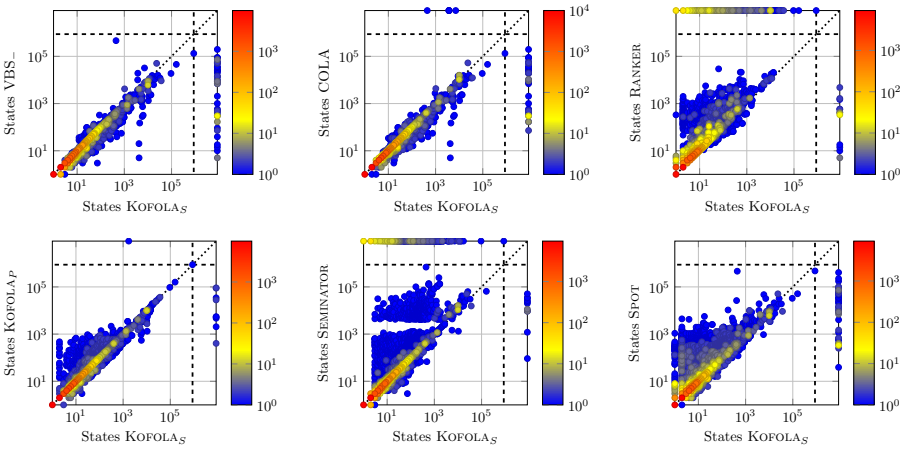


Fig. 2: Scatter plots comparing the numbers of states generated by the tools.

Intel 3.6 GHz i7-4790 CPU. To constrain and collect statistics about the executions of the tools, we used BenchExec [3] and imposed a memory limit of 12 GiB and a timeout of 10 minutes; we used Spot to cross-validate the equivalence of the automata generated by the different tools. An artifact reproducing our experiments is available as [26].

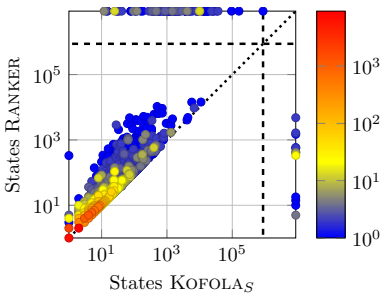
As our data set, we used 39,837 BAs from the automata-benchmarks repository [36] (used before by, e.g., [29,28,37]), which contains BAs from the following sources: (i) randomly generated BAs used in [52] (21,876 BAs), (ii) BAs obtained from LTL formulae from the literature and randomly generated LTL formulae [5] (3,442 BAs), (iii) BAs obtained from Ultimate Automizer [11] (915 BAs), (iv) BAs obtained from the solver for first-order logic over Sturmian words Pecan [31] (13,216 BAs), (v) BAs obtained from an S1S solver [23] (370 BAs), and (vi) BAs from LTL to SDBA translation [49] (18 BAs). From these BAs, 23,850 are deterministic, 6,147 are SDBAs (but not deterministic), 4,105 are elevator (but not SDBAs), and 5,735 are the rest.

In Table 1 we present an overview of the outcomes. Despite being a prototype, Kofola can already complement a large portion of the input automata, with very few cases that can be complemented successfully only by Spot or COLA. Regarding the mean number of states, Kofola$_S$ has the **least mean value** from all tools (except Ranker, which, however, had 1,000 unsolved cases) Moreover, Kofola **significantly decreased the mean number of states** when included into the VBS: from 96 to 78! We consider this to be a strong validation of the usefulness of our approach. Regarding the runtime, both versions of Kofola are rather similar; Kofola is just slightly slower than Spot and COLA but much faster than both Ranker and Seminator (cf. [27]).

In Fig. 2 we present a comparison of the number of states generated by Kofola$_S$ and other tools; we omit VBS$_+$ since the corresponding plot can be derived from the one for VBS$_-$ (since Ranker and Seminator only output BAs, we compare the sizes of outputs transformed into BAs for all tools to be fair). In the plots, the number of benchmarks represented by each mark is given by its colour; a mark above the diagonal means that Kofola$_S$ generated a BA smaller than the other tool while a mark on the top border means that the other tool failed while Kofola$_S$ succeeded, and symmetrically for the

bottom part and the right-hand border. Dashed lines represent the maximum number of states generated by one of the tools in the plot, axes are logarithmic.

From the results, Kofola$_S$ clearly dominates state-of-the-art tools that are not based on SCC decomposition (Ranker, Spot, Seminator). The outputs are quite comparable to COLA, which also uses SCC decomposition and can be seen as an instantiation of our framework. This supports our intuition that working on the single SCCs helps in reducing the size of the final automaton, confirming the validity of our modular mix-and-match Büchi complementation approach. Lastly, in the figure in the right we compare our algorithm for elevator automata with the one in Ranker (the only other tool with a dedicated algorithm for this subclass). Our new algorithm clearly dominates the one in Ranker.



8 Related Work

To the best of our knowledge, we provide the *first general framework* where one can plug-in different BA complementation algorithms while taking advantage of the specific structure of SCCs. We will discuss the difference between our work and the literature.

The breakpoint construction [42] was designed to complement BAs with only IWCs, while our construction treats it as a partial complementation procedure for IWCs and differs in the need to handle incoming states from other partition blocks. The NCSB family of algorithms [6,11,5,28] for SDBAs do not work when there are nondeterministic jumps between DACs; they can, however, be adapted as partial procedures for complementing DACs in our framework, cf. Sec. 4.2. In [29], a deelevation-based procedure is applied to elevator automata to obtain BAs with a fixed maximum rank of 3, for which a rank-based construction produces a result of the size in $O(16^n)$. In our work, we exploit the structure of the SCCs much more to obtain an exponentially better upper bound of $O(4^n)$ (the same as for SDBAs). The upper bound $O(4^n)$ for complementing unambiguous BAs was established in [39], which is orthogonal to our work, but seems to be possible to incorporate into our framework in the future.

There is a huge body of work on complementation of general BAs [8,50,7,34,21,22,10,24,29,48,2,46,43,45,5,52,32,53,19,20]; all of them work on the whole graph structure of the input BAs. Our framework is general enough to allow including all of them as partial complementation procedures for NACs. On the contrary, our framework does not directly allow (at least in the synchronous strategy) to use algorithms that *do not* work on the structure of the input BA, such as the learning-based complementation algorithm from [38]. The recent determinization algorithm from [37], which serves as our inspiration, also handles SCCs separately (it can actually be seen as an instantiation of our framework). Our current algorithm is, however, more flexible, allowing to mix-and-match various constructions, keep SCCs separate or merge them into partition blocks, and allows to obtain the complexity $O(4^n)$, while [37] only allowed $O(n!)$ (which is tight since SDBA determinization is in $\Omega(n!)$ [17,40]).

Regarding the tool Spot [15,16], it should not be perceived as a single complementation algorithm. Instead, Spot should be seen as a highly engineered platform

utilizing breakpoint construction for inherently weak BAs, NCSB [6,11] for SDBAs, and determinization-based complementation [46,43,45] for general BAs, while using many other heuristics along the way. Seminator uses semi-determinization [14,4,5] to make sure the input is an SDBA and then uses NCSB [6,11] to compute the complement.

9 Conclusion and Future Work

We have proposed a general framework for BA complementation where one can plug-in different partial complementation procedures for SCCs by taking advantage of their specific structure. Our framework not only obtains an exponentially better upper bound for elevator automata, but also complements existing approaches well. As shown by the experimental results (especially for the VBS), our framework significantly improves the current portfolio of complementation algorithms.

We believe that our framework is an ideal testbed for experimenting with different BA complementation algorithms, e.g., for the following two reasons: (i) One can develop an efficient complementation algorithm that only works for a quite restricted sub-class of BAs (such as the algorithm for initial deterministic SCCs that we showed in Sec. 5.1) and the framework can leverage it for complementation of all BAs that contain such a substructure. (ii) When one tries to improve a general complementation algorithm, they can focus on complementation of the structurally hard SCCs (mainly the nondeterministic accepting SCCs) and do not need to look for heuristics that would improve the algorithm if there were some easier substructure present in the input BA (as was done, e.g., in [29]). From how the framework is defined, it immediately offers opportunities for being used for on-the-fly BA *language inclusion* testing, leveraging the partial complementation procedures present. Finally, we believe that the framework also enables new directions for future research by developing smart ways, probably based on machine learning, of selecting which partial complementation procedure should be used for which SCC, based on their features. In future, we want to incorporate other algorithms for complementation of NACs, and identify properties of SCCs that allow to use more efficient algorithms (such as unambiguous NACs [39]). Moreover, it seems that generalizing the Delayed optimization from [24] on the top-level algorithm could also help reduce the state space.

Acknowledgements. We thank the reviewers for their useful remarks that helped us improve the quality of the paper and Alexandre Duret-Lutz for sharing a Ti*k*Z package for beautiful automata. This work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (grant no. XDA0320000); the National Natural Science Foundation of China (grants no. 62102407 and 61836005); the CAS Project for Young Scientists in Basic Research (grant no. YSBR-040); the Engineering and Physical Sciences Research Council (grant no. EP/X021513/1); the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme; the Czech Science Foundation project GA23-07565S; and the FIT BUT internal project FIT-S-23-8151.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant no. 101008233.

Data Availability Statement. An environment with the tools and data used for the experimental evaluation in the current study is available in the following Zenodo repository: https://doi.org/10.5281/zenodo.7505210.

References

1. Abdulla, P.A., Chen, Y., Holík, L., Vojnar, T.: Mediating for reduction (on minimizing alternating büchi automata). Theor. Comput. Sci. **552**, 26–43 (2014). https://doi.org/10.1016/j.tcs.2014.08.003, https://doi.org/10.1016/j.tcs.2014.08.003
2. Allred, J.D., Ultes-Nitsche, U.: A simple and optimal complementation algorithm for Büchi automata. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 46–55. ACM (2018). https://doi.org/10.1145/3209108.3209138, https://doi.org/10.1145/3209108.3209138
3. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y, https://doi.org/10.1007/s10009-017-0469-y
4. Blahoudek, F., Duret-Lutz, A., Klokocka, M., Kretínský, M., Strejcek, J.: Seminator: A tool for semi-determinization of omega-automata. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 356–367. EasyChair (2017). https://doi.org/10.29007/k5nl, https://doi.org/10.29007/k5nl
5. Blahoudek, F., Duret-Lutz, A., Strejcek, J.: Seminator 2 can complement generalized Büchi automata via improved semi-determinization. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 15–27. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_2, https://doi.org/10.1007/978-3-030-53291-8_2
6. Blahoudek, F., Heizmann, M., Schewe, S., Strejček, J., Tsai, M.: Complementing semi-deterministic Büchi automata. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 770–787. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_49, https://doi.org/10.1007/978-3-662-49674-9_49
7. Breuers, S., Löding, C., Olschewski, J.: Improved Ramsey-based Büchi complementation. In: Birkedal, L. (ed.) Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 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. Lecture Notes in Computer Science, vol. 7213, pp. 150–164. Springer (2012). https://doi.org/10.1007/978-3-642-28729-9_10, https://doi.org/10.1007/978-3-642-28729-9_10
8. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Mac Lane, S., Siefkes, D. (eds.) The Collected Works of J. Richard Büchi, pp. 425–435. Springer (1990). https://doi.org/10.1007/978-1-4613-8928-6_23, https://doi.org/10.1007/978-1-4613-8928-6_23
9. Bustan, D., Grumberg, O.: Simulation-based Minimization. ACM Transactions on Computational Logic **4**(2), 181–206 (2003)
10. Chen, Y., Havlena, V., Lengál, O.: Simulations in rank-based Büchi automata complementation. In: Lin, A.W. (ed.) Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11893, pp. 447–467. Springer (2019). https://doi.org/10.1007/978-3-030-34175-6_23, https://doi.org/10.1007/978-3-030-34175-6_23

11. Chen, Y., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., Zhang, L.: Advanced automata-based algorithms for program termination checking. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 135–150. ACM (2018). https://doi.org/10.1145/3192366.3192405, https://doi.org/10.1145/3192366.3192405
12. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8414, pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15, https://doi.org/10.1007/978-3-642-54792-8_15
13. Clemente, L., Mayr, R.: Efficient reduction of nondeterministic automata with application to language inclusion testing. Log. Methods Comput. Sci. **15**(1) (2019). https://doi.org/10.23638/LMCS-15(1:12)2019, https://doi.org/10.23638/LMCS-15(1:12)2019
14. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 338–345. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21950, https://doi.org/10.1109/SFCS.1988.21950
15. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 122–129 (2016). https://doi.org/10.1007/978-3-319-46520-3_8, https://doi.org/10.1007/978-3-319-46520-3_8
16. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9, https://doi.org/10.1007/978-3-031-13188-2_9
17. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From LTL and limit-deterministic Büchi automata to deterministic parity automata. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 426–442 (2017). https://doi.org/10.1007/978-3-662-54577-5_25, https://doi.org/10.1007/978-3-662-54577-5_25
18. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Büchi automata. SIAM J. Comput. **34**(5), 1159–1175 (2005). https://doi.org/10.1137/S0097539703420675, https://doi.org/10.1137/S0097539703420675
19. Fogarty, S., Kupferman, O., Vardi, M.Y., Wilke, T.: Profile trees for Büchi word automata, with application to determinization. Inf. Comput. **245**, 136–151 (2015). https://doi.org/10.1016/j.ic.2014.12.021, https://doi.org/10.1016/j.ic.2014.12.021
20. Fogarty, S., Kupferman, O., Wilke, T., Vardi, M.Y.: Unifying Büchi complementation constructions. Log. Methods Comput. Sci. **9**(1) (2013). https://doi.org/10.2168/LMCS-9(1:13)2013, https://doi.org/10.2168/LMCS-9(1:13)2013

21. Friedgut, E., Kupferman, O., Vardi, M.Y.: Büchi complementation made tighter. Int. J. Found. Comput. Sci. **17**(4), 851–868 (2006). https://doi.org/10.1142/S0129054106004145, https://doi.org/10.1142/S0129054106004145
22. Gurumurthy, S., Kupferman, O., Somenzi, F., Vardi, M.Y.: On complementing nondeterministic Büchi automata. In: Geist, D., Tronci, E. (eds.) Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2860, pp. 96–110. Springer (2003). https://doi.org/10.1007/978-3-540-39724-3_10, https://doi.org/10.1007/978-3-540-39724-3_10
23. Havlena, V., Lengál, O., Smahlíková, B.: Deciding S1S: down the rabbit hole and through the looking glass. In: Echihabi, K., Meyer, R. (eds.) Networked Systems - 9th International Conference, NETYS 2021, Virtual Event, May 19-21, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12754, pp. 215–222. Springer (2021). https://doi.org/10.1007/978-3-030-91014-3_15, https://doi.org/10.1007/978-3-030-91014-3_15
24. Havlena, V., Lengál, O.: Reducing (to) the ranks: Efficient rank-based Büchi automata complementation. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 2:1–2:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.2, https://doi.org/10.4230/LIPIcs.CONCUR.2021.2
25. Havlena, V., Lengál, O., Li, Y., Šmahlíková, B., Turrini, A.: Kofola (2022), https://github.com/VeriFIT/kofola
26. Havlena, V., Lengál, O., Li, Y., Šmahlíková, B., Turrini, A.: Artifact for the TACAS'23 paper "Modular Mix-and-Match Complementation of Büchi Automata" (Jan 2023). https://doi.org/10.5281/zenodo.7505210, https://doi.org/10.5281/zenodo.7505210
27. Havlena, V., Lengál, O., Li, Y., Šmahlíková, B., Turrini, A.: Modular mix-and-match complementation of Büchi automata (technical report). CoRR **abs/2301.01890** (2023). https://doi.org/10.48550/arXiv.2301.01890, https://doi.org/10.48550/arXiv.2301.01890
28. Havlena, V., Lengál, O., Šmahlíková, B.: Complementing Büchi automata with Ranker. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 188–201. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_10, https://doi.org/10.1007/978-3-031-13188-2_10
29. Havlena, V., Lengál, O., Šmahlíková, B.: Sky is not the limit: Tighter rank bounds for elevator automata in Büchi automata complementation. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 118–136. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_7, https://doi.org/10.1007/978-3-030-99527-0_7
30. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) 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. 797–813. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_53, https://doi.org/10.1007/978-3-319-08867-9_53
31. Hieronymi, P., Ma, D., Oei, R., Schaeffer, L., Schulz, C., Shallit, J.O.: Decidability for Sturmian words. In: Manea, F., Simpson, A. (eds.) 30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference). LIPIcs, vol. 216, pp. 24:1–24:23. Schloss Dagstuhl - Leibniz-Zentrum für

Informatik (2022). https://doi.org/10.4230/LIPIcs.CSL.2022.24, https://doi.org/10.4230/LIPIcs.CSL.2022.24

32. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games. Lecture Notes in Computer Science, vol. 5125, pp. 724–735. Springer (2008). https://doi.org/10.1007/978-3-540-70575-8_59, https://doi.org/10.1007/978-3-540-70575-8_59
33. Karmarkar, H., Chakraborty, S.: On minimal odd rankings for Büchi complementation. In: Liu, Z., Ravn, A.P. (eds.) Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5799, pp. 228–243. Springer (2009). https://doi.org/10.1007/978-3-642-04761-9_18, https://doi.org/10.1007/978-3-642-04761-9_18
34. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(3), 408–429 (2001). https://doi.org/10.1145/377978.377993, https://doi.org/10.1145/377978.377993
35. Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. J. Comput. Syst. Sci. **35**(1), 59–71 (1987). https://doi.org/10.1016/0022-0000(87)90036-5, https://doi.org/10.1016/0022-0000(87)90036-5
36. Lengál, O.: Automata benchmarks (2022), https://github.com/ondrik/automata-benchmarks
37. Li, Y., Turrini, A., Feng, W., Vardi, M.Y., Zhang, L.: Divide-and-conquer determinization of Büchi automata based on SCC decomposition. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 152–173. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_8, https://doi.org/10.1007/978-3-031-13188-2_8
38. Li, Y., Turrini, A., Zhang, L., Schewe, S.: Learning to complement Büchi automata. In: Dillig, I., Palsberg, J. (eds.) Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10747, pp. 313–335. Springer (2018). https://doi.org/10.1007/978-3-319-73721-8_15, https://doi.org/10.1007/978-3-319-73721-8_15
39. Li, Y., Vardi, M.Y., Zhang, L.: On the power of unambiguity in Büchi complementation. In: Raskin, J., Bresolin, D. (eds.) Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2020, Brussels, Belgium, September 21-22, 2020. EPTCS, vol. 326, pp. 182–198 (2020). https://doi.org/10.4204/EPTCS.326.12, https://doi.org/10.4204/EPTCS.326.12
40. Löding, C.: Optimal bounds for transformations of omega-automata. In: Rangan, C.P., Raman, V., Ramanujam, R. (eds.) Foundations of Software Technology and Theoretical Computer Science, 19th Conference, Chennai, India, December 13-15, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1738, pp. 97–109. Springer (1999). https://doi.org/10.1007/3-540-46691-6_8, https://doi.org/10.1007/3-540-46691-6_8
41. Löding, C., Pirogov, A.: New optimizations and heuristics for determinization of Büchi automata. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 317–333. Springer

(2019). https://doi.org/10.1007/978-3-030-31784-3_18, https://doi.org/10.1007/978-3-030-31784-3_18
42. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. Theor. Comput. Sci. **32**, 321–330 (1984). https://doi.org/10.1016/0304-3975(84)90049-5, https://doi.org/10.1016/0304-3975(84)90049-5
43. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3) (2007). https://doi.org/10.2168/LMCS-3(3:5)2007, https://doi.org/10.2168/LMCS-3(3:5)2007
44. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32, https://doi.org/10.1109/SFCS.1977.32
45. Redziejowski, R.R.: An improved construction of deterministic omega-automaton using derivatives. Fundam. Informaticae **119**(3-4), 393–406 (2012). https://doi.org/10.3233/FI-2012-744, https://doi.org/10.3233/FI-2012-744
46. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 319–327. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21948, https://doi.org/10.1109/SFCS.1988.21948
47. Safra, S., Vardi, M.Y.: On omega-automata and temporal logic (preliminary report). In: Johnson, D.S. (ed.) Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA. pp. 127–137. ACM (1989). https://doi.org/10.1145/73007.73019, https://doi.org/10.1145/73007.73019
48. Schewe, S.: Büchi complementation made tight. In: Albers, S., Marion, J. (eds.) 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings. LIPIcs, vol. 3, pp. 661–672. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2009). https://doi.org/10.4230/LIPIcs.STACS.2009.1854, https://doi.org/10.4230/LIPIcs.STACS.2009.1854
49. Sickert, S., Esparza, J., Jaax, S., Kretínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 312–332. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_17, https://doi.org/10.1007/978-3-319-41540-6_17
50. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theor. Comput. Sci. **49**, 217–237 (1987). https://doi.org/10.1016/0304-3975(87)90008-9, https://doi.org/10.1016/0304-3975(87)90008-9
51. The SV-COMP Community: International competition on software verification (2022), https://sv-comp.sosy-lab.org/
52. Tsai, M., Fogarty, S., Vardi, M.Y., Tsay, Y.: State of Büchi complementation. Log. Methods Comput. Sci. **10**(4) (2014). https://doi.org/10.2168/LMCS-10(4:13)2014, https://doi.org/10.2168/LMCS-10(4:13)2014
53. Vardi, M.Y., Wilke, T.: Automata: from logics to algorithms. In: Flum, J., Grädel, E., Wilke, T. (eds.) Logic and Automata: History and Perspectives. Texts in Logic and Games, vol. 2, pp. 629–736. Amsterdam University Press (2008)
54. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986. pp. 332–344. IEEE Computer Society (1986)

55. Yan, Q.: Lower bounds for complementation of omega-automata via the full automata technique. Log. Methods Comput. Sci. **4**(1) (2008). https://doi.org/10.2168/LMCS-4(1:5)2008, https://doi.org/10.2168/LMCS-4(1:5)2008


Validating Streaming JSON Documents with Learned VPAs*

Véronique Bruyère[1], Guillermo A. Pérez[2], and Gaëtan Staquet[1,2(✉)]

[1] University of Mons (UMONS), Mons, Belgium
{veronique.bruyere,gaetan.staquet}@umons.ac.be
[2] University of Antwerp (UAntwerp) – Flanders Make, Antwerp, Belgium
guillermo.perez@uantwerpen.be

Abstract. We present a new streaming algorithm to validate JSON documents against a set of constraints given as a JSON schema. Among the possible values a JSON document can hold, objects are unordered collections of key-value pairs while arrays are ordered collections of values. We prove that there always exists a visibly pushdown automaton (VPA) that accepts the same set of JSON documents as a JSON schema. Leveraging this result, our approach relies on learning a VPA for the provided schema. As the learned VPA assumes a fixed order on the key-value pairs of the objects, we abstract its transitions in a special kind of graph, and propose an efficient streaming algorithm using the VPA and its graph to decide whether a JSON document is valid for the schema. We evaluate the implementation of our algorithm on a number of random JSON documents, and compare it to the classical validation algorithm.

Keywords: Visibly pushdown automata · JSON · streaming validation

1 Introduction

JavaScript Object Notation (JSON) has overtaken XML as the de facto standard data-exchange format, in particular for web applications. JSON documents are easier to read for programmers and end users since they only have arrays and objects as structured types. Moreover, in contrast to XML, they do not include named open and end tags for all values, but open and end tags (braces actually) for arrays and objects only. *JSON schema* [13] is a simple schema language that allows users to impose constraints on the structure of JSON documents.

In this work, we are interested in the *validation* of *streaming* JSON documents against JSON schemas. Several previous results have been obtained about the formalization of XML schemas and the use of formal methods to validate XML documents (see, e.g., [5,15,16,18,24,25]). Recently, a standard to formalize JSON schemas has been proposed and (hand-coded) validation tools for such schemas can be found online [13]. Pezoa et al, in [19], observe that the standard

*This work was supported by the Belgian FWO "SAILor" project (G030020N). Gaëtan Staquet is a research fellow (Aspirant) of the Belgian F.R.S.-FNRS.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 271–289, 2023.
https://doi.org/10.1007/978-3-031-30823-9_14

of JSON documents is still evolving and that the formal semantics of JSON schemas is also still changing. Furthermore, validation tools seem to make different assumptions about both documents and schemas. The authors of [19] carry out an initial formalization of JSON schemas into formal grammars from which they are able to construct a *batch* validation tool from a given JSON schema.

In this paper, we rely on the formalization work of [19] and propose a *streaming* algorithm for validating JSON documents against JSON schemas. To our knowledge, this is the first JSON validation algorithm that is streaming. For XML, works that study streaming document validation base such algorithms on the construction of some automaton (see, e.g., [25], for XML). In [7], we first experimented with one-counter automata for this purpose. We submit that *visibly-pushdown automata* (VPAs) are a better fit for this task — this is in line with [15], where the same was proposed for streaming XML documents. In contrast to one-counter automata,[3] we show that VPAs are expressive enough to capture the language of JSON documents satisfying any JSON schema.

More importantly, we explain that *active learning à la* Angluin [4] is a good alternative to the automatic construction of such a VPA from the formal semantics of a given JSON schema. This is possible in the presence of labeled examples or a computer program that can answer membership and (approximate) equivalence queries about a set of JSON documents. This learning approach has two advantages. First, we derive from the learned VPA a streaming validator for JSON documents. Second, by automatically learning an automaton representation, we circumvent the need to write a schema and subsequently validate that it represents the desired set of JSON documents. Indeed, it is well known that one of the highest bars that users have to clear to make use of formal methods is the effort required to write a formal specification, in this case, a JSON schema.

Contributions. We present a VPA active learning framework to achieve what was mentioned above — though we fix an order on the keys appearing in objects. The latter assumption helps our algorithm learn faster. Secondly, we show how to bootstrap the learning algorithm by leveraging existing validation and document-generation tools to implement approximate equivalence checks. Thirdly, we describe how to validate streaming documents using our fixed-order learned automata — that is, our algorithm accepts other permutations of keys, not just the one encoded into the VPA. Finally, we present an empirical evaluation of our learning and validation algorithms, implemented on top of LearnLib [17].

All contributions, while complementary, are valuable in their own right. First, our learning algorithm for VPAs is a novel gray-box extension of TTT [9] that leverages side information about the language of all JSON documents. Second, our validation algorithm that uses a fixed-order VPA is novel and can be applied regardless of whether the automaton is learned or constructed from a schema. For the validation algorithm, we developed the concept of *key graph*, which allows us to efficiently realize the validation no matter the key-value order in the docu-

[3] By nesting objects and arrays, we obtain a set of JSON documents encoding $\{a^n b^m c^m d^n \mid n, m \in \mathbb{N}\}$, a context-free language that requires two counters.

ment, and might be of independent interest for other JSON-analysis applications using VPAs. Finally, we implemented our own batch validator to facilitate approximating equivalence queries as required by our learning algorithm. Both the new validator and the equivalence oracle are efficient, open-source, and easy to modify. We strongly believe the latter can be re-used in similar projects aiming to learn automata representations of sets of JSON documents.

A long version of this work is on arXiv: https://arxiv.org/abs/2211.08891.

2 Visibly Pushdown Languages

First, we recall the definition VPAs [3] and state some of their properties. We also recall how they can be actively learned following Angluin's approach [4].

Visibly Pushdown Automata An *alphabet* Σ is a finite set whose elements are called *symbols*. A *word* w over Σ is a finite sequence of symbols from Σ, with the *empty word* denoted by ε. The length of w is denoted $|w|$; the set of all words, Σ^*. Given two words $v, w \in \Sigma^*$, v is a *prefix* (resp. *suffix*) of w if there exists $u \in \Sigma^*$ such that $w = vu$ (resp. $w = uv$), and v is a *factor* of w if there exist $u, u' \in \Sigma^*$ such that $w = uvu'$. Given $L \subseteq \Sigma^*$, called a *language*, we denote by $\mathrm{Pref}(L)$ (resp. $\mathrm{Suff}(L)$) the set of prefixes (resp. suffixes) of words of L. Given a set Q, we write $\mathbb{I}_Q$ for the *identity relation* $\{(q, q) \mid q \in Q\}$ on Q.

VPA [3] are particular pushdown automata that we recall in this section. The *pushdown alphabet*, denoted $\tilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_i)$, is partitioned into pairwise disjoint alphabets $\Sigma_c, \Sigma_r, \Sigma_i$ such that Σ_c (resp. Σ_r, Σ_i) is the set of *call* symbols (resp. *return* symbols, *internal* symbols). In this paper, we work with the particular alphabet of return symbols $\Sigma_r = \{\bar{a} \mid a \in \Sigma_c\}$. For any such $\tilde{\Sigma}$, we denote by Σ the alphabet $\Sigma_c \cup \Sigma_r \cup \Sigma_i$. Given a pushdown alphabet $\tilde{\Sigma}$, the set $\mathrm{WM}(\tilde{\Sigma})$ of *well-matched* words over $\tilde{\Sigma}$ is defined:

- $\varepsilon \in \mathrm{WM}(\tilde{\Sigma})$, and $a \in \mathrm{WM}(\tilde{\Sigma})$ for all $a \in \Sigma_i$,
- if $w, w' \in \mathrm{WM}(\tilde{\Sigma})$, then $ww' \in \mathrm{WM}(\tilde{\Sigma})$,
- if $a \in \Sigma_c, w \in \mathrm{WM}(\tilde{\Sigma})$, then $aw\bar{a} \in \mathrm{WM}(\tilde{\Sigma})$.

Also, the *call/return balance* function $\beta : \Sigma^* \to \mathbb{Z}$ is defined as $\beta(\varepsilon) = 0$ and $\beta(ua) = \beta(u) + x$ with x being 1, -1, or 0 if a is in Σ_c, Σ_r, or Σ_i respectively. In particular, for all $w \in \mathrm{WM}(\tilde{\Sigma})$, we have $\beta(u) \geq 0$ for each prefix u of w and $\beta(u) \leq 0$ for each suffix u of w. Finally, the *depth* $d(w)$ of a well-matched word w is equal to $\max\{\beta(u) \mid u \in \mathrm{Pref}(\{w\})\}$, that is, the maximum number of unmatched call symbols among the prefixes of w.

Definition 1. *A* visibly pushdown automaton *(VPA) over a pushdown alphabet* $\tilde{\Sigma}$ *is a tuple* $(Q, \tilde{\Sigma}, \Gamma, \delta, Q_I, Q_F)$ *where* Q *is a finite non-empty set of* states, $Q_I \subseteq Q$ *is a set of* initial *states,* $Q_F \subseteq Q$ *is a set of* final *states,* Γ *is a* stack alphabet*, and* δ *is a finite set of* transitions *of the form* $\delta = \delta_c \cup \delta_r \cup \delta_i$ *where* $\delta_c \subseteq Q \times \Sigma_c \times Q \times \Gamma$ *is the set of* call *transitions,* $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$ *is the set of* return *transitions, and* $\delta_i \subseteq Q \times \Sigma_i \times Q$ *is the set of* internal *transitions. The* size *of* $\mathcal{A}$ *is denoted by* $|Q|$*, and its number of transitions by* $|\delta|$*.*

Let us describe the *transition system* $\mathrm{T}_{\mathcal{A}}$ of a VPA $\mathcal{A}$ whose vertices are configurations. A *configuration* is a pair $\langle q, \sigma \rangle$ where $q \in Q$ is a state and $\sigma \in \Gamma^*$ a stack content. A configuration is *initial* (resp. *final*) if $q \in Q_I$ (resp. $q \in Q_F$) and $\sigma = \varepsilon$. For $a \in \Sigma$, we write $\langle q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle$ in $\mathrm{T}_{\mathcal{A}}$ if there is either a call transition $(q, a, q', \gamma) \in \delta_c$ verifying $\sigma' = \gamma\sigma$,[4] or a return transition $(q, a, \gamma, q') \in \delta_r$ verifying $\sigma = \gamma\sigma'$, or an internal transition $(q, a, q') \in \delta_i$ such that $\sigma' = \sigma$.

The transition relation of $\mathrm{T}_{\mathcal{A}}$ is extended to words in the usual way. We say that $\mathcal{A}$ *accepts* a word $w \in \Sigma^*$ if there exists a path in $\mathrm{T}_{\mathcal{A}}$ from an initial configuration to a final configuration that is labeled by w. The *language of* $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is defined as $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists q \in Q_I, \exists q' \in Q_F, \langle q, \varepsilon \rangle \xrightarrow{w} \langle q', \varepsilon \rangle\}$, i.e., the set of all words accepted by $\mathcal{A}$. Any language accepted by some VPA is a *visibly pushdown language* (VPL). Notice that such a language is composed of well-matched words only.[5] Given a VPA $\mathcal{A}$ over $\tilde{\Sigma}$, the *reachability relation* $\mathrm{Reach}_{\mathcal{A}}$ of $\mathcal{A}$ is $\mathrm{Reach}_{\mathcal{A}} = \{(q, q') \in Q^2 \mid \exists w \in \mathrm{WM}(\tilde{\Sigma}), \langle q, \varepsilon \rangle \xrightarrow{w} \langle q', \varepsilon \rangle\}$.

Finally, we say that $p \in Q$ is a *bin state* if there exists no path in $\mathrm{T}_{\mathcal{A}}$ of the form $\langle q, \varepsilon \rangle \xrightarrow{w} \langle p, \sigma \rangle \xrightarrow{w'} \langle q', \varepsilon \rangle$ with $q \in Q_I$ and $q' \in Q_F$. If a VPA $\mathcal{A}$ has bin states, those states can be removed from Q as well as the transitions containing bin states without modifying the accepted language.

Minimal Deterministic VPAs Given a VPA $\mathcal{A} = (Q, \tilde{\Sigma}, \Gamma, \delta, Q_I, Q_F)$, we say that it is *deterministic* (det-VPA) if $|Q_I| = 1$ and $\mathcal{A}$ does not have two distinct transitions with the same left-hand side. By *left-hand side*, we mean (q, a) for a call transition $(q, a, q', \gamma) \in \delta_c$ or an internal transition $(q, a, q') \in \delta_i$, and (q, a, γ) for a return transition $(q, a, \gamma, q') \in \delta_r$.

Theorem 1 ([3,32]). *For any VPA $\mathcal{A}$ over $\tilde{\Sigma}$, one can construct a det-VPA $\mathcal{B}$ over $\tilde{\Sigma}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. Moreover, the size of $\mathcal{B}$ is in $\mathcal{O}(2^{|Q|^2})$ and the size of its stack alphabet is in $\mathcal{O}(|\Sigma_c| \cdot 2^{|Q|^2})$.*

Proof. Let us briefly recall this construction. Let $\mathcal{A} = (Q, \tilde{\Sigma}, \Gamma, \delta, Q_I, Q_F)$. The states of $\mathcal{B}$ are subsets R of the reachability relation $\mathrm{Reach}_{\mathcal{A}}$ of $\mathcal{A}$ and the stack symbols of $\mathcal{B}$ are of the form (R, a) with $R \subseteq \mathrm{Reach}_{\mathcal{A}}$ and $a \in \Sigma_c$. Let $w = u_1 a_1 u_2 a_2 \dots u_n a_n u_{n+1}$ be such that $n \geq 0$ and $u_i \in \mathrm{WM}(\tilde{\Sigma}), a_i \in \Sigma_c$ for all i. That is, we decompose w in terms of its unmatched call symbols. Let R_i be equal to $\{(p, q) \mid \langle p, \varepsilon \rangle \xrightarrow{u_i} \langle q, \varepsilon \rangle\}$ for all i. Then after reading w, the det-VPA $\mathcal{B}$ has its current state equal to R_{n+1} and its stack containing $(R_n, a_n) \dots (R_2, a_2)(R_1, a_1)$. Assume we are reading the symbol a after w, then $\mathcal{B}$ performs the following transition from R_{n+1}: (1) if $a \in \Sigma_c$, then push (R_{n+1}, a) on the stack and go to the state $R = \mathbb{I}_Q$ (a new unmatched call symbol is read); (2) if $a \in \Sigma_i$, then go to the state $R = \{(p, q) \mid \exists (p, p') \in R_{n+1}, (p', a, q) \in \delta_i\}$ (u_{n+1} is extended to the well-matched word $u_{n+1}a$); (3) if $a \in \Sigma_r$, then pop (R_n, a_n) from the stack if $\bar{a}_n = a$, and go to the state

$$R = \{(p, q) \mid \exists (p, p') \in R_n, (p', a_n, r', \gamma) \in \delta_c, (r', r) \in R_{n+1}, (r, a, \gamma, q) \in \delta_r\}$$

[4] The stack symbol γ is pushed on the left of σ.

[5] The original definition of VPA [3] allows acceptance of ill-matched words.

(the call symbol a_n is matched with the return symbol $a = \bar{a}_n$, leading to the well-matched word $u_n a_n u_{n+1} a$). Finally the initial state of $\mathcal{B}$ is $\mathbb{I}_{Q_I}$ and its final states are sets R containing some (p, q) with $p \in Q_I$ and $q \in Q_F$. □

Though a VPL L in general does not have a unique minimal det-VPA $\mathcal{A}$ accepting L, imposing the following subclass leads to a unique minimal acceptor.

Definition 2 ([2, 9]). *A* 1-module single entry VPA[6] *(1-SEVPA) is a det-VPA* $\mathcal{A} = (Q, \tilde{\Sigma}, \Gamma, \delta, Q_I = \{q_0\}, Q_F)$ *such that its stack alphabet* Γ *is equal to* $Q \times \Sigma_c$*, and all its call transitions* $(q, a, q', \gamma) \in \delta_c$ *are such that* $q' = q_0$ *and* $\gamma = (q, a)$*.*

Theorem 2 ([2]). *For any VPL* L*, there exists a unique minimal (with regards to the number of states) 1-SEVPA accepting* L*, up to a renaming of the states.*[7]

Learning VPAs Let us recall the concept of *learning* a deterministic finite automaton (DFA), as introduced in [4]. Let L be a regular language over an alphabet Σ. The task of the *learner* is to construct a DFA $\mathcal{H}$ such that $\mathcal{L}(\mathcal{H}) = L$ by interacting with the *teacher*. The two possible types of interactions are *membership queries* (does $w \in \Sigma^*$ belong to L?), and *equivalence queries* (does the DFA $\mathcal{H}$ accept L?). For the latter type, if the answer is negative, the teacher also provides a counterexample, i.e., a word w such that $w \in L \Leftrightarrow w \notin \mathcal{L}(\mathcal{H})$. The so-called L^* *algorithm* of [4] learns at least one representative per equivalence class of the Myhill-Nerode congruence of L [8] from which the minimal DFA $\mathcal{D}$ accepting L is constructed. This learning process terminates and it uses a polynomial number of membership and equivalence queries in the size of $\mathcal{D}$, and in the length of the longest counterexample returned by the teacher [4].

In [9], an extension of Angluin's learning algorithm is given for VPLs. The Myhill-Nerode congruence for regular languages is extended to VPLs as follows. Given a pushdown alphabet $\tilde{\Sigma}$ and a VPL L over $\tilde{\Sigma}$, we consider the set of *context pairs* $\mathrm{CP}(\tilde{\Sigma}) = \{(u, v) \in (\mathrm{WM}(\tilde{\Sigma}) \cdot \Sigma_c)^* \times \mathrm{Suff}(\mathrm{WM}(\tilde{\Sigma})) \mid \beta(u) = -\beta(v)\}$, and we define the equivalence relation $\simeq_L \subseteq \mathrm{WM}(\tilde{\Sigma}) \times \mathrm{WM}(\tilde{\Sigma})$ [2, 9] such that $w \simeq_L w'$ if and only if $\forall (u, v) \in \mathrm{CP}(\tilde{\Sigma}), uwv \in L \Leftrightarrow uw'v \in L$. The minimal 1-SEVPA accepting L as described in Theorem 2 is constructed from $\simeq_L$ such that its states are the equivalence classes of $\simeq_L$.

Theorem 3 ([9]). *Let* L *be a VPL over* $\tilde{\Sigma}$ *and* n *be the index of* $\simeq_L$*. queries and a number of membership queries polynomial in* n*,* $|\Sigma|$*, and* $\log \ell$*, where* ℓ *is the length of the longest counterexample returned by the teacher.*

The learning process designed in [9] extends to VPLs the Ttt *algorithm* proposed in [10] for regular languages. Ttt improves the efficiency of the L^* algorithm by eliminating redundancies in counterexamples provided by the teacher.

[6] The definitions of 1-SEVPA in [2] and [9] differ slightly. We follow the one in [9].

[7] This 1-SEVPA may be exponentially bigger than the size of a VPA accepting L.

3 JSON Format

In this section, we describe JSON documents [6] and JSON schemas [13] that impose some constraints on the structure of JSON documents. We also present the abstractions we make for the purpose of this paper.

JSON Documents We describe the structure of JSON documents. Our presentation is inspired by [19], though some details are skipped for readability (see [14] for a full description). The JSON format defines different types of *JSON values*:

- `true`, `false`, `null` are JSON values. Any decimal number (positive, negative) is a JSON value, called a *number*. In particular any number that is an integer is called an *integer*. Any finite sequence of characters starting and ending with `"` is a *string value*. All those values are called *primitive values*.
- If $v_1, v_2, \ldots, v_n$ are JSON values and $k_1, k_2, \ldots, k_n$ are *pairwise distinct* string values, then $\{k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n\}$ is a JSON value, called an *object*. Each $k_i : v_i$ is called a *key-value pair* such that k_i is the *key*. The collection of these pairs is *unordered*.
- If $v_1, v_2, \ldots, v_n$ are JSON values, then $[v_1, v_2, \ldots, v_n]$ is a JSON value, called an *array*. Each v_i is an *element* and the collection thereof is *ordered*.

In this work, *JSON documents* are supposed to be objects.[8] One can use *JSON pointers* to navigate through a document, e.g., if J is an object and k is a key, then $J[k]$ is the value v such that the key-value pair $k : v$ appears in J.

In this paper, we consider somewhat *abstract* JSON documents. We see JSON documents as well-matched words over the pushdown alphabet $\tilde{\Sigma}_{\text{JSON}}$ that we describe hereafter. We abstract all string values as `s`, and all numbers as `n` (as `i` when they are integers). We denote by $\Sigma_{\text{pVal}} = \{\texttt{true}, \texttt{false}, \texttt{null}, \texttt{s}, \texttt{n}, \texttt{i}\}$ the alphabet composed of the six primitive values. Concerning the key-value pairs appearing in objects, each key together with the symbol ":" following the key is abstracted as an alphabet symbol k. We assume knowledge of a *finite* alphabet Σ_{key} of keys. We define the pushdown alphabet $\tilde{\Sigma}_{\text{JSON}} = (\Sigma_c, \Sigma_r, \Sigma_i)$ with $\Sigma_i = \Sigma_{\text{key}} \cup \Sigma_{\text{pVal}} \cup \{\#\}$, where $\#$ is used in place of the comma; $\Sigma_c = \{\prec, \sqsubset\}$, where $\prec$ (resp. $\sqsubset$) is used in place of "{" (resp. "["); and $\Sigma_r = \{\succ, \sqsupset\}$, with $\overline{\prec} = \succ$ and $\overline{\sqsubset} = \sqsupset$. We denote by Σ_{JSON} the set $\Sigma_c \cup \Sigma_r \cup \Sigma_i$.

Example 1. An example of a JSON document is given in Listing 1. We can see that this document is an object containing three keys: `"title"`, whose associated value is a string value; `"keywords"`, whose value is an array containing string values; and `"conf"`, whose value is an object. This inner object contains two keys: `"name"`, whose value is a string value; `"year"`, whose value is an integer. The pointer `J[conf][name]`, where `J` is the root of the document, retrieves the value `"TACAS"`. The JSON document is abstracted as the word $\prec k_1\texttt{s}\#k_2\sqsubset\texttt{s}\#\texttt{s}\sqsupset\#k_3\prec k_4\texttt{s}\#k_5\texttt{i}\succ\succ \in \text{WM}(\tilde{\Sigma}_{\text{JSON}})$ where Σ_{key} contains the keys $k_i, i \in \{1, \ldots, 5\}$.

[8] In [6], a JSON document can be any JSON value and duplicated keys are allowed inside objects. In this paper, we follow what is commonly used in practice: JSON documents are objects, and keys are pairwise distinct inside objects.

```
{ "title": "Validating Streaming JSON Documents with Learned VPAs",
  "keywords": ["VPA", "JSON documents", "streaming validation"],
  "conf": { "name": "TACAS", "year": 2023 }
}
```

Listing 1: A JSON document.

```
{ "type": "object",
  "required": ["title", "conf"],
  "properties": {
    "title": { "type": "string" },
    "keywords": { "type": "array", "items": { "type": "string" } },
    "conf": {
      "type": "object",
      "required": ["name", "year"],
      "properties":{ "name":{"type": "string"},"year":{"type": "integer"}}}}}
```

Listing 2: A JSON schema.

JSON Schemas A *JSON schema* can impose some constraints on JSON documents by specifying any of the types of JSON values that appear in those documents. We say that a JSON document *satisfies* (or is *valid* for) the schema if it verifies the constraints imposed by this schema. We denote by $\mathcal{L}(S)$ the set of documents that are valid for S. In this section, we give a simplified presentation of JSON schemas and refer to [13] for a complete description and to [19] for a formalization (i.e. a formal grammar with its syntax and semantics).

A JSON schema is itself a JSON document that uses several keywords that help shape and restrict the set of JSON documents that this schema specifies. As we abstract JSON documents, JSON schemas we work on are also abstracted. We do not consider the restrictions that can be imposed on string values and numbers, for instance. We give here a few examples. See [13] for more details.

- Within object schemas, restrictions can be imposed on the key-value pairs of the objects, e.g., the value associated with some key has itself to satisfy a certain schema, or some particular keys must be present in the object.
- Within array schemas, it can be imposed that all elements of the array satisfy a certain schema, or that the array has a minimum/maximum size.
- Schemas can be combined with Boolean operations, e.g., a JSON document must satisfy a conjunction of several schemas.
- A schema can be defined as one referred to by a JSON pointer. This allows a *recursive* structure for the JSON documents satisfying a certain schema.

Example 2. The schema from Listing 2 describes objects that can have three keys: `"title"`, whose associated value must be a string value; `"keywords"`, an array of strings; and `"conf"`, an object. Among these, `"title"` and `"conf"` are required. The JSON document of Example 1 satisfies this JSON schema.

Under these abstractions, we can always construct a VPA that accept the same set of JSON documents than a schema S, as shown in the following theorem. We also extend this construction to the case where we fix an *order* $<$

on Σ_{key} and consider the set $\mathcal{L}_<(S)$ of documents valid for S whose key order inside objects respects this order $<$. The main idea of the proof is to define a formalism of JSON schemas as *extended context-free grammars*, and show that we can construct a VPA from such a grammar.

Theorem 4. *Let S be a JSON schema. Then, there exists a VPA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A})$ is the set $\mathcal{L}(S)$ of documents valid with regards to S. Moreover, for any order $<$ of Σ_{key}, there exists a VPA $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}_<(S)$.*

Our proof does not give a construction of the grammar from the schema S. The grammar depends on the formal semantics of JSON schemas which are still changing and being debated. Thus, to be more robust to changes in the semantics, we prefer to learn the minimal 1-SEVPA $\mathcal{B}$ accepting $\mathcal{L}_<(S)$ given a fixed order $<$, in the sense of Theorem 3.[9] For learning, equivalence queries require to generate a certain number of random JSON documents.[10] If S and the learner's hypothesis $\mathcal{H}$ disagree on a document, we have a counterexample. Otherwise, we say that $\mathcal{H}$ is correct. In both membership and equivalence queries, we only accept documents whose key order inside objects satisfy the order $<$. The randomness used in the equivalence queries implies that the learned 1-SEVPA may not exactly accept $\mathcal{L}_<(S)$. Setting the number of generated documents to be large would help reducing the probability that an incorrect 1-SEVPA is learned.

4 Validation of JSON Documents

For this section, let us fix a schema S, an order $<$ on Σ_{key}, and a 1-SEVPA $\mathcal{A} = (Q, \tilde{\Sigma}_{\text{JSON}}, \Gamma, \delta, \{q_0\}, Q_F)$ accepting $\mathcal{L}_<(S)$. We present a *streaming* algorithm to decide if a document J is in $\mathcal{L}(S)$. By "streaming", we mean an algorithm that processes the document in a single pass, symbol by symbol. Our new approach is as follows. We learn $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(S)$. As $\mathcal{L}_<(S) \neq \mathcal{L}(S)$, we design an algorithm that uses $\mathcal{A}$ in a clever way to allow arbitrary key orders in documents to validate. To do this, we use a *key graph* defined in the sequel.

Key Graph In this section, w.l.o.g. we suppose that $\mathcal{A}$ has *no bin states*. Let $\mathrm{T}_{\mathcal{A}}$ be the transition system of $\mathcal{A}$. We explain how to associate to $\mathcal{A}$ its *key graph* $\mathrm{G}_{\mathcal{A}}$: an abstraction of the paths of $\mathrm{T}_{\mathcal{A}}$ labeled by the contents of the objects appearing in words of $\mathcal{L}_<(S)$. This graph is essential in our validation algorithm.

Definition 3. *The* key graph $\mathrm{G}_{\mathcal{A}}$ *of* $\mathcal{A}$ *has:*

- *the vertices (p, k, p') with $p, p' \in Q$ and $k \in \Sigma_{\text{key}}$ if there exists in $\mathrm{T}_{\mathcal{A}}$ a path $\langle p, \varepsilon\rangle \xrightarrow{kv} \langle p', \varepsilon\rangle$ with $v \in \Sigma_{\text{pVal}} \cup \{au\bar{a} \mid a \in \Sigma_c, u \in \mathrm{WM}(\tilde{\Sigma}_{\text{JSON}})\}$,*[11]

[9] We use this automaton in the next section for the validation of JSON documents. We do not use a 1-SEVPA for $\mathcal{L}(S)$ as it could be exponentially larger.

[10] It is common to proceed this way in automata learning, as explained in [4, Sec. 4].

[11] Notice that each vertex (p, k, p') of $\mathrm{G}_{\mathcal{A}}$ only stores the key k and not the word kv.

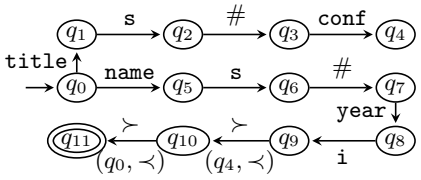


Fig. 1: A 1-SEVPA for the schema from Listing 2, without the key `keywords`.

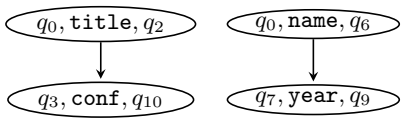


Fig. 2: The key graph for the 1-SEVPA from Figure 1.

– *the edges* $((p_1, k_1, p_1'), (p_2, k_2, p_2'))$ *if there exists* $(p_1', \#, p_2) \in \delta_i$.

We have the following property.

Lemma 1. *There exists a path* $((p_1, k_1, p_1') \dots (p_n, k_n, p_n'))$ *in* $\mathrm{G}_{\mathcal{A}}$ *with* $p_1 = q_0$ *if and only if there exist a factor* u *of a word in* $\mathcal{L}_<(S)$ *such that* $u = k_1 v_1 \# \dots \# k_n v_n$ *where each* $k_i v_i$ *is a key-value pair, and a path* $\langle q_0, \varepsilon\rangle \xrightarrow{u} \langle p_n', \varepsilon\rangle$ *in* $\mathrm{T}_{\mathcal{A}}$ *that decomposes as* $\langle p_i, \varepsilon\rangle \xrightarrow{k_i v_i} \langle p_i', \varepsilon\rangle, \forall i \in \{1, \dots, n\}$ *and* $\langle p_i', \varepsilon\rangle \xrightarrow{\#} \langle p_{i+1}, \varepsilon\rangle, \forall i \in \{1, \dots, n-1\}$. *Furthermore, there is no path* $((p_1, k_1, p_1') \dots (p_n, k_n, p_n'))$ *such that* $k_i = k_j$ *for some* $i \neq j$. *That is,* $\mathrm{G}_{\mathcal{A}}$ *contains a finite number of paths.*

Hence, paths in $\mathrm{G}_{\mathcal{A}}$ focus on contents of objects being part of JSON documents in $\mathcal{L}_<(S)$. Moreover, they abstract paths in $\mathrm{T}_{\mathcal{A}}$ in the sense that only keys k_i are stored and the subpaths labeled by the values v_i are implicit.

Example 3. Consider the schema from Listing 2, without the key `keywords`. A 1-SEVPA $\mathcal{A}$ accepting $\mathcal{L}_<(S)$ is given in Figure 1. For clarity, call transitions[12] and the bin state are not represented. In Figure 2, we depict its corresponding key graph $\mathrm{G}_{\mathcal{A}}$. Since we have the path $\langle q_0, \varepsilon\rangle \xrightarrow{\texttt{title s}} \langle q_2, \varepsilon\rangle$ in $\mathrm{T}_{\mathcal{A}}$, the triplet $(q_0, \texttt{title}, q_2)$ is a vertex of $\mathrm{G}_{\mathcal{A}}$. Likewise, $(q_0, \texttt{name}, q_6)$ and $(q_7, \texttt{year}, q_9)$ are vertices. As we have the path $\langle q_4, \varepsilon\rangle \xrightarrow{\prec} \langle q_0, (q_4, \prec)\rangle \xrightarrow{\texttt{name s \# year i}} \langle q_9, (q_4, \prec)\rangle \xrightarrow{\succ} \langle q_{10}, \varepsilon\rangle$, $(q_3, \texttt{conf}, q_{10})$ is also a vertex of $\mathrm{G}_{\mathcal{A}}$. Finally, as $\langle q_2, \varepsilon\rangle \xrightarrow{\#} \langle q_3, \varepsilon\rangle$, we have an edge from $(q_0, \texttt{title}, q_2)$ to $(q_3, \texttt{conf}, q_{10})$.

Computing the key graph can be done in polynomial time by first computing the reachability relation $\mathrm{Reach}_{\mathcal{A}}$. From this relation, the vertices can be easily found. Since the edges require to check whether a transition reading # exists, it is obvious that it can be done in polynomial time.

Validation Algorithm In this section, we provide a streaming algorithm that validates JSON documents against a given JSON schema S.

Given a word $w \in \Sigma_{\mathrm{JSON}}^* \setminus \{\varepsilon\}$, we want to check whether $w \in \mathcal{L}(S)$. The main difficulty is that the key-value pairs inside an object are arbitrarily ordered in w while a fixed key order $<$ is encoded in the 1-SEVPA $\mathcal{A}$ ($\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(S)$).

[12] Recall the form of call transitions for 1-SEVPAs, see Definition 2.

Our validation algorithm is inspired by the algorithm computing a det-VPA equivalent to some given VPA [3] (see Theorem 1 and its proof) and uses the key graph $\mathrm{G}_{\mathcal{A}}$ to treat arbitrary orders of the key-value pairs inside objects.

During the reading of $w \in \Sigma^*_{\mathrm{JSON}} \setminus \{\varepsilon\}$, in addition to checking whether $w \in \mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}})$, the algorithm updates a subset $R \subseteq \mathrm{Reach}_{\mathcal{A}}$ and modifies the content of a stack Stk (push, pop, modify the element on top of Stk).

First, let us explain the information stored in R. Assume that we have read the prefix zau of w such that $a \in \Sigma_c$ is the last unmatched call symbol (thus $za \in (\mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}}) \cdot \Sigma_c)^*$ and $u \in \mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}})$).

- If a is the symbol $\sqsubset$, then we have $R = \{(p, q) \mid \langle p, \varepsilon\rangle \xrightarrow{u} \langle q, \varepsilon\rangle\}$.
- If a is the symbol $\prec$, then we have $u = k_1 v_1 \# k_2 v_2 \# \dots k_{n-1} v_{n-1} \# u'$ such that $u' \in \mathrm{WM}(\tilde{\Sigma}_{\mathrm{JSON}})$ and u' is prefix of $k_n v_n$, where each $k_i v_i$ is a key-value pair. Then $R = \{(p, q) \mid \langle p, \varepsilon\rangle \xrightarrow{u'} \langle q, \varepsilon\rangle\}$.

In the first case, by using R as defined previously, we adopt the same approach as for the determinization of VPAs. In the second case, with u, we are currently reading the key-value pairs of an object in some order, not necessarily the one encoded in $\mathcal{A}$. In this case the set R is focused on the currently read key-value pair $k_n v_n$, that is, on the word u'. After reading of the whole object $\prec k_1 v_1 \# k_2 v_2 \# \dots \succ$, we will use the key graph $\mathrm{G}_{\mathcal{A}}$ to update the current set R.

Second, an element stored in the stack Stk is either a pair $(R, \sqsubset)$, or a 5-tuple $(R, \prec, K, k, Bad)$, where R is a set as described previously, $K \subseteq \Sigma_{\mathrm{key}}$ is a subset of keys, $k \in \Sigma_{\mathrm{key}}$ is a key, and Bad is a set containing some vertices of $\mathrm{G}_{\mathcal{A}}$.[13]

We now detail our streaming validation algorithm.[14] Before reading w, we initialize R to the set $\mathbb{I}_{\{q_0\}}$ and Stk to the empty stack. Let us explain how to update the current set R and the current content of the stack Stk while reading the input word w. Suppose that we are reading the symbol a in w. In some cases we will also peek the symbol b following a (*lookahead* of one symbol).

Case (1) Suppose that a is the symbol $\sqsubset$, i.e., we start an array. Hence $(R, \sqsubset)$ is pushed on Stk and R is updated to $R_{Upd} = \mathbb{I}_{\{q_0\}}$. We thus proceed as in the proof of Theorem 1 (with $\mathbb{I}_{\{q_0\}}$ instead of $\mathbb{I}_Q$, since $\mathcal{A}$ is a 1-SEVPA[12]).
Case (2) Suppose that $a \in \Sigma_i$ and $\sqsubset$ appears on top of Stk. We are thus reading the elements of an array. Hence R is updated to $R_{Upd} = \{(p, q) \mid \exists (p, q') \in R, (q', a, q) \in \delta_i\}$. Again we proceed as in the proof of Theorem 1.
Case (3) Suppose that a is the symbol $\sqsupset$. This means that we finished reading an array. If the stack is empty or its top element contains $\prec$, then $w \notin \mathcal{L}(S)$ and we stop the algorithm. Otherwise $(R', \sqsubset)$ is popped from Stk and R is updated to $R_{Upd} = \{(p, q) \mid \exists (p, p') \in R', (p', \sqsubset, q_0, \gamma) \in \delta_c, (q_0, r) \in R, (r, \sqsupset, \gamma, q) \in \delta_r\}$, as in the proof of Theorem 1.
Case (4) Suppose that a is the symbol $\prec$.

[13] In the particular case of the object $\prec\succ$, the 5-tuple $(R, \prec, K, k, Bad)$ is replaced by $(R, \prec)$. This situation will be clarified during the presentation of our algorithm.
[14] Note that the algorithm assumes we have a 1-SEVPA.

– Let us first consider the particular case where the symbol b following $\prec$ is equal to $\succ$, meaning that we will read the object $\prec\succ$. In this case, $(R, \prec)$ is pushed on Stk and R is updated to $R_{Upd} = \mathbb{I}_{\{q_0\}}$ as in Case (1).
– Otherwise, if b belongs to Σ_{key}, we begin to read a (non-empty) object whose treatment is different from that of an array as its key-value pairs can be read in any order. Then, R is updated to $R_{Upd} = \mathbb{I}_{P_b}$ where $P_b = \{p \in Q \mid \exists (p, b, p') \in \mathrm{G}_{\mathcal{A}}\}$, and $(R, \prec, K, b, Bad)$ is pushed on Stk such that K is the singleton $\{b\}$ and Bad is the empty set. The 5-tuple pushed on Stk indicates that the key-value pair that will be read next begins with key b; moreover $K = \{b\}$ because this is the first pair of the object. The meaning of Bad will be clarified later. The updated set R_{Upd} is equal to the identity relation on P_b since after reading $\prec$, we will start reading a key-value pair whose abstracted state in $\mathrm{G}_{\mathcal{A}}$ can be any state from P_b. Later while reading the object whose reading is here started, we will update the 5-tuple on top of Stk as explained below.
– Finally, it remains to consider the case where $b \notin \Sigma_{\text{key}} \cup \{\succ\}$. In this final case, we have that $w \notin \mathcal{L}(S)$ and we stop the algorithm.

Case (5) Suppose that $a \in \Sigma_i \setminus \{\#\}$ and $\prec$ appears on top of Stk. Therefore, we are currently reading a key-value pair of an object. Then R is updated to $R_{Upd} = \{(p, q) \mid \exists (p, q') \in R, (q', a, q) \in \delta_i\}$.
Case (6) Suppose that a is the symbol $\#$ and $\prec$ appears on top of Stk. This means that we just finished reading a key-value pair whose key k is stored in the 5-tuple $(R', \prec, K, k, Bad)$ on top of Stk, and that another key-value pair will be read after symbol $\#$. The set K in $(R', \prec, K, k, Bad)$ stores all the keys of the key-values pairs already read including k.

– If the symbol b following $\#$ does not belong to Σ_{key}, then $w \notin \mathcal{L}(S)$ and we stop the algorithm.
– Otherwise, if b belongs to K, this means that the object contains twice the same key, that is, $w \notin \mathcal{L}(S)$, and we also stop the algorithm.
– Otherwise, the set R is updated to $R_{Upd} = \mathbb{I}_{P_b}$ (as we begin the reading of a new key-value pair whose key is b) and the 5-tuple $(R', \prec, K, k, Bad)$ on top of Stk is updated such that *(i)* K is replaced by $K \cup \{b\}$, *(ii)* k is replaced by b, and *(iii)* all vertices (p, k, p') of $\mathrm{G}_{\mathcal{A}}$ such that $(p, p') \notin R$ are added to the set Bad. Recall that the vertex (p, k, p') of $\mathrm{G}_{\mathcal{A}}$ is a witness of a path $\langle p, \varepsilon\rangle \xrightarrow{kv} \langle p', \varepsilon\rangle$ in $\mathrm{T}_{\mathcal{A}}$ for some key-value pair kv. Hence by adding this vertex (p, k, p') to Bad, we mean that the pair that has just been read does not use such a path.

Case (7) Suppose that a is the symbol $\succ$. Therefore we end the reading of an object. If the stack is empty or its top element contains $\sqsubset$, then $w \notin \mathcal{L}(S)$ and we stop the algorithm. Otherwise the top of Stk contains either $(R', \prec)$ or $(R', \prec, K, k, Bad)$ that we pop from Stk.

– If $(R', \prec)$ is popped, then we are ending the reading of the object $\prec\succ$. Hence, we proceed as in Case (3): R is updated to $R_{Upd} = \{(p, q) \mid \exists (p, p') \in R', (p', \prec, q_0, \gamma) \in \delta_c, (q_0, \succ, \gamma, q) \in \delta_r\}$.[15]

[15] Notice that R does not appear in R_{Upd} as $R = \mathbb{I}_{\{q_0\}}$.

– If $(R', \prec, K, k, Bad)$ is popped, we are ending an object whose last seen key is k. As in Case (6), we add to Bad all vertices (p, k, p') such that $(p, p') \notin R$. Let $\mathrm{Valid}(K, Bad)$ be the set of pairs of states (q_0, r') such that there exists a path $((p_1, k_1, p'_1) \dots (p_n, k_n, p'_n))$ in $\mathrm{G}_{\mathcal{A}}$ with $p_1 = q_0$, $p'_n = r'$, $(p_i, k_i, p'_i) \notin Bad$ for all $i \in \{1, \dots, n\}$, and $K = \{k_1, \dots, k_n\}$. Then R is updated to $R_{Upd} = \{(p, q) \mid \exists (p, p') \in R', (p', \prec, q_0, \gamma) \in \delta_c, (q_0, r) \in \mathrm{Valid}(K, Bad), (r, \succ, \gamma, q) \in \delta_r\}$. We thus proceed as in Case (3) except that condition $(r', r) \in R$ is replaced by $(r', r) \in \mathrm{Valid}(K, Bad)$. That way, we check that the key-value pairs that have been read as composing an object of w label some path in $\mathrm{T}_{\mathcal{A}}$, once ordered by $<$. That is, the corresponding abstract path appears in $\mathrm{G}_{\mathcal{A}}$.

Case (8) Suppose that $a \in \Sigma_i$ and Stk is empty, then $w \notin \mathcal{L}(S)$ and we stop the algorithm. Indeed an internal symbol appears either in an array or in an object (see Cases (2), (5), and (6) above).

Finally, when the input word w is completely read, we check whether the stack Stk is empty and the computed set R contains a pair (q_0, q) with $q \in Q_F$.

The complexity of our algorithm is given in the following proposition.

Proposition 1. *Let S be a schema and $\mathcal{A}$ be a 1-SEVPA such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(S)$. Deciding if a document J is valid is in time $\mathcal{O}(|J| \cdot (|Q|^4 + |Q|^{|\Sigma_{\mathrm{key}}|} \cdot |\Sigma_{\mathrm{key}}|^{|\Sigma_{\mathrm{key}}|+1}))$, and uses $\mathcal{O}(|\delta| + |Q|^2 \cdot |\Sigma_{\mathrm{key}}| + d(J) \cdot (|Q|^2 + |\Sigma_{\mathrm{key}}|))$ memory.*

5 Implementation and Experiments

We present here our Java implementation of the learning process and the validation algorithm. First, we present classical validation algorithms and explain how to generate documents from a schema. We then explain how the required membership and equivalence queries are implemented. Finally, we present the schemas we evaluated, and the results for the learning, computation of the key graph, and validation experiments. The reader is referred to the code documentation for more details about our implementation [27–31].

In the remaining of this section, let us assume we have a JSON schema S_0.

Classical Validation Algorithm and Documents Generation Let us explain briefly the *classical* algorithm used in many implementations for validating a JSON document J_0 against S_0 [13]. It is a recursive algorithm that follows the constraints of S_0.[16] For instance, if the current value J is an object, we iterate over each key-value pair in J and its corresponding sub-schema in the current schema S. Then, J satisfies S if and only if the values in the key-value pairs all satisfy their corresponding sub-schema. As long as S_0 does not contain any Boolean operations, this algorithm is straightforward and linear in the size of both the initial document J_0 and schema S_0. However, if S_0 contains Boolean operations, then the current value J may be processed multiple times.

[16] Such a recursive algorithm is briefly presented in [19].

In order to match the abstractions we defined (see Section 3) and to have options to tune the learning process, we implemented our own classical validator. Alongside the validator, we implemented a tool to generate JSON documents whose structure is dictated by S_0. Due to the Boolean operations S_0 can contain, it may happen that choices must be made during the generation process. We have two generators: a *random* generator that makes a choice at random, and an *exhaustive* generator that exhaustively explores every choice, thus producing every valid document one by one. Moreover, we implemented modifications of these generators to allow the creation of invalid documents, by allowing *deviations*.[17] For instance, if the current schema describes an integer, we can instead decide to generate a string. To ensure we eventually produce a document, we can fix a *maximal depth* (i.e., the maximal number of nested objects or arrays). This is useful for recursive schemas, or when generating invalid documents.

Learning Algorithm Let us now focus on the learning algorithm itself, and in particular on the membership and equivalence queries. We recall that the equivalence queries are performed by generating a certain number of (valid and invalid) JSON documents and by verifying that the learned VPA $\mathcal{H}$ and the given schema S_0 agree on the documents' validity. As said in Section 2, we use the TTT algorithm [9] to learn a 1-SEVPA from S_0, relying on its implementation in the well-known Java libraries LearnLib and AutomataLib [11].

We use the random and exhaustive generators of valid and invalid documents as explained above and we fix two constants C and D depending on the schema to be learned.[18] For a *membership* query over a word $w \in \Sigma^*_{\text{JSON}}$, the teacher runs the classical validator on w and S_0. For an *equivalence* query over a learned 1-SEVPA $\mathcal{H}$, the teacher uses a generator to produce documents on which $\mathcal{H}$ is tested. If that generator is random, at each query, C documents are generated for each document depth between 0 and D. If none of the documents leads to a counterexample, the teacher checks whether $\mathrm{G}_{\mathcal{H}}$ does not satisfy Lemma 1, i.e., whether there is path $((p_1, k_1, p'_1) \dots (p_n, k_n, p'_n))$ with $p_1 = q_0$ such that $k_i = k_j$ for some $i \neq j$. In that case, we can create a counterexample.

Evaluated Schemas For the experimental evaluation of our algorithms, we consider the following schemas, sorted in increasing size: (1) A schema that accepts documents defined recursively. Each object contains a string and can contain an array whose single element satisfies the whole schema, i.e., this is a recursive list. (2) A schema that accepts documents containing each type of values, i.e., an object, an array, a string, a number, an integer, and a Boolean. (3) A schema that defines how snippets must be described in *Visual Studio Code* [23]. (4) A recursive schema that defines how the metadata files for *VIM plugins* must be written [22]. (5) A schema that defines how *Azure Functions Proxies* files must look like [20]. (6) A schema that defines the configuration file

[17] This is similar to mutation testing [1, 12].

[18] The values of C and D are given below.

for a code coverage tool called *codecov* [21]. Hence, we consider two schemas written by ourselves to test our framework, and four schemas that are used in real world cases. The last four schemas were modified to make all object keys mandatory and to remove unsupported keywords. All used schemas and scripts can be consulted on our repository [30]. In the rest of this section, the schemas are referred to by their order in the previous enumeration.

We present three types of experimental results: (1) the time and number of membership and equivalence queries to learn a 1-SEVPA $\mathcal{A}$ from a JSON schema, (2) the time and memory to compute the reachability relation $\mathrm{Reach}_{\mathcal{A}}$ and the key graph $\mathrm{G}_{\mathcal{A}}$, and (3) the time and memory to validate a document using both classical and new algorithms. The server used for the benchmarks ran OpenJDK version 11.0.12 on Debian 10 over Linux 5.4.73-1-pve with a 4-core Intel® Xeon® Silver 4214R Processor with 16.5M cache, and 64GB of RAM.

Learning VPAs First, we learn a 1-SEVPA from a schema. We use an exhaustive generator for the first three schemas (accepting a small set of documents), and a random generator[19] for the remaining three for which we fix $C = 10000$. For both generators, we set $D = \mathit{depth}(S) + 1$, where $\mathit{depth}(S)$ is the maximal number of nested objects and arrays in the schema S, except for the recursive list where $D = 10$, and for the recursive *VIM plugin* schema where $D = 7$.

For the first five schemas, we do not set a time limit and repeat the learning process ten times. For the last schema, we set a time limit of one week and, for time constraints, only perform the learning process once. After that, we stop the computation and retrieve the learned 1-SEVPA at that point. The retrieved automaton is therefore an approximation of this schema. Its key graph has repeated keys along some of its paths, a situation that cannot occur if the 1-SEVPA was correctly learned, see Lemma 1. Results are given in Table 1.

Comparing Validation Algorithms The second part of the preprocessing step is to construct the key graph of the learned 1-SEVPA. For each evaluated schema, we select the learned automaton with the largest set of states, in order to report a worst-case measure. Results after a single experiment are given in Table 2. We can see that the storage of the key graph does not consume more than one megabyte, except for *codecov* schema. That is, even for non-trivial schemas, the key graph is relatively lightweight.

Finally, we compare both classical and new streaming validation algorithms. For the latter, we use the 1-SEVPA (and its key graph) selected as described above. We first generate 5000 valid and 5000 invalid JSON documents using a random generator, with a maximal depth equal to $D = 20$. We then measure the time and memory required by both validation algorithms on these documents.[20]

[19] With the random generator, the learned 1-SEVPAS may differ each experiment.

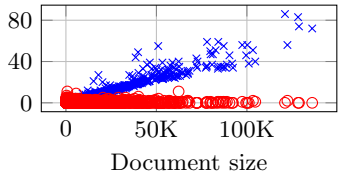
[20] Since obtaining a close approximation of the consumed memory requires Java to stop the execution and destroy all unused objects, we execute each algorithm twice: once to measure time, and a second time to measure memory.

Time (s)	Membership	Equivalence	$\lvert Q\rvert$	$\lvert\Sigma\rvert$	$\lvert\delta_c\rvert$	$\lvert\delta_r\rvert$	$\lvert\delta\rvert$	Diameter
2.2	2055.0	5.0	7.0	15.0	14.0	3.0	5.0	3.0
4.5	69514.0	3.0	24.0	20.0	48.0	3.0	26.0	12.0
9.0	21943.0	5.0	16.0	17.0	32.0	7.0	18.0	13.0
9590.3	4246085.0	36.4	150.0	27.0	300.0	2946.5	760.3	9.0
35008.2	4063971.7	30.5	121.0	35.0	242.0	2123.0	752.5	13.3
Timeout	633049534.0	192.0	884.0	77.0	1768.0	89695.0	8557.0	28.0

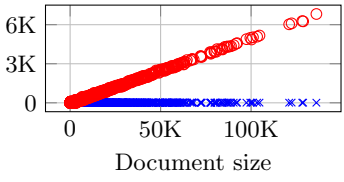
Table 1: Learning results. For the first five schemas, values are averaged out of ten experiments. For the last schema, a single experiment was conducted.

$\text{Reach}_{\mathcal{A}}$			$\text{G}_{\mathcal{A}}$			
Time (s)	Memory (kB)	Size	Time (s)	Computation (kB)	Storage (kB)	Size
34	492	31	100	2231	65	3
67	1152	213	234	2623	69	9
67	737	125	118	2223	69	10
1756	10316	5832	1715	11827	419	418
2208	13978	4420	2839	17968	667	541
377141	212970	270886	187659	120398	16335	6397

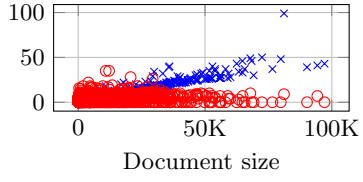
Table 2: Results for the computation of $\text{Reach}_{\mathcal{A}}$ and $\text{G}_{\mathcal{A}}$. The Computation (resp. Storage) column gives the memory required to compute $\text{G}_{\mathcal{A}}$ (resp. to store $\text{G}_{\mathcal{A}}$).



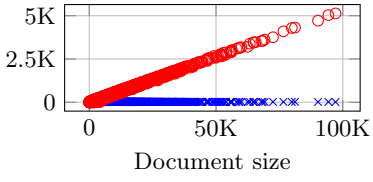
(a) Time usage (ms) for *VIM plugins*.



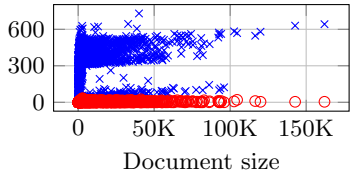
(b) Mem. usage (kB) for *VIM plugins*.



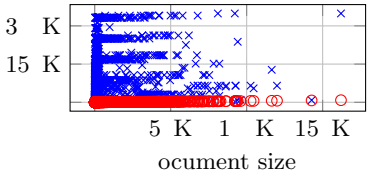
(c) Time usage for *Azure*.



(d) Mem. usage for *Azure*.



(e) Time usage for *codecov*.



(f) Mem. usage for *codecov*.

Fig. 3: Results of validation benchmarks.

On all considered documents, both algorithms return the same classification output, even for the partially learned 1-SEVPA.

For our algorithm, we only measure the memory required to execute the algorithm, as we do not need to store the whole document to be able to process it. We also do not count the memory to store the 1-SEVPA and its key graph. As the classical algorithm must have the complete document stored in memory, we sum the RAM consumption for the document and for the algorithm itself. This is coherent to what happens in actual web-service handling: Whenever a new validation request is received, we would spawn a new subprocess that handles a specific document. Since the 1-SEVPA and its key graph are the same for all subprocesses, they would be loaded in a memory space shared by all processes.

Experimental results indicate that our algorithm exhibits good performance. Results for the three smaller schemas are not presented here to save space, while they are given in Figure 3 for *VIM plugins*, *Azure Functions Proxies*, and *codecov*. The blue (resp. red) crosses (resp. circles) give the values for our (resp. the classical) algorithm. The x-axis gives the size of each (abstracted) document.

For both *VIM plugins* and *Azure Functions Proxies*, our algorithm consumes less memory than the classical one. For these benchmarks, memory and time usage seemingly trade off as we see that our algorithm usually requires more time to validate a document — a majority of that time is spent computing the set $\mathrm{Valid}(K, Bad)$. This tradeoff, however, does not hold in general: There are schemas for which our algorithm performs better than the classical one, both in terms of time and memory, as it does not have to backtrack to validate a document, which reduces the time and memory space required.

For the *codecov* schema, we recall that the learning process was not completed, leading to an approximated 1-SEVPA with repeated keys in its key graph. This means that the computation of $\mathrm{Valid}(K, Bad)$ explores some invalid paths, increasing the memory and time consumed by our algorithm. Thus, it appears that, while a not completely learned 1-SEVPA can still be used in our algorithm, stopping the learning process early may increase the time and space required.

6 Future Work

As future work, one could focus on constructing the VPA directly from the schema, without going through a learning algorithm. While this task is easy if the schema does not contain Boolean operations, it is not yet clear how to proceed in the general case. Second, it could be worthwhile to compare our algorithm against an implementation of a classical algorithm used in the industry. This would require either to modify the industrial implementations to support abstractions, or to modify our algorithm to work on unabstracted JSON schemas. Third, in our validation approach, we decided to use a VPA accepting the JSON documents satisfying a fixed key order — thus requiring to use the key graph and its costly computation of the set $\mathrm{Valid}(K, Bad)$. It could be interesting to make additional experiments to compare this approach with one where we instead use a VPA accepting the JSON documents and all their key permutations — in this

case, reasoning on the key graph would no longer be needed. Finally, motivated by obtaining efficient querying algorithms on XML trees, the authors of [26] have introduced the concept of mixed automata in a way to accept subsets of unranked trees where some nodes have ordered sons and some other have unordered sons. It would be interesting to adapt our validation algorithm to different formalisms of documents, such as the one of mixed automata.

Data-Availability Statement. The source code and experimental results that support the findings of this study are available in Zenodo with the identifier https://doi.org/10.5281/zenodo.7309690 [31].

References

1. Richard A. DeMillo, Richard J. Lipton, Frederick G. Sayward: Hints on Test Data Selection: Help for the Practicing Programmer. Computer **11**(4), 34–41 (1978). https://doi.org/10.1109/C-M.1978.218136
2. Rajeev Alur, Viraj Kumar, Madhusudan, P., Mahesh Viswanathan: Congruences for Visibly Pushdown Languages. In: Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, Moti Yung (eds.) Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3580, pp. 1102–1114. Springer (2005). https://doi.org/10.1007/11523468_89
3. Rajeev Alur, Madhusudan, P.: Visibly pushdown languages. In: László Babai (ed.) Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004. pp. 202–211. ACM (2004). https://doi.org/10.1145/1007352.1007390
4. Dana Angluin: Learning Regular Sets from Queries and Counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
5. Iovka Boneva, Radu Ciucanu, Slawek Staworko: Schemas for Unordered XML on a DIME. Theory Comput. Syst. **57**(2), 337–376 (2015). https://doi.org/10.1007/s00224-014-9593-1
6. Tim Bray: The JavaScript Object Notation (JSON) Data Interchange Format. RFC **8259**, 1–16 (2017). https://doi.org/10.17487/RFC8259
7. Véronique Bruyère, Guillermo A. Pérez, Gaëtan Staquet: Learning Realtime One-Counter Automata. In: Dana Fisman, Grigore Rosu (eds.) International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 13243, pp. 244–262. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_13
8. John E. Hopcroft, Jeffrey D. Ullman: Introduction to Automata Theory, Languages and Computation, Second Edition. Addison-Wesley (2000)
9. Malte Isberner: Foundations of active automata learning: an algorithmic perspective. Ph.D. thesis, Technical University Dortmund, Germany (2015), http://hdl.handle.net/2003/34282
10. Malte Isberner, Falk Howar, Bernhard Steffen: The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In: Borzoo Bonakdarpour, Scott A. Smolka (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer (2014). https://doi.org/10.1007/978-3-319-11164-3_26

11. 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)
12. Yue Jia, Mark Harman: An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Software Eng. **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
13. JSON Schema: JSON Schema website. Online (2015), https://json-schema.org
14. JSON.org: JSON.org website. Online (2013), https://www.json.org
15. Viraj Kumar, Madhusudan, P., Mahesh Viswanathan: Visibly pushdown automata for streaming XML. In: Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, Prashant J. Shenoy (eds.) Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007. pp. 1053–1062. ACM (2007). https://doi.org/10.1145/1242572.1242714
16. Wim Martens, Frank Neven, Matthias Niewerth, Thomas Schwentick: BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema. ACM Trans. Database Syst. **42**(3), 15:1–15:42 (2017). https://doi.org/10.1145/3105960
17. Maik Merten, Bernhard Steffen, Falk Howar, Tiziana Margaria: Next Generation LearnLib. In: Parosh Aziz Abdulla, Rustan M. Leino, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6605, pp. 220–223. Springer (2011). https://doi.org/10.1007/978-3-642-19835-9_18
18. Matthias Niewerth, Thomas Schwentick: Reasoning About XML Constraints Based on XML-to-Relational Mappings. Theory Comput. Syst. **62**(8), 1826–1879 (2018). https://doi.org/10.1007/s00224-018-9846-5
19. Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, Domagoj Vrgoc: Foundations of JSON Schema. In: Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, Ben Y. Zhao (eds.) Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016. pp. 263–273. ACM (2016). https://doi.org/10.1145/2872427.2883029
20. Schema for Azure Functions Proxies. Online, https://json.schemastore.org/proxies.json
21. Schema for codecov configuration. Online, https://json.schemastore.org/codecov.json
22. Schema for VIM plugins. Online, https://json.schemastore.org/vim-addon-info.json
23. Schema for Visual Studio Code snippets. Online, https://raw.githubusercontent.com/Yash-Singh1/vscode-snippets-json-schema/main/schema.json
24. Thomas Schwentick: Foundations of XML Based on Logic and Automata: A Snapshot. In: Thomas Lukasiewicz, Attila Sali (eds.) Foundations of Information and Knowledge Systems - 7th International Symposium, FoIKS 2012, Kiel, Germany, March 5-9, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7153, pp. 23–33. Springer (2012). https://doi.org/10.1007/978-3-642-28472-4_2
25. Luc Segoufin, Victor Vianu: Validating Streaming XML Documents. In: Lucian Popa, Serge Abiteboul, Phokion G. Kolaitis (eds.) Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA. pp. 53–64. ACM (2002). https://doi.org/10.1145/543613.543622

26. Helmut Seidl, Thomas Schwentick, Anca Muscholl: Counting in trees. In: Jörg Flum, Erich Grädel, Thomas Wilke (eds.) Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]. Texts in Logic and Games, vol. 2, pp. 575–612. Amsterdam University Press (2008)
27. Staquet, G.: AutomataLib, https://github.com/DocSkellington/automatalib
28. Staquet, G.: JSON Schema Tools, https://github.com/DocSkellington/JSONSchemaTools
29. Staquet, G.: LearnLib, https://github.com/DocSkellington/learnlib
30. Staquet, G.: Validating JSON Documents With Learned VPA, https://github.com/DocSkellington/ValidatingJSONDocumentsWithLearnedVPA
31. Staquet, G.: Validating Streaming JSON Documents with Learned VPAs (Nov 2022). https://doi.org/10.5281/zenodo.7309689
32. Nguyen Van Tang: A Tighter Bound for the Determinization of Visibly Pushdown Automata. In: Axel Legay (ed.) Proceedings International Workshop on Verification of Infinite-State Systems, INFINITY 2009, Bologna, Italy, 31th August 2009. EPTCS, vol. 10, pp. 62–76 (2009). https://doi.org/10.4204/EPTCS.10.5


Antichains Algorithms for the Inclusion Problem Between ω-VPL*

Kyveli Doveri[1,2(✉)], Pierre Ganty[1], and Luka Hadži-Đokić[1]

[1] IMDEA Software Institute, Madrid, Spain
{kyveli.doveri,pierre.ganty,luka.hadzi-dokic}@imdea.org
[2] Universidad Politécnica de Madrid, Madrid, Spain

Abstract. We define novel algorithms for the inclusion problem between two visibly pushdown languages of infinite words, an EXPTime-complete problem. Our algorithms search for counterexamples to inclusion in the form of ultimately periodic words i.e. words of the form uv^ω where u and v are finite words. They are parameterized by a pair of quasiorders telling which ultimately periodic words need not be tested as counterexamples to inclusion without compromising completeness. The pair of quasiorders enables distinct reasoning for prefixes and periods of ultimately periodic words thereby allowing to discard even more words compared to using the same quasiorder for both. We put forward two families of quasiorders: the state-based quasiorders based on automata and the syntactic quasiorders based on languages. We also implemented our algorithm and conducted an empirical evaluation on benchmarks from software verification.

1 Introduction

Visibly pushdown languages [4] (VPL) have applications in various domains including verification [22], theorem proving [27] or XML schema languages reasoning [26] where the inclusion problem plays a crucial role. For instance proving correctness relative to a specification reduces to a language inclusion problem and so does proving correctness of a theorem of the form $\forall x \exists y P(x) \implies Q(y)$. The extension to the case of visibly pushdown languages of infinite words (ω-VPL) has also been studied in the context of program verification [21] and it has applications in word combinatorics [23,25,27].

We distinguish two general approaches to solve the language inclusion problem $L \subseteq M$: (i) complement M, intersect with L and check for emptiness of the result; and (ii) reduce the inclusion check to finitely many *membership queries* asking whether $w \in M$ holds where $w \in L$ and each query aims at finding a counterexample to inclusion.

* This work was partially funded by the ESF Investing in your future, the RYC-2016-20281/MCIN/AEI/10.13039/501100011033, the Madrid regional government as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union, the PRODIGY Project (TED2021-132464B-I00) funded by MCIN and the European Union NextGenerationEU/ PRTR.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 290–307, 2023.
https://doi.org/10.1007/978-3-031-30823-9_15

In this paper we focus on the second approach. Previous work in that space leverage relations between words to select a finite subset of words of L on which we run the membership queries. A class of relations that consistently yields good results in practice are *quasiorders* which discard words subsumed (for the quasiorder) by others. A key feature of such quasiorders is that the subset of L selected via the quasiorder must contain a counterexample to inclusion if there exists one. Quasiorders are a versatile heuristic that has been applied to inclusion problems for languages such as languages of finite words [3,10,14] (including visibly pushdown language [6]) or infinite words [1,2,12,13,16,24] and even tree languages [3,5]. Algorithms leveraging quasiorders are commonly referred to as *antichains algorithms*. Subsequent improvements (e.g. [2] improving [1]) often attempt at defining coarser quasiorders because they enable the selection of an even smaller subset of L.

Let us now turn to the inclusion problem between ω-VPL, an EXPTIME-complete problem. For that problem the selection of words of L is limited to *ultimately periodic words*, i.e. words of the form uv^ω, where u and v are called *prefix* and *period* respectively. For an ultimately periodic word uv^ω subsumption (for a quasiorder) simply means subsumption of (u, v) relative to a pair $\leqq_1 \times \leqq_2$ of quasiorders on finite words. The quasiorders found in the literature [17,18] are all equivalences and are all such that $\leqq_1 = \leqq_2$.

In this paper, we propose a new family of algorithms for the inclusion problem between ω-VPL that leverages a subset of the ultimately periodic words, deemed *legitimate decompositions* and is parameterized by a pair of quasiorders and a decision procedure for the membership queries in M. We identify properties that such pair of quasiorders must satisfy so that the resulting algorithm actually decides the inclusion problem between two ω-VPL: (1) be decidable; (2) be well-quasiorders; (3) verify some monotonicity conditions w.r.t. word operations that are characteristic to ω-VPL and (4) satisfy a preservation property intuitively saying that a legitimate decomposition inside M cannot subsume a legitimate decomposition outside of M. We put forward two families of quasiorders satisfying (1) thru (4): the *state-based quasiorders* whose definition rely on a visibly pushdown automaton underlying M and the *syntactic quasiorders* whose definition is based solely on M. The syntactic orders are the "ideal" quasiorders in the sense they are the coarsest, hence they select the "smallest" subset of L. None of our quasiorders is symmetric, hence they are coarser than equivalences and in each and every pair we define the quasiorder on prefixes differs from the one on periods (i.e. $\leqq_1 \neq \leqq_2$). We further prove that when instantiated with the state-based quasiorders and with a state-based decision procedure for membership queries the resulting algorithm, which we call the *state-based algorithm*, has a runtime that matches the corresponding problem complexity.

Finally we implement the state-based algorithm and evaluate it on various benchmarks collected from Friedmann et al. [18] and from SV-COMP[3], the Software Verification competition. The empirical evaluation is carried out against Ultimate [21] which follows a complement, intersect and check for emptiness

[3] https://sv-comp.sosy-lab.org

approach. The preliminary conclusion of the empirical results is in favor of our approach as it scales up better.

Related Work. Bruyere et al. [6] proposed an antichain algorithm for the inclusion of VPL but they only tackle the problem for languages of finite words. The same limitation applies to Ganty et al. [19,20] where, moreover, they do not tackle the inclusion problem of VPL into VPL (the closest they tackle is CFL into regular). The extension from the finite to the infinite case was tackled in Doveri et al. [13] but they do not cover the case ω-VPL into ω-VPL (the closest they tackle is ω-CFL into ω-regular). Friedmann et al. [17,18] do tackle the ω-VPL into ω-VPL problem. However they do not leverage the full power of quasiorders (they use equivalence instead); they do not use distinct pruning techniques for prefix and periods; and they do not put forward syntactic quasiorders. A summary comparing our work (omegaVPLinc) with the closest works in the area is given at Table 1.

Table 1. Comparison of the closest work in the area based on the characteristics of the problem tackled (first two columns) and the techniques used (last three columns). N/A means non applicable, ○ means no support and ● means full support. The labels ω, VPL, qo, $\leqq_1 \neq \leqq_2$ and syntactic qo ask respectively whether the work thereof tackles the problem of infinite words, tackles the problem of VPL, leverage quasiorders, defines distinct quasiorders for prefixes and periods, and defines syntactic quasiorders.

	ω	VPL	qo	$\leqq_1 \neq \leqq_2$	syntactic qo
Bruyere et al. [6]	○	●	●	N/A	○
Ganty et al. [20]	○	○	●	N/A	●
Doveri et al. [13]	●	○	●	●	○
Friedmann et al. [18]	●	●	○	○	○
omegaVPLinc	●	●	●	●	●

2 Background

Fix $\Sigma \triangleq \Sigma_i \cup \Sigma_c \cup \Sigma_r$ an alphabet (a finite non empty set of symbols) comprising three disjoint alphabets. The set of finite words and the set of infinite words over Σ are denoted by Σ^* and Σ^ω respectively. We denote by ϵ the empty word and define $\Sigma^+ \triangleq \Sigma^* \backslash \{\epsilon\}$. Given a word $u = u_0 u_1 \cdots \in \Sigma^* \cup \Sigma^\omega$ we say that a position j where $j \in \mathbb{N}$, $j < |u|$ and $|u| \in \mathbb{N} \cup \{\omega\}$ is the length of u, is an *internal* (resp. *call*, resp. *return*) position if $u_j \in \Sigma_i$ (resp. $u_j \in \Sigma_c$, resp. $u_j \in \Sigma_r$).

Visibly Pushdown Languages. A Visibly Pushdown Automaton (VPA) over Σ is a tuple $\mathcal{A} = (Q, q_I, \Gamma, \delta, F)$, where Q is a finite set of states including an initial state $q_I \in Q$, $F \subseteq Q$ is the set of final states, Γ is the stack alphabet including a bottom-of-stack symbol $\perp$ and $\delta = \delta_i \cup \delta_c \cup \delta_r$ consists of three transition relations $\delta_i \subseteq Q \times \Sigma_i \times Q$, $\delta_c \subseteq Q \times \Sigma_c \times Q \times \Gamma \backslash \{\perp\}$ and $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$. *Configurations* in $\mathcal{A}$ are pairs in $Q \times \Gamma^*$. For $a \in \Sigma$ we define the relation $\vdash^a$ between configurations as follows:

- If $a \in \Sigma_i$ and $w \in \Gamma^*$ we have $(p, w) \vdash^a (q, w)$ if $(p, a, q) \in \delta_i$.
- If $a \in \Sigma_c$ and $w \in \Gamma^*$ we have $(p, w) \vdash^a (q, w\gamma)$ if $(p, a, q, \gamma) \in \delta_c$.
- If $a \in \Sigma_r$, $\gamma \in \Gamma \backslash \{\bot\}$ and $w \in \Gamma^*$ we have $(p, w\gamma) \vdash^a (q, w)$ if $(p, a, \gamma, q) \in \delta_r$.
- If $a \in \Sigma_r$ we have $(p, \bot) \vdash^a (q, \bot)$ if $(p, a, \bot, q) \in \delta_r$.

We lift the relation $\vdash$ to words by transitivity and reflexivity, that is, for all $u \in \Sigma^*$, $(q, w) \vdash^{*u} (p, w')$ when the configurations (q, w) and (p, w') are related by a sequence of transitions such that the concatenation of the corresponding labels is the word u. We write $(q, w) \vdash^{\circledast u} (p, w')$ when such a sequence includes a configuration whose state is final. A *trace* of $\mathcal{A}$ on a infinite word $\xi = a_0 a_1 \cdots \in \Sigma^\omega$ is an infinite sequence $(q_0, w_0) \vdash^{a_0} (q_1, w_1) \vdash^{a_1} \cdots$ It is a *final trace* when $q_j \in F$ for infinitely many j's. It is an *accepting trace* when it is a final trace and $(q_0, w_0) = (q_I, \bot)$. The ω-language accepted by $\mathcal{A}$ is $L^\omega(\mathcal{A}) \triangleq \{\xi \in \Sigma^\omega \mid$ there is an accepting trace of $\mathcal{A}$ on $\xi\}$. A language $L \subseteq \Sigma^\omega$ is ω-VPL if $L = L^\omega(\mathcal{A})$ for some VPA $\mathcal{A}$. Two examples of VPA are given at Fig. 1, $\mathcal{A}$ has an accepting trace on $c\,cr\,cr\,cr\ldots$ and so does $\mathcal{B}$ on $crr\,crr\ldots$

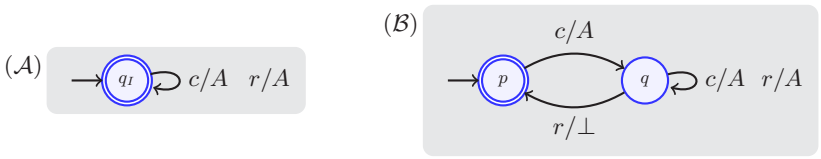


Fig. 1. Two ω-VPA with $\Gamma = \{A, \bot\}$, $\Sigma_i = \emptyset$, $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

Ultimately Periodic Words. An *ultimately periodic word* is an infinite word $\xi \in \Sigma^\omega$ such that $\xi = uv^\omega$ for some finite *prefix* $u \in \Sigma^*$ and some finite *period* $v \in \Sigma^+$. We call the couple $(u, v) \in \Sigma^* \times \Sigma^+$ a *decomposition* of ξ. Note that ξ admits infinitely many decompositions.

Ultimately periodic words play a central role in our approach as they suffice for the inclusion problem as shown by the following theorem. [4]

Theorem 1. *Let $L, M \subseteq \Sigma^\omega$ be ω-VPL. Then, $L \subseteq M$ iff $\forall uv^\omega \in L, uv^\omega \in M$.*

Matching Relation. The partition of the alphabet $\Sigma = \Sigma_i \cup \Sigma_c \cup \Sigma_r$ induces a unique matching relation between a word's *call* and *return* positions (see [18]). Given $u \in \Sigma^* \cup \Sigma^\omega$ define the *matching relation* of u, denoted $\curvearrowright_u$, as the unique relation on its call and return positions such that for every $j \curvearrowright_u k$ we have $0 \leq j < k < |u|$, $u_j \in \Sigma_c$, $u_k \in \Sigma_r$, $|\{n \mid j \curvearrowright_u n\}| \leq 1$, $|\{n \mid n \curvearrowright_u k\}| \leq 1$ and there are no j', k' with $j' \curvearrowright_u k'$ and $j < j' < k < k'$. Given $j \curvearrowright_u k$ we say that j and k are *matched* positions. A call (resp. return) position j in u is *unmatched*

[4] Theorem 1 can be easily obtained by adapting the proof of Fact 1 in [7].

if $j \curvearrowright_u k$ (resp. $k \curvearrowright_u j$) for no k. Furthermore, for every unmatched positions n in u there is no $j \curvearrowright_u k$ such that $j < n < k$, and if $u_n \in \Sigma_c$ (resp. $u_n \in \Sigma_r$) then there is no unmatched return (resp. call) position k with $n < k$ (resp. $k < n$). A word is said to be *well-matched* if it has no unmatched position.

3 Foundations

In this section we outline our approach which, given a VPA $\mathcal{A} = (Q, q_I, \Gamma, \delta, F)$ and an ω-VPL M, reduces the inclusion problem $L^\omega(\mathcal{A}) \subseteq M$ to finitely many membership queries in M. More precisely, we derive a finite subset S_{finite} of ultimately periodic words of $L^\omega(\mathcal{A})$ such that

$$L^\omega(\mathcal{A}) \subseteq M \iff \forall (u,v) \in S_{\text{finite}},\ uv^\omega \in M \ . \tag{†}$$

Reduction to Legitimate Decompositions. Our first step is to reduce the inclusion check to a subset of ultimately words of $L^\omega(\mathcal{A})$ given by legitimate decompositions. To do so, we define $\mathtt{W}$ as the set of well-matched finite words, $\mathtt{C}$ (resp. $\mathtt{R}$) as the set of finite words where all call (resp. return) positions are matched and $\mathtt{U_c}$ as the set of finite words with at least one unmatched call position. In turn, we define the set of *legitimate decompositions* given by

$$\mathtt{Ld} \triangleq \mathtt{C} \times \mathtt{C} \ \cup \ \mathtt{U_c} \times \mathtt{R}$$

which, as shown next, is sufficient for the inclusion problem between ω-VPL.

Theorem 2. *Let $L, M \subseteq \Sigma^\omega$ be ω-VPL. Then, $L \subseteq M$ iff $\forall (u,v) \in \mathtt{Ld}$, $uv^\omega \in L \implies uv^\omega \in M$.*

Next we leverage the relations $\vdash^*$ and $\vdash^\circledast$ of $\mathcal{A}$ to characterize the legitimate decompositions of the ultimately periodic words of $L^\omega(\mathcal{A})$. We start by defining the following languages of finite words for each pair $p, q \in Q$ of state of $\mathcal{A}$: $L_{p,q} \triangleq \{u \in \Sigma^* \mid \exists w \in \Gamma^*,\ (p, \bot) \vdash^{*u} (q, w)\}$ and $L^\circledast_{p,q} \triangleq \{u \in \Sigma^+ \mid \exists w \in \Gamma^*,\ (p, \bot) \vdash^{\circledast u} (q, w)\}$. Finally, define the following subset of $\mathtt{Ld}$:

$$S \triangleq \bigcup_{p \in Q} L_{q_I, p_{|\mathtt{C}}} \times L^\circledast_{p, p_{|\mathtt{C}}} \ \cup \ L_{q_I, p_{|\mathtt{U_c}}} \times L^\circledast_{p, p_{|\mathtt{R}}}$$

where $L_{|K}$ is defined to be $L \cap K$ to emphasize that L is restricted to K.

Example 1. Consider the VPA $\mathcal{A}$ and $\mathcal{B}$ depicted in Fig. 1. We have $L^\omega(\mathcal{A}) = \mathtt{R}^\omega$, $S = (\mathtt{W} \times \mathtt{W} \backslash \{\epsilon\}) \cup (\mathtt{R} \backslash \mathtt{C} \times \mathtt{R} \backslash \{\epsilon\})$ and $L^\omega(\mathcal{B}) = ((\mathtt{W} \backslash \{\epsilon\})r)^\omega$.

Proposition 1. *We have that $uv^\omega \in L^\omega(\mathcal{A}) \iff \exists (u', v') \in S,\ uv^\omega = u'v'^\omega$.*

By Theorem 2 and Proposition 1 the subset S verifies:

$$L^\omega(\mathcal{A}) \subseteq M \iff \forall (u,v) \in S,\ uv^\omega \in M \ . \tag{1}$$

Next we reduce the inclusion check to a finite subset of S using quasiorders.

Reduction to a Finite Basis. A *quasiorder* (qo) on a set E, is a reflexive and transitive relation $\ltimes \subseteq E \times E$. Given two subsets $X, Y \subseteq E$ the set Y is said to be a *basis* for X with respect to $\ltimes$ whenever $Y \subseteq X$ and $\forall x \in X, \exists y \in Y, y \ltimes x$. A qo $\ltimes$ is a *well-quasiorder* (wqo) if every subset of E admits a finite basis.

We obtain S_{finite} as a finite basis for S with respect to $\leqslant \times \preccurlyeq$ for a pair $\leqslant, \preccurlyeq$ of wqos.[5] To guarantee the direction $\Leftarrow$ in Eq. (†) we need the pair $\leqslant, \preccurlyeq$ to be *M-preserving*, a notion we introduce below.

A pair $\leqslant, \preccurlyeq$ of qos on Σ^* is said to be M-preserving if for all $(u, v), (u', v') \in \texttt{Ld}$ such that $(u, v), (u', v') \in \texttt{C} \times \texttt{C}$ or $(u, v), (u', v') \in \texttt{U}_\texttt{c} \times \texttt{R}$,

$$\text{if } uv^\omega \in M, u \leqslant u' \text{ and } v \preccurlyeq v' \text{ then } u'v'^\omega \in M \ .$$

Intuitively, M-preservation guarantees that if the inclusion does not hold then the finite basis S_{finite} contains a counterexample.

Next, we fix a pair of M-preserving wqos $\leqslant, \preccurlyeq$ and show the existence of a subset S_{finite} such that Eq. (†) holds. Since $\leqslant \times \preccurlyeq$ is a wqo, there exist two finite bases S_1 and S_2 for $S_{|\texttt{C}\times\texttt{C}}$ and $S_{|\texttt{U}_\texttt{c}\times\texttt{R}}$ respectively w.r.t. $\leqslant \times \preccurlyeq$. We define S_{finite} to be the union of such sets S_1, S_2, viz. $S_{\text{finite}} \triangleq S_1 \cup S_2 \subseteq S$. We have that: $\forall (u, v) \in S,\ uv^\omega \in M \implies \forall (u, v) \in S_{\text{finite}},\ uv^\omega \in M$. We now turn to the converse implication. Assume that $\forall (u, v) \in S_{\text{finite}},\ uv^\omega \in M$. Let $(u, v) \in S$. If $(u, v) \in S_{|\texttt{C}\times\texttt{C}}$ then there is $(u_0, v_0) \in S_1$ such that $(u_0, v_0) \leqslant \times \preccurlyeq (u, v)$. Since $S_1 \subseteq S_{|\texttt{C}\times\texttt{C}} \subseteq \texttt{C} \times \texttt{C}$ we have that $(u_0, v_0), (u, v) \in \texttt{C} \times \texttt{C}$. Since $u_0 v_0^\omega \in M$ and the pair $\leqslant, \preccurlyeq$ is M-preserving, we conclude that $uv^\omega \in M$. The case $(u, v) \in S_{|\texttt{U}_\texttt{c}\times\texttt{R}}$ proceeds analogously. It follows that $\forall (u, v) \in S,\ uv^\omega \in M \impliedby \forall (u, v) \in S_{\text{finite}},\ uv^\omega \in M$. Hence, we derive Equation (†) using Equation (1).

In Section 4, we give a fixpoint characterization of S and in Section 5 we show that under some monotonicity conditions on the wqos $\leqslant$ and $\preccurlyeq$ we can effectively compute a finite basis for S. We then give two examples of monotonic pairs of wqos in Section 6. In Section 7 we present our algorithm which given two VPA $\mathcal{A}$ and $\mathcal{B}$ decides the inclusion problem $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$. Therein we discuss the state-based algorithm and give an upper bound on its running time. Finally in Section 8 we report on an empirical evaluation.

4 Fixpoint Characterization

In this section we give a least fixpoint characterization of S for the VPA $\mathcal{A} = (Q, q_I, \Gamma, \delta, F)$. To this end we work with the complete lattice $(\wp(\Sigma^*)^{n\cdot|Q|^2}, \subseteq \times \cdots \times \subseteq)$, where $n \in \{4, 6\}$ and each Cartesian product consists of $n \cdot |Q|^2$ factors.

For a function $f\colon E \to E$ on a quasiordered set $(E, \ltimes)$ and for all $n \in \mathbb{N}$, we define the n-th iterate $f^n : E \to E$ of f inductively as follows: $f^0 \triangleq \lambda x.x$; $f^{n+1} \triangleq f \circ f^n$. The denumerable sequence of *Kleene iterates* of f starting from the bottom value $\bot \in E$ is given by $\{f^n(\bot)\}_{n\in\mathbb{N}}$. Recall that when $(E, \ltimes)$ is a complete lattice and $f\colon E \to E$ is a monotone function (i.e. $d \ltimes d' \implies$

[5] The qo $\leqslant \times \preccurlyeq$ is a wqo when both $\leqslant$ and $\preccurlyeq$ are wqos.

$f(d) \ltimes f(d'))$ then by the Knaster–Tarski theorem, f has a least fixpoint $\mathrm{lfp}\, f$ given by the supremum of the ascending[6] sequence of Kleene iterates of f.

Given a $n \cdot |Q|^2$-dimensional vector X and a $|Q|^2$-dimensional vector Y on $\wp(\Sigma^*)$ we write $X_{i,p,q}$, for the (i,p,q)-component of X and $Y_{p,q}$ for the (p,q)-component of Y. We define the following equations where $X, X' \in \wp(\mathtt{W})^{|Q|^2}$, $Y, Y' \in \wp(\mathtt{C})^{|Q|^2}$, $Z, Z' \in \wp(\mathtt{R})^{|Q|^2}$, and $T \in \wp(\mathtt{U_c})^{|Q|^2}$:

$$W(X) = \langle L_{p,q\,|\,(\Sigma_i \cup \{\epsilon\})} \cup \bigcup_{\substack{(p,c,p',\gamma)\in\delta_c,\\(q',r,\gamma,q)\in\delta_r}} cX_{p',q'}r \cup \bigcup_{q'\in Q} X_{p,q'}\, X_{q',q}\rangle_{p,q\in Q}$$

$$C(X,Y) = \langle L_{p,q\,|\,\Sigma_r} \cup X_{p,q} \cup \bigcup_{q'\in Q} Y_{p,q'}Y_{q',q}\rangle_{p,q\in Q}$$

$$R(X,Z) = \langle L_{p,q\,|\,\Sigma_c} \cup X_{p,q} \cup \bigcup_{q'\in Q} Z_{p,q'}Z_{q',q}\rangle_{p,q\in Q}$$

$$U(Y,Z,T) = \langle L_{p,q\,|\,\Sigma_c} \cup \bigcup_{p',q'\in Q,} Y_{p,p'}T_{p',q'}Z_{q',q}\rangle_{p,q\in Q}$$

$$W_{\circledast}(X,X') = \langle L^{\circledast}_{p,q\,|\,\Sigma_i} \cup \bigcup_{\substack{(p,c,p',\gamma)\in\delta_c,\\(q',r,\gamma,q)\in\delta_r,\\\{p,q\}\cap F\neq\emptyset}} cX_{p',q'}r \cup \bigcup_{\substack{(p,c,p',\gamma)\in\delta_c,\\(q',r,\gamma,q)\in\delta_r\\\{p,q\}\cap F=\emptyset}} cX'_{p',q'}r \cup \bigcup_{q'\in Q} (X'_{p,q'}X_{q',q} \cup X_{p,q'}X'_{q',q})\rangle_{p,q\in Q}$$

$$C_{\circledast}(X',Y,Y') = \langle L^{\circledast}_{p,q\,|\,\Sigma_r} \cup X'_{p,q} \cup \bigcup_{q'\in Q} (Y'_{p,q'}Y_{q',q} \cup Y_{p,q'}Y'_{q',q})\rangle_{p,q\in Q}$$

$$R_{\circledast}(X',Z,Z') = \langle L^{\circledast}_{p,q\,|\,\Sigma_c} \cup X'_{p,q} \cup \bigcup_{q'\in Q} (Z'_{p,q'}Z_{q',q} \cup Z_{p,q'}Z'_{q',q})\rangle_{p,q\in Q}\ .$$

The equations W, C, R and U are used to obtain the set of words in $\mathtt{W}$, $\mathtt{C}$, $\mathtt{R}$ and $\mathtt{U_c}$ respectively, that connect two configurations of $\mathcal{A}$. The equations $W_{\circledast}$, $C_{\circledast}$ and $R_{\circledast}$ refine those of W, C and R by filtering out words not visiting final states. In turn we define the functions $f_{\mathcal{A}}$ and $r_{\mathcal{A}}$ used to obtain the prefixes u and the periods v respectively for the decompositions $(u,v) \in S$. Define

$$\begin{aligned} f_{\mathcal{A}} \colon \wp(\Sigma^*)^{4\cdot|Q|^2} &\longrightarrow \wp(\Sigma^*)^{4\cdot|Q|^2} \\ (X,Y,Z,T) &\longmapsto (W(X), C(X,Y), R(X,Z), U(Y,Z,T)) \end{aligned}$$

for the prefixes, and for the periods define

$$\begin{aligned} r_{\mathcal{A}} \colon \wp(\Sigma^*)^{6\cdot|Q|^2} &\longrightarrow \wp(\Sigma^*)^{6\cdot|Q|^2} \\ (X,Y,Z,X',Y',Z') &\longmapsto \big(W(X), C(X,Y), R(X,Z), W_{\circledast}(X,X'), C_{\circledast}(X',Y,Y'), R_{\circledast}(X',Z,Z')\big)\ . \end{aligned}$$

The function $f_{\mathcal{A}}$ (resp. $r_{\mathcal{A}}$) is monotone and the supremum of the ascending sequence of its Kleene iterates starting at the bottom value $\vec{\emptyset} \triangleq (\emptyset, \ldots, \emptyset)$ of dimension $4 \cdot |Q|^2$ (resp. $6 \cdot |Q|^2$) is the vector $(\Lambda_{|\mathtt{W}}, \Lambda_{|\mathtt{C}}, \Lambda_{|\mathtt{R}}, \Lambda_{|\mathtt{U_c}})$ (resp. $(\Lambda_{|\mathtt{W}}, \Lambda_{|\mathtt{C}}, \Lambda_{|\mathtt{R}}, \Lambda^{\circledast}_{|\mathtt{W}}, \Lambda^{\circledast}_{|\mathtt{C}}, \Lambda^{\circledast}_{|\mathtt{R}})$) where $\Lambda_{|\mathtt{J}} \triangleq \langle L_{p,q\,|\,\mathtt{J}}\rangle_{p,q\in Q}$ and $\Lambda^{\circledast}_{|\mathtt{J}} \triangleq \langle L^{\circledast}_{p,q\,|\,\mathtt{J}}\rangle_{p,q\in Q}$ for $\mathtt{J} \in \{\mathtt{W}, \mathtt{C}, \mathtt{R}, \mathtt{U_c}\}$. Therefore, by the Knaster–Tarski theorem we obtain the following proposition.

Proposition 2. $\mathrm{lfp}\, f_{\mathcal{A}} = (\Lambda_{|\mathtt{W}}, \Lambda_{|\mathtt{C}}, \Lambda_{|\mathtt{R}}, \Lambda_{|\mathtt{U_c}})$ *and* $\mathrm{lfp}\, r_{\mathcal{A}} = (\Lambda_{|\mathtt{W}}, \Lambda_{|\mathtt{C}}, \Lambda_{|\mathtt{R}}, \Lambda^{\circledast}_{|\mathtt{W}}, \Lambda^{\circledast}_{|\mathtt{C}}, \Lambda^{\circledast}_{|\mathtt{R}})$.

[6] A sequence $\{s_n\}_{n\in\mathbb{N}} \in E^{\mathbb{N}}$ on an ordered set $(E, \ltimes)$ is ascending if for every $n \in \mathbb{N}$ we have $s_n \ltimes s_{n+1}$.

Finally, by Proposition 2, we obtain the desired fixpoint characterization of S:

$$S = \bigcup_{p\in Q}\Big(\big((\operatorname{lfp} f_{\mathcal{A}})_{2,q_I,p} \times (\operatorname{lfp} r_{\mathcal{A}})_{5,p,p}\big) \cup \big((\operatorname{lfp} f_{\mathcal{A}})_{4,q_I,p} \times (\operatorname{lfp} r_{\mathcal{A}})_{6,p,p}\big)\Big) \ . \quad (2)$$

Example 2. We derive from the VPA $\mathcal{A}$ depicted in Fig. 1 the following functions

$$W(X) \triangleq \{\epsilon\} \cup cXr \cup XX, \qquad C(X,Y) \triangleq X \cup YY,$$
$$R(X,Z) \triangleq \{c\} \cup X \cup ZZ, \qquad U(Y,Z,T) \triangleq \{c\} \cup YTZ \ .$$

Hence, we obtain the function

$$f_{\mathcal{A}} \colon \wp(\Sigma^*)^4 \longrightarrow \wp(\Sigma^*)^4$$
$$(X,Y,Z,T) \longmapsto (W(X), C(X,Y), R(X,Z), U(Y,Z,T)) \ .$$

The first three iterates of the least fixpoint computation of $\operatorname{lfp} f_{\mathcal{A}}$ are given by

$$f_{\mathcal{A}}(\vec{\emptyset}) = (\{\epsilon\}, \emptyset, \{c\}, \{c\}),$$
$${f_{\mathcal{A}}}^2(\vec{\emptyset}) = (\{\epsilon, cr\}, \{\epsilon\}, \{\epsilon, c, c^2\}, \{c\}),$$
$${f_{\mathcal{A}}}^3(\vec{\emptyset}) = (\{\epsilon, cr, c^2r^2, (cr)^2\}, \{\epsilon, cr\}, \{\epsilon, cr, c, c^2, c^3, c^4\}, \{c, c^2, c^3\})$$
$$\vdots$$
$$\operatorname{lfp} f_{\mathcal{A}} = (\mathtt{W}, \mathtt{W}, \mathtt{R}, \mathtt{R}\backslash\mathtt{C})$$

Since the unique state of $\mathcal{A}$ is a final state we have that $L_{q_I,q_I} = L^{\circledast}_{q_I,q_I}$. Consequently, the function $f_{\mathcal{A}}$ suffices to describe both the set of prefixes and the set of periods of S given by $\big((\operatorname{lfp} f_{\mathcal{A}})_2 \times (\operatorname{lfp} f_{\mathcal{A}})_2\backslash\{\epsilon\}\big) \cup \big((\operatorname{lfp} f_{\mathcal{A}})_4 \times (\operatorname{lfp} f_{\mathcal{A}})_3\backslash\{\epsilon\}\big)$.

Each (i,p,q)-component of the Kleene iterates of $f_{\mathcal{A}}$ and $r_{\mathcal{A}}$ keeps a finite set of words. However, if the language $L^{\omega}(\mathcal{A})$ is infinite, the fixpoint computations of $\operatorname{lfp} f_{\mathcal{A}}$ and $\operatorname{lfp} r_{\mathcal{A}}$ do not terminate in a finite number of steps. Nevertheless, under some monotonicity assumptions on our wqos we show in the following section that we can compute a finite basis for S w.r.t. $\leqslant \times \preccurlyeq$ as a terminating fixpoint computation.

5 Monotonicity Requirements

In order to detect finite bases among the Kleene iterates of the functions defined in the previous section we replace the set inclusion on $\wp(\Sigma^*)$, used so far, with the qo $\sqsubseteq_{\ltimes} \subseteq \wp(\Sigma^*) \times \wp(\Sigma^*)$ defined by $X \sqsubseteq_{\ltimes} Y \overset{\triangle}{\iff} \forall x \in X, \exists y \in Y, y \ltimes x$. The qo $\sqsubseteq_{\ltimes}$ leverage the notion of basis: given $X \in \wp(\Sigma^*)$ a subset $Y \subseteq X$ is a basis for X with respect to $\ltimes$ whenever $X \sqsubseteq_{\ltimes} Y$.

In the following we lift the notion of basis to n-dimensional vectors component wise and work with the quasiordered sets $(\wp(\Sigma^*)^{n\cdot|Q|^2}, \ \sqsubseteq_{\ltimes}^{n\cdot|Q|^2})$, where $n \in \{4,6\}$ and the ordering $\sqsubseteq_{\ltimes}^{n\cdot|Q|^2}$ is given by the product $\sqsubseteq_{\ltimes} \times \cdots \times \sqsubseteq_{\ltimes}$ of $n \cdot |Q|^2$

factors. Given a pair $\leqslant, \preccurlyeq$ of wqos, the orderings $\sqsubseteq_{\leqslant}^{4\cdot|Q|^2}$ and $\sqsubseteq_{\preccurlyeq}^{6\cdot|Q|^2}$ are used to compare the Kleene iterates of the functions $f_{\mathcal{A}}$ and $r_{\mathcal{A}}$ respectively. For them to be apt to detect finite bases for the least fixpoints of these functions the qos $\leqslant$ and $\preccurlyeq$ need to verify some monotonicity conditions.

We introduce the monotonicity conditions $\mathbf{W}, \mathbf{C}, \mathbf{R}, \boldsymbol{C}_{\circledast}, \mathbf{R}_{\circledast}$ and $\mathbf{U}$ on a qo $\ltimes \subseteq \Sigma^* \times \Sigma^*$ as follows: for all $u, u' \in \Sigma^*$ such that $u \ltimes u'$

$$
\begin{array}{rl}
(\mathbf{W}) & \text{if } u, u' \in \mathtt{W} \text{ and } c \in \Sigma_c, r \in \Sigma_r \text{ then } cur \ltimes cu'r, \\
(\mathbf{C}) & \text{if } u, u' \in \mathtt{C} \text{ and } s \in \mathtt{C}, t \in \Sigma^* \text{ then } sut \ltimes su't, \\
(\mathbf{R}) & \text{if } u, u' \in \mathtt{R} \text{ and } s \in \Sigma^*, t \in \mathtt{R} \text{ then } sut \ltimes su't, \\
(\mathbf{U}) & \text{if } u, u' \in \mathtt{U_c} \text{ and } s \in \mathtt{C}, t \in \mathtt{R} \text{ then } sut \ltimes su't, \\
(\mathbf{C}_{\circledast}) & \text{if } u, u' \in \mathtt{C} \text{ and } s \in \mathtt{C}, t \in \mathtt{C} \text{ then } sut \ltimes su't, \\
(\mathbf{R}_{\circledast}) & \text{if } u, u' \in \mathtt{R} \text{ and } s \in \mathtt{R}, t \in \mathtt{R} \text{ then } sut \ltimes su't.
\end{array}
$$

A pair of qos $\leqslant, \preccurlyeq$ is *monotonic* if $\leqslant$ verifies $\mathbf{W}, \mathbf{C}, \mathbf{R}, \mathbf{U}$ and $\preccurlyeq$ verifies $\mathbf{W}, \mathbf{C}_{\circledast}, \mathbf{R}_{\circledast}$.

Proposition 3. *Let $\leqslant, \preccurlyeq$ be a pair of wqos. There is a positive integer n such that ${f_{\mathcal{A}}}^{n+1}(\vec{\emptyset}) \sqsubseteq_{\leqslant}^{4\cdot|Q|^2} {f_{\mathcal{A}}}^{n}(\vec{\emptyset})$ (resp. ${r_{\mathcal{A}}}^{n+1}(\vec{\emptyset}) \sqsubseteq_{\preccurlyeq}^{6\cdot|Q|^2} {r_{\mathcal{A}}}^{n}(\vec{\emptyset})$); and, if the pair of wqos is monotonic then $\operatorname{lfp} f_{\mathcal{A}} \sqsubseteq_{\leqslant}^{4\cdot|Q|^2} {f_{\mathcal{A}}}^{n}(\vec{\emptyset})$ (resp. $\operatorname{lfp} r_{\mathcal{A}} \sqsubseteq_{\preccurlyeq}^{6\cdot|Q|^2} {r_{\mathcal{A}}}^{n}(\vec{\emptyset})$).*

Each Kleene iterate of $f_{\mathcal{A}}$ and $r_{\mathcal{A}}$ is computable and given a decidable qo $\ltimes$ on Σ^* and two finite sets $X, Y \subseteq \Sigma^*$ it is decidable whether $X \sqsubseteq_{\ltimes} Y$ holds. Thus, given a monotonic pair $\leqslant, \preccurlyeq$ of decidable wqos, by Proposition 3, we can compute a finite basis for $\operatorname{lfp} f_{\mathcal{A}}$ w.r.t. $\leqslant$ and a finite basis for $\operatorname{lfp} r_{\mathcal{A}}$ w.r.t. $\preccurlyeq$. Hence, by Equation (2) we can compute a finite basis for S w.r.t. $\leqslant \times \preccurlyeq$.

6 Quasiorders for ω-VPL

In the following we present two families of qos to solve the inclusion problem $L^{\omega}(\mathcal{A}) \subseteq M$, the state-based qos which are derived from a VPA-representation of M and compare words according to the set of configurations each word connects in the VPA, and the syntactic qos which rely on the syntactic structure of M. We say that a pair of qos is *M-suitable* if it is an M-preserving and monotonic pair of decidable wqos. Intuitively, if a pair of qos is M-suitable then it can be used in our algorithm to decide the inclusion $L^{\omega}(\mathcal{A}) \subseteq M$.

State-based Quasiorders. Given a VPA $\mathcal{B} = (\hat{Q}, \hat{q_I}, \hat{\Gamma}, \hat{\delta}, \hat{F})$ we associate with each word $u \in \Sigma^*$ its *context* $\operatorname{ctx}^{\mathcal{B}}[u]$ and *final context* $\operatorname{ctx}^{\mathcal{B}}_{\circledast}[u]$ *in* $\mathcal{B}$ as follows:

$$
\begin{aligned}
\operatorname{ctx}^{\mathcal{B}}[u] &\triangleq \{(p,q) \in \hat{Q}^2 \mid \exists w \in \hat{\Gamma}^*, (p, \bot) \vdash^{*u} (q, w)\}, \\
\operatorname{ctx}^{\mathcal{B}}_{\circledast}[u] &\triangleq \{(p,q) \in \hat{Q}^2 \mid \exists w \in \hat{\Gamma}^*, (p, \bot) \vdash^{\circledast u} (q, w)\} \ .
\end{aligned}
$$

Hence we define the following qos on words in Σ^*:

$$
u \leqslant^{\mathcal{B}} u' \overset{\triangle}{\iff} \operatorname{ctx}^{\mathcal{B}}[u] \subseteq \operatorname{ctx}^{\mathcal{B}}[u'], \quad u \preccurlyeq^{\mathcal{B}} u' \overset{\triangle}{\iff} u \leqslant^{\mathcal{B}} u' \wedge \operatorname{ctx}^{\mathcal{B}}_{\circledast}[u] \subseteq \operatorname{ctx}^{\mathcal{B}}_{\circledast}[u'] \ .
$$

Proposition 4. *Let $\mathcal{B}$ be a VPA. The pair $\leqslant^{\mathcal{B}}, \preccurlyeq^{\mathcal{B}}$ is $L^\omega(\mathcal{B})$-suitable.*

Example 3. Consider the pair of qos $\leqslant^{\mathcal{B}}, \preccurlyeq^{\mathcal{B}}$ derived as explained above from $\mathcal{B}$ (Fig. 1) and the set $S = (\mathtt{W} \times \mathtt{W}\backslash\{\epsilon\}) \cup (\mathtt{R}\backslash\mathtt{C} \times \mathtt{R}\backslash\{\epsilon\})$ from Example 1. We have that $\mathrm{ctx}^{\mathcal{B}}[\epsilon] = \{(p,p),(q,q)\}$, $\mathrm{ctx}^{\mathcal{B}}_{\circledast}[\epsilon] = \{(p,p)\}$, $\mathrm{ctx}^{\mathcal{B}}[u] = \{(p,q),(q,q)\}$ and $\mathrm{ctx}^{\mathcal{B}}_{\circledast}[u] = \{(p,q)\}$ for every $u \in \mathtt{R}\backslash\{\epsilon\}$. We have that $\{c\}$ is a basis for $\mathtt{R}\backslash\{\epsilon\}$ w.r.t. $\preccurlyeq^{\mathcal{B}}$ since $c \preccurlyeq^{\mathcal{B}} u$ for every $u \in \mathtt{R}\backslash\{\epsilon\}$. Since $\mathtt{R}\backslash\mathtt{C} \subseteq \mathtt{R}\backslash\{\epsilon\}$ and $\{c\} \subseteq \mathtt{R}\backslash\mathtt{C}$ we deduce that $\{c\}$ is also a basis for $\mathtt{R}\backslash\mathtt{C}$ w.r.t $\leqslant^{\mathcal{B}}$. Similarly we deduce that $\{\epsilon, cr\}$ is basis for $\mathtt{W}$ w.r.t $\leqslant^{\mathcal{B}}$ and that $\{cr\}$ is a basis for $\mathtt{W}\backslash\{\epsilon\}$ w.r.t. $\preccurlyeq^{\mathcal{B}}$. Hence, $(\{\epsilon, cr\} \times \{cr\}) \cup (\{c\} \times \{c\})$ is a basis for S w.r.t. $\leqslant^{\mathcal{B}} \times \preccurlyeq^{\mathcal{B}}$.

Syntactic Quasiorders. Given a ω-VPL M we associate with each word $u \in \Sigma^*$ its *context* $\mathrm{ctx}^M[u]$ and *final context* $\mathrm{ctx}^M_{\circledast}[u]$ *in* M as follows:

$$\mathrm{ctx}^M[u] \triangleq \{(s,\xi) \in \Sigma^* \times \Sigma^\omega \mid su\xi \in M\},$$
$$\mathrm{ctx}^M_{\circledast}[u] \triangleq \{(s,t) \in \Sigma^* \times \Sigma^* \mid s(ut)^\omega \in M\} \ .$$

At first glance, we are tempted to define the syntactic qos from ctx^M and $\mathrm{ctx}^M_{\circledast}$ in the analogue way we defined the state-based qos from the contexts and final contexts relatively to a VPA. Although, this definition provides a pair of M-preserving qos, it does not guarantee that the pair is M-suitable. To overcome this, we impose the respect of the partition $\mathcal{P} \triangleq \{\mathtt{W}, \mathtt{C}\backslash\mathtt{W}, \mathtt{R}\backslash\mathtt{W}, \mathtt{U_c}\backslash\mathtt{R}\}$ of Σ^*, meaning that two words compare only if they belong to a same subset of $\mathcal{P}$. Additionally, given $\mathtt{J} \in \mathcal{P}$ we compare two words of $\mathtt{J}$ by considering a restriction of their context and final context in M which depends on $\mathtt{J}$. More precisely, we define the qo $\leqslant^M$ on Σ^* as the union $\bigcup_{\mathtt{J}\in\mathcal{P}} \leqslant^M_{\mathtt{J}}$ where for every $\mathtt{J} \in \mathcal{P}$, the qo $\leqslant^M_{\mathtt{J}} \subseteq \mathtt{J} \times \mathtt{J}$ is defined by

$$\begin{aligned}
u \leqslant^M_{\mathtt{W}} u' &\overset{\triangle}{\iff} \mathrm{ctx}^M[u] \subseteq \mathrm{ctx}^M[u'],\\
u \leqslant^M_{\mathtt{C}\backslash\mathtt{W}} u' &\overset{\triangle}{\iff} \mathrm{ctx}^M[u]_{|\mathtt{C}\times\Sigma^\omega} \subseteq \mathrm{ctx}^M[u']_{|\mathtt{C}\times\Sigma^\omega},\\
u \leqslant^M_{\mathtt{R}\backslash\mathtt{W}} u' &\overset{\triangle}{\iff} \mathrm{ctx}^M[u]_{|\Sigma^*\times\mathtt{R}^\omega} \subseteq \mathrm{ctx}^M[u']_{|\Sigma^*\times\mathtt{R}^\omega},\\
u \leqslant^M_{\mathtt{U_c}\backslash\mathtt{R}} u' &\overset{\triangle}{\iff} \mathrm{ctx}^M[u]_{|\mathtt{C}\times\mathtt{R}^\omega} \subseteq \mathrm{ctx}^M[u']_{|\mathtt{C}\times\mathtt{R}^\omega} \ .
\end{aligned}$$

Similarly, we define the qo $\preccurlyeq^M \triangleq \bigcup_{\mathtt{J}\in\mathcal{P}} \preccurlyeq^M_{\mathtt{J}}$ on Σ^* where for every $\mathtt{J} \in \mathcal{P}$, $\preccurlyeq^M_{\mathtt{J}} \subseteq \mathtt{J} \times \mathtt{J}$ is the qo defined by

$$\begin{aligned}
u \preccurlyeq^M_{\mathtt{W}} u' &\overset{\triangle}{\iff} u \leqslant^M_{\mathtt{W}} u' \ \wedge \ \mathrm{ctx}^M_{\circledast}[u] \subseteq \mathrm{ctx}^M_{\circledast}[u'],\\
u \preccurlyeq^M_{\mathtt{C}\backslash\mathtt{W}} u' &\overset{\triangle}{\iff} u \leqslant^M_{\mathtt{C}\backslash\mathtt{W}} u' \ \wedge \ \big(\mathrm{ctx}^M_{\circledast}[u]_{|\mathtt{C}\times\mathtt{C}} \subseteq \mathrm{ctx}^M_{\circledast}[u']_{|\mathtt{C}\times\mathtt{C}}\big),\\
u \preccurlyeq^M_{\mathtt{R}\backslash\mathtt{W}} u' &\overset{\triangle}{\iff} u \leqslant^M_{\mathtt{R}\backslash\mathtt{W}} u' \ \wedge \ \big(\mathrm{ctx}^M_{\circledast}[u]_{|\Sigma^*\times\mathtt{R}} \subseteq \mathrm{ctx}^M_{\circledast}[u']_{|\Sigma^*\times\mathtt{R}}\big),\\
u \preccurlyeq^M_{\mathtt{U_c}\backslash\mathtt{R}} u' &\overset{\triangle}{\iff} u, u' \in \mathtt{U_c}\backslash\mathtt{R} \ .
\end{aligned}$$

Proposition 5. *Let $\mathcal{B}$ be a VPA. The pair $\leqslant^{L^\omega(\mathcal{B})}, \preccurlyeq^{L^\omega(\mathcal{B})}$ is $L^\omega(\mathcal{B})$-suitable.*

Proof (sketch). First we show that the pair $\leqslant^M, \preccurlyeq^M$ is M-preserving, where $M \triangleq L^\omega(\mathcal{B})$. Let $(u,v),(u',v') \in \mathtt{C}\times\mathtt{C}$ (resp. $\mathtt{U_c}\times\mathtt{R}$) such that $u \leqslant^M u'$, $v \preccurlyeq^M v'$ and $uv^\omega \in M$. From $u \leqslant^M u'$ and $uv^\omega \in M$ we deduce that $(\epsilon, v^\omega) \in \mathrm{ctx}^M_{|\mathtt{C}\times\Sigma^\omega}[u] \subseteq \mathrm{ctx}^M_{|\mathtt{C}\times\Sigma^\omega}[u']$ (resp. $(\epsilon,v^\omega) \in \mathrm{ctx}^M_{|\mathtt{C}\times\mathtt{R}^\omega}[u] \subseteq \mathrm{ctx}^M_{|\mathtt{C}\times\mathtt{R}^\omega}[u']$). Thus, $u'v^\omega \in M$. From $v \preccurlyeq^M v'$ and $u'v^\omega \in M$ we deduce that $(u',\epsilon) \in \mathrm{ctx}^M_{\circledast}[v]_{|\mathtt{C}\times\mathtt{C}} \subseteq \mathrm{ctx}^M_{\circledast_\mathtt{C}}[v']_{|\mathtt{C}\times\mathtt{C}}$ (resp. $(u',\epsilon) \in \mathrm{ctx}^M_{\circledast}[v]_{|\Sigma^*\times\mathtt{R}} \subseteq \mathrm{ctx}^M_{\circledast_\mathtt{C}}[v']_{|\Sigma^*\times\mathtt{C}}$). Thus, $u'v'^\omega \in M$.

We now show that the qo $\leqslant^M$ satisfies the monotonicity conditions **C** and **R**. Let $u \leqslant^M u'$ such that $u,u' \in \mathtt{C}$ (resp. $u,u' \in \mathtt{R}$). Let $s \in \mathtt{C}$ and $t \in \Sigma^*$ (resp. $s \in \Sigma^*$ and $t \in \mathtt{R}$). If $u,u' \in \mathtt{W}$ then it is easy to check that $sut \leqslant^M su't$. Otherwise $u,u' \in \mathtt{C}\backslash\mathtt{W}$ (resp. $u,u' \in \mathtt{R}\backslash\mathtt{W}$) and we distinguish two cases: if $t \in \mathtt{C}$ (resp. $s \in \mathtt{R}$) then $sut, su't \in \mathtt{C}\backslash\mathtt{W}$ (resp. $sut, su't \in \mathtt{R}\backslash\mathtt{W}$). We show that $sut \leqslant^M_{\mathtt{C}\backslash\mathtt{W}} su't$ (resp. $sut \leqslant^M_{\mathtt{R}\backslash\mathtt{W}} su't$). Let $(s',\xi) \in \mathrm{ctx}^M[sut]_{|\mathtt{C}\times\Sigma^\omega}$ (resp. $(s',\xi) \in \mathrm{ctx}^M[sut]_{|\Sigma^*\times\mathtt{R}^\omega}$). Since $s's \in \mathtt{C}$ (resp. $t\xi \in \mathtt{R}^\omega$), we deduce from $u \leqslant^M_{\mathtt{C}\backslash\mathtt{W}} u'$ (resp. $u \leqslant^M_{\mathtt{R}\backslash\mathtt{W}} u'$) that $(s',\xi) \in \mathrm{ctx}^M[su't]_{|\mathtt{C}\times\Sigma^\omega}$ (resp. $(s',\xi) \in \mathrm{ctx}^M[su't]_{|\Sigma^*\times\mathtt{R}^\omega}$). If $t \in \mathtt{U_c}$ (resp. $s \in \Sigma^*\backslash\mathtt{R}$) then $sut, su't \in \mathtt{U_c}\backslash\mathtt{R}$ and similarly we can show that $sut \leqslant^M_{\mathtt{U_c}\backslash\mathtt{R}} su't$. The proof that $\leqslant^M$ and $\preccurlyeq^M$ are wqos follows from [9, Prop 1.2] by observing that for every $\mathtt{J}$ in the partition $\mathcal{P}$ of Σ^* we have $\leqslant^{\mathcal{B}}_{|\mathtt{J}\times\mathtt{J}} \subseteq \leqslant^M_{\mathtt{J}}$ and $\preccurlyeq^{\mathcal{B}}_{|\mathtt{J}\times\mathtt{J}} \subseteq \preccurlyeq^M_{\mathtt{J}}$, where $\leqslant^{\mathcal{B}}$ and $\preccurlyeq^{\mathcal{B}}$ are the state-based qos previously defined. □

Deciding the syntactic qos can be easily shown to be as hard as the inclusion problem between ω-VPL generated by VPA. Nevertheless, the syntactic qos act as a gold standard for quasiorders in the sense formalized in the next proposition.

Proposition 6. *Let $M \subseteq \Sigma^\omega$ be an ω-VPL and $\leqslant, \preccurlyeq$ be a M-suitable pair of qos such that $\preccurlyeq \subseteq \leqslant$. For every $\mathtt{J} \in \mathcal{P}$ we have $\leqslant_{|\mathtt{J}\times\mathtt{J}} \subseteq \leqslant^M$ and $\preccurlyeq_{|\mathtt{J}\times\mathtt{J}} \subseteq \preccurlyeq^M$.*

By Propositions 5 and 6 the pair $\leqslant^{L^\omega(\mathcal{B})}, \preccurlyeq^{L^\omega(\mathcal{B})}$ is the greatest (w.r.t $\subseteq\times\subseteq$) among the $L^\omega(\mathcal{B})$-suitable pairs $\leqslant, \preccurlyeq$ of qos that respect the partition $\mathcal{P}$ and that verify $\preccurlyeq \subseteq \leqslant$.

7 Algorithm

We are now in position to present our algorithm which, given two VPA $\mathcal{A} = (Q, q_I, \Gamma, \delta, F)$ and $\mathcal{B} = (\hat{Q}, \hat{q_I}, \hat{\Gamma}, \hat{\delta}, \hat{F})$ and a pair of $L^\omega(\mathcal{B})$-suitable qos, decides the inclusion problem $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$.

Algorithm 1 computes a finite basis for S w.r.t. $\leqslant \times \preccurlyeq$ (lines 1–2) and afterwards checks membership in $L^\omega(\mathcal{B})$ on every ultimately periodic word uv^ω stemming from this finite basis (lines 3–7).

Theorem 3. *Given the required inputs, Algorithm 1 decides the inclusion problem $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$.*

Proof. As established by Proposition 3, given a monotonic pair $\leqslant, \preccurlyeq$ of decidable wqos, Algorithm 1 computes in line 1 (resp. line 2) a finite basis ${f_\mathcal{A}}^m(\vec{\emptyset})$ (resp.

Algorithm 1: Algorithm for deciding $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$

Data: VPA $\mathcal{A} = (Q, q_I, \Gamma, \delta, F)$ and $\mathcal{B} = (\hat{Q}, \hat{q_I}, \hat{\Gamma}, \hat{\delta}, \hat{F})$.
Data: $L^\omega(\mathcal{B})$-suitable pair $\leqslant, \preccurlyeq$.
Data: Procedure deciding $uv^\omega \in L^\omega(\mathcal{B})$ given (u, v).

1 Compute ${f_\mathcal{A}}^m(\vec{\emptyset})$ with least m s.t. ${f_\mathcal{A}}^{m+1}(\vec{\emptyset}) \sqsubseteq^{4\cdot|Q|^2}_{\leqslant} {f_\mathcal{A}}^m(\vec{\emptyset})$;
2 Compute ${r_\mathcal{A}}^{m'}(\vec{\emptyset})$ with least m' s.t. ${r_\mathcal{A}}^{m'+1}(\vec{\emptyset}) \sqsubseteq^{6\cdot|Q|^2}_{\preccurlyeq} {r_\mathcal{A}}^{m'}(\vec{\emptyset})$;
3 **foreach** $p \in Q$ **do**
4 **foreach** $u \in ({f_\mathcal{A}}^m(\vec{\emptyset}))_{2,q_I,p}$, $v \in ({r_\mathcal{A}}^{m'}(\vec{\emptyset}))_{5,p,p}$ **do**
5 **if** $uv^\omega \notin L^\omega(\mathcal{B})$ **then return** false;
6 **foreach** $u \in ({f_\mathcal{A}}^m(\vec{\emptyset}))_{4,q_I,p}$, $v \in ({r_\mathcal{A}}^{m'}(\vec{\emptyset}))_{6,p,p}$ **do**
7 **if** $uv^\omega \notin L^\omega(\mathcal{B})$ **then return** false;
8 **return** true;

${r_\mathcal{A}}^{m'}(\vec{\emptyset})$) for $\operatorname{lfp} f_\mathcal{A}$ (resp. $\operatorname{lfp} r_\mathcal{A}$) w.r.t. $\leqslant$ (resp. $\preccurlyeq$). Next define:

$$S_\mathcal{A}^{m,m'} \triangleq \bigcup_{p\in Q}\Big(\big(({f_\mathcal{A}}^m(\vec{\emptyset}))_{2,q_I,p} \times ({r_\mathcal{A}}^{m'}(\vec{\emptyset}))_{5,p,p}\big) \cup \big(({f_\mathcal{A}}^m(\vec{\emptyset}))_{4,q_I,p} \times ({r_\mathcal{A}}^{m'}(\vec{\emptyset}))_{6,p,p}\big)\Big) \ .$$

Using Equation (2) we deduce that $S_\mathcal{A}^{m,m'}$ is a finite basis for S w.r.t. $\leqslant \times \preccurlyeq$. Since the pair $\leqslant, \preccurlyeq$ is $L^\omega(\mathcal{B})$-preserving, by Section 3, we deduce that

$$L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B}) \iff \forall (u, v) \in S_\mathcal{A}^{m,m'},\ uv^\omega \in L^\omega(\mathcal{B}) \ .$$

□

We remark that Algorithm 1 can be easily adapted to decide the inclusion problem between visibly pushdown languages of finite words. The adaptation to the finite words case omits the fixpoint computation of line 2 and iterates over the components (i, q_I, p) where $i \in \{2, 3, 4\}$ and where $p \in F$ is a final state.

Example 4. Consider the iterates of the function $f_\mathcal{A}$ from Example 2. One can check that ${f_\mathcal{A}}^4(\vec{\emptyset}) \sqsubseteq^4_{\preccurlyeq^\mathcal{B}} {f_\mathcal{A}}^3(\vec{\emptyset})$ (thus also ${f_\mathcal{A}}^4(\vec{\emptyset}) \sqsubseteq^4_{\leqslant^\mathcal{B}} {f_\mathcal{A}}^3(\vec{\emptyset})$ since $\preccurlyeq^\mathcal{B} \subseteq \leqslant^\mathcal{B}$). Thus, we check whether the inclusion $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$ holds on the finite set $(\{\epsilon, cr\} \times \{cr\}) \cup (\{c, c^2, c^3\} \times \{cr, c, c^2, c^3, c^4\})$ and find the counterexample $c(cr)^\omega \in L^\omega(\mathcal{A}) \backslash L^\omega(\mathcal{B})$.

Antichains Everywhere. We show next that Algorithm 1 remains correct if, in the sequence of Kleene iterates of $f_\mathcal{A}$ or $r_\mathcal{A}$, for each application of $f_\mathcal{A}$ or $r_\mathcal{A}$ we first select a finite basis for their arguments instead (using $\leqslant^{4\cdot|Q|^2}$ for $f_\mathcal{A}$ and $\preccurlyeq^{6\cdot|Q|^2}$ for $r_\mathcal{A}$).

Proposition 7. *Let $\ltimes$ be a qo that verifies the monotonicity conditions $\mathbf{W}, \mathbf{C}, \mathbf{R}, \mathbf{U}$. If B is a basis for $(X, Y, Z, T) \in \wp(\mathtt{W})^{|Q|^2} \times \wp(\mathtt{C})^{|Q|^2} \times \wp(\mathtt{R})^{|Q|^2} \times \wp(\mathtt{U_c})^{|Q|^2}$ w.r.t. $\ltimes^{4\cdot|Q|^2}$, then $f_\mathcal{A}(B)$ is a basis for $f_\mathcal{A}(X, Y, Z, T)$ w.r.t. $\ltimes^{4\cdot|Q|^2}$. The analogue result holds for $r_\mathcal{A}$ when $\ltimes$ satisfies the monotonicity conditions $\mathbf{W}, \mathbf{C}_\circledast, \mathbf{R}_\circledast$.*

Since every Kleene iterate of $f_{\mathcal{A}}$ belongs to $\wp(\mathtt{W})^{|Q|^2} \times \wp(\mathtt{C})^{|Q|^2} \times \wp(\mathtt{R})^{|Q|^2} \times \wp(\mathtt{U_c})^{|Q|^2}$ given a basis B for ${f_{\mathcal{A}}}^n(\vec{\emptyset})$ w.r.t. $\leqslant^{4\cdot|Q|^2}$, by Proposition 7, $f_{\mathcal{A}}(B)$ is a basis for ${f_{\mathcal{A}}}^{n+1}(\vec{\emptyset})$ w.r.t. $\leqslant^{4\cdot|Q|^2}$. Hence, at each iteration we can select, for each (i,p,q)-component, a basis w.r.t. $\leqslant$ and then apply $f_{\mathcal{A}}$. In particular, we can keep antichains for each (i,p,q)-component, that is, finite bases of incomparable words. The analogue result holds for the Kleene iterates of $r_{\mathcal{A}}$.

7.1 State-based Algorithm

Next we consider Algorithm 1 instantiated with the pair of state-based qos (§ 6).

Data Structures. Comparing two words given a state-based qo requires to compute the corresponding sets of contexts in $\mathcal{B}$. Instead of computing contexts every time we need to compare two words we cache the context information along with each word for faster retrieval. More precisely, we cache $\mathrm{ctx}^{\mathcal{B}}[u]$ along with u when u is a prefix and we cache $(\mathrm{ctx}^{\mathcal{B}}[v], \mathrm{ctx}^{\mathcal{B}}_{\circledast}[v])$ along with v when v is a period. Next we go even further and explain that new context information can be computed inductively from already computed context information. Assume we are computing a new word during the fixpoint computation, for instance the word *cur* that is obtained by flanking c and r to u. We will show that the context information of *cur* can be computed directly from that of u, c and r instead of computing *cur* from "scratch".

Fixpoint Computation. Given an input vector the functions $f_{\mathcal{A}}$ and $r_{\mathcal{A}}$ add new words of type uu', and *cur* to its components, where c and r are fixed letters, and u, u' are words already present in some components of the vector. The following equalities show that we can inductively compute the contexts and final contexts in $\mathcal{B}$ of newly added words in these functions: for every $u, u' \in \mathtt{C} \cup \mathtt{R}$, $c \in \Sigma_c$, $r \in \Sigma_r$, we have

$$\mathrm{ctx}^{\mathcal{B}}[uu'] = \{(p,q) \in \hat{Q}^2 \mid \exists p_i \in \hat{Q}, (p,p_i) \in \mathrm{ctx}^{\mathcal{B}}[u], (p_i,q) \in \mathrm{ctx}^{\mathcal{B}}[u']\},$$
$$\mathrm{ctx}^{\mathcal{B}}[cur] = \{(p,q) \in \hat{Q}^2 \mid \exists (p',q') \in \mathrm{ctx}^{\mathcal{B}}[u], \exists \gamma \in \hat{\Gamma}, (p,c,p',\gamma) \in \hat{\delta}_c, (q',r,\gamma,q) \in \hat{\delta}_r\} \ .$$

The definitions for $\mathrm{ctx}^{\mathcal{B}}_{\circledast}[uu']$ and $\mathrm{ctx}^{\mathcal{B}}_{\circledast}[cur]$ are left as exercise to the reader.

Example 5. Using the above definition it is routine to check that $\mathrm{ctx}^{\mathcal{B}}[c\,r] = \{(p,q),(q,q)\}$ because $cr = c\epsilon r$, $\mathrm{ctx}^{\mathcal{B}}[\epsilon] = \{(p,p),(q,q)\}$ (Example 3) and $(p,c,q,A),(q,c,q,A) \in \hat{\delta}_c$, $(q,r,A,q) \in \hat{\delta}_r$.

Using the context information cached along words we check convergence of the fixpoint computations (lines 1–2) using the following qos directly on contexts $\sqsubseteq_{\subseteq}$ on $\wp(\wp(\hat{Q}^2))^4$ for prefixes and $\sqsubseteq_{\subseteq\times\subseteq}$ on $\wp(\wp(\hat{Q}^2) \times \wp(\hat{Q}^2))^6$ for periods.

Incidentally, as we show below, we can perform the membership checks of lines 5 and 7 (asking whether $uv^\omega \in L^\omega(\mathcal{B})$ given u and v) using the context information associated to the prefix u and period v and nothing else.

Membership Check. To decide membership in $L^\omega(\mathcal{B})$ we use the membership predicate $\mathrm{Inc}^{\mathcal{B}}$ defined for $x, y_1, y_2 \in \wp(\hat{Q}^2)$ as follows:

$$\mathrm{Inc}^{\mathcal{B}}(x, y_1, y_2) \triangleq \exists q, p \in \hat{Q}, (\hat{q_I}, q) \in x \wedge (q, p) \in y_1^* \wedge (p, p) \in y_1^* \circ y_2 \circ y_1^* \ ,$$

where, given two binary relations $y, y' \in \wp(\hat{Q}^2)$ on states of $\mathcal{B}$, the notation $y \circ y'$ denotes their composition, and y^* denotes the Kleene closure of y.

Proposition 8. *For all* $(u, v) \in \mathtt{Ld}$, $Inc^{\mathcal{B}}(ctx^{\mathcal{B}}[u], ctx^{\mathcal{B}}[v], ctx^{\mathcal{B}}_{\circledast}[v]) \iff uv^\omega \in L^\omega(\mathcal{B})$.

Proof. Let $(u, v) \in \mathtt{Ld}$. Note that if $v \in \mathtt{C}$ (resp. $v \in \mathtt{R}$) then for every positive integer n we have $v^n \in \mathtt{C}$ (resp. $v^n \in \mathtt{R}$) and $(p, q) \in \mathrm{ctx}^{\mathcal{B}}[v]^* \iff \exists n, (p, q) \in \mathrm{ctx}^{\mathcal{B}}[v^n]$. Therefore, if $\mathrm{Inc}^{\mathcal{B}}(\mathrm{ctx}^{\mathcal{B}}[u], \mathrm{ctx}^{\mathcal{B}}[v], \mathrm{ctx}^{\mathcal{B}}_{\circledast}[v])$ holds then there are $q, p \in \hat{Q}$ and two positive integers n, m such that $(\hat{q_I}, q) \in \mathrm{ctx}^{\mathcal{B}}[u]$, $(q, p) \in \mathrm{ctx}^{\mathcal{B}}[v^n]$ and $(p, p) \in \mathrm{ctx}^{\mathcal{B}}_{\circledast}[v^m]$. If $(u, v) \in \mathtt{C} \times \mathtt{C}$ then we deduce an accepting trace of $\mathcal{B}$ on uv^ω of the form $(\hat{q_I}, \bot) \vdash^{*u} (q, \bot) \vdash^{*v^n} (p, \bot) \vdash^{\circledast v^m} (p, \bot)$ for uv^ω. If $(u, v) \in \mathtt{U_c} \times \mathtt{R}$ then we deduce an accepting trace of $\mathcal{B}$ on uv^ω of the form $(\hat{q_I}, \bot) \vdash^{*u} (q, w) \vdash^{*v^n} (p, ww') \vdash^{\circledast v^m} (p, ww'w'')$ for some $w, w', w'' \in \Gamma$.

Conversely if $uv^\omega \in L^\omega(\mathcal{B})$ then there is an accepting trace of $\mathcal{B}$ on uv^ω.

- If $(u, v) \in \mathtt{C} \times \mathtt{C}$ then this trace is of the form

$$(\hat{q_I}, \bot) \vdash^{*u} (q, \bot) \vdash^{*v} (q_1, \bot) \vdash^{*v} (q_2, \bot) \vdash^{*v} \cdots$$

  Since $\hat{Q}$ is finite, there is $p \in \hat{Q}$ and a sequence $\{n_k\}_{k \in \mathbb{N}}$ such that $q_{n_k} = p$ for all $k \in \mathbb{N}$. Since the trace is accepting there is $m \in \mathbb{N}$ such that $(p, \bot) \vdash^{\circledast v^m} (p, \bot)$.
- If $(u, v) \in \mathtt{U_c} \times \mathtt{R}$ then it is of the form

$$(\hat{q_I}, \bot) \vdash^{*u} (q, w_0) \vdash^{*v} (q_1, w_1) \vdash^{*v} (q_2, w_1 w_2) \vdash^{*v} \cdots$$

  where for each $j \in \mathbb{N}$ no symbol of w_j is popped while reading v in the sequence of transitions $(q_j, w_j) \vdash^{*v} (q_{j+1}, w_j w_{j+1})$. Thus, we can derive sequences $(q_j, \bot) \vdash^{*v} (q_{j+1}, w_{j+1})$ for every $j \in \mathbb{N}$. There is $p \in \hat{Q}$ and a sequence $\{n_k\}_{k \in \mathbb{N}}$ such that $q_{n_k} = p$ for all $k \in \mathbb{N}$ and since the trace is accepting there is $m \in \mathbb{N}$ such that $(p, \bot) \vdash^{\circledast v^m} (p, w_{n_j} \cdots w_{n_{j+m}})$.

In both cases we deduce that $(\hat{q_I}, q) \in \mathrm{ctx}^{\mathcal{B}}[u]$, $(q, p) \in \mathrm{ctx}^{\mathcal{B}}[v^{n_0}]$ and $(p, p) \in \mathrm{ctx}^{\mathcal{B}}_{\circledast}[v^m]$. Thus, $\mathrm{Inc}^{\mathcal{B}}(\mathrm{ctx}^{\mathcal{B}}[u], \mathrm{ctx}^{\mathcal{B}}[v], \mathrm{ctx}^{\mathcal{B}}_{\circledast}[v])$ holds. □

By showing how to reason on contexts directly (for comparisons, for applying functions $f_{\mathcal{A}}$ and $r_{\mathcal{A}}$, for convergence check and for membership check) we removed the need to store words altogether since their contexts suffice. To sum up, Algorithm 1 instantiated with the state-based qos can be implemented by manipulating directly subsets of $\wp(\hat{Q}^2)$ (for the prefixes) and pairs of subsets of $\wp(\hat{Q}^2)$ (for the periods) thereby removing the need to store and manipulate words. We call this implementation of Algorithm 1 the *state-based algorithm*. We conclude this section with its complexity.

Proposition 9. *Let* $n \triangleq |Q|$, $\hat{n} \triangleq |\hat{Q}|$ *and* $m \triangleq max\{1, |\Sigma|\}$. *The running time of the state-based algorithm is* $2^{O(\hat{n}^2)} m^2 n^4$.

8 Experiments

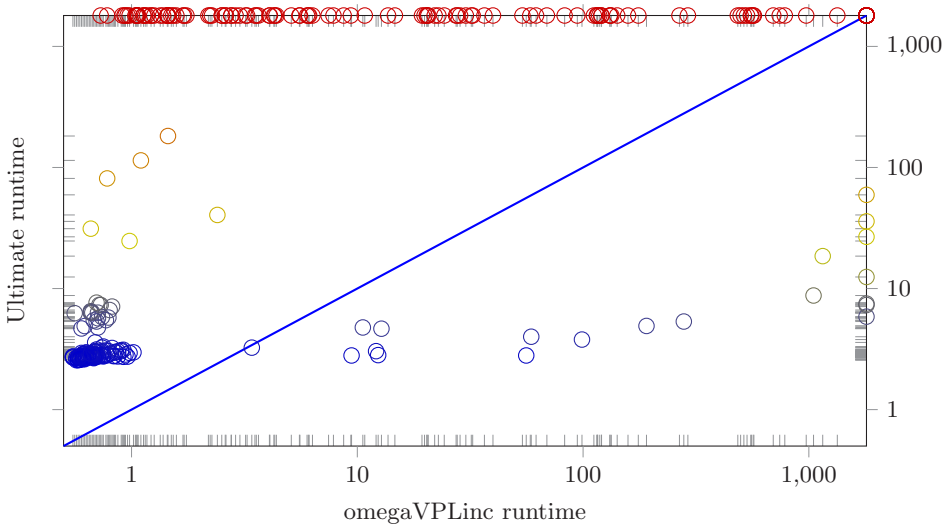


Fig. 2. Scatter plot comparing the runtime (in seconds) of Ultimate and omegaVPLinc on the Ultimate suite. Both axis feature a logarithmic scale. When a tool does not return an answer within 1800 seconds (it runs out of time or memory) the data point is plotted on the edge thereof (top edge for Ultimate, right edge for omegaVPLinc).

We implemented *omegaVPLinc* [11] , a Java prototype of the state-based algorithm and evaluated it against Ultimate from Heizmann et al. [21] which decides inclusion via complementation, intersection and emptiness check.[7]

Benchmarks. Our experiments use two sets of benchmarks. The first stems from [18] and consists of 5 queries $L^\omega(\mathcal{A}) \subseteq L^\omega(\mathcal{B})$ given $\mathcal{A}$ and $\mathcal{B}$. We first translated those VPA into the AutomataScript language that Ultimate and omegaVPLinc can use and then we minimized them with Ultimate. The second set of benchmarks consists of 281 instances of VPA $\mathcal{A}, \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ for which we run the query $L^\omega(\mathcal{A}) \subseteq \bigcup_{i=1}^{n} L^\omega(\mathcal{B}_i)$. These VPA were computed by Ultimate from randomly selected tasks in SV-COMP (Software Verification Competition) termination category. We used Ultimate to compute the unions of $\mathcal{B}_1, \ldots, \mathcal{B}_n$ and then minimize the result before running each query.

[7] We excluded FADecider [18] from our evaluation because it returned 22 false positive answers on a randomly chosen subset of 50 from our 286 benchmarks. Counterexamples to inclusion for these benchmarks were validated with Ultimate. The problem has been reported.

Experimental Setup. We ran our experiment in Debian/GNU Linux 11 (Bullseye) 64bit, running on a server with 20 GB of RAM and 2 Xeon E5640 2.6 GHz CPUs. We used Ultimate version 0.2.1, with openJDK 11.0.13, whereas omegaVPLinc uses openJDK 17.0.1. Maximal heap size for both programs was set to 6 GB and they were given a timeout of 30 minutes (or, equivalently, 1800 seconds).

Results. Of the 5 benchmarks in the FADecider suite, omegaVPLinc is faster on 4 of them. Our prototype times out on the remaining one, while Ultimate runs out of memory. Of the 281 benchmarks in the Ultimate suite, omegaVPLinc correctly returns an answer on 253 (165 $\subseteq$ and 88 $\not\subseteq$), times out on 27 and runs out of memory on 1. Ultimate, however, only terminates on 142 benchmarks, running out of memory on the remaining 139 (the red data points on the top edge in Fig. 2). There are 7 benchmarks for which Ultimate terminates, but omegaVPLinc doesn't (the data points on the right edge but not the top one), whereas there are 118 benchmarks for which omegaVPLinc terminates, but Ultimate doesn't (the red data points on the top edge but not the right one). Of the 135 benchmarks on which both tools terminate, omegaVPLinc is faster than Ultimate on 123 (data points touching no edges and above the diagonal). Moreover omegaVPLinc and Ultimate coincide on whether inclusion holds (98) or not (37). This empirical evaluation suggests that omegaVPLinc scales up better than Ultimate on both of these benchmark sets.

9 Conclusion and Future Work

We presented novel algorithms to solve the inclusion problem between visibly pushdown languages of infinite words that leverage antichain-like techniques as well as the use of separate quasiorders for prefixes and periods of ultimately periodic words. Our empirical evaluation suggests that our approach scales up better than the ones relying on an explicit complementation. A future work is to extend our approach to the class of operator-precedence languages [15] which also enjoy an EXPTime-complete inclusion problem and which is strictly contained in the class of deterministic CFL, and strictly contains VPL [8].

References

1. Abdulla, P.A., Chen, Y.F., Clemente, L., Holík, L., Hong, C.D., Mayr, R., Vojnar, T.: Simulation Subsumption in Ramsey-Based Büchi Automata Universality and Inclusion Testing. In: CAV'10: Proc. 20th Int. Conf. on Computer Aided Verification. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_14
2. Abdulla, P.A., Chen, Y.F., Clemente, L., Holík, L., Hong, C.D., Mayr, R., Vojnar, T.: Advanced Ramsey-Based Büchi Automata Inclusion Testing. In: CONCUR'11: Proc. 22nd Int. Conf. on Concurrency Theory. Springer (2011). https://doi.org/10.1007/978-3-642-23217-6_13
3. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When Simulation Meets Antichains. In: TACAS'10: Proc. 16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_14

4. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC'04: Proc. 36th Ann. ACM Symp. on Theory of Computing. ACM (2004). https://doi.org/10.1145/1007352.1007390
5. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In: CIAA'08: Proc. Int. Conf. on Implementation and Applications of Automata. LNCS, Springer (2008). https://doi.org/10.1007/978-3-540-70844-5_7
6. Bruyère, V., Ducobu, M., Gauwin, O.: Visibly Pushdown Automata: Universality and Inclusion via Antichains. In: LATA'13: Proc. Int. Conf. on Language and Automata Theory and Applications. LNCS, Springer (2013). https://doi.org/10.1007/978-3-642-37064-9_18
7. Calbrix, H., Nivat, M., Podelski, A.: Ultimately periodic words of rational ω-languages. In: Proc. Int. Symp. on Mathematical Foundations of Programming Semantics (MFPS). LNCS, Springer (1994). https://doi.org/10.1007/3-540-58027-1_27
8. Crespi Reghizzi, S., Mandrioli, D.: Operator precedence and the visibly pushdown property. Journal of Computer and System Sciences **78**(6), 1837–1867 (2012). https://doi.org/10.1016/j.jcss.2011.12.006
9. de Luca, A., Varricchio, S.: Well quasi-orders and regular languages. Acta Informatica **31**(6), 539–557 (1994). https://doi.org/10.1007/BF01213206
10. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: CAV'06: Proc. 16th Int. Conf. on Computer Aided Verification. Springer (2006). https://doi.org/10.1007/11817963_5
11. Doveri, K., Ganty, P., Hadzi-Djokic, L.: omegavplinc v1.1 (Jan 2023). https://doi.org/10.5281/zenodo.7506895
12. Doveri, K., Ganty, P., Mazzocchi, N.: FORQ-Based Language Inclusion Formal Testing. In: CAV'22: Proc. 32nd Int. Conf. on Computer Aided Verification. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_6
13. Doveri, K., Ganty, P., Parolini, F., Ranzato, F.: Inclusion Testing of Büchi Automata Based on Well-Quasiorders. In: CONCUR'21: Proc. 32nd Int. Conf. on Concurrency Theory. LIPIcs, Schloss Dagstuhl (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.3
14. Doyen, L., Raskin, J.F.: Antichain Algorithms for Finite Automata. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer (2010)
15. Floyd, R.W.: Syntactic analysis and operator precedence. J. ACM **10**(3), 316–333 (1963). https://doi.org/10.1145/321172.321179
16. Fogarty, S., Vardi, M.Y.: Efficient Büchi Universality Checking. In: TACAS'10: Proc. 16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_17
17. Friedmann, O., Klaedtke, F., Lange, M.: Ramsey goes visibly pushdown. In: ICALP'13: Proc. 40th Int. Coll. on Automata, Languages, and Programming. LNCS, Springer (2013). https://doi.org/10.1007/978-3-642-39212-2_22
18. Friedmann, O., Klaedtke, F., Lange, M.: Ramsey-based inclusion checking for visibly pushdown automata. ACM Transactions on Computational Logic **16**(4), 1–24 (2015). https://doi.org/10.1145/2774221
19. Ganty, P., Ranzato, F., Valero, P.: Language inclusion algorithms as complete abstract interpretations. In: Static Analysis. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_8

20. Ganty, P., Ranzato, F., Valero, P.: Complete abstractions for checking language inclusion. ACM Trans. Comput. Logic **22**(4) (2021). https://doi.org/10.1145/3462673
21. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate automizer and the search for perfect interpolants - (competition contribution). In: TACAS'18: Proc. 24th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
22. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. SIGPLAN Notices **45**(1), 471–482 (2010). https://doi.org/10.1145/1707801.1706353
23. Hieronymi, P., Ma, D., Oei, R., Schaeffer, L., Schulz, C., Shallit, J.: Decidability for Sturmian Words. In: Manea, F., Simpson, A. (eds.) CSL'22: Proc. 30th EACSL Ann. Conf. on Computer Science Logic. LIPIcs, Schloss Dagstuhl (2022). https://doi.org/10.4230/LIPIcs.CSL.2022.24
24. Meyer, R., Muskalla, S., Neumann, E.: Liveness verification and synthesis: New algorithms for recursive programs. CoRR **abs/1701.02947** (2017), http://arxiv.org/abs/1701.02947
25. Oei, R., Ma, D., Schulz, C., Hieronymi, P.: Pecan: An automated theorem prover for automatic sequences using Büchi automata (2021), https://arxiv.org/abs/2102.01727
26. Picalausa, F., Servais, F., Zimányi, E.: Xevolve: An XML schema evolution framework. In: SAC'11: Proc. ACM Symp. on Applied Computing. p. 1645–1650. ACM (2011). https://doi.org/10.1145/1982185.1982530
27. Rajasekaran, A., Shallit, J., Smith, T.: Additive number theory via automata theory. Theory of Computing Systems **64**(3), 542–567 (2019). https://doi.org/10.1007/s00224-019-09929-9

Stack-Aware Hyperproperties*

Ali Bajwa[2(✉)], Minjian Zhang[1], Rohit Chadha[2(✉)], and Mahesh Viswanathan[1]

[1] University of Illinois, Urbana-Champaign, USA
{minjian2,vmahesh}@illinois.edu
[2] University of Missouri, Columbia, USA
{azb9q8,chadhar}@missouri.edu

Abstract. A hyperproperty relates executions of a program and is used to formalize security objectives such as confidentiality, non-interference, privacy, and anonymity. Formally, a hyperproperty is a collection of allowable sets of executions. A program violates a hyperproperty if the set of its executions is not in the collection specified by the hyperproperty. The logic HyperCTL* has been proposed in the literature to formally specify and verify hyperproperties. The problem of checking whether a finite-state program satisfies a HyperCTL* formula is known to be decidable. However, the problem turns out to be undecidable for procedural (recursive) programs. Surprisingly, we show that decidability can be restored if we consider restricted classes of hyperproperties, namely those that relate only those executions of a program which have the same call-stack access pattern. We call such hyperproperties, *stack-aware hyperproperties.* Our decision procedure can be used as a proof method for establishing security objectives such as noninference for recursive programs, and also for refuting security objectives such as observational determinism. Further, if the call stack size is observable to the attacker, the decision procedure provides exact verification.

Keywords: Hyperproperties · Temporal Logic · Recursive Programs · Model Checking · Pushdown Systems · Visibly Pushdown Automata.

1 Introduction

Temporal logics H perLTL and H perCTL* [5] were designed to express and reason about security guarantees that are *hyperproperties* [6]. A hyperproperty [6] is a security guarantee that does not depend solely on individual executions. Instead, a hyperproperty relates multiple executions. For example, non-interference, a confidentiality property, states that any *two* executions of a program that differ only in high-level security inputs must have the same *low*-security observations. As pointed out in [6], several security guarantees are hyperproperties. The logic H perCTL* subsumes H perLTL, and the problem of checking a finite-state system against a H perCTL* formula is decidable [5].

* Ali Bajwa was partially supported by NSF CNS 1553548. Rohit Chadha was partially supported by NSF CNS 1553548 and NSF SHF 1900924. Mahesh Viswanathan and Minjian Zhang were partially supported by NSF SHF 1901069 and NSF SHF 2007428.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 308–325, 2023.
https://doi.org/10.1007/978-3-031-30823-9_16

In this paper, we consider the problem of model checking procedural (recursive) programs against security hyperproperties. Recall recursive programs are naturally modeled as a pushdown system. Unlike the case of finite-state transition systems, the problem of checking whether a pushdown system satisfies a H PERCTL* formula is undecidable [16]. In contrast, CTL* model checking is decidable for pushdown systems [3,18].

Our contributions. We consider restricted classes of hyperproperties for recursive programs, namely those that relate only those executions that have the same *call-stack access pattern*. Intuitively, two executions have the same stack access pattern if they access the call stack in the same manner at each step, i.e., if in one execution there is a push (pop) at a point, then there is a push (pop) at the same point in the other execution. Observe that if two executions have the same stack access pattern, then their stack sizes are the same at all times. We call such hyperproperties, *stack-aware hyperproperties.*

In order to specify stack-aware hyperproperties, we extend H PERCTL* to SHCTL*. The logic SHCTL* has a two level syntax. At the first level, the syntax is identical to H PERCTL* formulas, and is interpreted over executions of the pushdown system with the same stack access pattern. At the top-level, we quantify over different stack access patterns. The formula $E\psi$ is true if for some stack access pattern ρ of the system, the pushdown system restricted to executions with stack access pattern ρ satisfies the H PERCTL* formula ψ. The formula $A\psi$ is true if for each stack access pattern ρ of the system, the pushdown system restricted to executions with stack access pattern ρ satisfies the H PERCTL* formula ψ. See Figure 1 on Page 8 for a side-by-side comparison of the syntax for H PERCTL* and SHCTL*. H PERLTL is extended to SHLTL similarly. Please note that SHCTL* subsumes SHLTL, and that SHCTL* (SHLTL) coincides with H PERCTL* (H PERLTL) for finite state systems as all executions of finite state systems have the same stack access pattern.

We show that the model checking problem for SHCTL* is decidable. We demonstrate three different ways this result can aid in verifying recursive programs. First, for security guarantees such as noninference [14], which are expressible in the $\forall\exists^*$ fragment of H PERLTL, we can use the model checking algorithm to establish that a recursive program satisfies the H PERLTL property. Secondly, for the $\forall^*$ fragment of H PERLTL, the model checking algorithm can be used to detect security flaws by establishing that a recursive program does not satisfy security guarantees. Observational determinism [13,19] is an example of such a property. Finally, when the attacker can observe stack access patterns (or, equivalently, stack sizes), we can get exact verification for several properties. The assumption of the attacker observing stack access patterns holds, for example, in the program counter security model [15] in which the attacker has access to program counters at each step. As argued in [15], the program security model is appropriate to capture control-flow side channels such as those arising from timing behavior and/or disclosure of errors.

The decision procedure uses an automata-theoretic approach inspired by the model checking algorithm for finite state systems and H PERCTL* given

in [10]. Since stack-aware hyperproperties relate only executions with the same stack access-pattern, a set of executions with the same stack access pattern can be encoded as a word over a *pushdown* alphabet, [3] and the problem of model checking a sHCTL* formula can be reduced to the problem of checking emptiness of a *non-deterministic visibly pushdown automaton (NVPA)* over infinite words [1]. The reduction of the model checking problem to the emptiness problem is based on a compositional construction of an automaton for each sub-formula which accepts exactly the set of assignments to path variables that satisfy the sub-formula. For this construction to be optimal, we carefully leverage two equi-expressive classes of automata on infinite words, namely NVPAs and *1-way alternating jump automata (1-AJA)* [4]. The model checking algorithm for sHCTL* against procedural programs has a complexity that is very close to the complexity of model checking finite state systems against H perCTL*. If $g(k, n)$ denotes a tower of exponentials of height k, where the top most exponent is $\mathsf{poly}(n)$, then for a formula with formula complexity r, [4] and a system and formula whose size is bounded by n, our algorithm is in $\mathsf{DTIME}(g(\lceil \frac{r}{2} \rceil, n))$. In comparison, model checking finite state systems against H perCTL* is in $\mathsf{NSPACE}(g(\lceil \frac{r}{2} \rceil - 1, n))$. This slight difference in complexity is consistent with checking other properties like invariants for finite state systems (NL) versus procedural programs (P).

We also prove that our model checking algorithm is optimal by proving a matching lower bound. Our proof showing $\mathsf{DTIME}(g(\lceil \frac{r}{2} \rceil, n)$-hardness of the model checking problem for formulas with (formula) complexity r, relies on reducing the membership problem for $g(\lceil \frac{r}{2} \rceil - 1, n)$ space bounded *alternating Turing machines* (ATM) to the model checking problem. The reduction requires identifying an encoding of computations of ATMs, which are trees, as strings that can be guessed and generated by pushdown systems. The pushdown system we construct for the model checking problem guesses potential computations of the ATM, while the sHCTL* formula we construct checks if the guessed computation is a valid accepting computation.

Related work. Clarkson and Schneider introduced *hyperproperties* [6] and demonstrated their need to capture complex security properties. Temporal logics H perLTL and H perCTL*, that describe hyperproperties, were introduced by Clarkson et al. [5]. They also characterized the complexity of model checking finite state transition systems against H perCTL* specifications by a reduction to the satisfiability problem of QPTL [17]. Subsequently, other model checking algorithms for verifying finite state systems against H perCTL* properties have been proposed [10,7]. Tools that check satisfiability [8] and runtime verification [9] for H perLTL formulas have also been developed. Finkbeiner et al. introduced the automata-theoretic approach to model checking H perCTL* for finite-state systems [10].

[3] A pushdown alphabet is an alphabet that is partitioned into three sets: a set of call symbols, a set of internal symbols, and a set of return symbols. See Section 4.1.

[4] Our definition of formula complexity is roughly double the usual notion of quantifier alternation. For a precise definition, see Definition 4.

The model checking problem for H perLTL, and consequently H perCTL*, was shown to be undecidable for pushdown systems in [16]. For restricted fragments of H perLTL, Pommellet and Tayssir [16] introduced over-approximations and under-approximations to establish/refute that a pushdown system satisfies a H perLTL formula in those fragments. Gutsfeld et al. introduced stuttering H_μ, a *linear* time logic for checking asynchronous hyperproperties for recursive programs in [12]. The authors present complexity results for the model checking problem under an assumption of *fairness*, and a restriction of *well-alignment*. While the restriction to paths with the same *stack access pattern* is similar to the well-alignment restriction, we do not assume any fairness condition to establish decidability. However, as sHCTL* is a branching time logic and only considers synchronous hyperproperties, the two logics are not directly comparable. It is also worth mentioning that the branching nature of sHCTL* requires us to "copy" a potentially unbounded stack, from the most recently quantified path variable, when assigning a path to the "current" quantified path variable. In contrast, all path assignments in [12] start with an empty stack.

For lack of space reasons, some proofs are omitted and can be located in [2].

2 Motivation

Clarkson and Schneider [6] argue that many important *security* guarantees are expressible only as *hyperproperties*. We discuss two examples of security hyperproperties, and the logics H perLTL and H perCTL* used to specify them.

Hyperproperties and temporal logics. We discuss two variants of noninterference [11] that model confidentiality requirements. In non-interference, the inputs of a system are partitioned into *low*-level input security variables and *high*-level input security variables. The attacker is assumed to know the values of low-level security inputs. During an execution, the attacker can observe parts of the system configuration such as system outputs, or the memory usage. A system satisfies *non-interference* if the attacker cannot deduce the values of high-level inputs from the low-level observations. To formally specify the variants, we use the logic H perLTL [5], a fragment of the logic H perCTL* [5]. The precise syntax of H perLTL and H perCTL* is shown in Fig. 1. In the syntax, π is a path variable and the formula a_π is true if the proposition a is true along the path "π". Intuitively, the formula $\exists \pi.\, \psi$ is existential quantification over paths, and is true if there is a path that can be assigned to π such that ψ is true. Universal quantification ($\forall \pi.\, \psi$), and other logical connectives such as conjunction ($\wedge$), implication ($\rightarrow$), equivalence ($\leftrightarrow$) and the temporal operators G and F can be defined in the standard way. By having explicit path variables, H perLTL and H perCTL* allow quantification over multiple paths simultaneously.

Example 1. The first variant, noninference [14], states that for each execution σ of a program, there is another execution σ' such that (a) σ' is obtained from σ by replacing the high-level security inputs by a dummy input, and (b) σ and σ' have the same low-level observations. Noninference is a hyperliveness property [5,6].

Let us assume that the low-level observations of a configuration are determined by the values of the propositions in $L = \{\ell_1, \cdots \ell_m\}$. As shown in [5], noninference is expressible by the H PERLTL formula: $\mathsf{NI} \stackrel{\text{def}}{=} \forall\pi.\, \exists\pi'.(\mathsf{G}\,\lambda_{\pi'}) \wedge \pi \equiv_L \pi'$. Here $\mathsf{G}\,\lambda_{\pi'}$ expresses that G*lobally* (or in each configuration of the execution) the high input of π' is the dummy input λ, and $\pi \equiv_L \pi' \stackrel{\text{def}}{=} \mathsf{G}(\wedge_{\ell \in L}(\ell_\pi \leftrightarrow \ell_{\pi'}))$ expresses that π and π' have the same low-level observations.

Example 2. The second variant, observational determinism [13,19], states that any two executions that have the same low-level initial inputs, must have the same low-level output observations. Observational determinism is a hypersafety property [5,6], and is also expressible in H PERLTL using the formula [5]: $\mathsf{OD} \stackrel{\text{def}}{=} \forall\pi.\, \forall\pi'.(\pi[0] \equiv_{L,in} \pi'[0]) \rightarrow \pi \equiv_{L,ou} \pi'$. Here $\equiv_{L,in}$ and $\equiv_{L,ou}$ express the fact that π and π' have the same low-security inputs and outputs respectively.

Procedural (recursive) programs and Stack-aware hyperproperties. Pushdown systems model procedural programs that do not dynamically allocate memory, and whose program variables take values in finite domains. Unlike finite-state transition systems, the problem of checking whether a pushdown system satisfies a H PERCTL* formula is undecidable [16]. However, we identify a natural class of hyperproperties for which the model checking problem becomes decidable. As we shall shortly see, this class of hyperproperties not only enjoys decidability, but is also useful in reasoning about security hyperproperies such as noninference and observational determinism.

We consider a restricted class of hyperproperties for recursive programs, which relate only executions that access the call stack in the same manner, i.e., push or pop at the same time. An execution of a pushdown system $\mathcal{P}$ is a sequence of configurations (control state + stack) $\sigma = c_1 c_2 \cdots$, such that the stacks of consecutive configurations c_i and c_{i+1} differ only due to the possible presence of an additional element at the top of the stack of either c_i or c_{i+1}. For such a sequence, we can associate a sequence $\mathsf{pr}(\sigma) = \mathsf{o}_1\mathsf{o}_2 \cdots$ such that $\mathsf{o}_i \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}$ such that $\mathsf{o}_i = \mathsf{call}$ (ret respectively) if and only if the stack in c_{i+1} has one more (less respectively) element than c_i. The sequence $\mathsf{pr}(\sigma)$ is said to be the *stack access pattern* of σ. Observe that the stack sizes of two executions with the same stack access pattern evolve in a similar fashion. Thus, equivalently, we can consider this class of hyperproperties to be the hyperproperties that relate executions with identical memory usage.

To specify these hyperproperties, we propose the logic sHCTL* which extends H PERCTL*. sHCTL* has a two level syntax. At the innermost level, the syntax is identical to that of H PERCTL* formulas, but is interpreted over executions of the pushdown system with the same stack access pattern. At the outer level, we quantify over different stack access patterns. Intuitively, the formula $E\psi$ is true if there is a stack access pattern ρ exhibited by the system such that the set of executions with access pattern ρ satisfy the hyperproperty ψ. The dual formula $A\psi$, defined as $\neg E \neg \psi$, is true if for each stack access pattern ρ exhibited by the system, the set of all executions with stack access pattern ρ

satisfy ψ. The syntax of sHLTL is obtained from H perLTL in a similar fashion. Please see Fig. 1 on Page 8 for a side-by-side comparison of the syntax of H perCTL* (H perLTL) and sHCTL* (sHLTL). Unlike H perCTL*, we show that the problem of checking sHCTL* is decidable for pushdown systems (Theorem 3). Formal definitions of stack access patterns, syntax and semantics of sHCTL* are in Section 3.

For the rest of the paper, hyperproperties expressible in sHCTL* will be called *stack-aware hyperproperties*. Restricting to stack-aware hyperproperties is useful in verifying security guarantees of recursive programs as discussed below.

Proving $\forall\exists^*$ hyperproperties. The *noninference* property (Example 1) can be expressed in H perLTL as $\mathsf{NI} \stackrel{\text{def}}{=} \forall\pi.\,\exists\pi'.(\mathsf{G}\,\lambda_{\pi'}) \wedge \pi \equiv_L \pi'$. Consider the sHLTL formula $A(\mathsf{NI})$ obtained by putting an A in front NI. A pushdown system satisfies $A(\mathsf{NI})$ only if for each execution σ of the system, there is another execution σ' with *the same stack access pattern as* σ such that σ, σ' together satisfy $(\mathsf{G}\,\lambda_{\sigma'}) \wedge \sigma \equiv_L \sigma'$. Thus, if the pushdown system satisfies the sHLTL formula $A(\mathsf{NI})$, then it also satisfies noninference. Thus, a decision procedure for sHLTL can be used to prove that a recursive program satisfies noninference.

The above observation generalizes to H perLTL formulas of the form $\psi = \forall\pi.\exists\pi_1.\ldots\exists\pi_k.\psi'$ — if a system satisfies the sHLTL formula $A\psi$ then it must also satisfy the H perLTL formula ψ. Though the model checking problem is undecidable for pushdown systems even when restricted to such H perLTL formulas, we gain decidability by restricting the search space for $\pi, \pi_1, \ldots, \pi_k$.

Refuting $\forall^*$ hyperproperties. *Observational determinism* (Example 2) can be written in H perLTL as $\mathsf{OD} \stackrel{\text{def}}{=} \forall\pi.\forall\pi'.(\pi[0] \equiv_{L,in} \pi'[0]) \rightarrow \pi \equiv_{L,ou} \pi'$. Consider the sHLTL formula $A(\mathsf{OD})$. A pushdown system *fails* to satisfy the sHLTL formula $A(\mathsf{OD})$ only if there is a stack access pattern ρ and executions σ_1 and σ_2 with stack access pattern ρ such that the pushdown system does not satisfy $(\sigma[0] \equiv_{L,in} \sigma'[0]) \rightarrow \sigma \equiv_{L,ou} \sigma'$.

This observation generalizes to H perLTL formulas of the form $\psi = \forall\pi_1.\ldots\forall\pi_k.\psi'$ — if a pushdown system fails to satisfy the sHLTL formula $A\psi$ then it does not satisfy ψ. Even though model checking pushdown systems against such restricted specifications is undecidable, our decision procedure can be used to show that a recursive program does not meet such properties.

Exact verification when stack access pattern is observable. Often, it is reasonable to assume that the attacker can observe the stack access pattern. For example, in the program counter security model [15], the attacker has access to the program counter transcript, i.e., the sequence of program counters during an execution. Access to the program counter transcript implies that the attacker can observe stack access pattern. The assumption that the program counter transcript is observable helps model control flow side channel attacks which include timing attacks and error disclosure attacks [15]. sHCTL* can be used to verify security guarantees in this security model. For example, the sHCTL* formula $A(\mathsf{NI})$ models noninference faithfully by introducing a unique proposition for

each control state. Observational determinism can also be verified in this model by suitably transforming the pushdown automaton.

Another scenario in which stack access patterns are observable is when the attacker can observe the memory usage of a program in terms of stack size. As observing stack size may lead to information leakage, stack size should be considered a low-level observation. Since the stack size can be unbounded, it cannot be modeled as a proposition. sHCTL*, however, can still be used to verify security guarantees in this case. For example, $A(\mathsf{NI}) = A(\forall\pi.\,\exists\pi.'(\mathsf{G}\,\lambda_{\pi'}) \wedge \pi \equiv_L \pi')$ faithfully models non-inference as semantics of sHCTL* forces π and π' to have the same call-stack size in addition to other low-level observations. Once again, observational determinism can also be verified in this model by suitably transforming the pushdown automaton.

3 Stack-aware Hyper Computation Tree Logic (sHCTL*)

Stack-aware Hyper Computation Tree Logic (sHCTL*), and its sub-logic Stack-aware Hyper Linear Temporal Logic (sHLTL) are formally presented. We begin by establishing some conventions over strings.

Strings. A *string/word* w over a finite alphabet Σ is a sequence $w = a_0a_1\cdots$ of finite or infinitely many symbols from Σ, i.e., $a_i \in \Sigma$ for all i. The *length* of a string w, denoted $|w|$, is the number of symbols appearing in it — if $w = a_0a_1\cdots a_{n-1}$ is finite then $|w| = n$, and if $w = a_0a_1\cdots$ is infinite then $|w| = \omega$. The *unique* string of length 0, the *empty string*, is denoted ε. For a string $w = a_0a_1\cdots a_i\cdots$, $w(i) = a_i$ denotes the ith symbol, $w[:i] = a_0a_1\cdots a_{i-1}$ denotes the prefix of length i, $w[i:] = a_ia_{i+1}\cdots$ denotes the suffix of w starting at position i, and $w[i:j] = a_ia_{i+1}\cdots a_{j-1}$ denotes the substring from position i (included) to position j (not included). Thus $w[0:] = w$. By convention, when $i \leq 0$, we take $w[:i] = \varepsilon$. Over Σ, the set of all finite strings is denoted Σ^*, and the set of all infinite strings is denoted Σ^ω. For a finite string u and a (finite or infinite) string v, uv denotes the *concatenation* of u and v.

3.1 Pushdown Systems

Pushdown systems naturally model for sequential recursive programs. Formally, an AP-*labeled pushdown system* is a tuple $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$, where S is a finite set of *control states*, Γ is a finite set of *stack symbols*, $s_{\mathsf{in}} \in S$ is the *initial control state*, $L : S \to 2^{\mathsf{AP}}$ is the *labeling function*, and Δ is the transition relation. The transition relation $\Delta = \Delta_{\mathsf{int}} \uplus \Delta_{\mathsf{call}} \uplus \Delta_{\mathsf{ret}}$ is the disjoint union of *internal transitions* $\Delta_{\mathsf{int}} \subseteq S \times S$ where the stack is unchanged, *call transitions* $\Delta_{\mathsf{call}} \subseteq S \times (S \times \Gamma)$ where a single symbol is *pushed* onto the stack, and *return transitions* $\Delta_{\mathsf{ret}} \subseteq (S \times \Gamma) \times S$ where a single symbol is *popped* from the stack. When AP is clear from the context, we simply refer to them as pushdown systems.

Transition System Semantics. We recall the standard semantics of a pushdown system as a transition system. Let us fix a pushdown system $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$. A *configuration* c of $\mathcal{P}$ is a pair (s, α) where $s \in S$ and $\alpha \in \Gamma^*$.

$$a \in \mathsf{AP}, \pi \in \mathcal{V}$$

$$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi \,\mathsf{U}\, \psi \mid \exists \pi.\, \psi$$

(a) HyperCTL*

$$\theta ::= E\psi \mid \neg\theta \mid \theta \vee \theta$$
$$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi \,\mathsf{U}\, \psi \mid \exists \pi.\, \psi$$

(b) sHCTL*

Fig. 1: BNF for HyperCTL* and sHCTL*. Let $\forall$ denote $\neg\exists\neg$ and A denote $\neg E \neg \psi$. HyperLTL is the set of HyperCTL* formulas $Q_1\pi_1.\cdots Q_r\pi_r.\psi$ where $Q_i \in \{\exists, \forall\}$ and ψ is quantifier-free. sHLTL is the set of sHCTL* formulas $\text{Æ}\varphi$, where $\text{Æ} \in \{A, E\}$ and φ is in HyperLTL.

The set of all configurations of $\mathcal{P}$ will be denoted $\mathsf{Conf}_\mathcal{P} = S \times \Gamma^*$. The *labeled transition system* associated with $\mathcal{P}$ is $[\![\mathcal{P}]\!] := (\mathsf{Conf}_\mathcal{P}, c_{\mathsf{in}}, \longrightarrow, \mathsf{AP}, L)$ where $c_{\mathsf{in}} = (s_{\mathsf{in}}, \varepsilon)$ is the *initial configuration*, $\longrightarrow \subseteq \mathsf{Conf}_\mathcal{P} \times (\{\mathsf{call}, \mathsf{ret}, \mathsf{int}\} \times S \times (\Gamma \cup \{\varepsilon\}) \times S) \times \mathsf{Conf}_\mathcal{P}$ is the *transition relation*, and L is the *labeling function* that extends the labeling function of $\mathcal{P}$ to configurations as follows: $L(s, \alpha) = L(s)$. The transition relation $\longrightarrow$ is defined to capture the informal semantics of internal, call, and return transitions — for any $\alpha \in \Gamma^*$, (int) $(s, \alpha) \xrightarrow{(\mathsf{int}, s, \varepsilon, s')} (s', \alpha)$ iff $(s, s') \in \Delta_{\mathsf{int}}$; (call) $(s, \alpha) \xrightarrow{(\mathsf{call}, s, a, s')} (s', a\alpha)$ iff $(s, (s', a)) \in \Delta_{\mathsf{call}}$; and (ret) $(s, a\alpha) \xrightarrow{(\mathsf{ret}, s, a, s')} (s', \alpha)$ iff $((s, a), s') \in \Delta_{\mathsf{ret}}$.

A *path* of $[\![\mathcal{P}]\!]$ is an infinite sequence of configurations $\sigma = c_0, c_1, \ldots$ such that for each i, $c_i \xrightarrow{(\mathsf{o}, s, a, s')} c_{i+1}$ for some $\mathsf{o} \in \{\mathsf{int}, \mathsf{call}, \mathsf{ret}\}$, $s, s' \in S$ and $a \in \Gamma \cup \{\varepsilon\}$. The path σ is said to *start* in configuration c_0 (the first configuration in the sequence). We will use $\mathsf{Paths}([\![\mathcal{P}]\!], c)$ to denote the set of paths of $[\![\mathcal{P}]\!]$ starting in the configuration c and $\mathsf{Paths}([\![\mathcal{P}]\!])$ to denote all paths of $[\![\mathcal{P}]\!]$.

We conclude this section by introducing some notation on configurations. For $c = (s, \alpha)$, its *stack height* is $|\alpha|$, its *control state* is $\mathsf{state}(c) = s$, and its *top of stack symbol* is $\mathsf{top}(c) = a \in \Gamma$ if $\alpha = a\alpha'$ and is undefined if $\alpha = \varepsilon$.

3.2 Syntax of sHCTL*

Let us fix a set of atomic propositions AP, and a set of path variables, $\mathcal{V}$. The BNF grammar for sHCTL* formulas is given in Figure 1(b). In the BNF grammar, $a \in \mathsf{AP}$ is an *atomic proposition*, π is a *path variable*, ψ is a *cognate formula*, and θ is a sHCTL* formula. The syntax has two levels, with the inner level identical to H perCTL* formulas, while the outer level allows quantification over different stack access patterns (see Section 3.3). Also, following [5,10], we assume that the until operator U only occurs within the scope of a path quantifier.

Remark 1. We have chosen to not have A, the dual of E, and conjunction as explicit logical operators to keep our exposition simple. This choice does makes the automata constructions presented here less efficient for formulas involving

conjunction. Adding them explicitly does not pose a technical challenge to our setup and our automata constructions can be extended to handle them explicitly. In addition, we will sometimes use other quantifiers and logical operators to write formulas. Some standard examples include: $\theta_1 \wedge \theta_2 = \neg(\neg\theta_1 \vee \neg\theta_2)$, where θ_i ($i \in \{1, 2\}$) is either a sHCTL* or cognate formula; $\forall\pi.\psi = \neg\,\exists\pi.\,\neg\psi$; $\mathsf{F}\,\psi = \mathsf{true}\,\mathsf{U}\,\psi$, where $\mathsf{true} = a_\pi \vee \neg a_\pi$; $\mathsf{G}\,\psi = \neg\,\mathsf{F}\,\neg\psi$.

We call formulas of the form $\text{Æ}\psi$ (where $\text{Æ} \in \{A, E\}$ and ψ is a cognate formula) *basic formulas*. Observe that any sHCTL* formula is a Boolean combination of basic formulas. A sHCTL* formula θ is a *sentence* if in each basic sub-formula $\text{Æ}\psi$, ψ is a sentence, i.e., every path variable appearing in ψ is quantified. Without loss of generality, we assume that in any cognate formula ψ, all bound variables in ψ are renamed to ensure that any path variable is quantified at most once. We will only consider sHCTL* sentences in this paper. The logic sHLTL is the sub-logic of sHCTL* consisting of all formulas of the form $\text{Æ}Q_1\pi_1.\cdots Q_r\pi_r.\psi$ where $\text{Æ} \in \{A, E\}$, $Q_i \in \{\exists, \forall\}$ and ψ is quantifier free.

3.3 Semantics of sHCTL*

The syntax of cognate formulas is identical to that H PERCTL* formulas. Their semantics will be described in a similar manner, in a context where free path variables in the formula are interpreted as executions of a system. However, we will require that the interpretations of every path variable share a *common* stack access pattern — hence the term *cognate*. Thus, before defining the semantics, we will define what we mean by the *stack access pattern* of a path and a *path environment* that assigns an interpretation to path variables.

For the rest of this section let us fix a pushdown system $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$. A string $w \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^*$ is said to be *well matched* if either $w = \varepsilon$ or $w = \mathsf{int}$ or $w = \mathsf{call}\ u\ \mathsf{ret}$ or $w = uv$, where $u, v \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^*$ are (recursively) well matched. In a string $\rho \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$, $\rho(i)$ is an *unmatched return*, if $\rho[\,:i+1] = w\ \mathsf{ret}$, where w is well matched. We are now ready to present the definition of a stack access pattern.

Definition 1 (Stack access pattern). *A string $\rho \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$ is a* stack access pattern *if the set $\{i \in \mathbb{N} \mid \rho(i)$ is an unmatched return$\}$ is finite.*

A path $\sigma = c_0c_1c_2\cdots \in \mathsf{Paths}(\llbracket\mathcal{P}\rrbracket)$ is said to have a stack access pattern $\rho = o_0o_1\cdots$ (denoted $\mathsf{pr}(\sigma) = \rho$) if for every i: (a) $o_i = \mathsf{call}$ if and only if $\mathsf{stack}(c_{i+1}) = \mathsf{top}(c_{i+1})\,\mathsf{stack}(c_i)$, (b) $o_i = \mathsf{int}$ if and only if $\mathsf{stack}(c_{i+1}) = \mathsf{stack}(c_i)$, and (c) $o_i = \mathsf{ret}$ if and only if $\mathsf{stack}(c_i) = \mathsf{top}(c_i)\,\mathsf{stack}(c_{i+1})$.

We now present the definition of *path environment* that interprets the free path variables in a cognate formula as paths of $\llbracket\mathcal{P}\rrbracket$ such that they share a common stack access pattern. This plays a key role in defining the semantics of sHCTL*. For a set of path variables $\mathcal{V}$, let $\mathcal{V}^\dagger$ be defined as the set $\mathcal{V} \cup \{\dagger\}$.

Definition 2 (Path Environment). *A* path environment *for pushdown system $\mathcal{P}$ over variables $\mathcal{V}$ is function $\Pi : \mathcal{V}^\dagger \to \mathsf{Paths}(\llbracket\mathcal{P}\rrbracket) \cup \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$ such*

that $\Pi(\dagger)$ is a stack access pattern , and for every $\pi \in \mathcal{V}$, $\Pi(\pi) \in \mathsf{Paths}([\![\mathcal{P}]\!])$ with $\mathsf{pr}(\Pi(\pi)) = \Pi(\dagger)$. When the pushdown system is clear from the context, we will simply refer to it as a path environment over $\mathcal{V}$.

When $\mathcal{V} = \emptyset$, we additionally require that there is a path $\sigma \in \mathsf{Paths}([\![\mathcal{P}]\!], c_{\mathsf{in}})$ (where c_{in} is the initial configuration of $[\![\mathcal{P}]\!]$) such that $\mathsf{pr}(\sigma) = \Pi(\dagger)$.

We introduce some notation related to path environments. Let us fix a path environment Π over variables $\mathcal{V}$. Given a path $\sigma \in \mathsf{Paths}([\![\mathcal{P}]\!])$, $\Pi[\pi \mapsto \sigma]$ denotes the path environment over $\mathcal{V} \cup \{\pi\}$ such that $\Pi[\pi \mapsto \sigma](\pi) = \sigma$, and $\Pi[\pi \mapsto \sigma](\pi') = \Pi(\pi')$, for any $\pi' \in \mathcal{V}^{\dagger}$ with $\pi' \neq \pi$. Finally, for $i \in \mathbb{N}$, $\Pi[i:]$ denotes the *suffix* path environment, where every variable is mapped to the suffix of the path starting at position i. More formally, for every $\pi' \in \mathcal{V}^{\dagger}$, $\Pi[i:](\pi') = \Pi(\pi')[i:]$.

We now define when a pushdown system $\mathcal{P}$ satisfies a sHCTL* sentence θ, denoted $\mathcal{P} \models \theta$. The definition of satisfaction of θ relies on a definition of satisfaction for cognate formulas. To inductively to define the semantics of cognate formulas, we will interpret free path variables using a path environment. Finally, as in H perCTL*, it is important to track the most recently quantified path variable because that influences the semantics of $\exists\pi(\cdot)$. Thus satisfaction of cognate formulas takes the form $\mathcal{P}, \Pi, \pi' \models \psi$, where π' is the most recently quantified path variable, and Π is a path environment over the free variables of ψ. Finally, by convention, we will take $\mathsf{Paths}([\![\mathcal{P}]\!], \Pi(\dagger)(0))$ to mean $\mathsf{Paths}([\![\mathcal{P}]\!], c_{\mathsf{in}})$, where c_{in} is the initial configuration of $[\![\mathcal{P}]\!]$ [5]. Below, θ, θ_1, and θ_2 are sHCTL* sentences, while ψ, ψ_1, ψ_2 are cognate formulas.

$$
\begin{aligned}
&\mathcal{P} \models \neg\theta \text{ iff } \mathcal{P} \not\models \theta\\
&\mathcal{P} \models \theta_1 \vee \theta_2 \text{ iff } \mathcal{P} \models \theta_1 \text{ or } \mathcal{P} \models \theta_2\\
&\mathcal{P} \models E\psi \text{ iff for some path environment } \Pi \text{ over } \emptyset, \mathcal{P}, \Pi, \dagger \models \psi\\
&\mathcal{P}, \Pi, \pi' \models a_\pi \text{ iff } a \in L(\Pi(\pi)(0))\\
&\mathcal{P}, \Pi, \pi' \models \neg\psi \text{ iff } \mathcal{P}, \Pi, \pi' \not\models \psi\\
&\mathcal{P}, \Pi, \pi' \models \psi_1 \vee \psi_2 \text{ iff } \mathcal{P}, \Pi, \pi' \models \psi_1 \text{ or } \mathcal{P}, \Pi, \pi' \models \psi_2\\
&\mathcal{P}, \Pi, \pi' \models \mathsf{X}\psi \text{ iff } \mathcal{P}, \Pi[1:], \pi' \models \psi\\
&\mathcal{P}, \Pi, \pi' \models \psi_1 \,\mathsf{U}\, \psi_2 \text{ iff } \exists i \geq 0 : \mathcal{P}, \Pi[i:], \pi' \models \psi_2 \text{ and } \forall j, 0 \leq j < i,\\
&\qquad\qquad \mathcal{P}, \Pi[j:], \pi' \models \psi_1\\
&\mathcal{P}, \Pi, \pi' \models \exists\pi.\, \psi \text{ iff } \exists \sigma \in \mathsf{Paths}([\![\mathcal{P}]\!], \Pi(\pi')(0)) \text{ with } \mathsf{pr}(\sigma) = \Pi(\dagger),\\
&\qquad\qquad \text{such that } \mathcal{P}, \Pi[\pi \mapsto \sigma], \pi \models \psi
\end{aligned}
$$

4 A Decision Procedure for sHCTL*

Given a pushdown system $\mathcal{P}$ and a sHCTL* sentence θ, we present an algorithm that determines if $\mathcal{P} \models \theta$. Our approach is similar to the one in [10]. Given a finite state transition system $\mathcal{K}$ and a H perCTL* formula φ, Finkbeiner et. al. [10], construct an alternating (finite state) Büchi automaton $\mathcal{A}_{\mathcal{K},\varphi}$, by induction on φ, such that an input word σ is accepted by $\mathcal{A}_{\mathcal{K},\varphi}$ if and only if σ is the encoding

[5] The convention is needed because $\Pi(\dagger)(0)$ is not a configuration but an element of the set $\{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}$.

of a path environment Π such that $\mathcal{K}, \Pi \models \varphi$. Determining if $\mathcal{K} \models \varphi$ then reduces to checking if $\mathcal{A}_{\mathcal{K},\varphi}$ accepts any string.

Extending these ideas to sHCTL* and pushdown systems, requires one to answer two questions: (a) What is an encoding of path environments for cognate formulas where path variables are mapped to sequences of configurations (control state + stack)?; (b) Which automata models can capture the collection of path environments satisfying a cognate formula with respect to a pushdown system? We encode path environments for cognate formulas using strings over a *pushdown alphabet* — pushdown tags on symbols adds structure that helps encode sequences of configurations. And for automata, we consider automata that process such strings and accept *visibly pushdown languages*. A natural generalization of the approach outlined in [10] would suggest the use of alternating visibly pushdown automata (AVPA) on infinite strings [4]. However, using AVPAs results in an inefficient algorithm. To get a more efficient algorithm, we instead rely on a careful use of *nondeterministic visibly pushdown automata (NVPA)* [1] and *1-way alternating jump automata (*1*-AJA)* [4]. The advantage of using NVPA and 1-AJA can be seen in the case of existential quantification ($\exists\pi.$) which requires converting an alternating automaton to a nondeterministic one [10]: Converting from 1-AJA to NVPA leads to exponential blowup while converting AVPA to NVPA leads to a doubly exponential blowup [4].

The rest of this section is organized as follows. We begin by introducing the automata models on pushdown alphabets (Section 4.1). Next we present our encoding of path environments, and finally our automata constructions that establish the decidability result (Section 4.2).

4.1 Automata on Pushdown Alphabets

A *pushdown alphabet* is a finite set Σ that is partitioned into three sets $\Sigma_{\mathsf{call}} \uplus \Sigma_{\mathsf{int}} \uplus \Sigma_{\mathsf{ret}}$, where Σ_{call} is the set of *call symbols*, Σ_{int} is the set of *internal symbols*, and Σ_{ret} is the set of *return symbols*. Automata models processing strings over a pushdown alphabet are restricted to perform certain types of transitions based on whether the read symbol is a call, internal, or return symbol. We introduce, informally, two such automata models next. Precise definition and its semantics can be found in the detailed version of this paper [2].

Nondeterministic Visibly Pushdown Büchi Automata. A *nondeterministic visibly pushdown automaton (NVPA)* [1] is like a pushdown system. It has finitely many control states and uses an unbounded stack for storage. However, unlike a pushdown system, it is an automaton that processes an infinite sequence of input symbols from a pushdown alphabet $\Sigma = \Sigma_{\mathsf{call}} \uplus \Sigma_{\mathsf{int}} \uplus \Sigma_{\mathsf{ret}}$. Transitions are constrained to conform to pushdown alphabet — whenever a Σ_{call} symbol is read, a symbol onto the stack, whenever a Σ_{ret} symbol is read, the top stack symbol is popped, and whenever Σ_{int} symbol is read, the stack is unchanged.

1-way Alternating Jump Automata. Our second automaton model is *1-way Alternating Parity Jump Automata (1-AJA)* [4]. 1-AJA are computationally equivalent to NVPAs (i.e., accept the same class of languages) but provide

greater flexibility in describing algorithms. 1-AJAs are alternating automata, which means that they can define acceptance based on multiple runs of the machine on an input word. Though they are finite state machines with no auxiliary storage, their ability to spawn a computation thread that jumps to a future portion of the input string on reading a symbol, allows them to have the same computational power as a more conventional machine with storage (like NVPAs).

We present some useful properties of NVPA and 1-AJA. The two models are equi-expressive with the size of automata constructed by the translation known.

Theorem 1 ([4]). *For any NVPA N of size n, there is a 1-AJA $\mathcal{A}_N$ of size $O(n^2)$, such that $\mathcal{L}(\mathcal{A}_N) = \mathcal{L}(N)$. Conversely, for any 1-AJA $\mathcal{A}$ of size n, there is a NVPA $N_{\mathcal{A}}$ of size $2^{O(n)}$, such that $\mathcal{L}(N_{\mathcal{A}}) = \mathcal{L}(\mathcal{A})$. Constructions can be carried out in time proportional to the size of the resulting automaton.*

Both 1-AJA and NVPAs are closed for language operations like complementation, union and prefixing. We also recall the following result.

Theorem 2 ([1]). *For NVPAs, the emptiness problem is PTIME-complete.*

4.2 Algorithm for sHCTL*

Let us fix a pushdown system $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$ and a sHCTL* sentence θ. Our goal is to decide if $\mathcal{P} \models \theta$. We will reduce this problem to checking the emptiness of multiple NVPAs (Theorem 2). Our approach is similar to [10] — for each cognate sub-formula ψ (not necessarily sentence) of θ, we will compositionally construct an automaton that accepts the path environments satisfying ψ. Path environments will be encoded by strings over pushdown alphabets as follows.

For a path $\sigma = c_0 c_1 c_2 \cdots$ of $[\![\mathcal{P}]\!]$, the *trace* of σ, denoted $\mathsf{tr}(\sigma)$, is the (unique) sequence $(\mathsf{o}_0, q_0, a_0, q_1)(\mathsf{o}_1, q_1, a_1, q_2)\cdots$ such that for every $i \in \mathbb{N}$, $c_i \xrightarrow{(\mathsf{o}_i, q_i, a_i, q_{i+1})} c_{i+1}$ where $\mathsf{o}_i \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}$, $q_i, q_{i+1} \in Q$, and $a_i \in \Gamma \cup \{\varepsilon\}$ [6].

While $\mathsf{tr}(\sigma)$ is uniquely determined by the path σ, the converse is not true — different paths may have the same trace. To see this, consider the following example. For configuration c and $\gamma \in \Gamma^*$, let $\gamma(c)$ denote the configuration $(\mathsf{state}(c), \mathsf{stack}(c)\gamma)$, i.e., the configuration with the same control state, but with stack containing the symbols in γ at the bottom. Observe that, for any $\gamma \in \Gamma^*$, if $\sigma = c_0 c_1 c_2 \cdot$ is a path then so is $\gamma(\sigma) = \gamma(c_0)\gamma(c_1)\gamma(c_2)\cdots$. Additionally, $\mathsf{tr}(\sigma) = \mathsf{tr}(\gamma(\sigma))$. Two paths σ_1 and σ_2 of $[\![\mathcal{P}]\!]$ will be said to be equivalent if $\mathsf{tr}(\sigma_1) = \mathsf{tr}(\sigma_2)$ and will be denoted as $\sigma_1 \simeq \sigma_2$. Observe that equivalent paths have the same stack access pattern , i.e. if $\sigma_1 \simeq \sigma_2$ then $\mathsf{pr}(\sigma_1) = \mathsf{pr}(\sigma_2)$. The semantics of sHCTL* doesn't distinguish between equivalent paths.

[6] Observe that even when σ is not a path in $[\![\mathcal{P}]\!]$ (i.e., corresponds to an actual sequence of transitions of $\mathcal{P}$), the trace of σ is uniquely defined as long as stacks of successive configurations of σ can be obtained by leaving the stack unchanged, or pushing/popping one symbol.

Proposition 1. *Let φ be a cognate formula with $\mathcal{V}$ as the set of free path variables. Let Π_1 and Π_2 be two path environments such that for every $\pi \in \mathcal{V}$, $\Pi_1(\pi) \simeq \Pi_2(\pi)$. Then, $\mathcal{P}, \Pi_1, \pi \models \varphi$ if and only if $\mathcal{P}, \Pi_2, \pi \models \varphi$.*

The proof of Proposition 1 follows by induction on cognate formulas. Proposition 1 establishes that the set of path environments satisfying a cognate formula is a union of equivalence classes with respect to path equivalence. Thus, instead of constructing automata that accept path environments, we will construct automata that accept mappings from path variables to traces of paths. For $m \in \mathbb{N}$, let $\Sigma[m] = \Sigma[m]_{\mathsf{call}} \uplus \Sigma[m]_{\mathsf{int}} \uplus \Sigma[m]_{\mathsf{ret}}$ be the pushdown alphabet where $\Sigma[m]_{\mathsf{call}} = \{\mathsf{call}\} \times S^m \times \Gamma^m$, $\Sigma[m]_{\mathsf{int}} = \{\mathsf{int}\} \times S^m \times \{\varepsilon\}^m$, and $\Sigma[m]_{\mathsf{ret}} = \{\mathsf{ret}\} \times S^m \times \Gamma^m$. Observe $\Sigma[0]$ is (essentially) the set $\{\mathsf{int}, \mathsf{call}, \mathsf{ret}\}$.

Definition 3 (Encoding Path Environments). *Consider a set of m path variables $\mathcal{V} = \{\pi_1, \pi_2, \ldots \pi_m\}$. A string $w \in \Sigma[m]^\omega$ where for any $j \in \mathbb{N}$, $w(j) = (\mathsf{o}_j, (s_1^j, s_2^j, \ldots s_m^j), (a_1^j, a_2^j, \ldots a_m^j))$ encodes all path environments Π such that*

$$\Pi(\dagger) = \mathsf{o}_0\mathsf{o}_1\mathsf{o}_2 \cdots \mathsf{o}_j \cdots$$
$$\mathsf{tr}(\Pi(\pi_i)) = (\mathsf{o}_0, s_i^0, a_i^0, s_i^1)(\mathsf{o}_1, s_i^1, a_i^1, s_i^2) \cdots$$

for any $i \in \{1, 2, \ldots m\}$. The string encoding a path environment Π is denoted as $\mathsf{enc}(\Pi)$ (= w, in this case).

Based on the definitions, the following observation about traces and encodings can be concluded.

Proposition 2. *For any path $\sigma \in \mathsf{Paths}([\![\mathcal{P}]\!])$ and $i \in \mathbb{N}$, $\mathsf{tr}(\sigma[i:]) = \mathsf{tr}(\sigma)[i:]$. For any path environment Π and $i \in \mathbb{N}$, $\mathsf{enc}(\Pi[i:]) = \mathsf{enc}(\Pi)[i:]$.*

The encoding of path environments as strings over $\Sigma[m]$ (for an appropriate value of m) is used in our decision procedure, which compositionally constructs automata that accept path environments satisfying each cognate formula. The size of our constructed automata, like in [10], will be tower of exponentials that depends on the *formula complexity* of the cognate formula φ.

Definition 4 (Formula Complexity). *The* formula complexity *of a* sHCTL* *formula φ, denoted $\mathsf{fc}(\varphi)$, is inductively defined as follows. Let $\mathsf{odd} : \mathbb{N} \to \mathbb{N}$ be the function that maps a number n to the smallest odd number $\geq n$, i.e., $\mathsf{odd}(n) = n$ if n is odd and $\mathsf{odd}(n) = n + 1$ if n is even. Similarly, $\mathsf{even} : \mathbb{N} \to \mathbb{N}$ maps n to the smallest even number $\geq n$, i.e., $\mathsf{even}(n) = \mathsf{odd}(n + 1) - 1$. Below ψ_1, ψ_2 denote cognate formulas, and θ_1, θ_2 denote* sHCTL* *sentences.*

$$\mathsf{fc}(a_\pi) = 0 \qquad \mathsf{fc}(\neg\psi_1) = \mathsf{even}(\mathsf{fc}(\psi_1)) \qquad \mathsf{fc}(\mathsf{X}\psi_1) = \mathsf{fc}(\psi_1)$$
$$\mathsf{fc}(\psi_1 \vee \psi_2) = \max(\mathsf{fc}(\psi_1), \mathsf{fc}(\psi_2)) \qquad \mathsf{fc}(\psi_1 \,\mathsf{U}\, \psi_2) = \mathsf{even}(\max(\mathsf{fc}(\psi_1), \mathsf{fc}(\psi_2)))$$
$$\mathsf{fc}(\exists\pi.\, \psi_1) = \mathsf{odd}(\mathsf{fc}(\psi_1)) \qquad \mathsf{fc}(E\psi_1) = \mathsf{odd}(\mathsf{fc}(\psi_1))$$
$$\mathsf{fc}(\neg\theta_1) = \mathsf{fc}(\theta_1) \qquad \mathsf{fc}(\theta_1 \vee \theta_2) = \max(\mathsf{fc}(\theta_1), \mathsf{fc}(\theta_2))$$

Observe the difference in the definition of $\mathsf{fc}(\neg\theta_1)$ and $\mathsf{fc}(\neg\psi_1)$; for $\neg\theta_1$ there is no change in formula complexity, while for $\neg\psi_1$ we move to the next even level.

Our main technical lemma is a compositional construction of an automaton for cognate formulas ψ. Depending on the parity of $\mathsf{fc}(\psi)$, the automaton we construct will either be a 1-AJA or a NVPA. Before presenting this lemma, we define a function that is a tower of exponentials. For $c, k, n \in \mathbb{N}$, the value $g_c(k, n)$ is defined inductively on k as follows: $g_c(0, n) = cn \log n$, and $g_c(k+1, n) = 2^{g_c(k,n)}$. We use $g_{O(1)}(k, n)$ to denote the family of functions $\{g_c(k, n) \mid c \in \mathbb{N}\}$.

Lemma 1. *Consider pushdown system $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$ and* sHCTL* *sentence θ. Let ψ be a cognate subformula of θ with free path variables in the set $\mathcal{V} = \{\pi_1, \ldots \pi_m\}$ for $m \in \mathbb{N}$. We assume, without loss of generality, that the variables $\pi_1, \ldots \pi_m$ are in the order in which they are quantified in θ with π_m being the first free variable of ψ that will be quantified in the context θ. In addition, we assume that the size of both ψ and $\mathcal{P}$ is bounded by n. There is an automaton $\mathcal{A}_\psi$ over pushdown alphabet $\Sigma[m]$ such that for any path environment Π over $\mathcal{V}$,*

$$\mathcal{P}, \Pi, \pi_m \models \psi \text{ if and only if } \mathsf{enc}(\Pi) \in \mathcal{L}(\mathcal{A}_\psi).$$ [7]

The automaton $\mathcal{A}_\psi$ is a NVPA if $\mathsf{fc}(\psi)$ is odd, and a 1-AJA if $\mathsf{fc}(\psi)$ is even. The size of $\mathcal{A}_\psi$ is at most $g_{O(1)}(\lceil \frac{\mathsf{fc}(\psi)}{2} \rceil, n)$[8].

Before presenting the proof of Lemma 1, we would like to highlight a subtlety about its statement. The result guarantees that for *valid* path environments Π, encoding $\mathsf{enc}(\Pi)$ is accepted by $\mathcal{A}_\psi$ if and only if Π satisfies ψ. It says nothing about path environments that are not valid. In particular, there may be functions that map path variables to traces that do not correspond to actual paths of $[\![\mathcal{P}]\!]$, but which are nonetheless accepted by $\mathcal{A}_\psi$. Notice, however, when $\psi = \exists \pi. \psi_1$ is a cognate sentence, a string over $\{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}$ will, by conditions guaranteed in Lemma 1, be accepted if and only if it corresponds to a stack access pattern of a path from the initial state that satisfies $\exists \pi. \psi_1$.

Proof (Sketch of Lemma 1). Our construction of $\mathcal{A}_\psi$ will proceed inductively. The type of automaton constructed will be consistent with the parity of $\mathsf{fc}(\psi)$, i.e., an NVPA if $\mathsf{fc}(\varphi)$ is odd and a 1-AJA if $\mathsf{fc}(\psi)$ is even. We sketch the main ideas here, with the full proof in [2].

For a_π, $\neg\psi_1$, $\psi_1 \vee \psi_2$, and $\mathsf{X}\psi_1$, the construction essentially proceeds by converting $\mathcal{A}_{\psi_i}$ ($i \in \{1, 2\}$) if needed, into the type (NVPA or 1-AJA) of the target automaton using Theorem 1, and then using standard closure properties to combine them to get the desired automaton. In case of $\psi = \psi_1 \,\mathsf{U}\, \psi_2$, we first convert (if needed) $\mathcal{A}_{\psi_i}$ ($i \in \{1, 2\}$) into a 1-AJA. At each step, the automaton for ψ will choose to either run $\mathcal{A}_{\psi_2}$, or run $\mathcal{A}_{\psi_1}$ *and* restart itself. Correctness relies on the fact that our encoding for path environments satisfies Proposition 2.

The most interesting case is that of $\psi = \exists \pi. \psi_1$. We will first convert (if needed) the automaton for ψ_1 into a NVPA $\mathcal{A}_1$. The automaton for ψ will essentially guess the encoding of a path that is consistent with the transitions of

[7] When $m = 0$, we take π_m to be $\dagger$.

[8] When the size of the specification ψ is considered constant, the size of $\mathcal{A}_\psi$ is at most $g_{O(1)}(\lceil \frac{\mathsf{fc}(\psi)}{2} \rceil - 1, n)$

$\mathcal{P}$, and check if assigning the guessed path to variable π satisfies ψ_1 by running the automaton $\mathcal{A}_1$. The additional requirement we have is that the guessed path start at the *same configuration* as the current configuration of the path assigned to variable π_m which introduces some subtle challenges. In order to be able to guess a path, $\mathcal{A}_\psi$ will keep track of $\mathcal{P}$'s control state in its control state, and use its stack to track $\mathcal{P}$'s stack operations along the guessed path. Since the stacks of all paths are synchronized, it makes it possible for $\mathcal{A}_\psi$ to use its (single stack) to track the stack of both $\mathcal{P}$ and the stack of $\mathcal{A}_1$. □

Using Lemma 1, we can establish the main result of this section.

Theorem 3. *Given a $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$ and a* sHCTL* *sentence θ, the problem of determining if $\mathcal{P} \models \theta$ is in $\cup_c \mathsf{DTIME}(g_c(\lceil \frac{\mathsf{fc}(\theta)}{2} \rceil, n))$, where n is a bound on the size of $\mathcal{P}$ and θ.*

Proof. Recall that a sHCTL* sentence is a Boolean combination of formulas of the form $E\psi$, where ψ is a cognate sentence. Results on whether $\mathcal{P} \models E\psi$ for each such subformula can be combined to determine whether $\mathcal{P} \models \theta$. Given this, the time to determine if $\mathcal{P} \models \theta$ is at most the time to decide if $\mathcal{P}$ satisfies each subformula of the form $E\psi$ plus $O(n)$ (to compute the Boolean combination of these results). Next, recall that the construction in Lemma 1 ensures that for a cognate sentence of the form $\exists \pi.\, \psi$, $\mathcal{L}(\mathcal{A}_{\exists \pi.\, \psi})$ consists exactly of strings in $\{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$ that encode a path environment over $\emptyset$ that satisfy $\exists \pi.\, \psi$.

Consider a sHCTL* sentence $E\psi$. Let π be a path variable that does not appear in the sentence ψ. Based on the semantics of sHCTL* the following observation holds: $\mathcal{P} \models E\psi$ if and only if for some path environment Π over $\emptyset$, $\mathcal{P}, \Pi, \dagger \models \exists \pi.\, \psi$. Which is equivalent to saying that $\mathcal{P} \models E\psi$ if and only if $\mathcal{L}(\mathcal{A}_{\exists \pi.\, \psi}) \neq \emptyset$. Since $\mathsf{fc}(E\psi) = \mathsf{fc}(\exists \pi.\, \psi)$, and the emptiness problem of NVPA can be decided in polynomial time (Theorem 2), our theorem follows. □

5 Lower Bound

In this section, we establish a lower bound for the problem of model checking sHCTL* sentences against pushdown systems. Our proof establishes a hardness result for the sHLTL sub-fragment of sHCTL*. Before presenting this lower bound, we introduce the function $h_c(\cdot, \cdot)$, which is another tower of exponentials, inductively defined as follows: $h_c(0, n) = n$, and $h_c(k+1, n) = h_c(k, n) \cdot c^{h_c(k,n)}$.

Theorem 4. *Let $\mathcal{P}$ be a pushdown system and θ be a* sHLTL *sentence such that the sizes of both $\mathcal{P}$ and θ is bounded by n and $\mathsf{fc}(\theta) = 2k - 1$ for some $k \in \mathbb{N}$. The problem of checking if $\mathcal{P} \models \theta$ is $\mathsf{DTIME}(h_c(k, n))$-hard, for every $c \in \mathbb{N}$.*

Proof (Sketch). We sketch the main intuitions behind the proof. To highlight the novelties of this proof, it is useful to recall how $\mathsf{NSPACE}(h_c(k-1, n))$-hardness for H PERLTL model checking is proved [5]. The idea is to reduce the language of a nondeterministic $h_c(k-1, n)$ space bounded machine M to the model checking

problem by constructing a finite state transition system that guesses a run of M, and a H PERLTL formula that checks if the path is a valid accepting run.

To get the stricter bound of $\mathsf{DTIME}(h_c(k, n))$, we use the fact that we are checking pushdown systems. The stack of the pushdown system can be used to guess a *tree*, as opposed to a simple trace. Therefore, we reduce a $h_c(k - 1, n)$ space bounded *alternating* Turing machine, instead of a nondeterministic machine. Since $\mathsf{ASPACE}(f(n)) = \mathsf{DTIME}(2^{O(f(n))})$ for $f(n) \geq \log n$, the theorem will follow if the reduction succeeds.

Recall that a run of an alternating Turing machine M is a rooted, labeled tree, where vertices are labeled by configurations of M in a manner that is consistent with the transition function of M. To faithfully encode a tree as a sequence of symbols, we record the DFS traversal of the tree, making explicit the stack operations performed during such a traversal. Consider a labeled, rooted tree T with root r whose label is $\ell(r)$ with T_1 as a the left sub-tree and T_2 as the right sub-tree. The DFS traversal of T will push $\ell(r)$, traverse T_1 recursively, pop $\ell(r)$, push $\ell(r)$, traverse T_2, and then pop $\ell(r)$. We will use such a DFS traversal to guess and encode runs of M. Popping and pushing $\ell(r)$ between the traversals of T_1 and T_2 may seem redundant. Why not simply do nothing between the traversals of T_1 and T_2? For T to be a valid run of M, the configuration labeling of the root of T_2 must be the result of taking one step from $\ell(r)$. Such checks will be encoded in our sHLTL sentence, and for that to be possible, we need successive configurations of M to be consecutive in the string encoding.

To highlight some additional consistency checks, let us continue with our example tree T from the previous paragraph. For a string to be a correct encoding of T, it is necessary that the string pushed before the traversal of T_i ($i \in \{1, 2\}$) be the same as the string popped after the traversal. This can be ensured by the pushdown system by actually pushing and popping those symbols. In addition, the string popped after T_1's traversal must be the same as the string pushed before T_2's traversal. Neither the stack nor the finite control of the pushdown system can be used to ensure this. Instead this must be checked by the sHLTL sentence we construct. But the symbols while popping $\ell(r)$ will be in reverse order of the symbols being pushed, and it is challenging to perform this check in the formula. To overcome this, we push/pop the label *and its reverse* at the same time. This ensures that if we want to check if a string pushed is the same as a string that was just popped, then we can check for string *equality*, and this check is easier to do using formulas in sHLTL. Additional checks to ensure that the tree encodes a valid accepting run are performed by the sHLTL sentence using ideas from [17]. Full details can be found in [2]. □

6 Conclusions

In this paper, we introduced a branching time temporal logic sHCTL* that can be used to specify synchronous hyperproperties for recursive programs modeled as pushdown systems. The primary difference from the standard branching time logic H PERCTL* for synchronous hyperproperties is that sHCTL* considers

a restricted class of hyperproperties, namely, those that relate only executions that the same stack access pattern. We call such hyperproperties stack-aware hyperproperties. We showed that the problem of model checking pushdown systems sHCTL* specifications is decidable, and characterized its complexity. We also showed how this result can potentially be used to aid security verification.

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing. pp. 202–211. ACM (2004)
2. Bajwa, A., Zhang, M., Chadha, R., Viswanathan, M.: Stack-aware hyperproperties. https://arxiv.org/abs/2301.11521 (2023)
3. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Concurrency Theory, 8th International Conference. pp. 135–150. Springer (1997)
4. Bozzelli, L.: Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In: Concurrency Theory, 18th International Conference. pp. 476–491. Springer (2007)
5. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Principles of Security and Trust - Third International Conference. pp. 265–284. Springer (2014)
6. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium. pp. 51–65. IEEE Computer Society (2008)
7. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Computer Aided Verification - 31st International Conference. pp. 121–139. Springer (2019)
8. Finkbeiner, B., Hahn, C., Stenger, M.: EAHyper: Satisfiability, implication, and equivalence checking of hyperproperties. In: Computer Aided Verification - 29th International Conference. pp. 564–570. Springer (2017)
9. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: A runtime verification tool for temporal hyperproperties. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference. pp. 194–200. Springer (2018)
10. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Computer Aided Verification - 27th International Conference. pp. 30–48. Springer (2015)
11. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. pp. 11–20. IEEE Computer Society (1982)
12. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Deciding asynchronous hyperproperties for recursive programs. CoRR **abs/2201.12859** (2022)
13. McLean, J.: Proving noninterference and functional correctness using traces. Journal of Computer Security **1**(1), 37–58 (1992)
14. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: IEEE Computer Society Symposium on Research in Security and Privacy. pp. 79–93. IEEE Computer Society (1994)
15. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: Proceedings of the 8th international conference on Information Security and Cryptology. p. 156–168. Springer-Verlag (2005)

16. Pommellet, A., Touili, T.: Model-checking HyperLTL for pushdown systems. In: Model Checking Software - 25th International Symposium. pp. 133–152. Springer (2018)
17. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for büchi automata with appplications to temporal logic. Theoretical Computer Science **49**, 217–237 (1987)
18. Walukiewicz, I.: Pushdown processes: Games and model checking. In: Computer Aided Verification, 8th International Conference. pp. 62–74. Springer (1996)
19. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings of the 16th IEEE Computer Security Foundations Workshop. p. 29. IEEE Computer Society (2003)

Proofs

Propositional Proof Skeletons*

Joseph E. Reeves[1](✉), Benjamin Kiesl-Reiter[2], and Marijn J. H. Heule[1,2]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
{jereeves,mheule}@cs.cmu.edu
[2] Amazon Web Services, Seattle, WA, USA
benkiesl@amazon.com

Abstract. Modern SAT solvers produce proofs of unsatisfiability to justify the correctness of their results. These proofs, which are usually represented in the well-known DRAT format, can often become huge, requiring multiple gigabytes of disk storage. We present a technique for semantic proof compression that selects a subset of important clauses from a proof and stores them as a so-called proof skeleton. This proof skeleton can later be used to efficiently reconstruct a full proof by exploiting parallelism. We implemented our approach on top of the award-winning SAT solver CaDiCaL and the proof checker DRAT-trim. In an experimental evaluation, we demonstrate that we can compress proofs into skeletons that are 100 to 5,000 times smaller than the original proofs. For almost all problems, proof reconstruction using a skeleton improves the solving time on a single core, and is around five times faster when using 24 cores.

Keywords: SAT solving · proofs · compression.

1 Introduction

Solvers for the Boolean satisfiability problem (SAT) take as input a formula of propositional logic and decide if the formula is satisfiable. In case of satisfiability, they usually return an assignment of truth values to the variables of the formula; by plugging these truth values into the formula, users can easily convince themselves that the solver was right and that the formula is indeed satisfiable. In case of unsatisfiability, however, things are more complicated: to justify their answer, solvers need to produce an independently checkable proof that none of the—exponentially many—potential truth assignments make the formula true.

In practical SAT solving, proofs of unsatisfiability are represented in the DRAT format [10], and they are often huge, requiring several gigabytes (in some cases even terabytes [12] or petabytes [11]) of disk storage. Storing proofs is thus costly, especially since users might not require access to the proofs until sometime long after solving, at a point when proof verification or further analysis is desired.

* Supported by the U.S. National Science Foundation under grant CCF-2229099, and supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 329–347, 2023.
https://doi.org/10.1007/978-3-031-30823-9_17

Up to now, the only options to deal with this problem were either to not store proofs and instead recompute them on demand—a laborious but plausible approach considering that proof checking typically takes longer than solving—or to use compression methods to reduce proof size. However, syntactic compression techniques (such as LZMA or DEFLATE, as supported by the ZIP file format) only provide moderate levels of compression. The same can be said about existing semantic compression techniques for proofs in SAT and SMT (c.f. [4, 18, 21]), which only achieve 20% compression on average.

In this paper, we present a novel approach to semantic compression that stores only a small subset of the clauses derived by a solver, called a *proof skeleton*. We can achieve strong compression rates with proof skeletons (around 100 to 5,000 times smaller than the original proof), while still retaining enough information to allow for a quick on-demand reconstruction of a complete proof that might differ from the original proof. This is similar to how a mathematician might put down the most important reasoning steps of a proof in a proof sketch, enabling a moderately talented reader to fill in the gaps. In our case, the gaps can even be filled independently, meaning that multiple readers can work in parallel.

We present both an online version (creating a proof skeleton during solving) and an offline version (creating a proof skeleton from a full proof) of our approach. We select the clauses that end up in a proof skeleton by relying on several heuristics such as *glue* (a heuristic used internally by solvers to estimate the usefulness of clauses) for online and *clause activity* (a measure of how often a clause is used to derive new clauses) for offline. To reconstruct a full proof from a proof skeleton, we utilize multiple incremental SAT solvers that can run in parallel. We implemented all our algorithms on top of the award-winning SAT solver CaDiCaL [2] and the proof checker DRAT-trim [22]. In an extensive empirical evaluation, we demonstrate the feasibility of our approach, with all code and data available at https://github.com/amazon-science/unsat-proof-skeletons.

Beyond being a tool for compression, proof skeletons can also serve as a source of insight into a solver's reasoning. Getting any sort of intuition from a million-line proof is difficult; by computing a skeleton, we obtain a small set of facts—logically implied by the problem—that can give us an idea of how a solver established the unsatisfiability of a formula. This can lead to a feedback loop that improves solver performance. For example, when inspecting skeletons for some bounded-model-checking benchmarks, we observed many unit clauses and binary clauses of a certain type. From this, we hypothesized that the problems required more preprocessing, which did indeed improve performance.

Our main contributions are as follows: (1) We present a semantic approach for proof compression that selects only the most important clauses of a proof. (2) We implemented an online version and an offline version of our approach on top of the SAT solver CaDiCaL and the proof checker DRAT-trim. (3) In an extensive empirical evaluation, we demonstrate that our approach can drastically reduce proof size while still enabling efficient proof reconstruction.

The rest of this paper is structured as follows. In Section 2, we discuss background required to understand our paper and review related work. In Section 3,

we outline the main idea behind our proof-compression approach. In Section 4, we show multiple ways to create proof skeletons, and in Section 5 we show how to reconstruct full proofs from skeletons. Finally, in Section 6, we present an empirical evaluation of our approach before concluding in Section 7.

2 Background and Related Work

The Boolean satisfiability problem (SAT) takes as input a formula of propositional logic and asks if there exists a truth assignment under which the formula evaluates to true. As is common in SAT solving, we consider propositional formulas in *conjunctive normal form* (CNF), which are defined as follows. A *literal* is either a variable x (a *positive literal*) or the negation $\bar{x}$ of a variable x (a *negative literal*). The *complement* $\bar{l}$ of a literal l is defined as $\bar{l} = \bar{x}$ if $l = x$ and as $\bar{l} = x$ if $l = \bar{x}$. For a literal l, we denote the variable of l by $var(l)$. A *clause* is a finite disjunction of the form $(l_1 \vee \cdots \vee l_n)$, where $l_1, \ldots, l_n$ are literals. Clauses with only one literal are called *unit clauses* and clauses with two literals are called *binary clauses*. We denote the empty clause by $\bot$. A *formula* is a finite conjunction of the form $C_1 \wedge \cdots \wedge C_m$, where $C_1, \ldots, C_m$ are clauses. For example, $(x \vee \bar{y}) \wedge (z) \wedge (\bar{x} \vee \bar{z})$ is a formula consisting of the clauses $(x \vee \bar{y})$, (z), and $(\bar{x} \vee \bar{z})$.

A *truth assignment* (or *assignment* for short) is a function from a set of variables to the truth values 1 (*true*) and 0 (*false*). A literal l is *satisfied* by an assignment α if l is positive and $\alpha(var(l)) = 1$ or if l is negative and $\alpha(var(l)) = 0$. A literal l is *falsified* by an assignment if its complement $\bar{l}$ is satisfied by the assignment. A clause C is satisfied by an assignment α if α satisfies at least one of C's literals. A formula ψ is satisfied by an assignment α if α satisfies all of ψ's clauses. A formula is *satisfiable* if there exists an assignment that satisfies it, otherwise it is *unsatisfiable*. A clause $C = (l_1 \vee \cdots \vee l_k)$ is *implied* by a formula ψ, denoted by $\psi \models C$, if all satisfying assignments of ψ satisfy C, or equivalently, if $\psi \wedge \bar{C}$ is unsatisfiable, where $\bar{C} = (\bar{l}_1) \wedge \cdots \wedge (\bar{l}_k)$. In case a formula is satisfiable, modern solvers can output a satisfying assignment; in case the formula is unsatisfiable, most solvers can output a proof of unsatisfiability.

Proofs of Unsatisfiability. State-of-the-art SAT solvers produce so-called *clausal proofs*. Intuitively, a clausal proof is a list of clause additions and clause deletions. Formally, a clausal proof is a list of pairs $\langle s_1, C_1 \rangle, \ldots, \langle s_m, C_m \rangle$, where for each $i \in 1, \ldots, m$, $s_i \in \{\mathsf{a}, \mathsf{d}\}$ and C_i is a clause. If $s_i = \mathsf{a}$, the pair is called an *addition*, and if $s_i = \mathsf{d}$, it is called a *deletion*. For a given input formula ψ_0, a clausal proof gives rise to *accumulated formulas* ψ_i $(i \in 1, \ldots, m)$ as follows:

$$\psi_i = \begin{cases} \psi_{i-1} \cup \{C_i\} & \text{if } s_i = \mathsf{a} \\ \psi_{i-1} \setminus \{C_i\} & \text{if } s_i = \mathsf{d} \end{cases}$$

The clauses of an accumulated formula ψ_i are also called the *active clauses* at point i. Clause additions must preserve satisfiability, which is usually guaranteed by requiring the added clauses to fulfill some efficiently decidable syntactic

criterion that itself implies satisfiability is preserved. Deletions are unrestricted and are not useful for proving unsatisfiability as they only make a formula "more satisfiable"; their main purpose is to speed up proof checking by keeping the set of active clauses small. A valid proof of unsatisfiability must end with the addition of the empty clause. As the empty clause is trivially unsatisfiable, and since all proof steps preserve satisfiability, the unsatisfiability of the original formula can then be concluded.

Clausal proof systems are distinguished by the syntactic criterion they impose on clause additions. The standard SAT solving paradigm *conflict-driven clause learning* (CDCL) [15,16] adds so-called *RUP* (short for *reverse unit propagation*) clauses [20], whose definition is based on the notion of *unit propagation*. Unit propagation is the process of repeatedly applying the *unit-clause rule* to a formula until no unit clauses are left. Given a formula ψ, the unit-clause rule takes a unit clause (l) and makes its literal l true, meaning that (1) all clauses that contain l are removed from ψ, and (2) the negation $\bar{l}$ of l is removed from all remaining clauses. If unit propagation produces the empty clause, we say it derived a *conflict*. For example, unit propagation derives a conflict on $(x) \wedge (\bar{x} \vee y) \wedge (\bar{x} \vee \bar{y})$ as the application of the unit-clause rule for (x) produces the formula $(y) \wedge (\bar{y})$, on which another application of the unit-clause rule, with either of (y) or $(\bar{y})$, produces the empty clause. If unit propagation derives a conflict on a formula, the formula is clearly unsatisfiable, but not vice versa.

A clause $C = (l_1 \vee \cdots \vee l_k)$ is a RUP for a formula ψ if unit propagation derives a conflict on $\psi \wedge \bar{C}$. If C is a RUP for ψ, it is implied by ψ since $\psi \wedge \bar{C}$ is unsatisfiable; we thus sometimes write $\psi \vdash_1 C$ to denote that C is a RUP for ψ. The clausal proof system allowing the addition of RUP clauses together with deletions is called DRUP. Solvers participating in the SAT competition must produce DRAT proofs, but since each DRUP proof is also a DRAT proof (but not vice versa) and since all state-of-the-art solvers actually produce DRUP proofs by default, we restrict this study of proof compression to DRUP proofs.

A *proof checker* is an independent tool that verifies the correctness of proofs. There exist formally verified proof checkers that provide strong correctness guarantees (c.f., [5, 9, 14, 19]). Because these tools are inefficient, proofs are often passed through an—efficient but unverified—intermediary proof checker (such as DRAT-TRIM [22]) that transforms a DRAT proof into a so-called *LRAT proof* [5]. The resulting LRAT proof includes additional information (called *hints*), which allows a formally verified checker to efficiently check the proof.

3 Problem Overview

We want to compress proofs into small representations that can be efficiently decompressed into full proofs. Existing techniques for SAT and SMT focus on transformations and substitutions that preserve validity to generate smaller proofs [4,18,21]. We achieve greater compression by storing only a so-called *proof skeleton*, which itself is not a valid proof.

Tools like Sledgehammer [3] automatically solve proof obligations from interactive theorem provers, filling gaps in the proof by translating lower-level reasoning into the theorem provers' logic. More recent work proposed a method for constructing proofs for complex SMT rewriting steps on demand in a post-processing step [17]. In a similar way, we use proof skeletons to efficiently reconstruct valid proofs that can differ from the original proofs.

Suppose you solved an unsatisfiable CNF formula ψ, and out of the many facts you learned during solving, there were three facts A, B, and C, which you deem particularly important for showing the unsatisfiability of ψ. You can then build a proof skeleton from A, B, and C. Later, you can rephrase the question $\psi \models \bot$ (*"does ψ imply the empty clause?"*, or equivalently, *"is ψ unsatisfiable?"*) into the following questions:

$$\psi \models A \qquad \psi \wedge A \models B \qquad \psi \wedge A \wedge B \models C \qquad \psi \wedge A \wedge B \wedge C \models \bot$$

Not only do A, B, and C provide a way to partition the proof effort, when ordered carefully, they can be used as assumptions in subsequent questions. Each question can be submitted to a solver independently, and combining the four resulting proofs will give a proof of the original claim that ψ is unsatisfiable.

Our work translates this general schema to the realm of SAT by (1) determining which learned clauses from a SAT solver are most useful and should be stored in a proof skeleton; (2) carefully grouping solver calls to prevent repeated work when producing partial proofs from a proof skeleton; and (3) stitching the partial proofs together to generate a complete proof.

Determining which clauses are stored in a proof skeleton. We co-opt the clause-importance metrics used by CDCL solvers. We give a brief overview of these metrics in the following. CDCL solvers make progress by continuously learning new clauses that help them prune the search space of possible truth assignments. To limit memory usage, they occasionally perform a clause database *reduction*, removing a large portion of learned clauses based on some usefulness heuristics. Most solvers keep clauses that are short, have low *glue* value, are *reason clauses*, or have been used recently. The glue of a clause (also known as its *literal block distance*, or LBD) is a positive integer that estimates the usefulness of a clause. Intuitively, a low glue value means that few decisions are required to falsify the clause, which is considered good. For a more extensive discussion of glue, we refer to the respective literature [1]. A *reason clause* is a clause that was used by the solver when performing unit propagation, meaning that the clause became a unit clause under a partial assignment. The number of times a reason clause is *used* during conflict analysis is considered the clause's *activity*.

Grouping solver calls for partial proofs. We leverage incremental SAT to construct partial proofs. An incremental SAT solver solves a problem with several related steps, with the solver retaining state (e.g., learned clauses and heuristics) between steps; it also allows solving under so-called *assumptions*, which are

literals assumed to be true in a step. Solving a sequence of related steps incrementally is often much faster than solving them independently of each other (for more details on incremental SAT see, e.g., [6]).

Given a formula ψ and a sequence $C_1, \ldots, C_n$ of clauses, we want to produce a DRUP proof of $\psi \models C_i$ for each $i \in 1, \ldots, n$. We use an incremental solver to produce partial proofs, with each solving step corresponding to a clause C_i. For the first step, $\psi \models C_1$, we pass the assumptions $\bar{C}_1 = \bar{l}_1 \wedge \cdots \wedge \bar{l}_k$ to the incremental solver. Given the formula ψ, the solver assigns the literals in the assumptions, then runs the CDCL algorithm until it derives the empty clause. During solving, CDCL guarantees that all learned clauses are RUPs for the input formula ψ. Let ϕ_1 denote the sequence of clauses learned by the solver. Then, since unit propagation under the assumptions $\bar{l}_1 \wedge \cdots \wedge \bar{l}_k$ derived the empty clause, C_1 is by definition a RUP for $\psi \wedge \phi_1$. This means that C_1 can be appended to the corresponding proof of the solver (which derives all clauses in ϕ_1) to obtain a valid DRUP derivation of C_1 from ψ.

In the next step, the clause C_2 is handled similarly, except the solver retains the learned clauses $\phi_1 \wedge C_1$ when proving that C_2 is a RUP clause. This continues until all $n + 1$ steps corresponding to the n clauses of the proof skeleton are completed (step $n + 1$ corresponds to the derivation of the empty clause).

To parallelize this reasoning, we use an approach akin to *divide-and-conquer* techniques established in parallel SAT solving [13]. Divide-and-conquer solvers first partition a problem into multiple subproblems and then solve the subproblems in parallel. Similarly, we divide the incremental solver steps into so-called *chunks*, which are independent groups of subsequent solver steps. For example, we can split the solver steps into one chunk containing the first half of steps and another chunk containing the second half of steps. Both chunks can then be solved in parallel by two independent incremental SAT solvers.

Stitching partial proofs together. Once we have partial proofs for all $n+1$ solving steps, a full proof of unsatisfiability can be constructed as the sequence of clause additions arising from $\phi_1, C_1, \phi_2, C_2, \ldots, C_n, \phi_{n+1}, \bot$, where ϕ_i is the sequence of learned clauses by the i-th solver step, as explained above. In general, clauses are added and deleted during solving, so the proof can be augmented with the deletion information contained in the proofs emitted by a solver. But, we need to ensure clauses are not deleted in the proof and then implicitly reintroduced into a solver, which can occur when inprocessing techniques touch variables in the assumptions. We use *variable freezing* [7] to freeze all variables occurring in $C_1, \ldots, C_n$; this avoids any unsound inprocessing [8], and is required to ensure correctness of the proofs.

4 Creating Proof Skeletons

Given a clausal proof $P = \langle s_1, C_1 \rangle, \ldots, \langle s_m, C_m \rangle$, we define a proof skeleton of P to be a sequence of clauses obtained from clause additions in P. Ideally, a skeleton is small but contains enough useful clauses to guide reasoning during proof

reconstruction. A proof skeleton can be constructed *online*, during the solver's execution, by applying a filter to clauses as they are traced to a proof. Alternatively, a proof skeleton can be constructed *offline*, after solving, by processing the full proof and selecting important clauses.

4.1 Online Generation of Proof Skeletons

We create proof skeletons online by filtering clause additions as the solver traces them to a proof. Clauses that pass a usefulness threshold are added to the skeleton. As mentioned earlier, the filter applies usefulness heuristics from CDCL including *glue* and *clause activity*. Additionally, at certain intervals we add *reason clauses* to the skeleton. We implemented the filter within the solver CaDiCaL, giving us access to these values as well as to the reason clauses (through the *trail* of assignments). We also enabled logging, giving every clause a unique identifier, in order to sort the skeletons. We evaluate three different configurations:

- Glue: Clauses with glue lower than 3.
- Glue+Trail: Clauses with glue lower than 3, and all reason clauses on the trail before each clause-database reduction.
- Dynamic: Clauses with glue lower than some dynamically adjusted threshold $glue_d$, and all reason clauses on the trail every 50,000 learned clauses.

The first two configurations combine low-glue clauses with either no or some reason clauses. Increasing the glue value threshold often led to a compression of less than 1,000 times and slower reconstruction. Reason clauses are important because they are actively used by the solver whereas for low-glue clauses this is not guaranteed (although low glue is associated with high usage in general). Clause-database reductions are sparse, so reason clauses (which are added only during these reductions) will be added infrequently. We evaluate the impact of including reason clauses in the skeletons in Section 6.3.

In the first two configurations, all clauses passing the filter are accepted into the skeleton. For some formulas, a solver will produce many low-glue clauses and the skeleton will become too large, and for others too few low-glue clauses will lead to a small skeleton. Our third configuration accounts for the differences between formulas by adjusting heuristics dynamically to meet a desired compression ratio. The heuristics are updated based on the number of clauses added to the skeleton within some number of conflicts, denoted as $window_c$. For a compression ratio between 500 and 1,000, and a $window_c$ value of 5,000, we tuned the Dynamic configuration in the following way: every 5,000 conflicts, if more than 25 ($window_c$/200) lemmas passed the filter, the $glue_d$ value is decreased, and if less than 3 lemmas ($window_c$/2,000) passed the filter, the $glue_d$ value is increased. Reasons from the trail are added every 50,000 conflicts ($window_c \times 10$).

For configurations using reason clauses, the unique clause IDs are used to sort the skeleton. This is necessary because reason clauses are traced during reductions, so they may initially appear in the skeleton long after they were learned by the solver. During proof reconstruction it is important that clauses appear in the skeleton in an order that corresponds with a solver's reasoning.

We implemented additional configurations using clause activities. For this, we incremented an *activity* field for each clause every time it was used during conflict analysis. An evaluation of these additional configurations is beyond the scope of this paper, but data can be found in the paper's repository.

4.2 Offline Generation of Proof Skeletons

We create proof skeletons offline by processing a full proof and selecting the most active clauses. Given a DRAT proof, the tool DRAT-trim uses backwards checking to generate an optimized LRAT proof and, optionally, an UNSAT core (i.e., an unsatisfiable subset of the original formula). From the LRAT proof, we can estimate a clause's *activity* by counting the number of times the clause appears in a hint of a clause-addition step. We then add the clauses with the highest activity to the skeleton until a target compression ratio is met. We found for most problems the target 1,000 provided optimal reconstruction performance. We sort the skeleton by each clause's first use as a hint in the LRAT proof, signifying when a clause is actually used as opposed to when it is learned. We evaluate three configurations for offline generation:

- Offline: Select 1,000 times fewer clauses than in the original DRAT proof.
- Offline+Units: Additionally include all unit clauses from the proof.
- Offline-Opt: Select 1,000 times fewer clauses than in the optimized LRAT proof.

The motivation for Offline-Opt is that some optimized LRAT proofs have significantly fewer clauses than the DRAT proofs, resulting from many unused lemmas, which suggests that stronger compression is possible.

Offline construction requires expensive post-processing with DRAT-trim. However, during online construction we can only *guess* the future usefulness of clauses when they are derived, by relying on heuristics such as glue, but we cannot know how often a clause will actually be used. For instance, it may be that a clause has low glue (predicting high usefulness) but is learned and then never used in the rest of the proof, making it worthless in the skeleton. In contrast, when constructing a skeleton offline—after solving—we know already how often the clause was actually used in reasoning throughout the proof, and whether it was used to derive the empty clause. Also, we can use the UNSAT core instead of the original formula when reconstructing a proof for the original problem.

5 Reconstructing Proofs from Skeletons

We reconstruct proofs by filling the gaps of a proof skeleton with a SAT solver. Once we have proofs for all gaps, we stitch them together with the clauses of the skeleton to create a complete proof. We can utilize information obtained during proof reconstruction to further shrink skeletons by removing less useful clauses. Finally, we can also use a skeleton to create an optimized LRAT proof.

Proof	Skeleton	Reconstruction	Incremental Reconstruction
C_1	C_2	$\psi \models C_2$	$\psi \models C_2 : \phi_1$
C_2	C_5	C_2	C_2
C_3	$\vdots$	$\psi \wedge C_2 \models C_5$	$\psi \wedge C_2 \wedge \phi_1 \models C_5 : \phi_2$
C_4		C_5	C_5
C_5		$\vdots$	$\vdots$
$\vdots$		$\psi \wedge Ske\ eton \models \bot$	$\psi \wedge Ske\ eton \wedge \phi \models \bot$

Fig. 1. Proof reconstruction from a proof skeleton and a formula ϕ by filling in the gaps between skeleton clauses. This can be done with independent SAT calls or with an incremental SAT solver that keeps learned clauses (ϕ_i) between steps.

5.1 Filling Skeletons Using Incremental Solvers

We consider two ways of filling a proof skeleton's gaps—*reconstruction* and *incremental reconstruction*; both are illustrated in Fig. 1. Given a formula ϕ and a skeleton $C_1, \ldots, C_n$, reconstruction fills each gap $\psi \wedge C_1 \wedge \cdots \wedge C_{i-1} \models C_i$ using independent SAT solver calls, with $\psi_1 \wedge C_1 \wedge \cdots \wedge C_n \models \bot$ as the final call. Filling a gap for $C_i = (l_1 \vee \cdots \vee l_k)$ involves assuming $\bar{l}_1 \wedge \cdots \wedge \bar{l}_k$ and deriving the empty clause with proof ϕ, which proves that C_i is a RUP for $\psi \wedge C_1 \wedge \cdots \wedge C_{i-1} \wedge \phi$. Each gap has an associated DRUP proof ϕ_i emitted by the solver. Since RUP is a monotonic property, the clauses added in ϕ_i will not affect the validity of ϕ_j for $i < j$. However, clause deletions could make the proof $\phi_1, \langle \mathsf{a}, C_1 \rangle, \phi_2, \langle \mathsf{a}, C_2 \rangle, \ldots, \langle \mathsf{a}, C_n \rangle, \phi_{n+1}, \bot$ incorrect. For example, if a skeleton clause C_1 is deleted in ϕ_2, then ϕ_3 (stemming from $\psi \wedge C_1 \wedge C_2 \models C_3$) may use C_2—a clause already deleted in the proof. The same problem could occur if formula clauses are deleted. Therefore, we must remove any deletion steps for clauses of the skeleton or of the formula clauses from each ϕ_i.

The second approach, *incremental reconstruction*, uses an incremental SAT solver, which allows the use of learned clauses when filling subsequent gaps. Specifically, we create an incremental problem with the steps $assume(\bar{C}_1)$, …, $assume(\bar{C}_n)$, $assume(\emptyset)$, where each step $assume(\bar{C}_i)$, with $C_i = (l_1 \vee \cdots \vee l_k)$, involves assuming $\bar{l}_1 \wedge \cdots \wedge \bar{l}_k$ and deriving the empty clause. Each step produces a proof ϕ_i, and the complete proof $\phi_1, \langle \mathsf{a}, C_1 \rangle, \phi_2, \langle \mathsf{a}, C_2 \rangle, \ldots, \langle \mathsf{a}, C_n \rangle, \phi_{n+1}, \langle \mathsf{a}, \bot \rangle$ is correct as long as variables occurring in skeleton clauses are frozen (as described in Section 3). With this approach, we no longer need to worry about deletions of skeleton clauses or formula clauses because the solver fills each gap using the current clause database, i.e., each gap is proved without clauses formerly deleted by the solver.

To parallelize incremental reconstruction, we partition the incremental problem into several independent incremental problems, which we call *chunks*. We assign k clauses $C_l, \ldots, C_{l+k-1}$ from the skeleton to each chunk, and we then use an incremental solver to compute partial proofs for each of the clauses, starting

from the formula $\psi \wedge C_1 \wedge \cdots \wedge C_{l-1}$. For each partial proof corresponding to a clause C_i, we call the solver with the assumptions negating the clause, i.e., with $assume(\bar{C}_i)$. Again, we must remove any deletion steps of skeleton clauses or formula clauses since they may be used in later chunks. All added clauses are then RUPs, and so the concatenation of chunk proofs is a complete proof.

Each chunk can be solved independently in parallel. The more skeleton clauses in each chunk, the more clauses the incremental solver can learn and reuse in subsequent steps. However, gaps might differ in hardness, meaning that some gaps can be filled quickly while others require a significant amount of solving time. A chunk can thus become a bottleneck during parallelization if it includes many difficult gaps. In our evaluation, we partitioned the skeleton into chunks of equal size, one for each core. For instance, on a single core, one incremental problem spanning the entire skeleton was given to a solver instance whereas for 24 cores, the skeleton was partitioned into 24 chunks. In principle, we could partition a skeleton into more chunks than cores, but this would require an intermediary level of problem scheduling that we leave for future work.

5.2 Shrinking Skeletons

The runtimes for filling each gap of a proof skeleton could provide insight into the usefulness of the skeleton clauses. For example, if the solver can quickly fill a gap, the corresponding skeleton clause may be trivially implied, and if the solver takes long, the clause may be useful since its derivation requires a lot of reasoning. Alternatively, the difference in runtime might not be explained by clause usefulness. Take, for example, the two gaps $\psi \models C_2$ and $\psi \wedge C_2 \models C_5$ from Fig. 1, and assume that the solver fills the first gap in a millisecond and the second gap in ten seconds. If the difference is a result of C_2 being trivially implied, it makes sense to remove C_2 from the skeleton; otherwise, if the difference is due to factors unrelated to usefulness, it is better to remove C_5. Based on this observation, we try to shrink a given skeleton by sorting gap reconstruction times and removing a certain share of the slowest or fastest clauses.

Our empirical evaluation in Section 6 indicates that removing the fastest clauses is the right approach for improving compression and (sometimes) reducing reconstruction time. Even though gap runtime and clause usefulness are correlated, the correlation is not perfect. For instance, sometimes the incremental solver is able to quickly fill a gap because of learning from previous steps of the incremental problem. Even if it takes a long time to fill a gap, there is no guarantee that the corresponding skeleton clause is useful for filling future gaps. We examine in detail how shrinking skeletons affects reconstruction time.

5.3 Reconstructing LRAT Proofs from Skeletons

The proof reconstruction described above will produce DRAT proofs. Formally verified checkers typically require LRAT proofs, forcing a conversion via a proof checker such as DRAT-trim, which can take much longer than the original

solving time. Instead, we can reconstruct DRAT proofs for each chunk, then convert the DRAT proofs to LRAT in parallel, and finally concatenate them.

We use DRAT-trim to convert chunk DRAT proofs to LRAT. This required us to modify DRAT-trim (e.g., by changing the way it performs backwards checking, and how it handles unit clauses). By default, DRAT-trim starts backwards checking at the empty clause. But, only the last chunk will derive the empty clause, and further, we must ensure all skeleton clauses are included in the backwards check, as they may be used in later chunks. To account for this, we mark each skeleton clause in the DRAT proof before performing the backwards check. The backwards check verifies that each marked clause is RAT (or RUP, in our case), including the clauses in the LRAT proof. When combining the chunk LRAT proofs, we map the skeleton clauses in each chunk to the index of the LRAT step where they were initially added. Finally, we remove all deletions from the LRAT proof, but this will not affect proof-checking time, mainly since LRAT checkers perform unit propagation in linear time using hints. While the following evaluation focuses on DRAT proof reconstruction from skeletons, we tested our implementation of parallel LRAT proof reconstruction on 24 cores, and verified several proofs with Cake-LPR [19].

6 Experimental Evaluation

We evaluated our approach on SAT competition 2021 Main Track benchmarks, using all (65) unsatisfiable formulas that were solved between 500 and 5,000 seconds by the solver CaDiCaL [2]. By requiring at least 500 seconds of solving time, we ensured that proofs are of reasonable size (around 1 GB) and therefore good candidates for compression. We ran experiments on an AWS EC2 m5d.metal instance, with 96 virtual CPUs and 500 GB of memory, running at most 24 parallel processes at a time. We used a timeout of 5,000 seconds for solving a problem and constructing a DRAT proof. For proof reconstruction on a single core we used a single incremental problem spanning the entire skeleton. For proof reconstruction on 24 cores, we evenly divided the proof skeleton into 24 incremental problems (chunks) passed to 24 instances of CaDiCaL. We report real time for proof reconstruction, not including skeleton extraction.

6.1 Single-Core Proof Reconstruction

Fig. 2 shows the best configurations on each formula using online skeletons (left) and offline skeletons (right), for the single-core experiments (i.e., the entire skeleton on a single core). Almost all proofs were reconstructed faster than the original solving time (below the red dotted line), and in some cases more than five times faster (below the blue dotted line). Each configuration was the best for some formulas. The Glue configuration led the online skeletons. With a single incremental problem, learned clauses from earlier incremental calls can be kept for the entire execution, meaning that clauses that occur later in large skeletons (e.g., Glue+Trail) may be trivially implied by previously learned clauses.

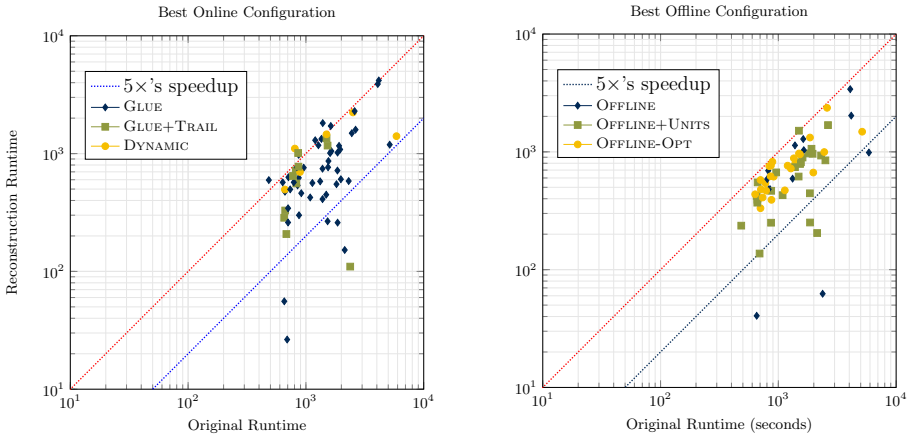


Fig. 2. Runtimes (in seconds) of best online (left) and offline (right) configurations for proof reconstruction using a proof skeleton and a single core.

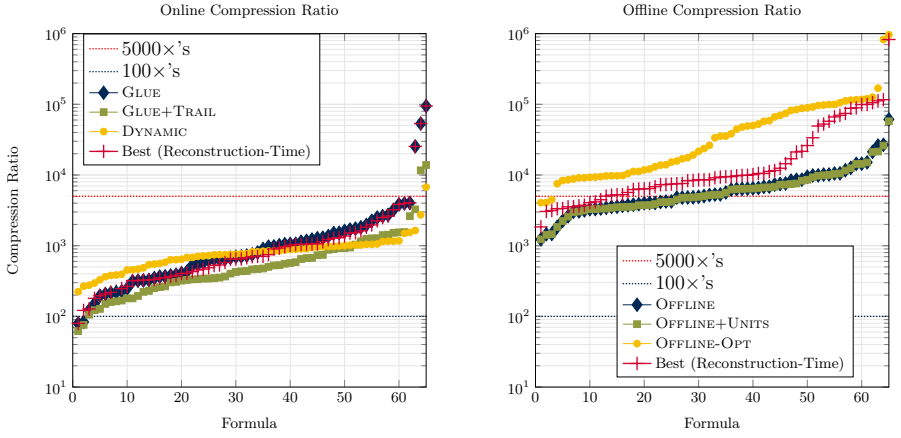


Fig. 3. Proof skeleton compression ratio for online (left) and offline (right).

6.2 Skeleton Compression Ratio

Fig. 3 shows the sorted compression ratios (w.r.t. file size) between proof skeletons and the original DRAT proofs for each configuration as well as the compression ratios for the configuration with the fastest reconstruction time on each formula (Best). For online configurations (left), the Dynamic skeletons have the most consistent compression ratios, with a tradeoff in reconstruction times. In some cases, skeletons can have higher compression (10,000 times) without a loss in performance, witnessed by the right-hand-side tail of the plot.

For offline configurations (right), Offline selects 1/1,000 of the clauses from the original DRAT proof. The ratios are much greater than 1,000 because skele-

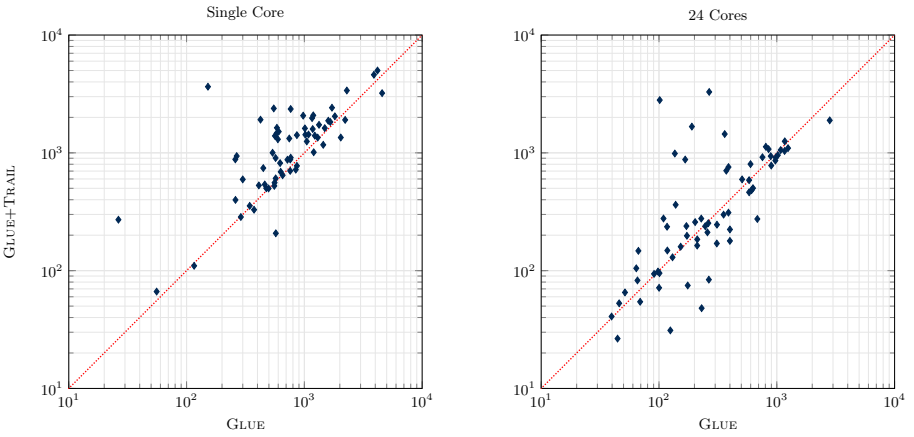


Fig. 4. Runtimes (in seconds) for proof reconstruction of multiple online configurations with a single core (left) and 24 cores (right).

tons have no deletion information and the most active clauses are typically much shorter than the average clause. Offline-Opt provides around a factor 10 more compression, and these smaller skeletons provide faster reconstruction for about half of the formulas. In general, the compression is much better when using *clause activity* as a measure for clause importance as opposed to online heuristics (such as *glue*), with similar reconstruction times seen in Fig. 2.

6.3 Impact of Reason Clauses in Online Skeletons

Fig. 4 shows a comparison of reconstruction times between the Glue and the Glue+Trail online configurations, both on a single core (left) and on 24 cores (right). On a single core, creating skeletons with only low-glue clauses performs better than creating skeletons with low-glue clauses *and* reasons from the trail. On multiple cores, however, the reason clauses are beneficial for many reconstructions. This may be because for parallel reconstruction, each individual chunk only has access to lemmas earlier in the skeleton during solving. Therefore, having more clauses in the skeleton will aid the later chunks. In contrast, for a single chunk on one core, learned clauses are kept throughout solving, and these learned clauses supplement the smaller skeletons.

6.4 Impact of the UNSAT Core on Offline Skeletons

Fig. 5 shows the effect of using an UNSAT core during reconstruction for offline skeletons on a single core (left) and on 24 cores (right). For the experiments using an UNSAT core, we remove formula clauses that are not in the UNSAT core before passing the formula to the solver during the incremental SAT call for the chunk proof. Using the UNSAT core greatly improves performance during

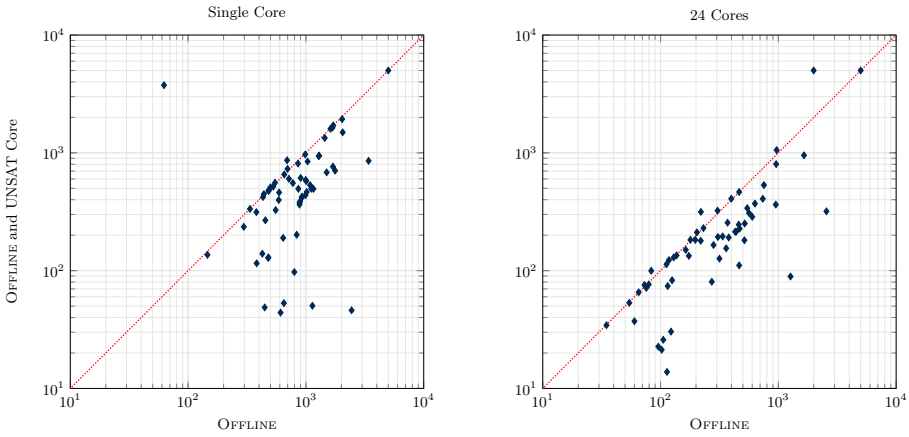


Fig. 5. Runtimes (in seconds) for Offline proof reconstruction with and without an UNSAT core with a single core (left) and 24 cores (right).

reconstruction on a single core. This may be because the skeleton is built from reasoning based on the UNSAT core, so focusing the solver on these specific formula clauses makes filling the gaps in the skeleton easier. The UNSAT core is useful in parallel reconstruction as well, producing the overall best configuration between online and offline skeletons. To give an idea, it takes approximately 125 KB to store an UNSAT core as a bit vector (each bit indicating whether or not a clause is part of the core) for a formula with one million clauses. For most formulas, this data would be dominated by the size of the proof skeleton.

6.5 Skeleton Shrinking after Reconstruction

We discussed in Section 5.2 that it might make sense to shrink a skeleton by removing some amount of the fastest or of the slowest skeleton clauses. Fig. 6 shows results for reconstruction on 24 cores using the online skeleton, removing either the fastest 90% or the slowest 10% of clauses. To perform the shrinking, we performed proof reconstruction from the skeleton and measured the solve times for the incremental calls, with each call corresponding to a skeleton clause. Removing the fastest 90% has a small impact on reconstruction time, performing slower for the majority of formulas. In some cases, shrinking the skeleton even improves performance because redundant or unnecessary clauses are removed from the skeleton. Removing the slowest solved clauses causes a wider variation in reconstruction time. This might be because these clauses are important for guiding the solver during reconstruction, and sometimes they lead the solver into unprofitable search regions that waste time. This shows two things: (1) For some formulas, removing only a fraction of clauses from the skeleton can lead to a big or small improvement, and (2) skeleton clauses are mostly nontrivial and cannot be added or removed randomly without a potentially consequential impact.

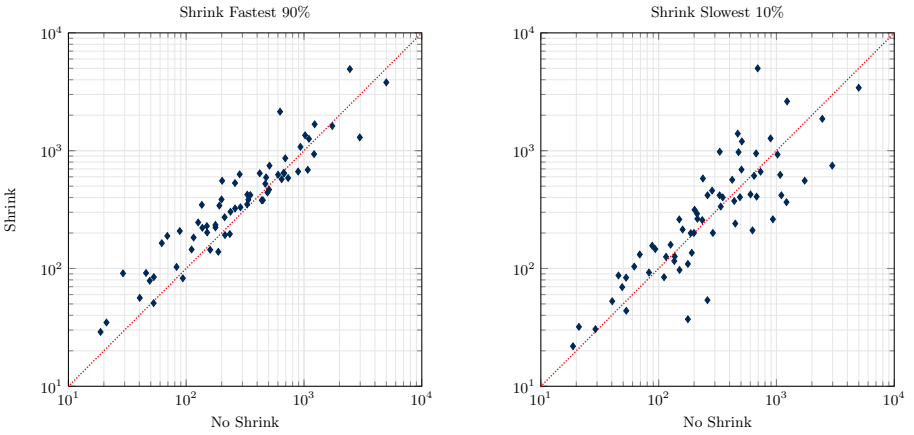


Fig. 6. Runtimes (in seconds) of proof reconstruction on 24 cores after skeleton shrinking for the DYNAMIC online configuration, removing the fastest 90% (left) or the slowest 10% (right) of clauses from the skeleton.

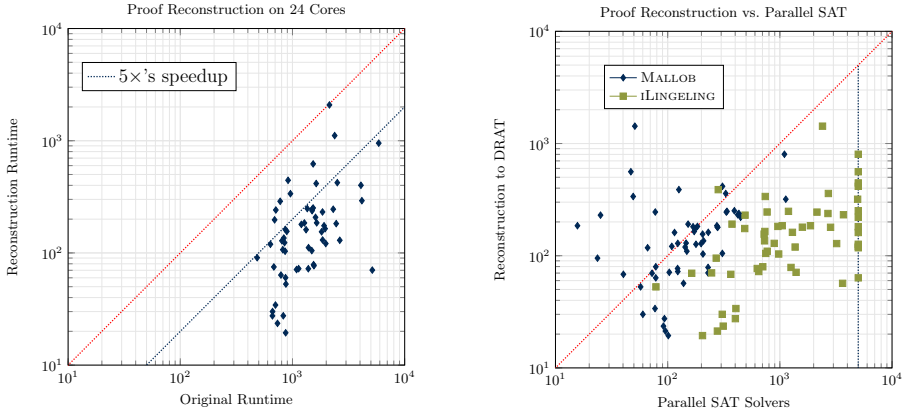


Fig. 7. Left: Runtimes (in seconds) of original solver on a single core against proof reconstruction on 24 cores with the best offline-skeleton configurations OFFLINE+UNITS using UNSAT cores. Right: Runtimes (in seconds) of parallel SAT solvers MALLOB and LINGELING without proof logging against proof reconstruction with the best offline skeleton configurations using an UNSAT core, each using 24 cores.

6.6 Comparison With Sequential and Parallel SAT Solvers

Alternatives to our proof reconstruction could be to compute a proof on demand by solving a formula from scratch (either with a sequential or with a parallel SAT solver) or to run a parallel incremental solver that fills the gaps of a skeleton.

The left plot of Fig. 7 shows the difference between running a sequential solver on a single core versus running our parallel proof reconstruction on 24 cores. For

the majority of formulas, parallel proof reconstruction is over five times faster, and in some cases closer to ten times faster. One formula had little improvement for reconstruction (on the red dotted line). For this formula, the final chunk took around 2,000 seconds to solve, and the next slowest chunk took only 24 seconds, meaning the hardest gaps were all clustered in the final chunk. For these sorts of problems, a smaller chunk size could break up the hard gaps, therefore improving utilization across cores and reducing the reconstruction time.

To our knowledge, there exist no portfolio solvers or parallel incremental solvers that produce proofs. However, it might be possible to add proof support to solvers like Mallob (a clause-sharing portfolio solver) or iLingeling (a parallel incremental solver); we thus compare our approach to these solvers in the right plot of Fig. 7.

The comparison to Mallob suggests that some form of clause sharing between solvers that solve independent chunks may improve performance. This could be achieved with *forward clause sharing*, where learned clauses can only be sent to solvers running on subsequent chunks. Also, Mallob has full core utilization by running each solver until one derives the empty clause, but our proof reconstruction does not since some chunks take longer than others. With smaller chunk sizes and good scheduling, proof reconstruction could get closer to full utilization.

iLingeling, which is based on Lingeling [2], takes an incremental problem and greedily assigns steps to solver instances, terminating when one instance derives the empty clause. There is no clause sharing between solvers. We ran iLingeling using the incremental problem derived from the proof skeleton. In proof reconstruction, chunks can use skeleton clauses from previous chunks, leading to consistently better performance than iLingeling.

7 Conclusion

We presented a semantic approach for compressing propositional proofs by selecting important clauses that summarize the reasoning of a solver. We store these clauses in a so-called proof skeleton, from which we can reconstruct a complete proof in parallel by performing multiple incremental SAT solver calls. We implemented our approach on top of the SAT solver CaDiCaL and the proof checker DRAT-trim. In an empirical evaluation, we showed that our approach can produce skeletons that are 100 to 5,000 times smaller than the original proofs. On a single core, almost all proofs were reconstructed faster than the original solving time, and when using 24 cores, the majority of proofs was reconstructed around five times faster. This is significant since proof checking typically takes longer than solving, and since existing parallel solvers cannot produce proofs while maintaining strong performance. We observed that proof skeletons not only serve as a compression mechanism but also provide insight into a problem. In future work, we thus plan to explore the connection between skeletons, proofs, and solver performance.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 399–404 (2009), http://ijcai.org/Proceedings/09/Papers/074.pdf
2. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
3. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. J. Autom. Reason. **51**(1), 109–128 (2013)
4. Boudou, J., Fellner, A., Paleo, B.W.: Skeptik: A proof compression system. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8562, pp. 374–380. Springer (2014), https://doi.org/10.1007/978-3-319-08587-6_29
5. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 220–236. Springer (2017), https://doi.org/10.1007/978-3-319-63046-5_14
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003), https://doi.org/10.1007/978-3-540-24605-3_37
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003), https://doi.org/10.1016/S1571-0661(05)82542-3
8. Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 136–154. Springer (2019), https://doi.org/10.1007/978-3-030-24258-9_9
9. Heule, M., Jr., W.A.H., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10499, pp. 269–284. Springer (2017), https://doi.org/10.1007/978-3-319-66107-0_18
10. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **abs/1610.06229** (2016), http://arxiv.org/abs/1610.06229
11. Heule, M.J.H.: Schur number five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18). pp. 6598–6606. AAAI Press (2018), https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16952