Constantin Enea
Akash Lal (Eds.)

LNCS 13965

Computer Aided Verification

35th International Conference, CAV 2023
Paris, France, July 17–22, 2023
Proceedings, Part II

2 Part II

Springer

Lecture Notes in Computer Science 13965

The series Lecture Notes in Computer Science (LNCS), including its subseries Lecture Notes in Artificial Intelligence (LNAI) and Lecture Notes in Bioinformatics (LNBI), has established itself as a medium for the publication of new developments in computer science and information technology research, teaching, and education.

LNCS enjoys close cooperation with the computer science R & D community, the series counts many renowned academics among its volume editors and paper authors, and collaborates with prestigious societies. Its mission is to serve this international community by providing an invaluable service, mainly focused on the publication of conference and workshop proceedings and postproceedings. LNCS commenced publication in 1973.

Constantin Enea · Akash Lal
Editors

Computer Aided Verification

35th International Conference, CAV 2023
Paris, France, July 17–22, 2023
Proceedings, Part II

Springer

Editors

Constantin Enea
LIX, Ecole Polytechnique, CNRS and Institut
Polytechnique de Paris
Palaiseau, France

Akash Lal
Microsoft Research
Bangalore, India

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-37702-0 ISBN 978-3-031-37703-7 (eBook)
https://doi.org/10.1007/978-3-031-37703-7

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

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

Preface

It was our privilege to serve as the program chairs for CAV 2023, the 35th International Conference on Computer-Aided Verification. CAV 2023 was held during July 19–22, 2023 and the pre-conference workshops were held during July 17–18, 2023. CAV 2023 was an in-person event, in Paris, France.

CAV is an annual conference dedicated to the advancement of the theory and practice of computer-aided formal analysis methods for hardware and software systems. The primary focus of CAV is to extend the frontiers of verification techniques by expanding to new domains such as security, quantum computing, and machine learning. This puts CAV at the cutting edge of formal methods research, and this year's program is a reflection of this commitment.

CAV 2023 received a large number of submissions (261). We accepted 15 tool papers, 3 case-study papers, and 49 regular papers, which amounts to an acceptance rate of roughly 26%. The accepted papers cover a wide spectrum of topics, from theoretical results to applications of formal methods. These papers apply or extend formal methods to a wide range of domains such as concurrency, machine learning and neural networks, quantum systems, as well as hybrid and stochastic systems. The program featured keynote talks by Ruzica Piskac (Yale University), Sumit Gulwani (Microsoft), and Caroline Trippel (Stanford University). In addition to the contributed talks, CAV also hosted the CAV Award ceremony, and a report from the Synthesis Competition (SYNTCOMP) chairs.

In addition to the main conference, CAV 2023 hosted the following workshops: Meeting on String Constraints and Applications (MOSCA), Verification Witnesses and Their Validation (VeWit), Verification of Probabilistic Programs (VeriProP), Open Problems in Learning and Verification of Neural Networks (WOLVERINE), Deep Learning-aided Verification (DAV), Hyperproperties: Advances in Theory and Practice (HYPER), Synthesis (SYNT), Formal Methods for ML-Enabled Autonomous Systems (FoMLAS), and Verification Mentoring Workshop (VMW). CAV 2023 also hosted a workshop dedicated to Thomas A. Henzinger for this 60th birthday.

Organizing a flagship conference like CAV requires a great deal of effort from the community. The Program Committee for CAV 2023 consisted of 76 members—a committee of this size ensures that each member has to review only a reasonable number of papers in the allotted time. In all, the committee members wrote over 730 reviews while investing significant effort to maintain and ensure the high quality of the conference program. We are grateful to the CAV 2023 Program Committee for their outstanding efforts in evaluating the submissions and making sure that each paper got a fair chance. Like recent years in CAV, we made artifact evaluation mandatory for tool paper submissions, but optional for the rest of the accepted papers. This year we received 48 artifact submissions, out of which 47 submissions received at least one badge. The Artifact Evaluation Committee consisted of 119 members who put in significant effort to evaluate each artifact. The goal of this process was to provide constructive feedback to tool developers and help make the research published in CAV more reproducible. We are also very grateful to the Artifact Evaluation Committee for their hard work and dedication in evaluating the submitted artifacts.

CAV 2023 would not have been possible without the tremendous help we received from several individuals, and we would like to thank everyone who helped make CAV 2023 a success. We would like to thank Alessandro Cimatti, Isil Dillig, Javier Esparza, Azadeh Farzan, Joost-Pieter Katoen and Corina Pasareanu for serving as area chairs. We also thank Bernhard Kragl and Daniel Dietsch for chairing the Artifact Evaluation Committee. We also thank Mohamed Faouzi Atig for chairing the workshop organization as well as leading publicity efforts, Eric Koskinen as the fellowship chair, Sebastian Bardin and Ruzica Piskac as sponsorship chairs, and Srinidhi Nagendra as the website chair. Srinidhi, along with Enrique Román Calvo, helped prepare the proceedings. We also thank Ankush Desai, Eric Koskinen, Burcu Kulahcioglu Ozkan, Marijana Lazic, and Matteo Sammartino for chairing the mentoring workshop. Last but not least, we would like to thank the members of the CAV Steering Committee (Kenneth McMillan, Aarti Gupta, Orna Grumberg, and Daniel Kroening) for helping us with several important aspects of organizing CAV 2023.

We hope that you will find the proceedings of CAV 2023 scientifically interesting and thought-provoking!

June 2023

Constantin Enea
Akash Lal

Organization

Conference Co-chairs

Constantin Enea	LIX, École Polytechnique, France
Akash Lal	Microsoft Research, India

Artifact Co-chairs

Bernhard Kragl	Amazon Web Services, USA
Daniel Dietsch	Qt Group/University of Freiburg, Germany

Workshop Chair

Mohamed Faouzi Atig	Uppsala University, Sweden

Verification Mentoring Workshop Organizing Committee

Ankush Densai	AWS CA, USA
Eric Koskinen	Stevens Institute of Technology, USA
Burcu Kulahcioglu Ozkan	TU Delft, The Netherlands
Marijana Lazic	TU Munich, Germany
Matteo Sammartino	Royal Holloway, University of London, UK

Fellowship Chair

Eric Koskinen	Stevens Institute of Technology, USA

Website Chair

Srinidhi Nagendra	Université Paris Cité, CNRS, IRIF, France and Chennai Mathematical Institute, India

Sponsorship Co-chairs

Sebastian Bardin	CEA LIST, France
Ruzica Piskac	Yale University, USA

Proceedings Chairs

Srinidhi Nagendra	Université Paris Cité, CNRS, IRIF, France and Chennai Mathematical Institute, India
Enrique Román Calvo	Université Paris Cité, CNRS, IRIF, France

Program Committee

Aarti Gupta	Princeton University, USA
Abhishek Bichhawat	IIT Gandhinagar, India
Aditya V. Thakur	University of California, USA
Ahmed Bouajjani	University of Paris, France
Aina Niemetz	Stanford University, USA
Akash Lal	Microsoft Research, India
Alan J. Hu	University of British Columbia, Canada
Alessandro Cimatti	Fondazione Bruno Kessler, Italy
Alexander Nadel	Intel, Israel
Anastasia Mavridou	KBR, NASA Ames Research Center, USA
Andreas Podelski	University of Freiburg, Germany
Ankush Desai	Amazon Web Services
Anna Slobodova	Intel, USA
Anthony Widjaja Lin	TU Kaiserslautern and Max-Planck Institute for Software Systems, Germany
Arie Gurfinkel	University of Waterloo, Canada
Arjun Radhakrishna	Microsoft, India
Aws Albarghouthi	University of Wisconsin-Madison, USA
Azadeh Farzan	University of Toronto, Canada
Bernd Finkbeiner	CISPA Helmholtz Center for Information Security, Germany
Bettina Koenighofer	Graz University of Technology, Austria
Bor-Yuh Evan Chang	University of Colorado Boulder and Amazon, USA
Burcu Kulahcioglu Ozkan	Delft University of Technology, The Netherlands
Caterina Urban	Inria and École Normale Supérieure, France
Cezara Dragoi	Amazon Web Services, USA

Christoph Matheja	Technical University of Denmark, Denmark
Claudia Cauli	Amazon Web Services, UK
Constantin Enea	LIX, CNRS, Ecole Polytechnique, France
Corina Pasareanu	CMU, USA
Cristina David	University of Bristol, UK
Dirk Beyer	LMU Munich, Germany
Elizabeth Polgreen	University of Edinburgh, UK
Elvira Albert	Complutense University, Spain
Eunsuk Kang	Carnegie Mellon University, USA
Gennaro Parlato	University of Molise, Italy
Hossein Hojjat	Tehran University and Tehran Institute of Advanced Studies, Iran
Ichiro Hasuo	National Institute of Informatics, Japan
Isil Dillig	University of Texas, Austin, USA
Javier Esparza	Technische Universität München, Germany
Joost-Pieter Katoen	RWTH-Aachen University, Germany
Juneyoung Lee	AWS, USA
Jyotirmoy Deshmukh	University of Southern California, USA
Kenneth L. McMillan	University of Texas at Austin, USA
Kristin Yvonne Rozier	Iowa State University, USA
Kshitij Bansal	Google, USA
Kuldeep Meel	National University of Singapore, Singapore
Kyungmin Bae	POSTECH, South Korea
Marcell Vazquez-Chanlatte	Alliance Innovation Lab (Nissan-Renault-Mitsubishi), USA
Marieke Huisman	University of Twente, The Netherlands
Markus Rabe	Google, USA
Marta Kwiatkowska	University of Oxford, UK
Matthias Heizmann	University of Freiburg, Germany
Michael Emmi	AWS, USA
Mihaela Sighireanu	University Paris Saclay, ENS Paris-Saclay and CNRS, France
Mohamed Faouzi Atig	Uppsala University, Sweden
Naijun Zhan	Institute of Software, Chinese Academy of Sciences, China
Nikolaj Bjorner	Microsoft Research, USA
Nina Narodytska	VMware Research, USA
Pavithra Prabhakar	Kansas State University, USA
Pierre Ganty	IMDEA Software Institute, Spain
Rupak Majumdar	Max Planck Institute for Software Systems, Germany
Ruzica Piskac	Yale University, USA

Sebastian Junges	Radboud University, The Netherlands
Sébastien Bardin	CEA, LIST, Université Paris Saclay, France
Serdar Tasiran	Amazon, USA
Sharon Shoham	Tel Aviv University, Israel
Shaz Qadeer	Meta, USA
Shuvendu Lahiri	Microsoft Research, USA
Subhajit Roy	Indian Institute of Technology, Kanpur, India
Suguman Bansal	Georgia Institute of Technology, USA
Swarat Chaudhuri	UT Austin, USA
Sylvie Putot	École Polytechnique, France
Thomas Wahl	GrammaTech, USA
Tomáš Vojnar	Brno University of Technology, FIT, Czech Republic
Yakir Vizel	Technion - Israel Institute of Technology, Israel
Yu-Fang Chen	Academia Sinica, Taiwan
Zhilin Wu	State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China

Artifact Evaluation Committee

Alejandro Hernández-Cerezo	Complutense University of Madrid, Spain
Alvin George	IISc Bangalore, India
Aman Goel	Amazon Web Services, USA
Amit Samanta	University of Utah, USA
Anan Kabaha	Technion, Israel
Andres Noetzli	Cubist, Inc., USA
Anna Becchi	Fondazione Bruno Kessler, Italy
Arnab Sharma	University of Oldenburg, Germany
Avraham Raviv	Bar Ilan University, Israel
Ayrat Khalimov	TU Clausthal, Germany
Baoluo Meng	General Electric Research, USA
Benjamin Jones	Amazon Web Services, USA
Bohua Zhan	Institute of Software, Chinese Academy of Sciences, China
Cayden Codel	Carnegie Mellon University, USA
Charles Babu M.	CEA LIST, France
Chungha Sung	Amazon Web Services, USA
Clara Rodriguez-Núñez	Universidad Complutense de Madrid, Spain
Cyrus Liu	Stevens Institute of Technology, USA
Daniel Hausmann	University of Gothenburg, Sweden

Daniela Kaufmann	TU Wien, Austria
Debasmita Lohar	MPI SWS, Germany
Deivid Vale	Radboud University Nijmegen, Netherlands
Denis Mazzucato	Inria, France
Dorde Žikelić	Institute of Science and Technology Austria, Austria
Ekanshdeep Gupta	New York University, USA
Enrico Magnago	Amazon Web Services, USA
Ferhat Erata	Yale University, USA
Filip Cordoba	Graz University of Technology, Austria
Filipe Arruda	UFPE, Brazil
Florian Dorfhuber	Technical University of Munich, Germany
Florian Sextl	TU Wien, Austria
Francesco Parolini	Sorbonne University, France
Frédéric Recoules	CEA LIST, France
Goktug Saatcioglu	Cornell, USA
Goran Piskachev	Amazon Web Services, USA
Grégoire Menguy	CEA LIST, France
Guy Amir	Hebrew University of Jerusalem, Israel
Habeeb P.	Indian Institute of Science, Bangalore, India
Hadrien Renaud	UCL, UK
Haoze Wu	Stanford University, USA
Hari Krishnan	University of Waterloo, Canada
Hünkar Tunç	Aarhus University, Denmark
Idan Refaeli	Hebrew University of Jerusalem, Israel
Ignacio D. Lopez-Miguel	TU Wien, Austria
Ilina Stoilkovska	Amazon Web Services, USA
Ira Fesefeldt	RWTH Aachen University, Germany
Jahid Choton	Kansas State University, USA
Jie An	National Institute of Informatics, Japan
John Kolesar	Yale University, USA
Joseph Scott	University of Waterloo, Canada
Kevin Lotz	Kiel University, Germany
Kirby Linvill	CU Boulder, USA
Kush Grover	Technical University of Munich, Germany
Levente Bajczi	Budapest University of Technology and Economics, Hungary
Liangcheng Yu	University of Pennsylvania, USA
Luke Geeson	UCL, UK
Lutz Klinkenberg	RWTH Aachen University, Germany
Marek Chalupa	Institute of Science and Technology Austria, Austria

Mario Bucev	EPFL, Switzerland
Mário Pereira	NOVA LINCS—Nova School of Science and Technology, Portugal
Marius Mikucionis	Aalborg University, Denmark
Martin Jonáš	Masaryk University, Czech Republic
Mathias Fleury	University of Freiburg, Germany
Matthias Hetzenberger	TU Wien, Austria
Maximilian Heisinger	Johannes Kepler University Linz, Austria
Mertcan Temel	Intel Corporation, USA
Michele Chiari	TU Wien, Austria
Miguel Isabel	Universidad Complutense de Madrid, Spain
Mihai Nicola	Stevens Institute of Technology, USA
Mihály Dobos-Kovács	Budapest University of Technology and Economics, Hungary
Mikael Mayer	Amazon Web Services, USA
Mitja Kulczynski	Kiel University, Germany
Muhammad Mansur	Amazon Web Services, USA
Muqsit Azeem	Technical University of Munich, Germany
Neelanjana Pal	Vanderbilt University, USA
Nicolas Koh	Princeton University, USA
Niklas Metzger	CISPA Helmholtz Center for Information Security, Germany
Omkar Tuppe	IIT Bombay, India
Pablo Gordillo	Complutense University of Madrid, Spain
Pankaj Kalita	Indian Institute of Technology, Kanpur, India
Parisa Fathololumi	Stevens Institute of Technology, USA
Pavel Hudec	HKUST, Hong Kong, China
Peixin Wang	University of Oxford, UK
Philippe Heim	CISPA Helmholtz Center for Information Security, Germany
Pritam Gharat	Microsoft Research, India
Priyanka Darke	TCS Research, India
Ranadeep Biswas	Informal Systems, Canada
Robert Rubbens	University of Twente, Netherlands
Rubén Rubio	Universidad Complutense de Madrid, Spain
Samuel Judson	Yale University, USA
Samuel Pastva	Institute of Science and Technology Austria, Austria
Sankalp Gambhir	EPFL, Switzerland
Sarbojit Das	Uppsala University, Sweden
Sascha Klüppelholz	Technische Universität Dresden, Germany
Sean Kauffman	Aalborg University, Denmark

Shaowei Zhu	Princeton University, USA
Shengjian Guo	Amazon Web Services, USA
Simmo Saan	University of Tartu, Estonia
Smruti Padhy	University of Texas at Austin, USA
Stanly Samuel	Indian Institute of Science, Bangalore, India
Stefan Pranger	Graz University of Technology, Austria
Stefan Zetzsche	Amazon Web Services, USA
Sumanth Prabhu	TCS Research, India
Sumit Lahiri	Indian Institute of Technology, Kanpur, India
Sunbeom So	Korea University, South Korea
Syed M. Iqbal	Amazon Web Services, USA
Tobias Meggendorfer	Institute of Science and Technology Austria, Austria
Tzu-Han Hsu	Michigan State University, USA
Verya Monjezi	University of Texas at El Paso, USA
Wei-Lun Tsai	Academia Sinica, Taiwan
William Schultz	Northeastern University, USA
Xiao Liang Yu	National University of Singapore, Singapore
Yahui Song	National University of Singapore, Singapore
Yasharth Bajpai	Microsoft Research, USA
Ying Sheng	Stanford University, USA
Yuriy Biktairov	University of Southern California, USA
Zafer Esen	Uppsala University, Sweden

Additional Reviewers

Azzopardi, Shaun
Baier, Daniel
Belardinelli, Francesco
Bergstraesser, Pascal
Boker, Udi
Ceska, Milan
Chien, Po-Chun
Coglio, Alessandro
Correas, Jesús
Doveri, Kyveli
Drachsler Cohen, Dana
Durand, Serge
Fried, Dror
Genaim, Samir
Ghosh, Bishwamittra
Gordillo, Pablo
Guillermo, Roman Diez
Gómez-Zamalloa, Miguel
Hernández-Cerezo, Alejandro
Holík, Lukáš
Isabel, Miguel
Ivrii, Alexander
Izza, Yacine
Jothimurugan, Kishor
Kaivola, Roope
Kaminski, Benjamin Lucien
Kettl, Matthias
Kretinsky, Jan
Lengal, Ondrej
Losa, Giuliano
Luo, Ning
Malik, Viktor

Markgraf, Oliver
Martin-Martin, Enrique
Meller, Yael
Perez, Mateo
Petri, Gustavo
Pote, Yash
Preiner, Mathias
Rakamaric, Zvonimir
Rastogi, Aseem
Razavi, Niloofar
Rogalewicz, Adam
Sangnier, Arnaud
Sarkar, Uddalok
Schoepe, Daniel
Sergey, Ilya

Stoilkovska, Ilina
Stucki, Sandro
Tsai, Wei-Lun
Turrini, Andrea
Vafeiadis, Viktor
Valiron, Benoît
Wachowitz, Henrik
Wang, Chao
Wang, Yuepeng
Wies, Thomas
Yang, Jiong
Yen, Di-De
Zhu, Shufang
Žikelić, Đorđe
Zohar, Yoni

Contents – Part II

Decision Procedures

Model Checking

Neural Networks and Machine Learning

Decision Procedures

Bitwuzla

Aina Niemetz(✉) and Mathias Preiner

Stanford University, Stanford, USA
{niemetz,preiner}@cs.stanford.edu

Abstract. Bitwuzla is a new SMT solver for the quantifier-free and quantified theories of fixed-size bit-vectors, arrays, floating-point arithmetic, and uninterpreted functions. This paper serves as a comprehensive system description of its architecture and components. We evaluate Bitwuzla's performance on all benchmarks of supported logics in SMT-LIB and provide a comparison against other state-of-the-art SMT solvers.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers serve as back-end reasoning engines for a wide range of applications in formal methods (e.g., [13,14,21,23,35]). In particular, the theory of fixed-size bit-vectors, in combination with arrays, uninterpreted functions and floating-point arithmetic, have received increasing interest in recent years, as witnessed by the high and increasing numbers of benchmarks submitted to the SMT-LIB benchmark library [5] and the number of participants in corresponding divisions in the annual SMT competition (SMT-COMP) [42]. State-of-the-art SMT solvers supporting (a subset of) these theories include Boolector [31], cvc5 [3], MathSAT [15], STP [19], Yices2 [17] and Z3 [25]. Among these, Boolector had been largely dominating the quantifier-free divisions with bit-vectors and arrays in SMT-COMP over the years [2].

Boolector was originally published in 2009 by Brummayer and Biere [11] as an SMT solver for the quantifier-free theories of fixed-size bit-vectors and arrays. Since 2012, Boolector has been mainly developed and maintained by the authors of this paper, who have extended it with support for uninterpreted functions and lazy handling of non-recursive lambda terms [32,38,39], local search strategies for quantifier-free bit-vectors [33,34], and quantified bit-vector formulas [40].

While Boolector is still competitive in terms of performance, it has several limitations. Its code base consists of largely monolithic C code, with a rigid architecture focused on a very specialized, tight integration of bit-vectors and arrays. Consequently, it is cumbersome to maintain, and adding new features is difficult and time intensive. Further, Boolector requires manual management of memory and reference counts from API users; terms and sorts are tied to a specific solver instance and cannot be shared across instances; all preprocessing

This work was supported in part by the Stanford Center for Automated Reasoning, the Stanford Agile Hardware Center, the Stanford Center for Blockchain Research and a gift from Amazon Web Services.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 3–17, 2023.
https://doi.org/10.1007/978-3-031-37703-7_1

techniques are destructive, which disallows incremental preprocessing; and due to architectural limitations, incremental solving with quantifiers is not supported.

In 2018, we forked Boolector in preparation for addressing these issues, and entered an improved and extended version of this fork as Bitwuzla in the SMT competition 2020 [26]. At that time, Bitwuzla extended Boolector with: support for floating-point arithmetic by integrating SymFPU [8] (a C++ library of bit-vector encodings of floating-point operations); a novel generalization of its propagation-based local search strategy [33] to ternary values [27]; unsat core extraction; and since 2022, support for reasoning about quantified formulas for all supported theories and their combinations. This version of Bitwuzla was already made available on GitHub at [28], but not officially released. However, architectural and structural limitations inherited from Boolector remained. Thus, to overcome these limitations and address the above issues, we decided to discard the existing code base and rewrite Bitwuzla from scratch.

In this paper, we present the first official release of Bitwuzla, an SMT solver for the (quantified and quantifier-free) theories of fixed-size bit-vectors, arrays, floating-point arithmetic, uninterpreted functions and their combinations. Its name (pronounced as *bitvootslah*) is derived from an Austrian dialect expression that can be translated as *someone who tinkers with bits*. Bitwuzla is written in C++, inspired by techniques implemented in Boolector. That is, rather than only redesigning problematic aspects of Boolector, we carefully dissected and (re)evaluated its parts to serve as guidance when writing a new solver from scratch. In that sense, it is not a reimplementation of Boolector, but can be considered its superior successor. Bitwuzla is available on GitHub [28] under the MIT license, and its documentation is available at [29].

2 Architecture

Bitwuzla supports reasoning about quantifier-free and quantified formulas over fixed-size bit-vectors, floating-point arithmetic, arrays and uninterpreted functions as standardized in SMT-LIB [4]. In this section, we provide an overview of Bitwuzla's system architecture and its core components as given in Fig. 1.

Bitwuzla consists of two main components: the *Solving Context* and the *Node Manager*. The Solving Context can be seen as a solver instance that determines satisfiability of a set of formulas and implements the lazy, abstraction/refinement-based SMT paradigm *lemmas on demand* [6,24] (in contrast to SMT solvers like cvc5 and Z3, which are based on the CDCL($\mathcal{T}$) [36] framework). The Node Manager is responsible for constructing and maintaining nodes and types and is shared across multiple Solving Context instances.

Bitwuzla provides a comprehensive C++ API as its main interface, with a C and Python API built on top. All features of the C++ API are also accessible to C and Python users. The API documentation is available at [29]. The C++ API exports Term, Sort, Bitwuzla, and Option classes for constructing nodes and types, configuring solver options and constructing Bitwuzla solver instances (the external representation of Solving Contexts). Term and Sort objects may be

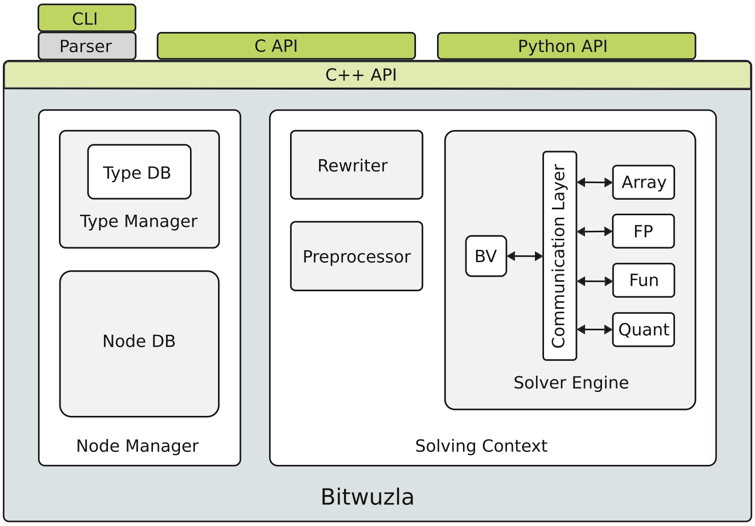


Fig. 1. Bitwuzla system architecture.

used in multiple Bitwuzla instances. The parser interacts with the solver instance via the C++ API. A textual command line interface (CLI) builds on top of the parser, supporting SMT-LIBv2 [4] and BTOR2 [35] as input languages.

2.1 Node Manager

Bitwuzla represents formulas and terms as reference-counted, immutable nodes in a directed acyclic graph. The Node Manager is responsible for constructing and managing these nodes and employs hash-consing to maximize sharing of subgraphs. Automatic reference counting allows the Node Manager to determine when to delete nodes. Similarly, types are constructed and managed by the *Type Manager*, which is maintained by the Node Manager. Nodes and types are stored globally (thread-local) in the Node Database and Type Database, which has the key advantage that they can be shared between arbitrarily many solving contexts within one thread. This is one of the key differences to Boolector's architecture, where terms and types are manually reference counted and tied to a single solver instance, which does not allow sharing between solver instances.

2.2 Solving Context

A *Solving Context* is the internal equivalent of a solver instance and determines the satisfiability of a set of asserted formulas (assertions). Solving Contexts are fully configurable via options and provide an incremental interface for adding and removing assertions via push and pop. Incremental solving allows users to perform multiple satisfiability checks with similar sets of assertions while reusing work from earlier checks. On the API level, Bitwuzla also supports satisfiability queries under a given set of assumptions (SMT-LIB command `check-sat-assuming`), which are internally handled via push and pop.

Nodes and types constructed via the Node Manager may be shared between multiple Solving Contexts. If the set of assertions is satisfiable, the Solving Context provides a model for the input formula. It further allows to query model values for any term, based on this model (SMT-LIB command `get-value`). In case of unsatisfiable queries, the Solving Context can be configured to extract an unsatisfiable core and unsat assumptions.

A Solving Context consists of three main components: a *Rewriter*, a *Preprocessor* and a *Solver Engine*. The Rewriter and Preprocessor perform local (node level) and global (over all assertions) simplifications, whereas the Solver Engine is the central solving engine, managing theory solvers and their interaction.

Preprocessor. As a first step of each satisfiability check, prior to solving, the preprocessor applies a pipeline of preprocessing passes in a predefined order to the current set of assertions until fixed-point. Each preprocessing pass implements a set of satisfiability-preserving transformations. All passes can be optionally disabled except for one mandatory transformation, the reduction of the full set of operators supported on the API level to a reduced operator set: Boolean connectives are expressed by means of $\{\neg, \wedge\}$, quantifier $\exists$ is represented in terms of $\forall$, inequalities are represented in terms of $<$ and $>$, signed bit-vector operators are expressed in terms of unsigned operators, and more. These reduction transformations are a subset of the term rewrites performed by the Rewriter, and rewriting is implemented as one preprocessing pass. Additionally, Bitwuzla implements 7 preprocessing passes, which are applied sequentially, after rewriting, until no further transformations are possible: *and flattening*, which splits a top-level $\wedge$ into its subformulas, e.g., $a \wedge (b \wedge (c = d))$ into $\{a, b, c = d\}$; *substitution*, which replaces all occurrences of a constant x with a term t if $x = t$ is derived on the top level; *skeleton preprocessing*, which simplifies the Boolean skeleton of the input formula with a SAT solver; *embedded constraints*, which substitutes all occurrences of top-level constraints in subterms of other top-level constraints with *true*; *extract elimination*, which eliminates bit-vector extracts over constants; *lambda elimination*, which applies beta reduction on lambda terms; and *normalization* of arithmetic expressions.

Preprocessing in Bitwuzla is *fully incremental*: all passes are applied to the current set of assertions, from all assertion levels, and simplifications derived from lower levels are applied to all assertions of higher levels (including assumptions). Assertions are processed per assertion level i, starting from $i = 0$, and for each level $i > 0$, simplifications are applied based on information from all levels $j \leq i$. Note that when solving under assumptions, Bitwuzla internally pushes an assertion level and handles these assumptions as assertions of that level. When a level i is popped, the assertions of that level are popped, and the state of the preprocessor is backtracked to the state that was associated with level $i - 1$. Note that preprocessing assertion levels $i < j$ with information derived from level j requires to not only restore the state of the preprocessor, but to also reconstruct the assertions on levels $i < j$ when level j is popped to the state before level j was pushed, and is left to future work.

Boolector, on the other hand, only performs preprocessing based on top-level assertions (assertion level 0) and does not incorporate any information from assumptions or higher assertion levels.

Rewriter. The rewriter transforms terms via a predefined set of rewrite rules into semantically equivalent normal forms. This transformation is local in the sense that it is independent from the current set of assertions. We distinguish between required and optional rewrite rules, and further group rules into so-called rewrite levels from 0–2. The set of required rules consists of operator elimination rewrites, which are considered level 0 rewrites and ensure that nodes only contain operators from a reduced base set. For example, the two's complement $-x$ of a bit-vector term x is rewritten to $({\sim} x + 1)$ by means of one's complement and bit-vector addition. Optional rewrite rules are grouped into level 1 and level 2. Level 1 rules perform rewrites that only consider the immediate children of a node, whereas level 2 rules may consider multiple levels of children. If not implemented carefully, level 2 rewrites can potentially destroy sharing of subterms and consequently increase the overall size of the formula. For example, rewriting $(t + 0)$ to t is considered a level 1 rewrite rule, whereas rewriting $(a - b = c)$ to $(b + c = a)$ is considered a level 2 rule since it may introduce an additional bit-vector addition $(b + c)$ if $(a - b)$ occurs somewhere else in the formula. The maximum rewrite level of the rewriter can be configured by the user.

Rewriting is applied on the current set of assertions as a preprocessing pass and, as all other passes, applied until fixed-point. That is, on any given term, the rewriter applies rewrite rules until no further rewrite rules can be applied. For this, the rewriter must guarantee that no set of applied rewrite rules may lead to cyclic rewriting of terms. Additionally, all components of the solving context apply rewriting on freshly created nodes to ensure that all nodes are always fully normalized. In order to avoid processing nodes more than once, the rewriter maintains a cache that maps nodes to their fully rewritten form.

Solver Engine. After preprocessing, the solving context sends the current set of assertions to the Solver Engine, which implements a lazy SMT paradigm called *lemmas on demand* [6,24]. However, rather than using a propositional abstraction of the input formula as in [6,24], it implements a bit-vector abstraction similar to Boolector [12,38]. At its core, the Solver Engine maintains a bit-vector theory solver and a solver for each supported theory. Quantifier reasoning is handled by a dedicated quantifiers module, implemented as a theory solver. The Solver Engine manages all theory solvers, the distribution of relevant terms, and the processing of lemmas generated by the theory solvers.

The bit-vector solver is responsible for reasoning about the bit-vector abstraction of the input assertions and lemmas generated during solving, which includes all propositional and bit-vector terms. Theory atoms that do not belong to the bit-vector theory are abstracted as Boolean constants, and bit-vector terms whose operator does not belong to the bit-vector theory are abstracted as bit-vector constants. For example, an array select operation of type bit-vector is abstracted as a bit-vector constant, while an equality between two arrays is abstracted as a Boolean constant.

If the bit-vector abstraction is satisfiable, the bit-vector solver produces a satisfying assignment, and the floating-point, array, function and quantifier solvers check this assignment for theory consistency. If a solver finds a theory inconsistency, i.e., a conflict between the current satisfying assignment and the solver's theory axioms, it produces a lemma to refine the bit-vector abstraction and rule out the detected inconsistency. Theory solvers are allowed to send any number of lemmas, with the only requirement that if a theory solver does not send a lemma, the current satisfying assignment is consistent with the theory.

Finding a satisfying assignment for the bit-vector abstraction and the subsequent theory consistency checks are implemented as an abstraction/refinement loop as given in Algorithm 1. Whenever a theory solver sends lemmas, the loop is restarted to get a new satisfying assignment for the refined bit-vector abstraction. The loop terminates if the bit-vector abstraction is unsatisfiable, or if the bit-vector abstraction is satisfiable and none of the theory solvers report any theory inconsistencies. Note that the abstraction/refinement algorithm may return *unknown* if the input assertions include quantified formulas.

Algorithm 1. Abstraction/refinement loop in Solver Engine. Function SOLVE ($\mathcal{A}$) is called on the current set of preprocessed assertions $\mathcal{A}$, which is iteratively refined with a set of Lemmas $\mathcal{L}$.

```
function SOLVE($\mathcal{A}$)
    $r \leftarrow$ UNKNOWN, $\mathcal{L} \leftarrow \emptyset$
    repeat
        $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{L}$
        $r, \mathcal{M} \leftarrow T_{BV}$::SOLVE($\mathcal{A}$)          ▷ Solve bit-vector abstraction of $\mathcal{A}$
        if $r =$ UNSAT then break end if
        $\mathcal{L} \leftarrow T_{FP}$::CHECK($\mathcal{M}$)             ▷ Check FP theory consistency of $\mathcal{M}$
        if $\mathcal{L} \neq \emptyset$ then continue end if
        $\mathcal{L} \leftarrow T_{\mathcal{A}}$::CHECK($\mathcal{M}$)          ▷ Check array theory consistency of $\mathcal{M}$
        if $\mathcal{L} \neq \emptyset$ then continue end if
        $\mathcal{L} \leftarrow T_{UF}$::CHECK($\mathcal{M}$)             ▷ Check UF theory consistency of $\mathcal{M}$
        if $\mathcal{L} \neq \emptyset$ then continue end if
        $\mathcal{L} \leftarrow T_{Q}$::CHECK($\mathcal{M}$)              ▷ Check quantified formulas in $\mathcal{M}$
    until $\mathcal{L} = \emptyset$
    return $r$
end function
```

Backtrackable Data Structures. Every component of the Solver Context except for the Rewriter depends on the current set of assertions. When solving incrementally, the assertion stack is modified by adding (SMT-LIB command `push`) and removing (SMT-LIB command `pop`) assertions. In contrast to Boolector, Bitwuzla supports saving and restoring the internal solver state, i.e., the state of the Solving Context, corresponding to these push and pop operations by means of *backtrackable data structures*. These data structures are custom variants of mutable data structures provided in the C++ standard library, extended with an interface to save and restore their state on push and pop calls. This allows the solver to take full advantage of incremental solving by reusing work from previous satisfiability checks and backtracking to previous states. Further, this enables incremental preprocessing. Bitwuzla's backtrable data structures are conceptually similar to context-dependent data structures in cvc5 [3].

3 Theory Solvers

The Solver Engine maintains a theory solver for each supported theory and implements a module for handling quantified formulas as a dedicated theory solver. The central engine of the Solver Engine is the bit-vector theory solver, which reasons about a bit-vector abstraction of the current set of input assertions, refined with lemmas generated by other theory solvers. The theories of fixed-size bit-vectors, arrays, floating-point arithmetic, and uninterpreted functions are combined via a model-based theory combination approach similar to [12,38].

Theory combination is based on candidate models produced by the bit-vector theory solver for the bit-vector abstraction (function T_{BV}::solve() in Algorithm 1). For each candidate model, each theory solver checks consistency with the axioms of the corresponding theory (functions T_*::check() in Algorithm 1). If a theory solver requests a model value for a term that is not part of the current bit-vector abstraction, the theory solver who "owns" that term is queried for a value. If this value or the candidate model is inconsistent with the axioms of the theory querying the value, it sends a lemma to refine the bit-vector abstraction.

3.1 Arrays

The array theory solver implements and extends the array procedure from [12] with support for reasoning over (equalities of) nested arrays and non-extensional constant arrays. This is in contrast to Boolector, which generalizes the lemmas on demand procedure for extensional arrays as described in [12] to non-recursive first-order lambda terms [37,38], without support for nested arrays. Generalizing arrays to lambda terms allows to use the same procedure for arrays and uninterpreted functions and enables a natural, compact representation and extraction of extended array operations such as *memset*, *memcpy* and array initialization patterns as described in [39]. As an example, $\mathit{memset}(a, i, n, e)$, which updates n elements of array a within range $[i, i+n[$ to a value e starting from index i, can be represented as $\lambda j \,.\, \mathit{ite}(i \leq j < i+n, e, a[j])$. Reasoning over equalities involving arbitrary lambda terms (including these operations), however, requires higher-order reasoning, which is not supported by Boolector. Further, extensionality over standard array operators that are represented as lambda terms (e.g., `store`) requires special handling, which makes the procedure unnecessarily complex. Bitwuzla, on the other hand, implements separate theory solvers for arrays and uninterpreted functions. Consequently, since it does not generalize arrays to lambda terms, it cannot utilize the elegant representation of Boolector for the extended array operations of [39]. Thus, currently, extracting and reasoning about these operations is not yet supported. Instead of representing such operators as lambda terms, we plan to introduce specific array operators. This will allow a seamless integration into Bitwuzla's array procedure, with support for reasoning about extensionality involving these operators. We will also add support for reasoning about extensional constant arrays in the near future.

3.2 Bit-Vectors

The bit-vector theory solver implements two orthogonal approaches: the classic *bit-blasting* technique employed by most state-of-the-art bit-vector solvers, which eagerly translates the current bit-vector abstraction to SAT; and the *ternary propagation-based local search* approach presented in [27]. Since local search procedures only allow to determine satisfiability, they are particularly effective as a complementary strategy, in combination with (rather than instead of) bit-blasting [27,33]. Bitwuzla's bit-vector solver allows to combine local search with bit-blasting in a sequential portfolio setting: the local search procedure is run until a predefined resource limit is reached before falling back on the bit-blasting procedure. Currently, Bitwuzla allows combining these two approaches only in this particular setting. We plan to explore more interleaved configurations, possibly while sharing information between the procedures as future work.

Bit-Blasting. Bitwuzla implements the eager reduction of the bit-vector abstraction to propositional logic in two phases. First, it constructs an And-Inverter-Graph (AIG) circuit representation of the abstraction while applying AIG-level rewriting techniques [10]. This AIG circuit is then converted into Conjunctive Normal Form (CNF) via Tseitin transformation and sent to the SAT solver back-end. Note that for assertions from levels > 0, Bitwuzla leverages solving under assumptions in the SAT solver in order to be able to backtrack to lower assertion levels on pop. Bitwuzla supports CaDiCaL [7], CryptoMiniSat [41], and Kissat [7] as SAT back-ends and uses CaDiCaL as its default SAT solver.

Local Search. Bitwuzla implements an improved version of the ternary propagation-based local search procedure described in [27]. This procedure is a generalization of the propagation-based local search approach implemented in Boolector [33] and addresses one of its main weaknesses: its obliviousness to bits that can be simplified to constant values. Propagation-based local search is based on propagating target values from the outputs to the inputs, does not require bit-blasting, brute-force randomization or restarts, and lifts the concept of backtracing of Automatic Test Pattern Generation (ATPG) [22] to the word-level.

Boolector additionally implements the stochastic local search (SLS) approach presented in [18], optionally augmented with a propagation-based strategy [34]. Bitwuzla, however, only implements our ternary propagation-based approach since it was shown to significantly outperform these approaches [33].

3.3 Floating-Point Arithmetic

The solver for the theory of floating-point arithmetic implements an eager translation of floating-point atoms in the bit-vector abstraction to equisatisfiable formulas in the theory of bit-vectors, a process sometimes referred to as *word-blasting*. To translate floating-point expressions to the word-level, Bitwuzla integrates SymFPU [9], a C++ library of bit-vector encodings of floating-point operations. SymFPU uses templated types for Booleans (un)signed bit-vectors, rounding modes and floating-point formats, which allows utilizing solver-specific representations. SymFPU has also been integrated into cvc5 [3].

3.4 Uninterpreted Functions

For the theory of uninterpreted functions (UF), Bitwuzla implements *dynamic Ackermannization* [16], which is a lazy form of Ackermann's reduction. The UF solver checks whether the current satisfying assignment of the bit-vector abstraction is consistent with the function congruence axiom $\bar{a} = \bar{b} \rightarrow f(\bar{a}) = f(\bar{b})$ and produces a lemma whenever the axiom is violated.

3.5 Quantifiers

Quantified formulas are handled by the quantifiers module, which is treated as a theory solver and implements model-based quantifier instantiation [20] for all supported theories and their combinations. In the bit-vector abstraction, quantified formulas are abstracted as Boolean constants. Based on the assignment of these constants, the quantifiers solver produces instantiation or Skolemization lemmas. If the constant is assigned to true, the quantifier is treated as universal quantifier and the solver produces instantiation lemmas. If the constant is assigned to false, the solver generates a Skolemization lemma. Bitwuzla allows to combine quantifiers with all supported theories as well as incremental solving and unsat core extraction. This is in contrast to Boolector, which only supports sequential reasoning about quantified bit-vector formulas and, generally, does not provide unsat cores for unsatisfiable instances.

4 Evaluation

We evaluate the overall performance of Bitwuzla on all non-incremental and incremental benchmarks of all supported logics in SMT-LIB [5]. We further include logics with floating-point arithmetic that are classified as containing linear integer arithmetic (LRA). Bitwuzla does not support LRA reasoning, but

Table 1. Solved instances and total runtime on solved instances (non-incremental).

Logic	**Boolector**	**Z3**	**cvc5**	**SC22**	**Bitwuzla**
ABV (169)	–	**89**	32	0	1
ABVFP (30)	–	**25**	19	0	16
ABVFPLRA (75)	–	**47**	36	0	31
AUFBV (1,522)	–	403	486	597	**983**
AUFBVFP (57)	–	7	21	24	**39**
BV (6,045)	5,659	5,593	**5,818**	5,624	5,705
BVFP (205)	–	176	171	148	**188**
BVFPLRA (209)	–	189	107	140	**199**
FP (2,669)	–	2,128	2,353	**2,513**	2,481
FPLRA (87)	–	72	51	55	**83**
QF_ABV (15,084)	15,041	14,900	14,923	**15,043**	15,041
QF_ABVFP (18,129)	–	18,017	18,113	**18,125**	**18,125**
QF_ABVFPLRA (74)	–	69	**74**	34	**74**
QF_AUFBV (67)	45	50	42	46	**55**
QF_AUFBVFP (1)	–	1	1	1	1
QF_BV (42,472)	41,958	40,876	41,574	42,039	**42,049**
QF_BVFP (17,244)	–	17,229	17,238	**17,242**	17,241
QF_FP (40,409)	–	40,303	40,357	**40,368**	40,358
QF_FPLRA (57)	–	41	48	**56**	**56**
QF_UFBV (1,434)	1,403	1,404	1,387	**1,413**	1,411
QF_UFFP (2)	–	2	2	2	2
UFBV (192)	–	**156**	141	146	147
UFBVFP (2)	–	1	1	1	1
Total (146,235)	64,106	141,778	142,995	143,617	**144,287**
Time (solved) [s]	417,643	1,212,584	1,000,466	563,832	580,435

the benchmarks in these logics currently only involve to-floating-point conversion (SMT-LIB command `to_fp`) from real values, which is supported.

We compare against Boolector [31] and the SMT-COMP 2022 version of Bitwuzla [26] (configuration SC22), which, at that time, was an improved and extended version of Boolector and won several divisions in all tracks of SMT-COMP 2022 [2]. Boolector did not participate in SMT-COMP 2022, thus we use the current version of Boolector available on GitHub (commit `13a8a06d`) [1]. Further, since Boolector does not support logics involving floating-point arithmetic, quantified logics other than pure quantified bit-vectors and incremental solving when quantifiers are involved, we also compare against the SMT-COMP 2022 versions of cvc5 [3] and Z3 [25]. Both solvers are widely used, high performance SMT solvers with support for a wide range of theories, including the theories supported by Bitwuzla. Note that this version of cvc5 uses a sequential portfolio of multiple configurations for some logics.

Table 2. Solved queries and total runtime on solved queries (incremental).

Logic	**Boolector**	**Z3**	**cvc5**	**SC22**	**Bitwuzla**
ABVFPLRA (2,269)	–	2,220	818	55	**2,269**
BV (38,856)	–	**37,188**	36,169	35,567	35,246
BVFP (458)	–	**458**	**458**	274	**458**
BVFPLRA (5,597)	–	**5,507**	2,964	3,144	4,797
QF_ABV (3,411)	3,238	2,866	2,746	**3,242**	2,939
QF_ABVFP (550,088)	–	515,714	534,629	550,034	**550,041**
QF_ABVFPLRA (1,876)	–	48	**1,876**	**1,876**	**1,876**
QF_AUFBV (967)	23	860	320	23	**956**
QF_BV (53,684)	52,218	51,826	51,683	51,581	**52,305**
QF_BVFP (3,465)	–	3,403	3,437	**3,444**	3,438
QF_BVFPLRA (32,736)	–	31,287	32,681	**32,736**	**32,736**
QF_FP (663)	–	663	663	663	663
QF_FPLRA (48)	–	48	48	48	48
QF_UFBV (5,492)	4,634	5,422	5,148	2,317	**5,489**
QF_UFFP (2)	–	2	2	2	2
Total (699,612)	60,113	657,512	673,642	685,006	**693,263**
Time (solved) [s]	102,812	3,359,645	1,516,672	157,083	172,534

We ran all experiments on a cluster with Intel Xeon E5-2620 v4 CPUs. We allocated one CPU core and 8GB of RAM for each solver and benchmark pair, and used a 1200 s s time limit, the same time limit as used in SMT-COMP 2022 [2].

Table 1 shows the number of solved benchmarks for each solver in the non-incremental quantifier-free (QF_) and quantified divisions. Overall, Bitwuzla solves the largest number of benchmarks in the quantified divisions, considerably improving over SC22 and Boolector with over 600 and 4,200 solved benchmarks, respectively. Bitwuzla also takes the lead in the quantifier-free divisions, with 44 more solved instances compared to SC22, and more than 650 solved benchmarks compared to cvc5. On the 140,438 commonly solved instances between Bitwuzla, SC22, cvc5, and Z3 over all divisions, Bitwuzla is the fastest solver with 203,838s, SC22 is slightly slower with 208,310s, cvc5 is 2.85× slower (586,105s), and Z3 is 5.1× slower (1,049,534s).

Table 2 shows the number of solved incremental check-sat queries for each solver in the incremental divisions. Again, Bitwuzla solves the largest number of queries overall and in the quantifier-free divisions. For the quantified divisions, Bitwuzla solves 42,770 queries, the second largest number of solved queries after Z3 (45,373), and more than 3700 more queries than SC22 (39,040). On benchmarks of the ABVFPLRA division, Bitwuzla significantly outperforms SC22 due to the occurrence of nested arrays, which were unsupported in SC22.

The artifact of this evaluation is archived and available in the Zenodo open-access repository at https://zenodo.org/record/7864687.

5 Conclusion

Our experimental evaluation shows that Bitwuzla is a state-of-the-art SMT solver for the quantified and quantifier-free theories of fixed-size bit-vectors, arrays, floating-point arithmetic, and uninterpreted functions. Bitwuzla has been extensively tested for robustness and correctness with Murxla [30], an API fuzzer for SMT solvers, which is an integral part of its development workflow. We have outlined several avenues for future work throughout the paper. We further plan to add support for the upcoming SMT-LIB version 3 standard, when finalized.

References

1. Boolector. (2023). https://github.com/boolector/boolector
2. The International Satisfiability Modulo Theories Competition (SMT-COMP) (2023). https://smt-comp.github.io
3. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep. Department of Computer Science, The University of Iowa (2017). http://smt-lib.org
5. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2023). http://smt-lib.org
6. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 236–249. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_18
7. 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)
8. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 79–98. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_5
9. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 79–98. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_5
10. Brummayer, R., Biere, A.: Local two-level and-inverter graph minimization without blowup. In: Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06), Mikulov, Czechia, October 2006 (2006)
11. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_16
12. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. J. Satisf. Boolean Model. Comput. **6**(1–3), 165–201 (2009). https://doi.org/10.3233/sat190067

13. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008 (December), pp. 8–10, 2008. San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008). http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
14. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 510–517. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_29
15. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
16. Dutertre, B., de Moura, L.: The Yices SMT Solver (2006). https://yices.csl.sri.com/papers/tool-paper.pdf
17. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
18. Fröhlich, A., Biere, A., Wintersteiger, C.M., Hamadi, Y.: Stochastic local search for satisfiability modulo theories. In: Bonet, B., Koenig, S. (eds.) Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, 25–30 January 2015, Austin, Texas, USA, pp. 1136–1143. AAAI Press (2015). http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9896
19. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
20. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
21. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012). https://doi.org/10.1145/2093548.2093564
22. Kunz, W., Stoffel, D.: Reasoning in Boolean Networks - Logic Synthesis and Verification Using Testing Techniques. Frontiers in Electronic Testing. Springer (1997). https://doi.org/10.1007/978-1-4757-2572-8
23. Mann, M., et al.: Pono: a flexible and extensible SMT-based model checker. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 461–474. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_22
24. Moura, L.D., Rueß, H.: Lemmas on demand for satisfiability solvers. In: The 5th International Symposium on the Theory and Applications of Satisfiability Testing, SAT 2002, Cincinnati, 15 May 2002 (2002)
25. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. arXiv preprint (2020). https://arxiv.org/abs/2006.01621
27. Niemetz, A., Preiner, M.: Ternary propagation-based local search for more bit-precise reasoning. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, 21–24 September 2020, pp. 214–224. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29

28. Niemetz, A., Preiner, M.: Bitwuzla (2023). https://github.com/bitwuzla/bitwuzla
29. Niemetz, A., Preiner, M.: Bitwuzla Documentation (2023). https://bitwuzla.github.io
30. Niemetz, A., Preiner, M., Barrett, C.W.: Murxla: a modular and highly extensible API fuzzer for SMT solvers. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, 7–10 August 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 92–106. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_5
31. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). https://doi.org/10.3233/sat190101
32. Niemetz, A., Preiner, M., Biere, A.: Turbo-charging lemmas on demand with don't care reasoning. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 21–24 October 2014. pp. 179–186. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987611
33. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. Formal Methods Syst. Design **51**(3), 608–636 (2017). https://doi.org/10.1007/s10703-017-0295-6
34. Niemetz, A., Preiner, M., Biere, A., Fröhlich, A.: Improving local search for bit-vector logics in SMT with path propagation. In: Proceedings of the Fourth International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS), affiliated with FMCAD, Austin, pp. 1–10 (2015)
35. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 587–595. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_32
36. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006). https://doi.org/10.1145/1217856.1217859
37. Preiner, M.: Lambdas, Arrays and Quantifiers. Ph.D. thesis, Informatik, Johannes Kepler University Linz (2017)
38. Preiner, M., Niemetz, A., Biere, A.: Lemmas on demand for lambdas. In: Ganai, M.K., Sen, A. (eds.) Proceedings of the Second International Workshop on Design and Implementation of Formal Tools and Systems, Portland, OR, USA, October 19, 2013. CEUR Workshop Proceedings, vol. 1130. CEUR-WS.org (2013). http://ceur-ws.org/Vol-1130/paper_7.pdf
39. Preiner, M., Niemetz, A., Biere, A.: Better lemmas with lambda extraction. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, 27–30 September 2015, pp. 128–135. IEEE (2015). https://doi.org/10.1109/FMCAD.2015.7542262
40. Preiner, M., Niemetz, A., Biere, A.: Counterexample-guided model synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 264–280. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_15
41. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24
42. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT competition 2015–2018. J. Satisf. Boolean Model. Comput. **11**(1), 221–259 (2019). https://doi.org/10.3233/SAT190123

Decision Procedures for Sequence Theories

Artur Jeż[1], Anthony W. Lin[2,3], Oliver Markgraf[2](✉), and Philipp Rümmer[4,5]

[1] University of Wrocław, Wrocław, Poland
[2] TU Kaiserslautern, Kaiserslautern, Germany
markgraf@cs.uni-kl.de
[3] Max Planck Institute for Software Systems, Kaiserslautern, Germany
[4] University of Regensburg, Regensburg, Germany
[5] Uppsala University, Uppsala, Sweden

Abstract. Sequence theories are an extension of theories of strings with an infinite alphabet of letters, together with a corresponding alphabet theory (e.g. linear integer arithmetic). Sequences are natural abstractions of extendable arrays, which permit a wealth of operations including append, map, split, and concatenation. In spite of the growing amount of tool support for theories of sequences by leading SMT-solvers, little is known about the decidability of sequence theories, which is in stark contrast to the state of the theories of strings. We show that the decidable theory of strings with concatenation and regular constraints can be extended to the world of sequences over an alphabet theory that forms a Boolean algebra, while preserving decidability. In particular, decidability holds when regular constraints are interpreted as parametric automata (which extend both symbolic automata and variable automata), but fails when interpreted as register automata (even over the alphabet theory of equality). When length constraints are added, the problem is Turing-equivalent to word equations with length (and regular) constraints. Similar investigations are conducted in the presence of symbolic transducers, which naturally model sequence functions like map, split, filter, *etc.* We have developed a new sequence solver, SeCo, based on parametric automata, and show its efficacy on two classes of benchmarks: (i) invariant checking on array-manipulating programs and parameterized systems, and (ii) benchmarks on symbolic register automata.

1 Introduction

Sequences are an extension of strings, wherein elements might range over an infinite domain (e.g., integers, strings, and even sequences themselves). Sequences

A. Jeż was supported under National Science Centre, Poland project number 2017/26/E/ST6/00191. A. Lin and O. Markgraf were supported by the ERC Consolidator Grant 101089343 (LASD). P. Rümmer was supported by the Swedish Research Council (VR) under grant 2018-04727, the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and the Wallenberg project UPDATE.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 18–40, 2023.
https://doi.org/10.1007/978-3-031-37703-7_2

are ubiquitous and commonly used data types in modern programming languages. They come under different names, e.g., Python/Haskell/Prolog lists, Java ArrayList (and to some extent Streams) and JavaScript arrays. Crucially, sequences are *extendable*, and a plethora of operations (including append, map, split, filter, concatenation, etc.) can naturally be defined and are supported by built-in library functions in most modern programming languages.

Various techniques in software model checking [30] — including symbolic execution, invariant generation — require an appropriate SMT theory, to which verification conditions could be discharged. In the case of programs operating on sequences, we would consequently require an SMT theory of sequences, for which leading SMT solvers like Z3 [6,38] and cvc5 [4] already provide some basic support for over a decade. The basic design of sequence theories, as done in Z3 and cvc5, as well as in other formalisms like symbolic automata [15], is in fact quite natural. That is, sequence theories can be thought of as extensions of theories of strings with an infinite alphabet of letters, together with a corresponding alphabet theory, e.g. Linear Integer Arithmetic (LIA) for reasoning about sequences of integers. Despite this, very little is known about what is decidable over theories of sequences.

In the case of finite alphabets, sequence theories become theories over strings, in which a lot of progress has been made in the last few decades, barring the long-standing open problem of string equations with length constraints (e.g. see [26]). For example, it is known that the existential theory of concatenation over strings with regular constraints is decidable (in fact, PSpace-complete), e.g., see [17,29,36,40,43]. Here, a *regular constraint* takes the form $x \in L(E)$, where E is a regular expression, mandating that the expression E matches the string represented by x. In addition, several natural syntactic restrictions — including straight-line, acylicity, and chain-free (e.g. [1,2,5,11,12,26,35]) — have been identified, with which string constraints remain decidable in the presence of more complex string functions (e.g. transducers, replace-all, reverse, etc.). In the case of infinite alphabets, only a handful of results are available. Furia [25] showed that the existential theory of sequence equations over the alphabet theory of LIA is decidable by a reduction to the existential theory of concatenation over strings (over a finite alphabet) *without regular constraints*. Loosely speaking, a number (e.g. 4) can be represented as a string in unary (e.g. 1111), and addition is then simulated by concatenation. Therefore, his decidability result does not extend to other data domains and alphabet theories. Wang et al. [45] define an extension of the array property fragment [9] with concatenation. This fragment imposes strong restrictions, however, on the equations between sequences (here called finite arrays) that can be considered.

"Regular Constraints" Over Sequences. One answer of what a regular constraint is over sequences is provided by *automata modulo theories*. Automata modulo theories [15,16] are an elegant framework that can be used to capture the notion of regular constraints over sequences: Fix an alphabet theory T that forms a Boolean algebra; this is satisfied by virtually all existing SMT theories. In this framework, one uses formulas in T to capture multiple (possibly infinitely many)

transitions of an automaton. More precisely, between two states in a *symbolic automaton* one associates a unary[1] formula $\varphi(x) \in T$. For example, $q \rightarrow_\varphi q'$ with $\varphi := x \equiv 0 \pmod 2$ over LIA corresponds to all transitions $q \rightarrow_i q'$ with any even number i. Despite their nice properties, it is known that many simple languages cannot be captured using symbolic automata; e.g., one cannot express the language consisting of sequences containing the same even number i *throughout* the sequence.

There are essentially two (expressively incomparable) extensions of symbolic automata that address the aforementioned problem: (i) Symbolic Register Automata (SRA) [14] and (ii) Parametric Automata (PA) [21,23,24]. The model SRA was obtained by combining register automata [31] and symbolic automata. The model PA extends symbolic automata by allowing *free variables* (a.k.a. *parameters*) in the transition guards, i.e., the guard will be of the form $\varphi(x, \bar{p})$, for parameters $\bar{p}$. In an accepting path of PA, a parameter p used in multiple transitions has to be instantiated with the same value, which enables comparisons of different positions in an input sequence. For example, we can assert that only sequences of the form i^*, for an even number i, are accepted by the PA with a single transition $q \rightarrow_\varphi q$ with $\varphi(x,p) := x = p \wedge x \equiv 0 \pmod 2$ and q being the start and final state. PA can also be construed as an extension of both variable automata [27] and symbolic automata. SRA and PA are not comparable: while parameters can be construed as read-only registers, SRA can only compare two different positions using equality, while PA may use a general formula in the theory in such a comparison (e.g., order).

Contributions. The main contribution of this paper is to provide *the first decidable fragments of a theory of sequences parameterized in the element theory.* In particular, we show how to leverage string solvers to solve theories over sequences. We believe this is especially interesting, in view of the plethora of existing string solvers developed in the last 10 years (e.g. see the survey [3]). This opens up new possibilities for verification tasks to be automated; in particular, we show how verification conditions for Quicksort, as well as Bakery and Dijkstra protocols, can be captured in our sequence theory. This formalization was done in the style of *regular model checking* [8,34], whose extension to infinite alphabets has been a longstanding challenge in the field. We also provide a new (dedicated) sequence solver SeCo We detail our results below.

We first show that the quantifier-free theory of sequences with concatenation and PA as regular constraints is decidable. Assuming that the theory is solvable in PSpace (which is reasonable for most SMT theories), we show that our algorithm runs in ExpSpace (i.e., double-exponential time and exponential space). We also identify conditions on the SMT theory T under which PSpace can be achieved and as an example show that Linear Real Arithmetic (LRA) satisfies those conditions. This matches the PSpace-completeness of the theory of strings with concatenation and regular constraints [18].

We consider three different variants/extensions:

[1] This can be generalized to any arity, which has to be set uniformly for the automaton.

(i) *Add length constraints.* Length constraints (e.g., $|\mathbf{x}| = |\mathbf{y}|$ for two sequence variables $\mathbf{x}, \mathbf{y}$) are often considered in the context of string theories, but the decidability of the resulting theory (i.e., strings with concatenation and length constraints) is still a long-standing open problem [26]. We show that the case for sequences is Turing-equivalent to the string case.

(ii) *Use SRA instead of PA.* We show that the resulting theory of sequences is undecidable, even over the alphabet theory T of equality.

(iii) *Add symbolic transducers.* Symbolic transducers [15,16] extend finite-state input/output transducers in the same way that symbolic automata extend finite-state automata. To obtain decidability, we consider formulas satisfying the straight-line restriction that was defined over strings theories [35]. We show that the resulting theory is decidable in 2-ExpTime and is ExpSpace-hard, if T is solvable in PSpace.

We have implemented the solver SeCo based on our algorithms, and demonstrated its efficacy on two classes of benchmarks: (i) invariant checking on array-manipulating programs and parameterized systems, and (ii) benchmarks on Symbolic Register Automata (SRA) from [14]. For the first benchmarks, we model as sequence constraints invariants for QuickSort, Dijkstra's Self-Stabilizing Protocol [20] and Lamport's Bakery Algorithm [33]. For (ii), we solve decision problems for SRA on benchmarks of [14] such as emptiness, equivalence and inclusion on regular expressions with back-references. We report promising experimental results: our solver SeCo is up to three orders of magnitude faster than the SRA solver in [14].

Organization. We provide a motivating example of sequence theories in Sect. 2. Section 3 contains the syntax and semantics of the sequence constraint language, as well as some basic algorithmic results. We deal with equational and regular constraints in Sect. 4. In Sect. 5, we deal with the decidable fragments with equational constraints, regular constraints, and transducers. We deal with extensions of these languages with length and SRA constraints in Sect. 6. In Sect. 7 we report our implementation and experimental results. We conclude in Sect. 8. Missing details and proofs can be found in the full version.

2 Motivating Example

We illustrate the use of sequence theories in verification using a implementation of QuickSort [28], shown in Listing 1. The example uses the Java Streams API and resembles typical implementations of QuickSort in functional languages; the program uses high-level operations on streams and lists like *filter* and *concatenation.* As we show, the data types and operations can naturally be modelled using a theory of sequences over integer arithmetic, and our results imply decidability of checks that would be done by a verification system.

The function `quickSort` processes a given list `l` by picking the first element as the pivot `p`, then creating two sub-lists `left`, `right` in which all numbers

```
/*@
 * ensures \forall int i; \result.contains(i) == l.contains(i);
 */
public static List<Integer> quickSort(List<Integer> l) {
  if (l.size() < 1) return l;
  Integer p = l.get(0);
  List<Integer> left   = l.stream().filter(i -> i < p)
                          .collect(Collectors.toList());
  List<Integer> right  = l.stream().skip(1).filter(i -> i >= p)
                          .collect(Collectors.toList());
  List<Integer> result = quickSort(left);
  result.add(p); result.addAll(quickSort(right));
  return result;
}
```

Listing 1. Implementation of QuickSort with Java Streams.

≥p (resp., <p) have been eliminated. The function `quickSort` is then recursively invoked on the two sub-lists, and the results are finally concatenated and returned.

We focus on the verification of the post-condition shown in the beginning of Listing 1: sorting does not change the set of elements contained in the input list. This is a weaker form of the permutation property of sorting algorithms, and as such known to be challenging for verification methods (e.g., [42]). Sortedness of the result list can be stated and verified in a similar way, but is not considered here. Following the classical design-by-contract approach [37], to verify the partial correctness of the function it is enough to show that the post-condition is established in any top-level call of the function, assuming that the post-condition holds for all recursive calls. For the case of non-empty lists, the verification condition, expressed in our logic, is:

$$\begin{pmatrix} \mathbf{left} = T_{<\mathbf{l}_0}(\mathbf{l}) \wedge \mathbf{right} = T_{\geq \mathbf{l}_0}(\mathit{skip}_1(\mathbf{l})) \wedge \\ \forall i.\,(i \in \mathbf{left} \leftrightarrow i \in \mathbf{left}') \wedge \forall i.\,(i \in \mathbf{right} \leftrightarrow i \in \mathbf{right}') \wedge \\ \mathbf{res} = \mathbf{left}' \,.\, [\mathbf{l}_0] \,.\, \mathbf{right}' \end{pmatrix} \\ \rightarrow \forall i.\,(i \in \mathbf{l} \leftrightarrow i \in \mathbf{res})$$

The variables $\mathbf{l}, \mathbf{res}, \mathbf{left}, \mathbf{right}, \mathbf{left}', \mathbf{right}'$ range over sequences of integers, while i is a bound integer variable. The formula uses several operators that a useful sequence theory has to provide: (i) $\mathbf{l}_0$: the first element of input list $\mathbf{l}$; (ii) $\in$ and $\notin$: membership and non-membership of an integer in a list, which can be expressed using symbolic parametric automata; (iii) skip_1, $T_{<\mathbf{l}_0}$, $T_{\geq \mathbf{l}_0}$: sequence-to-sequence functions, which can be represented using symbolic parametric transducers; (iv) $\cdot\,.\,\cdot$: concatenation of several sequences. The formula otherwise is a direct model of the method in Listing 1; the variables $\mathbf{left}', \mathbf{right}'$ are the results of the recursive calls, and concatenated to obtain the result sequence.

In addition, the formula contains quantifiers. To demonstrate validity of the formula, it is enough to eliminate the last quantifier $\forall i$ by instantiating with a Skolem symbol k, and then instantiate the other quantifiers (left of the implication) with the same k:

$$\begin{pmatrix} \mathbf{left} = T_{<\mathbf{l}_0}(\mathbf{l}) \wedge \mathbf{right} = T_{\geq \mathbf{l}_0}(skip_1(\mathbf{l})) \wedge \\ (k \in \mathbf{left} \leftrightarrow k \in \mathbf{left}') \wedge (k \in \mathbf{right} \leftrightarrow k \in \mathbf{right}') \wedge \\ \mathbf{res} = \mathbf{left}' \,.\, [\mathbf{l}_0] \,.\, \mathbf{right}' \end{pmatrix} \rightarrow (k \in \mathbf{l} \leftrightarrow k \in \mathbf{res})$$

As one of the results of this paper, we prove that this final formula is in a decidable logic. The formula can be rewritten to a disjunction of straight-line formulas, and shown to be valid using the decision procedure presented in Sect. 5.

3 Models

In this section, we will define our sequence constraint language, and prove some basic results regarding various constraints in the language. The definition is a natural generalization of string constraints (e.g. see [12,17,26,29,35]) by employing an alphabet theory (a.k.a. element theory), as is done in symbolic automata and automata modulo theories [15,16,44].

For simplicity, our definitions will follow a model-theoretic approach. Let σ be a vocabulary. We fix a σ-structure $\mathfrak{S} = (D; I)$, where D can be a finite or an infinite set (i.e., the universe) and I maps each function/relation symbol in σ to a function/relation over D. The elements of our sequences will range over D. We assume that the quantifier-free theory $T_{\mathfrak{S}}$ over $\mathfrak{S}$ (including equality) is decidable. Examples of such $T_{\mathfrak{S}}$ are abound from SMT, e.g., LRA and LIA. We write T instead of $T_{\mathfrak{S}}$, when $\mathfrak{S}$ is clear. Our quantifier-free formula will use *uninterpreted* T*-constants* $a, b, c, \ldots$, and may also use variables $x, y, z, \ldots$. (The distinction between uninterpreted constants and variables is made only for the purpose of presentation of sequence constraints, as will be clear shortly.) We use $\mathcal{C}$ to denote the set of all uninterpreted T-constants. A formula φ is satisfiable if there is an assignment that maps the uninterpreted constants and variables to concrete values in D such that the formula becomes true in $\mathfrak{S}$.

Next, we define how we lift T to sequence constraints, using T as the *alphabet theory* (a.k.a. *element theory*). As in the case of strings (over a finite alphabet), we use standard notation like D^* to refer to the set of all sequences over D. By default, elements of D^* are written as standard in mathematics, e.g., $7, 8, 100$, when $D = \mathbb{Z}$. Sometimes we will disambiguate them by using brackets, e.g., $(7, 8, 100)$ or $[7, 8, 100]$. We will use the symbol s (with/without subscript) to refer to concrete sequences (i.e., a member of D^*). We will use $\mathbf{x}, \mathbf{y}, \mathbf{z}$ to refer to T-sequence variables. Let $\mathcal{V}$ denote the set of all T-sequence variables, and $\Gamma := \mathcal{C} \cup D$. We will define constraint languages syntactically at the beginning, and will instantiate them to specific sequence operations. The theory T^* of T-sequences consists of the following constraints:

$$\varphi ::= R(\mathbf{x}_1, \ldots, \mathbf{x}_r) \mid \varphi \wedge \varphi$$

where R is an r-ary relation symbol. In our definition of each atom R below, we will specify if an assignment μ, which maps each $\mathbf{x}_i$ to a T-sequence and each uninterpreted constant to a T-element, satisfies R. If μ satisfies all atoms, we say that μ is a *solution* and the *satisfiability problem* is to decide whether there is a solution for a given φ.

A few remarks about the missing boolean operators in the constraint language above are in order. Disjunctions can be handled easily using the DPLL(T) framework (e.g. see [32]), so we have kept our theory conjunctive. As in the case of strings, negations are usually handled separately because they can sometimes (but not in all cases) be eliminated while preserving decidability.

Equational Constraints. A *T-sequence equation* is of the form

$$L = R$$

where each of L and R is a concatenation of concrete T-elements, uninterpreted constants, and T-sequence variables. That is, if $\Theta := \Gamma \cup \mathcal{V}$, then $L, R \in \Theta^*$.

For example, in the equation

$$0.1.\mathbf{x} = \mathbf{x}.0.1$$

the set of all solutions is of the form $\mathbf{x} \mapsto (01)^*$. To make this more formal, we extend each assignment μ to a homomorphism on Θ^*. We write $\mu \models L = R$ if $\mu(L) = \mu(R)$. Notice that this definition is just direct extension of that of *word equations* (e.g. see [17]), i.e., when the domain D is finite.

In most cases the inequality constraints $L \neq R$ can be reduced to equality in our case this requires also element constraints, described below.

Element Constraints. We allow T-formulas to constrain the uninterpreted constants. More precisely, given a T-sentence (i.e., no free variables) φ that uses $\mathcal{C}$ as uninterpreted constants, we obtain a proposition P (i.e., 0-ary relation) that $\mu \models P$ iff $T \models_\mu \varphi$.

Negations in the equational constraints can be removed just like in the case of strings, i.e., by means of additional variables/constants and element constraints. For example, $\mathbf{x} \neq \mathbf{y}$ can be replaced by $(\mathbf{x} = \mathbf{z}a\mathbf{x}' \wedge \mathbf{y} = \mathbf{z}b\mathbf{y}' \wedge a \neq b) \vee \mathbf{x} = \mathbf{y}a\mathbf{z} \vee \mathbf{x}a\mathbf{z} = \mathbf{y}$. Notice that $a \neq b$ is a T-formula because we assume the equality symbol in T.

Regular Constraints. Over strings, regular constraints are simply unary constraints $U(\mathbf{x})$, where U is an automaton. The interpretation is $\mathbf{x}$ is in the language of U. We define an analogue of regular constraints over sequences using *parametric automata* [21,23,24], which generalize both symbolic automata [15,16] and variable automata [27].

A *parametric automaton* (PA) over T is of the form $\mathcal{A} = (\mathcal{X}, Q, \Delta, q_0, F)$, where $\mathcal{X}$ is a finite set of parameters, Q is a finite set of control states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq_{\text{fin}} Q \times T(\mathit{curr}, \mathcal{X}) \times Q$. Here, *parameters* are simply uninterpreted T-constants, i.e., $\mathcal{X} \subseteq \mathcal{C}$. Formulas that appear in transitions in Δ will be referred to as *guards*, since they restrict which transitions are enabled at a given state. Note that *curr* is an uninterpreted constant that refers to the "current" position in the sequence. The semantics is quite simply defined: a sequence $(d_1, d_2, \ldots, d_n)$ is in the language of $\mathcal{A}$ under the assignment of parameters μ, written as $(d_1, \ldots, d_n) \in L_\mu(\mathcal{A})$, when there is a sequence of Δ-transitions

$$(q_0, \varphi_1(curr, \mathcal{X}), q_1), (q_1, \varphi_2(curr, \mathcal{X}), q_2), \ldots, (q_{n-1}, \varphi_n(curr, \mathcal{X}), q_n),$$

such that $q_n \in F$ and $T \models \varphi_i(d_i, \mu(\mathcal{X}))$. Finally, for a regular constraint $\mathcal{A}(\mathbf{x})$ is satisfied by μ, when $\mu(\mathbf{x}) \in L_\mu(\mathcal{A})$.

Note, that it is possible to complement a PA $\mathcal{A}$, one has to be careful with the semantics: we treat $\mathcal{A}$ as a symbolic automaton, which are closed under boolean operations [15]. So we are looking for μ such that $\mu(\mathbf{x}) \in \overline{L_\mu(\mathbf{x})}$. What we cannot do using the complementation, is a universal quantification over the parameters; note that already theory of strings with universal and existential quantifiers is undecidable.

We state next a lemma showing that PAs using only "local" parameters, together with equational constraints, can encode the constraint language that we have defined so far.

Lemma 1. *Satisfiability of sequence constraints with equation, element, and regular constraints can be reduced in polynomial-time to satisfiability of sequence constraints with equation and regular constraints (i.e., without element constraints). Furthermore, it can be assumed that no two regular constraints share any parameter.*

Proposition 1. *Assume that T is solvable in* NP *(resp.* PSpace*). Then, deciding nonemptiness of a parametric automaton over T is in* NP *(resp.* PSpace*).*

The proof is standard (e.g. see [21,23,24]), and only sketched here. The algorithm first nondeterministically guesses a simple path in the automaton $\mathcal{A}$ from an initial state q_0 to some final state q_F. Let us say that the guards appearing in this path are $\psi_1(curr, \mathcal{X}), \ldots, \psi_k(curr, \mathcal{X})$. We need to check if this path is realizable by checking T-satisfiability of

$$\exists \mathcal{X}. \bigwedge_{i=1}^{k} \exists curr. (\psi_i(curr, \mathcal{X})).$$

It is easy to see that this is an NP (resp. NPSPACE = PSpace) procedure.

Parametric Transducers. We define a suitable extension of symbolic transducers over parameters following the definition from Veanes et al. [44]. A *transducer constraint* is of the form $\mathbf{y} = \mathcal{T}(\mathbf{x})$, for a parametric transducer $\mathcal{T}$. A *parametric transducer* over T is of the form $\mathcal{T} = (\mathcal{X}, Q, \Delta, q_0, F)$, where $\mathcal{X}$, Q, q_0, F are just like in parametric automata. Unlike parametric automata, Δ is a finite set of tuples of the form $(p, (\varphi, \mathbf{w}), q)$, where (p, φ, q) is a standard transition in parametric automaton, and $\mathbf{w}$ is a (possibly empty) sequence of T-terms over variable *curr* and constants $\mathcal{X}$, e.g., $\mathbf{w} = (\mathit{curr}+7, \mathit{curr}+2)$. One can think of $\mathbf{w}$ as the output produced by the transition. Given an assignment μ of parameters and the sequence variables, the constraint $\mathbf{y} = \mathcal{T}(\mathbf{x})$ is satisfied when there is a sequence of Δ-transitions

$$(q_0, \varphi_1(\mathit{curr}, \mathcal{X}), \mathbf{w}_1, q_1), (q_1, \varphi_2(\mathit{curr}, \mathcal{X}), \mathbf{w}_2, q_2), \ldots (q_{n-1}, \varphi_n(\mathit{curr}, \mathcal{X}), \mathbf{w}_n, q_n),$$

such that $q_n \in F$ and $T \models \varphi_i(d_i, \mu(\mathcal{X}))$, where $\mu(\mathbf{x}) = (d_1, \ldots, d_n)$, and finally

$$\mu(\mathbf{y}) = \mu_1(\mathbf{w}_1) \cdots \mu_n(\mathbf{w}_n)$$

where μ_i is μ but maps *curr* to d_i. The definition assumes that μ_i is extended to terms and concatenation thereof by homomorphism, e.g., in LRA, if $\mathbf{w}_1 = (\mathit{curr}+7, \mathit{curr}+2)$ and μ_1 maps *curr* to 10, then $\mathbf{w}_1$ will get mapped to 17, 12. Given a set $S \subseteq D^*$ and an assignment μ (mapping the constants to D), we define the *pre-image* $\mathcal{T}_\mu^{-1}(S)$ of S under $\mathcal{T}$ with respect to μ as the set of sequences $\mathbf{w} \in D^*$ such that $\mathbf{w}' = \mathcal{T}(\mathbf{w})$ holds with respect to μ.

4 Solving Equational and Regular Constraints

Here we present results on solving equational constraints, together with regular constraints, by a reduction to the string case, for which a wealth of results are already available. In general, this reduction causes an exponential blow-up in the resulting string constraint, which we show to be unavoidable in general. That said, we also provide a more refined analysis in the case when the underlying theory is LRA, where we can avoid this exponential blow-up.

Prelude: The Case of Strings. We start with some known results about the case of strings. The satisfiability of word equations with regular constraints is PSPACE-complete [18,19]. This upper bound can be extended to full quantifier-free theory [10]. When no regular constraints are given, the problem is only known to be NP-hard, and it is widely believed to be in NP. In the absence of regular constraints, without loss of generality Γ can be assumed to contain only letters from the equations; this is not the case in presence of regular constraints. The algorithm solving word equations [19] does not need an explicit access to Γ: it is enough to know whether there is a letter which labels a given set of transitions in the NFAs used in the regular constraints. In principle, there could be exponentially many different (i.e., inducing different transitions in the NFAs) letters. When oracle access to such alphabet is provided, the satisfiability can still be decided in PSPACE: while not explicitly claimed, this is exactly the scenario in [19, Sect. 5.2]

Other constraints are also considered for word equations; perhaps the most widely known are the length constraints, which are of the form: $\sum_{x\in\mathcal{V}} a_x \cdot |x| \leq c$, where $\{a_x\}_{x\in\mathcal{V}}, c$ are integer constants and $|x|$ denotes the length $|\mu(x)|$, with an obvious semantics. It is an open problem, whether word equations with length constraints are decidable, see [26].

Reduction to Word Equations. We assume Lemma 1, i.e. that the parameters used for different automata-based constraints are pairwise different. In particular, when looking for a satisfying assignment μ we can first fix assignment for $\mathcal{X}$ and then try to extend it to $\mathcal{V}$. To avoid confusion, we call this partial assignment $\pi : \mathcal{X} \to D$.

Consider a set Φ of all atoms in all guards in the regular constraints together with the set of formulas $\{x = c\}$ over all constants $c \in D$ that appear in all equational constraints and the negations of both types of formulas. Fix an assignment $\pi : \mathcal{X} \to D$. The type $\mathrm{type}_\pi(a)$ of a (under assignment π) is the set of formulas in Φ satisfied by a, i.e. $\{\varphi \in \Phi \ : \ \varphi(\pi(\mathcal{X}), a) \text{ holds}\}$. Clearly there are at most exponentially many different types (for a fixed π). A type t is realizable (for π) when $t = \mathrm{type}_\pi(a)$ and it is realized by a.

If the constraints are satisfiable (for some parameters assignment π) then they are satisfiable over a subset $D_\pi {\subseteq_{\mathrm{fin}}} D$, in the sense that we assign uniterpreted constants elements from D_π and T-sequence variables elements of D_π^*, where D_π is created by taking (arbitrarily) one element of a realizable type. Note that for each constant c in the equational constraints there is a formula "$x = c$" in Φ, in particular $\mathrm{type}_\pi(c)$ is realizable (only by c) and so $c \in D_\pi$.

Lemma 2. *Given a system of constraints and a parameter assignment π let $D_\pi \subseteq D$ be obtained by choosing (arbitrarily) for each realizable type a single element of this type. Then the set of constraints is satisfiable (for π) over D if and only if they are satisfiable (for π) over D_π. To be more precise, there is a letter-to-letter homomorphism $\psi : D^* \to D_\pi^*$ such that if μ is a solution of a system of constraints then $\psi \circ \mu$ is also a solution.*

The proof can be found in the full version, its intuition is clear: we map each letter $a \in D$ to the unique letter in D_π of the same type.

Once the assignment is fixed (to π) and domain restricted to a finite set (D_π), the equational and regular constraints reduce to word equations with regular constraints: treat D_π as a finite alphabet, for a parametric automaton $\mathcal{A} = (\mathcal{X}, Q, \Delta, q_0, F)$ create an NFA $\mathcal{A}' = (D_\pi, Q, \Delta', q_0, F)$, i.e. over the alphabet D_π, with the same set of states Q, same starting state q_0 and accepting states F and the relation defined as $(q, a, q') \in \Delta'$ if and only if there is $(q, \varphi(\mathit{curr}, \mathcal{X}), q') \in \Delta$ such that $\varphi(a, \pi(\mathcal{X}))$ holds, i.e. we can move from q to q' by a in $\mathcal{A}'$ if and only if we can make this move in $\mathcal{A}$ under assignment π. Clearly, from the construction

Lemma 3. *Given an assignment of parameters π let D_μ be a set from Lemma 2, $\mathcal{A}$ be a parametric automaton and $\mathcal{A}'$ the automaton as constructed above. Then*

$$L_\pi(\mathcal{A}) \cap D_\pi^* = L(\mathcal{A}') \ .$$

We can rewrite the parametric automata-constraints with regular constraints and treat equational constraints as word equations (over the finite alphabet D_π). From Lemma 2 and Lemma 3 it follows that the original constraints have a solution for assignment π if and only if the constructed system of constraints has a solution. Therefore once the appropriate assignment π is fixed, the validity of constraints can be verified [19]. It turns out that we do not need the actual π, it is enough to know which types are realisable for it, which translates to an exponential-size formula. We will use letter τ to denote subset of Φ; the idea is that $\tau = \{\text{type}_\pi(a) \ : \ a \in D\} \subseteq 2^\Phi$ and if different π, π' give the same sets of realizable types, then they both yield a satisfying assignment or both not. Hence it is enough to focus on τ and not on actual π.

Lemma 4. *Given a system of equational and regular constraints we can non-deterministically reduce them to a formula of a form*

$$\exists_{t \in \tau} a_t \in D. \, \exists \mathcal{X} \in D^+. \bigwedge_{t \in \tau} \bigwedge_{\varphi \in t} \varphi(\mathcal{X}, a_t) \ , \tag{1}$$

where $\tau \subseteq 2^\Phi$ is of at most exponential size, and a system of word equations with regular constraints of linear size and over an $|\tau|$-size alphabet, using auxiliary $\mathcal{O}(n|\tau|)$ space. The solution of the latter word equations (for which also (1) holds) are solutions of the original system, by appropriate identifications of symbols.

Proof. We guess the set τ of types of the assignment of parameters π, i.e. $\tau = \{\text{type}_\pi(a) \ : \ a \in D\}$ such that there is an assignment μ extending π; note that as Φ has linearly many atoms and $\tau \subseteq 2^\Phi$, then $|\tau|$ may be of exponential size, in general. The (1) verifies the guess: we validate whether there are values of $\mathcal{X}$ such that for each type $t \in \tau$ there is a value a such that $\text{type}_\pi(a) = t$.

Let D_π be a set having one symbol per every type in τ, as in Lemma 2; note that this includes all constants in the equational constraints. The algorithm will not have access to particular values, instead we store each $t \in \tau$, say as a bitvector describing which atoms in Φ this letter satisfies. In particular, $|D_\pi| = |\tau|$ and it is at most exponential. In the following we will consider only solutions over D_π.

For each $a \in D_\pi$ we can validate, which transitions in $\mathcal{A}$ it can take: the transition is labelled by a guard which is a conjunction of atoms from Φ and either each such atom is in $\text{type}_\pi(a)$ or not. Hence we can treat $\mathcal{A}$ as an NFA for D_π. We do not need to construct nor store it, we can use $\mathcal{A}$: when we want to make a transition by $\varphi(\mathcal{X}, a)$ we look up, whether each atom of φ is in $\text{type}_\pi(a)$ or not. Similarly, the constraint $\mathcal{A}(\mathbf{x})$ is restricted to $\mathbf{x} \in L_\pi(\mathcal{A})$ and for $\mathbf{x} \in D_\pi^*$ this is a usual regular constraint.

We treat equational constraints as word equations over alphabet D_π.

Concerning the correctness of the reduction: if the system of word equations (with regular constraints) is satisfiable and the formula (1) is also satisfiable, then there is a satisfying assignment μ over D_π and D_π^* in particular, there is an assignment of parameters for which there are letters of the given types (note that in principle it could be that μ induces more types, i.e. there is a value a such that $\text{type}_\mu(a) \notin \tau$ and so it is not represented in D_π, but this is fine: enlarging the alphabet cannot invalidate a solution), i.e. the transitions for a_t in the automata after the reduction are the same as in the corresponding parametric automata for the assignment π, this is guaranteed by the satisfiability of (1) and the way we construct the instance, see Lemma 3.

On the other hand, when there is a solution of the input constraints, there is one for some assignment of parameters π. Hence, by Lemma 2, there is a solution over D_π. The algorithm guesses $\tau = \{\mathrm{type}_\pi(a) \,:\, a \in D\}$ and (1) is true for it. Then by Lemma 2 there is a solution over D_π as constructed in the reduction and by Lemma 3 the regular constraints define the same subsets of D_π^* both when interpreted as parametric automata and NFAs. □

Theorem 1. *If theory T is in* PSpace *then sequence constraints are in* ExpSpace.

If τ is polynomial size and the formula (1) *can be verified in* PSpace*, then sequence constraints can be verified in* PSpace.

One of the difficulties in deciding sequence constraints using the word equations approach is the size of set of realizable types τ, which could be exponential. For some concrete theories it is known to be smaller and thus a lower upper bound on complexity follows. For instance, it is easy to show that for LRA there are linearly many realizable types, which implies a PSpace upper bound.

Corollary 1. *Sequence constraints for Linear Real Arithmetic are in* PSpace.

In general, the ExpSpace upper bound from Theorem 1 cannot be improved, as even non-emptiness of intersection of parametric automata is ExpSpace-complete for some theories decidable in PSpace. This is in contrast to the case of symbolic automata, for which the non-emptiness of intersection (for a theory T decidable in PSpace) is in PSpace. This shows the importance of parameters in our lower bound proof.

Theorem 2. *There are theories with existential fragment decidable in* PSpace *and whose non-emptiness of intersection of parametric automata is* ExpSpace*-complete.*

When no regular constraints are allowed, we can solve the equational and element constraints in PSpace (note that we do not use Lemma 1).

Theorem 3. *For a theory T decidable in* PSpace*, the element and equational constraints (so no regular constraints) can be decided in* PSpace.

5 Algorithm for Straight-Line Formulas

It is known that adding finite transducers into word equations results in an undecidable model (e.g. see [35]). Therefore, we extend the *straight-line restriction* [12,35] to sequences, and show that it suffices to recover decidability for equational constraints, together with regular and transducer constraints. In fact, we will show that deciding problems in the straight-line fragment is solvable in doubly exponential time and is ExpSpace-hard, if T is solvable in PSpace. It has been observed that the straight-line fragment for the theory of strings already covers many interesting benchmarks [12,35], and similarly many properties of sequence-manipulating programs can be proven using the fragment, including the QuickSort example from Sect. 2 and other benchmarks shown in Sect. 7.

The Straight-Line Fragment SL. We start by defining recognizable formulas over sequences, followed by the syntactic and semantic restrictions on our constraint language. This definition follows closely the definition of recognizable relations over finite alphabets, except that we replace finite automata with parametric automata.

Definition 1 (Recognizable formula). *A formula* $R(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_r)$ *is* recognizable *if it is equivalent to a positive Boolean combination of regular constraints.*

Note that this is simply a generalization of regular constraints to multiple variables, i.e., 1-ary recognizable formula can be turned into a regular constraint, which is closed under intersection and union.

To define the straight-line fragment, we use the approach of [12]; that is, the fragment is defined in terms of "feasibility of a symbolic execution". Here, a *symbolic execution* is just a sequence of assignments and assertions, whereas the *feasibility* problem amounts to deciding whether there are concrete values of the variables so that the symbolic execution can be run and none of the assertions are violated. We now make this intuition formal. A symbolic execution is syntactically generated by the following grammar:

$$S \quad ::= \quad \mathbf{y} := f(\mathbf{x}_1, \ldots, \mathbf{x}_k, \mathcal{X}) \mid \mathbf{assert}(R(\mathbf{x}_1, \ldots, \mathbf{x}_r)) \mid \mathbf{assert}(\varphi) \mid S; S \tag{2}$$

where $f : (D^*)^k \times D^{|\mathcal{X}|} \to D$ is a function, R are recognizable formulas, and φ are element constraints.

The symbolic execution S can be turned into a sequence constraint as follows. Firstly, we can turn S into the standard *Static Single Assignment (SSA)* form by means of introducing new variables on the left-hand-side of an assignment. For example, $\mathbf{y} := f(\mathbf{x}); \mathbf{y} := g(\mathbf{z})$ becomes $\mathbf{y} := f(\mathbf{x}_1); \mathbf{y}' := g(\mathbf{z})$. Then, in the resulting constraint, each variable appears *at most once* on the left-hand-side of an assignment. That way, we can simply replace each assignment symbol $:=$ with an equality symbol $=$. We then treat each sequential composition as the conjunction operator $\wedge$ and assertion as a conjunct. Note that individual assertions are already sequence constraints. Next, we define how an interpretation μ satisfies the constraint $\mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_r, \mathcal{X})$:

$$\mu \models \mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_r, \mathcal{X}) \quad \text{iff} \quad \mu(\mathbf{y}) = f(\mu(\mathbf{x}_1), \ldots, \mu(\mathbf{x}_r), \mu(\mathcal{X})).$$

Note that '=' on the l.h.s. is syntactic, while the '=' on the r.h.s. is in the metalanguage. The definition of the semantics of the language is now inherited from Sect. 3.

In addition to the syntactic restrictions, we also need a semantic condition: in our language, we only permit functions f such that the pre-image of each regular constraint under f is effectively a recognizable formula:

(RegInvRel) A function f is permitted if for each regular constraint $\mathcal{A}(\mathbf{y})$, it is possible to compute a recognizable formula that is equivalent to the formula $\exists \mathbf{y} : \mathcal{A}(\mathbf{y}) \wedge \mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_r, \mathcal{X})$.

Two functions satisfying (RegInvRel) are the concatenation function $\mathbf{x} := \mathbf{y}.\mathbf{z}$ (here $\mathbf{y}$ could be the same as $\mathbf{z}$) and parametric transducers $\mathbf{y} := \mathcal{T}(\mathbf{x})$. We will only use these two functions in the paper, but the result is generalizable to other functions.

Proposition 2. *Given a regular constraint $\mathcal{A}(\boldsymbol{y})$ and a constraint $\boldsymbol{y} = \boldsymbol{x}.\boldsymbol{z}$, we can compute a recognizable formula $\psi(\boldsymbol{x}, \boldsymbol{z})$ equivalent to $\exists \boldsymbol{y} : \mathcal{A}(\boldsymbol{y}) \wedge \boldsymbol{y} = \boldsymbol{x}.\boldsymbol{z}$. Furthermore, this can be achieved in polynomial time.*

The proof of this proposition is exactly the same as in the case of strings, e.g., see [12,35].

Proposition 3. *Given a regular constraint $\mathcal{A}(\boldsymbol{y})$ and a parametric transducer constraint $\boldsymbol{y} = \mathcal{T}(\boldsymbol{x})$, we can compute a regular constraint $\mathcal{A}'(\boldsymbol{x})$ that is equivalent to $\exists \boldsymbol{y} : \mathcal{A}(\boldsymbol{y}) \wedge \boldsymbol{y} = \mathcal{T}(\boldsymbol{x})$. This can be achieved in exponential time.*

The construction in Proposition 3 is essentially the same as the pre-image computation of a symbolic automaton under a symbolic transducer [44]. The complexity is exponential in the maximum number of output symbols of a single transition (i.e. the maximum length of $\mathbf{w}$ in the transducer), which is in practice a small natural number.

The following is our main theorem on the SL fragment with equational constraints, regular constraints, and transducers.

Theorem 4. *If T is solvable in* PSPACE*, then the SL fragment with concatenation and parametric transducers over T is in 2-*EXPTIME *and is* EXPSPACE*-hard.*

Proof. We give a decision procedure. We assume that S is already in SSA (i.e. each variable appears at most once on the left-hand side). Let us assume that S is of the form $S'; \mathbf{y} := f(\mathbf{x}_1, ...\mathbf{x}_r)$, for some symbolic execution S'. Without loss of generality, we may assume that each recognizable constraint is of the form $\mathcal{A}(\mathbf{x})$. This is no limitation: (1) since each R in the assertion is a recognizable formula, we simply have to "guess" one of the implicants for each R, and (2) $\mathbf{assert}(\psi_1 \wedge \psi_2)$ is equivalent to $\mathbf{assert}(\psi_1); \mathbf{assert}(\psi_2)$.

Assume now that $\{\mathcal{A}_1(\mathbf{y}), \ldots, \mathcal{A}_m(\mathbf{y})\}$ are all the regular constraints on $\mathbf{y}$ in S. By our assumption, it is possible to compute a recognizable formula equivalent to

$$\psi(\mathbf{x}_1, \ldots, \mathbf{x}_r) := \exists \mathbf{y} : \bigwedge_{i=1}^{m} \mathcal{A}_i(\mathbf{y}) \wedge \mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_r).$$

There are two ways to see this. The first way is that regular constraints are closed under intersection. This is in general computationally quite expensive because of a product automata construction before applying the pre-image computation. A better way to do this is to observe that ψ is equivalent to the conjunction of ψ_i's over $i = 1, \ldots, m$, where

$$\psi_i := \exists \mathbf{y} : \mathcal{A}_i(\mathbf{y}) \wedge \mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_r).$$

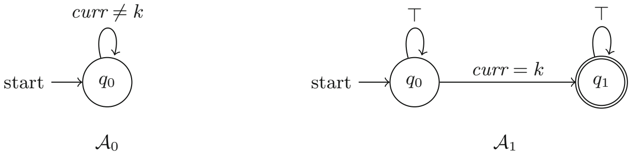


Fig. 1. $\mathcal{A}_0$ accepts all words not containing k and $\mathcal{A}_1$ accepts all words containing k.

By our semantic condition, we can compute recognizable formulas $\psi'_i, \dots, \psi'_m$ equivalent to $\psi_1, \dots, \psi_m$ respectively. Therefore, we simply replace S by

$$S'; \textbf{assert}(\psi'_1); \cdots ; \textbf{assert}(\psi'_m),$$

in which every occurrence of $\mathbf{y}$ has been completely eliminated. Applying the above variable elimination iteratively, we obtain a conjunction of regular constraints. We now end up with a conjunction of regular constraints and element constraints, which as we saw from Sect. 4 is decidable. □

Example 1. We consider the example from Sect. 2 where a weaker form of the permutation property is shown for QuickSort. The formula that has to be proven is a disjunction of straight-line formulas and in the following we execute our procedure only on one disjunct without redundant formulas:

$$\textbf{assert}(\mathcal{A}_0(\textbf{left}')); \textbf{assert}(\mathcal{A}_0(\textbf{right}')); \textbf{res} = \textbf{left}' \,.\, [\mathbf{l}_0] \,.\, \textbf{right}'; \textbf{assert}(\mathcal{A}_1(\textbf{res}))$$

We model $L(\mathcal{A}_1)$ as the language which accepts all words which contain one letter equal to k and $L(\mathcal{A}_0)$ as the language which accepts only words not containing k, where k is an uninterpreted constant, so a single element. See Fig. 1. We begin by removing the operation $\textbf{res} = \textbf{left}' \,.\, [\mathbf{l}_0] \,.\, \textbf{right}'$. The product automaton for all assertions that contain $\textbf{res}$ is just $\mathcal{A}_1$. Hence, we can remove the assertion $\textbf{assert}(\mathcal{A}_1(\textbf{res}))$. The concatenation function . satisfies **RegInvRel** and the pre-image g can be represented by

$$\bigvee_{0 \leq i,j \leq 1} \mathcal{A}_1^{q_0,\{q_i\}}(\textbf{left}') \wedge \mathcal{A}_1^{q_i,\{q_j\}}([\mathbf{l}_0]) \wedge \mathcal{A}_1^{q_j,\{q_1\}}(\textbf{right}'),$$

where $\mathcal{A}_i^{p,F'}$ is $\mathcal{A}_i$ with start state set to p and finals to F'.

In the next step, the assertion g is added to the program and all assertions containing $\textbf{res}$ and the concatenation function are removed.

$$\textbf{assert}(\mathcal{A}_0(\textbf{left}')); \textbf{assert}(\mathcal{A}(\textbf{right}')); \textbf{assert}(g(\textbf{left}', [\mathbf{l}_0], \textbf{right}'))$$

From here, we pick a tuple from g, lets say $i = j = 1$, and obtain

$$\textbf{assert}(\mathcal{A}_0(\textbf{left}')); \textbf{assert}(\mathcal{A}_0(\textbf{right}')); \textbf{assert}(\textbf{left}' \in \mathcal{A}_1^{q_0,\{q_1\}});$$
$$\textbf{assert}([\mathbf{l}_0] \in \mathcal{A}_1^{q_1,\{q_1\}}); \textbf{assert}(\textbf{right}' \in \mathcal{A}_1^{q_1,\{q_1\}})$$

Finally, the product automata $\mathcal{A}_0 \times \mathcal{A}_1^{q_0,\{q_1\}}$ and $\mathcal{A}_0 \times \mathcal{A}_1^{q_0,\{q_1\}}$ are computed for the variables $\mathbf{left}', \mathbf{right}'$ and a non-emptiness check over the product automata and the automaton for $[\mathbf{l}_0]$ is done. The procedure will find no combination of paths for each automaton which can be satisfied, since $\mathbf{left}'$ is forced to accept no words containing k by $\mathcal{A}_0$ and only accepts by reading a k from $\mathcal{A}_1^{q_0,\{q_1\}}$. Next, the procedure needs to exhaust all tuples from $(\mathcal{A}_1^{q_0,\{q_i\}}, \mathcal{A}_1^{q_i,\{q_j\}}, \mathcal{A}_1^{q_j,\{q_1\}})_{0 \le i,j \le 1}$ before it is proven that this disjunct is unsatisfiable.

6 Extensions and Undecidability

Length Constraints. We consider the extension of our model by allowing *length-constraints* on the sequence variables: for each sequence variable $\mathbf{x}$ we consider the associated length variable $\ell_{\mathbf{x}}$, let the set of length variables be $\mathcal{L} = \{\ell_{\mathbf{x}} : \mathbf{x} \in \mathcal{V}\}$, we extend μ to $\mathcal{L}$, it assigns natural numbers to them. The length constraints are of the form $\sum_{\mathbf{x}} a_{\mathbf{x}} \ell_{\mathbf{x}} ?0$, where $? \in \{<, \le, =, \ne, \ge, >\}$ and each $a_{\mathbf{x}}$ is an integer constant, i.e., linear arithmetic formulas on the length-variables. The semantics is natural: we require that $|\mu(\mathbf{x})| = \mu(\ell_{\mathbf{x}})$ (the assigned values are the true lengths of sequences) and that $\mu(\mathcal{L})$ satisfies each length constraint.

There is, however, another possible extensions: if we the theory $T_{\mathfrak{S}}$ is the Presburger arithmetic, then the parameter automata could use the values $\ell_{\mathbf{x}}$. We first deal with a more generic, though restricted case, when this is not allowed: then all reductions from Sect. 4 generalize and we can reduce to the word equations with regular and length constraints. However, the decidability status of this problem is unknown. When we consider Presburger arithmetic and allow the automata to employ the length variables, then it turns out that we can interpret the formula (1) as a collection of length constraints, and again we reduce to word equations with regular and length constraints.

Automata Oblivious of Lengths. We first consider the setting, in which the length variables $\mathcal{L}$ can only be used in length constraints. It is routine to verify that the reduction from Sect. 4 generalize to the case of length constraints: it is possible to first fix μ for parameters, calling it again π. Then Lemma 2 shows that each solution μ can be mapped by a letter-to-letter homomorphism to a finite alphabet D_π, and this mapping preserves the satisfiability/unsatisfiability of length constraints, so Lemma 2 still holds when also length constraints are allowed. Similarly, Lemma 3 is also not affected by the length constraints and finally Lemma 4 deals with regular and equational constraints, ignoring the other possible constraints and the length of substitutions for variables are the same. Hence it holds also when the length constraints are allowed then the resulting word equations use regular and length constraints.

Unfortunately, the decidability of word equations with linear length constraints (even without regular constraints) is a notorious open problem. Thus instead of decidability, we get Turing-equivalent problems.

Theorem 5. *Deciding regular, equational and length constraints for T-sequences of a decidable theory T is Turing-equivalent to word equations with regular and length constraints.*

Automata Aware of the Sequence Lengths. We now consider the case when the underlying theory $T_{\mathfrak{S}}$ is the Presburger arithmetic, i.e. $\mathfrak{S}$ is the natural numbers and we can use addition, constants $0, 1$ and comparisons (and variables). The additional functionality of the parametric automaton $\mathcal{A}$ is that $\Delta \subseteq_{\text{fin}} Q \times T(\mathit{curr}, \mathcal{X}, \mathcal{L}) \times Q$, i.e. the guards can also use the length variables; the semantics is extended in the natural way.

Then the type $\text{type}_\pi(a)$ of $a \in \mathbb{N}$ now depends on μ values on $\mathcal{X}$ and $\mathcal{L}$, hence we denote by π the restriction of μ to $\mathcal{X} \cup \mathcal{L}$. Then Lemma 2, 3 still hold, when we fix π. Similarly, Lemma 4 holds, but the analogue of (1) now uses also the length variables, which are also used in the length constraints. Such a formula can be seen as a collection of length constraints for original length variables $\mathcal{L}$ as well as length variables $\mathcal{X} \cup \{a_t \ : \ t \in \tau\}$. Hence we validate this formula as part of the word equations with length constraints. Note that a_t has two roles: as a letter in D_π and as a length variable. However, the connection is encoded in the formula from the reduction (analogue of (1)) and we can use two different sets of symbols.

Theorem 6. *Deciding conjunction of regular, equational and length constraints for sequences of natural numbers with Presburger arithmetic, where the regular constraints can use length variables, is Turing-equivalent to word equations with regular and (up to exponentially many) length constraints.*

Undecidability of Register Automata Constraints. One could use more powerful automata for regular constraints; one such popular model are register automata; informally, such automaton has k registers $r_1, \ldots, r_k$ and its transition depends on state and a value of formula using the registers and *curr*: the read value [23]; note that the registers can be updated: to *curr* or to one of register's values; this is specified in the transition. In "classic" register automata guards can only use equality and inequality between registers and *curr*; in SRA model more powerful atoms are allowed. We show that sequence constraints and register automata constraints (which use quantifier-free formulas with equality and inequality as only atoms, i.e. do not employ the SRA extension) lead to undecidability (over infinite domain D).

Theorem 7. *Satisfiability of equational constraints and register automata constraints, which use equality and inequality only, over infinite domain, is undecidable.*

7 Implementations, Optimizations and Benchmarks

Implementation. We have implemented our decision procedure for problems in the constraint language SL for the theory of sequences in a new tool SeCo

(Sequence Constraint Solver) on top of the SMT solver Princess [41]. We extend a publicly available library for symbolic automata and transducers [13] to parametric automata and transducers by connecting them to the uninterpreted constants in our theory of sequences. Our tool supports symbolic transducers, concatenation of sequences and reversing of sequences. Any additional function which satisfies **RegInvRel** such as a replace function which replaces only the first and leftmost longest match can be added in the future.

Our algorithm is an adaption of the tool OSTRICH [12] and closely follows the proof of Theorem 4. To summarize the procedure, a depth-first search is employed to remove all functions in the given input and splitting on the pre-images of those functions. When removing a function, new assertions are added to the pre-image constraints. After all functions have been removed and only assertions are left a nonemptiness check is called over all parametric automata which encoded the assertions. If the check is successful a corresponding model can be constructed, otherwise the procedure computes a conflict set and back-jumps to the last split in the depth search.[2]

Benchmarks. We have performed experiments on two benchmark suites. The first one concerns itself with the verification of properties for programs manipulating sequences. The second benchmark suite compares our tool against an algorithm using symbolic register automata [13] on decision procedures of regular expressions with back-references such as emptiness, equivalence and inclusion.

Both benchmark suites require universal quantification over the parameters; there are existing methods for eliminating these universal quantifiers, one such class are the *semantically deterministic* (SD) [22] PAs; despite its name, being SD is algorithmically checkable. Most of considered the PAs are SD, in particular all in benchmark suite 2.

Experiments were conducted on an AMD Ryzen 5 1600 Six-Core CPU with 16 GB of RAM running on Windows 10. The results for second benchmark suite is shown Table 1. The timeout for all benchmarks is 300 s.

In the first benchmarks suite we are looking to verify a weaker form of the permutation property of sorting as shown in Sect. 2. Furthermore, we verify properties of two self-stabilizing algorithms for mutual exclusion on parameterized systems. The first one is Lamport's bakery algorithm [33], for which we proved that the algorithm ensures mutual exclusion. The system is modelled in the style of regular model checking [8], with system states represented as words, here over an infinite alphabet: the character representing a thread stores the thread control state, a Boolean flag, and an integer as the number drawn by the thread. The system transitions are modelled as parametric transducers, and invariants as parametric automata. The second algorithm is known as Dijkstra's Self-Stabilizing Protocol [20], in which system states are encoded as sequences of integers, and in which we verify that the set of states in which exactly one processor is privileged forms an invariant. The mentioned benchmarks require

[2] For a more detailed write-up of the depth-first search algorithm see OSTRICH [12] Algorithm 1.

Table 1. Benchmark suite 2. SRA is used for the algorithm for symbolic register automata and SEQ for our tool. The symbol $\emptyset$ indicates the column where emptiness was checked, $\equiv$ indicates self equivalence and $\subseteq$ inclusion of languages.

$\mathcal{L}_1$	$\mathcal{L}_2$	$SRA_\emptyset(\mathcal{L}_1)$	$\textsc{SeCo}_\emptyset(\mathcal{L}_1)$	$SRA_\equiv(\mathcal{L}_1)$	$\textsc{SeCo}_\equiv(\mathcal{L}_1)$	$SRA_\subseteq(\mathcal{L}_2, \mathcal{L}_1)$	$\textsc{SeCo}_\subseteq(\mathcal{L}_2, \mathcal{L}_1)$
Pr-C2	Pr-CL2	0.03 s	0.65 s	0.43 s	0.10 s	4.7 s	0.10 s
Pr-C3	Pr-CL3	0.58 s	0.70 s	10.73 s	0.12 s	36.90 s	0.10 s
Pr-C4	Pr-CL4	18.40 s	0.77 s	98.38 s	0.14 s	–	0.10 s
Pr-C6	Pr-CL6	–	1.00 s	–	0.12 s	–	0.10 s
Pr-CL2	Pr-C2	0.33 s	0.30 s	1.03 s	0.13 s	0.52 s	0.76 s
Pr-CL3	Pr-C3	14.04 s	0.38 s	20.44 s	0.13 s	10.52 s	0.76 s
Pr-CL4	Pr-C4	–	0.41 s	0.43 s	0.12 s	–	0.82 s
Pr-CL6	Pr-C6	–	0.62 s	0.43 s	0.12 s	–	1.27 s
IP-2	IP-3	0.11 s	1.53 s	0.63 s	0.14 s	2.43 s	0.15 s
IP-3	IP-4	1.83 s	1.45 s	4.66 s	0.14 s	28.60 s	0.17 s
IP-4	IP-6	30.33 s	1.75 s	80.03 s	0.14 s	–	0.17 s
IP-6	IP-9	–	1.60 s	0.43 s	0.13 s	–	0.17 s

universal quantification, but similar to the motivating example from Sect. 2 one can eliminate quantifiers by Skolemization and instantiation which was done by hand.

The second benchmark suite consists of three different types of benchmarks, summarized in Table 1. The benchmark PR-Cn describes a regular expression for matching products which have the same code number of length n, and PR-CLn matches not only the code number but also the lot number. The last type of benchmark is IP-n, which matches n positions of 2 IP addresses. The benchmarks are taken from the regular-expression crowd-sourcing website RegExLib [39] and are also used in experiments for symbolic register automata [14] which we also compare our results against. To apply our decision procedure to the benchmarks, we encode each of the benchmarks as a parametric automaton, using parameters for the (bounded-size) back-references. The task in the experiments is to check emptiness, language equivalence, and language inclusion for the same combinations of the benchmarks as considered in [14].

Results of the Experiments. All properties can be encoded by parametric automata with very few states and parameters. As a result the properties for each program can be verified in < 2.6 s, in detail the property for Dijkstra's algorithm was proven in 0.6 s, QuickSort in 1.1 s and Lamport's bakery algorithm in 2.5 s.

The results for the second benchmark suite are shown in Table 1. The algorithm for symbolic register automata times out on 11 of the 36 benchmarks and our tool solves most benchmarks in <1 s. One thing to observe that the symbolic register automata scales poorly when more registers are needed to capture the back-references while the performance of our approach does not change noticeably when more parameters are introduced.

8 Conclusion and Future Work

In this paper, we have performed a systematic investigation of decidability and complexity of constraints on sequences. Our starting point is the subcase of string constraints (i.e. over a finite set of sequence elements), which include equational constraints with concatenation, regular constraints, length constraints, and transducers. We have identified parametric automata (extending symbolic automata and variable automata) as suitable notion of "regular constraints" over sequences, and parametric transducers (extending symbolic transducers) as suitable notion of transducers over sequences. We showed that decidability results in the case of strings carry over to sequences, although the complexity is in general higher than in the case of strings (sometimes exponentially higher). For certain element theory (e.g. Linear Real Arithmetic), it is possible to retain the same complexity as in the string case. We also delineate the boundary of the suitable notion of "regular constraints" by showing that the equational constraints with symbolic register automata [14] yields undecidable satisfiability. Finally, our new sequence solver SeCo shows promising experimental results.

There are several future research avenues. Firstly, the complexity of sequence constraints over other specific element theories (e.g. Linear Integer Arithmetic) should be precisely determined. Secondly, is it possible to recover decidability with other fragments of register automata (e.g., single-use automata [7])? On the implementation side, there are some algorithmic improvements, e.g., better nonemptiness checks for parametric automata in the case of a single automaton, as well as product of multiple automata.

Acknowledgment. We thank anonymous reviewers for their thorough and helpful feedback. We are grateful to Nikolaj Bjørner, Rupak Majumdar and Margus Veanes for the inspiring discussion.

References

1. Abdulla, P.A., et al.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_10
2. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P.: Chain-free string constraints. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 277–293. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_16
3. Amadini, R.: A survey on string constraint solving. ACM Comput. Surv. **55**(2), 16:1-16:38 (2023). https://doi.org/10.1145/3484198
4. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
5. Barceló, P., Figueira, D., Libkin, L.: Graph logics with rational relations. Log. Methods Comput. Sci. **9**(3) (2013). https://doi.org/10.2168/LMCS-9(3:1)2013
6. Bjørner, N., de Moura, L., Nachmanson, L., Wintersteiger, C.M.: Programming Z3. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) SETSS 2018. LNCS, vol. 11430, pp. 148–201. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17601-3_4

7. Bojanczyk, M., Stefanski, R.: Single-use automata and transducers for infinite alphabets. In: Czumaj, A., Dawar, A., Merelli, E. (eds.) 47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8–11, 2020, Saarbrücken, Germany (Virtual Conference). LIPIcs, vol. 168, pp. 113:1–113:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ICALP.2020.113
8. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular Model Checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_31
9. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_28
10. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. In: The Collected Works of J. Richard Büchi, pp. 671–683. Springer, New York (1990). https://doi.org/10.1007/978-1-4613-8928-6_37
11. Chen, T., et al.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498707
12. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019). https://doi.org/10.1145/3290362
13. D'Antoni, L.: SVPAlib. Symbolic Automata Library (2018). https://github.com/lorisdanto/symbolicautomata. Accessed 2 Feb 2023
14. D'Antoni, L., Ferreira, T., Sammartino, M., Silva, A.: Symbolic register automata. In: Dillig, I., Tasiran, S. (eds.) CAV. vol. 11561, pp. 3–21. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_1
15. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 47–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_3
16. D'Antoni, L., Veanes, M.: Automata modulo theories. Commun. ACM **64**(5), 86–95 (2021). https://doi.org/10.1145/3419404
17. Diekert, V.: Makanin's algorithm. In: Lothaire, M. (ed.) Algebraic Combinatorics on Words, Encyclopedia of Mathematics and its Applications, vol. 90, chap. 12, pp. 387–442. Cambridge University Press (2002)
18. Diekert, V., Gutiérrez, C., Hagenah, C.: The existential theory of equations with rational constraints in free groups is PSPACE-complete. Inf. Comput. **202**(2), 105–140 (2005). https://doi.org/10.1016/j.ic.2005.04.002
19. Diekert, V., Jeż, A., Plandowski, W.: Finding all solutions of equations in free groups and monoids with involution. Inf. Comput. **251**, 263–286 (2016). https://doi.org/10.1016/j.ic.2016.09.009
20. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974). https://doi.org/10.1145/361179.361202
21. Faran, R., Kupferman, O.: On synthesis of specifications with arithmetic. In: Chatzigeorgiou, A., et al. (eds.) SOFSEM 2020. LNCS, vol. 12011, pp. 161–173. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-38919-2_14
22. Faran, R., Kupferman, O.: On synthesis of specifications with arithmetic. In: Chatzigeorgiou, A., et al. (eds.) SOFSEM 2020. LNCS, vol. 12011, pp. 161–173. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-38919-2_14

23. Figueira, D., Jeż, A., Lin, A.W.: Data path queries over embedded graph databases. In: PODS '22: International Conference on Management of Data, Philadelphia, 12–17 June, 2022. pp. 189–201 (2022). https://doi.org/10.1145/3517804.3524159
24. Figueira, D., Lin, A.W.: Reasoning on data words over numeric domains. In: LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, 2–5 August 2022, pp. 37:1–37:13 (2022). https://doi.org/10.1145/3531130.3533354
25. Furia, C.A.: What's decidable about sequences? In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 128–142. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_11
26. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 209–226. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_21
27. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable automata over infinite alphabets. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 561–572. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13089-2_47
28. Hoare, C.A.R.: Quicksort. Comput. J. **5**(1), 10–15 (1962). https://doi.org/10.1093/comjnl/5.1.10
29. Jeż, A.: Recompression: a simple and powerful technique for word equations. J. ACM **63**(1), 4:1–4:51 (2016). https://doi.org/10.1145/2743014
30. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009). https://doi.org/10.1145/1592434.1592438
31. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994). https://doi.org/10.1016/0304-3975(94)90242-9
32. Kroening, D., Strichman, O.: Decision Procedures. Springer (2008)
33. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974). https://doi.org/10.1145/361082.361093
34. Lin, A.W., Rümmer, P.: Regular model checking revisited. In: Olderog, E.-R., Steffen, B., Yi, W. (eds.) Model Checking, Synthesis, and Learning. LNCS, vol. 13030, pp. 97–114. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91384-7_6
35. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, 20–22 January 2016, pp. 123–136 (2016). https://doi.org/10.1145/2837614.2837641
36. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Sbornik: Mathematics **32**(2), 129–198 (1977)
37. Meyer, B.: Applying "Design by contract." IEEE Comput. **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
38. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
39. None: RegExLib (2017). https://regexlib.com/. Accessed 2 Feb 2023
40. Plandowski, W.: On PSPACE generation of a solution set of a word equation and its applications. Theor. Comput. Sci. **792**, 20–61 (2019). https://doi.org/10.1016/j.tcs.2018.10.023
41. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_20

42. Safari, M., Huisman, M.: A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In: Dongol, B., Troubitsyna, E. (eds.) IFM 2020. LNCS, vol. 12546, pp. 257–275. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63461-2_14
43. Schulz, K.U.: Makanin's algorithm for word equations–two improvements and a generalization. In: Schulz, K.U. (ed.) IWWERT. Lecture Notes in Computer Science, vol. 572, pp. 85–150. Springer, Cham (1990). https://doi.org/10.1007/3-540-55124-7_4
44. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjorner, N.: Symbolic finite state transducers: algorithms and applications. SIGPLAN Not. **47**(1), 137–150 (2012). https://doi.org/10.1145/2103621.2103674
45. Wang, Q., Appel, A.W.: A solver for arrays with concatenation. J. Autom. Reason. **67**(1), 4 (2023). https://doi.org/10.1007/s10817-022-09654-y

Exploiting Adjoints in Property Directed Reachability Analysis

Mayuko Kori[1,2(✉)], Flavio Ascari[3], Filippo Bonchi[3], Roberto Bruni[3], Roberta Gori[3], and Ichiro Hasuo[1,2]

[1] National Institute of Informatics, Tokyo, Japan
{mkori,hasuo}@nii.ac.jp
[2] The Graduate University for Advanced Studies (SOKENDAI), Hayama, Japan
[3] Dipartimento di Informatica, Università di Pisa, Pisa, Italy
flavio.ascari@phd.unipi.it,
{filippo.bonchi,roberto.bruni,roberta.gori}@unipi.it

Abstract. We formulate, in lattice-theoretic terms, two novel algorithms inspired by Bradley's property directed reachability algorithm. For finding safe invariants or counterexamples, the first algorithm exploits over-approximations of both forward and backward transition relations, expressed abstractly by the notion of adjoints. In the absence of adjoints, one can use the second algorithm, which exploits lower sets and their principals. As a notable example of application, we consider quantitative reachability problems for Markov Decision Processes.

Keywords: PDR · Lattice theory · Adjoints · MDPs · Over-approximation

1 Introduction

Property directed reachability analysis (PDR) refers to a class of verification algorithms for solving safety problems of transition systems [5,12]. Its essence consists of 1) interleaving the construction of an *inductive invariant* (a *positive chain*) with that of a *counterexample* (a *negative sequence*), and 2) making the two sequences *interact*, with one narrowing down the search space for the other.

PDR algorithms have shown impressive performance both in hardware and software verification, leading to active research [15,18,28,29] going far beyond its original scope. For instance, an abstract domain [8] capturing the over-approximation exploited by PDR has been recently introduced in [13], while PrIC3 [3] extended PDR for quantitative verification of probabilistic systems.

Research supported by MIUR PRIN Project 201784YSZ5 *ASPRA*, by JST ERATO HASUO Metamathematics for Systems Design Project JPMJER1603, by JST CREST Grant JPMJCR2012, by JSPS DC KAKENHI Grant 22J21742 and by EU Next-GenerationEU (NRRP) SPOKE 10, Mission 4, Component 2, Investment N. 1.4, CUP N. I53C22000690001.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 41–63, 2023.
https://doi.org/10.1007/978-3-031-37703-7_3

To uncover the abstract principles behind PDR and its extensions, Kori et al. proposed LT-PDR [19], a generalisation of PDR in terms of lattice/category theory. LT-PDR can be instantiated using domain-specific *heuristics* to create effective algorithms for different kinds of systems such as Kripke structures, Markov Decision Processes (MDPs), and Markov reward models. However, the theory in [19] does not offer guidance on devising concrete heuristics.

Adjoints in PDR. Our approach shares the same vision of LT-PDR, but we identify different principles: *adjunctions* are the core of our toolset.

An adjunction $f \dashv g$ is one of the central concepts in category theory [23]. It is prevalent in various fields of computer science, too, such as abstract interpretation [8] and functional programming [22]. Our use of adjoints in this work comes in the following two flavours.

$$\begin{tikzcd} A \arrow[r, bend right, "g"'] \arrow[r, phantom, "\perp"] & C \arrow[l, bend right, "f"'] \end{tikzcd}$$

- (forward-backward adjoint) f describes the *forward semantics* of a transition system, while g is the *backward* one, where we typically have $A = C$.
- (abstraction-concretization adjoint) C is a concrete semantic domain, and A is an abstract one, much like in abstract interpretation. An adjoint enables us to convert a fixed-point problem in C to that in A.

Our Algorithms. The problem we address is the standard lattice theoretical formulation of safety problems, namely whether the least fixed point of a continuous map b over a complete lattice $(L, \sqsubseteq)$ is below a given element $p \in L$. In symbols $\mu b \sqsubseteq_? p$. We present two algorithms.

The first one, named `AdjointPDR`, assumes to have an element $i \in L$ and two adjoints $f \dashv g \colon L \to L$, representing respectively initial states, forward semantics and backward semantics (see right) such that $b(x) = f(x) \sqcup i$ for all $x \in L$. Under this assumption, we have the following equivalences (they follow from the Knaster-Tarski theorem, see §2):

$$\begin{tikzcd} L \arrow[r, bend right, "g"'] \arrow[r, phantom, "\perp"] & L \arrow[l, bend right, "f"'] \end{tikzcd}$$

$$\mu b \sqsubseteq p \quad \Leftrightarrow \quad \mu(f \sqcup i) \sqsubseteq p \quad \Leftrightarrow \quad i \sqsubseteq \nu(g \sqcap p),$$

where $\mu(f \sqcup i)$ and $\nu(g \sqcap p)$ are, by the Kleene theorem, the limits of the *initial* and *final* chains illustrated below.

$$\bot \sqsubseteq i \sqsubseteq f(i) \sqcup i \sqsubseteq \cdots \qquad\qquad \cdots \sqsubseteq g(p) \sqcap p \sqsubseteq p \sqsubseteq \top$$

As positive chain, PDR exploits an over-approximation of the initial chain: it is made greater to accelerate convergence; still it has to be below p.

The distinguishing feature of `AdjointPDR` is to take as a negative sequence (that is a sequential construction of potential counterexamples) an over-approximation of the final chain. This crucially differs from the negative sequence of LT-PDR, namely an under-approximation of the computed positive chain.

We prove that `AdjointPDR` is sound (Theorem 5) and does not loop (Proposition 7) but since, the problem $\mu b \sqsubseteq_? p$ is not always decidable, we cannot prove termination. Nevertheless, `AdjointPDR` allows for a formal theory of heuristics that are essential when instantiating the algorithm to concrete problems.

The theory prescribes the choices to obtain the boundary executions, using initial and final chains (Proposition 10); it thus identifies a class of heuristics guaranteeing termination when answers are negative (Theorem 12).

AdjointPDR's assumption of a forward-backward adjoint $f \dashv g$, however, does not hold very often, especially in probabilistic settings. Our second algorithm AdjointPDR$^{\downarrow}$ circumvents this problem by extending the lattice for the negative sequence, from L to the lattice $L^{\downarrow}$ of *lower sets* in L.

$$\begin{tikzcd} L \arrow[loop left, "b"] \arrow[r, bend right, "(-)^{\downarrow}"'] \arrow[r, phantom, "\perp"] & L^{\downarrow} \arrow[l, bend right, "\bigsqcup"'] \arrow[loop right, "b^{\downarrow} \dashv b^{\downarrow}_{r}"] \end{tikzcd}$$

Specifically, by using the second form of adjoints, namely an abstraction-concretization pair, the problem $\mu b \sqsubseteq_? p$ in L can be translated to an equivalent problem on $b^{\downarrow}$ in $L^{\downarrow}$, for which an adjoint $b^{\downarrow} \dashv b^{\downarrow}_{r}$ is guaranteed. This allows one to run AdjointPDR in the lattice $L^{\downarrow}$. We then notice that the search for a positive chain can be conveniently restricted to principals in $L^{\downarrow}$, which have representatives in L. The resulting algorithm, using L for positive chains and $L^{\downarrow}$ for negative sequences, is AdjointPDR$^{\downarrow}$.

The use of lower sets for the negative sequence is a key advantage. It not only avoids the restrictive assumption on forward-backward adjoints $f \dashv g$, but also enables a more thorough search for counterexamples. AdjointPDR$^{\downarrow}$ can simulate step-by-step LT-PDR (Theorem 17), while the reverse is not possible due to a single negative sequence in AdjointPDR$^{\downarrow}$ potentially representing multiple (Proposition 18) or even all (Proposition 19) negative sequences in LT-PDR.

Concrete Instances. Our lattice-theoretic algorithms yield many concrete instances: the original IC3/PDR [5,12] as well as Reverse PDR [27] are instances of AdjointPDR with L being the powerset of the state space; since LT-PDR can be simulated by AdjointPDR$^{\downarrow}$, the latter generalizes all instances in [19].

As a notable instance, we apply AdjointPDR$^{\downarrow}$ to MDPs, specifically to decide if the maximum reachability probability [1] is below a given threshold. Here the lattice $L = [0,1]^S$ is that of fuzzy predicates over the state space S. Our theory provides guidance to devise two heuristics, for which we prove negative termination (Corollary 20). We present its implementation in Haskell, and its experimental evaluation, where comparison is made against existing probabilistic PDR algorithms (PrIC3 [3], LT-PDR [19]) and a non-PDR one (Storm [11]). The performance of AdjointPDR$^{\downarrow}$ is encouraging—it supports the potential of PDR algorithms in probabilistic model checking. The experiments also indicate the importance of having a variety of heuristics, and thus the value of our adjoint framework that helps coming up with those.

Additionally, we found that abstraction features of Haskell allows us to code lattice-theoretic algorithms almost literally (∼100 lines). Implementing a few heuristics takes another ∼240 lines. This way, we found that mathematical abstraction can directly help easing implementation effort.

Related Work. Reverse PDR [27] applies PDR from unsafe states using a backward transition relation $\mathbf{T}$ and tries to prove that initial states are unreachable. Our right adjoint g is also backward, but it differs from $\mathbf{T}$ in the presence of nondeterminism: roughly, $\mathbf{T}(X)$ is the set of states which *can* reach X in one step, while $g(X)$ are states which *only* reach X in one step. *fb*PDR [28,29] runs PDR and Reverse PDR in parallel with shared information. Our work uses both forward and backward directions (the pair $f \dashv g$), too, but approximate differently: Reverse PDR over-approximates the set of states that can reach an unsafe state, while we over-approximate the set of states that only reach safe states.

The comparison with LT-PDR [19] is extensively discussed in Sect. 4.2. PrIC3 [3] extended PDR to MDPs, which are our main experimental ground: Sect. 6 compares the performances of PrIC3, LT-PDR and $\texttt{AdjointPDR}^{\downarrow}$.

We remark that PDR has been applied to other settings, such as software model checking using theories and SMT-solvers [6,21] or automated planning [30]. Most of them (e.g., software model checking) fall already in the generality of LT-PDR and thus they can be embedded in our framework.

It is also worth to mention that, in the context of abstract interpretation, the use of adjoints to construct initial and final chains and exploit the interaction between their approximations has been investigated in several works, e.g., [7].

Structure of the Paper. After recalling some preliminaries in Sect. 2, we present AdjointPDR in Sect. 3 and $\texttt{AdjointPDR}^{\downarrow}$ in Sect. 4. In Sect. 5 we introduce the heuristics for the max reachability problems of MDPs, that are experimentally tested in Sect. 6.

2 Preliminaries and Notation

We assume that the reader is familiar with lattice theory, see, e.g., [10]. We use $(L, \sqsubseteq)$, $(L_1, \sqsubseteq_1)$, $(L_2, \sqsubseteq_2)$ to range over complete lattices and x, y, z to range over their elements. We omit subscripts and order relations whenever clear from the context. As usual, $\bigsqcup$ and $\bigsqcap$ denote least upper bound and greatest lower bound, $\sqcup$ and $\sqcap$ denote join and meet, $\top$ and $\bot$ top and bottom. Hereafter we will tacitly assume that all maps are monotone. Obviously, the identity map $id\colon L \to L$ and the composition $f \circ g\colon L_1 \to L_3$ of two monotone maps $g\colon L_1 \to L_2$ and $f\colon L_2 \to L_3$ are monotone. For a map $f\colon L \to L$, we inductively define $f^0 = id$ and $f^{n+1} = f \circ f^n$. Given $l\colon L_1 \to L_2$ and $r\colon L_2 \to L_1$, we say that l is the *left adjoint* of r, or equivalently that r is the *right adjoint* of l, written $l \dashv r$, when it holds that $l(x) \sqsubseteq_2 y$ iff $x \sqsubseteq_1 r(y)$ for all $x \in L_1$ and $y \in L_2$. Given a map $f\colon L \to L$, the element $x \in L$ is a *post-fixed point* iff $x \sqsubseteq f(x)$, a *pre-fixed point* iff $f(x) \sqsubseteq x$ and a *fixed point* iff $x = f(x)$. Pre, post and fixed points form complete lattices: we write μf and νf for the least and greatest fixed point.

Several problems relevant to computer science can be reduced to check if $\mu b \sqsubseteq p$ for a monotone map $b\colon L \to L$ on a complete lattice L. The Knaster-Tarski fixed-point theorem characterises μb as the least upper bound of all pre-fixed points of b and νb as the greatest lower bound of all its post-fixed points:

$$\mu b = \bigsqcap\{x \mid b(x) \sqsubseteq x\} \qquad\qquad \nu b = \bigsqcup\{x \mid x \sqsubseteq b(x)\} \ .$$

This immediately leads to two proof principles, illustrated below:

$$\frac{\exists x,\ b(x) \sqsubseteq x \sqsubseteq p}{\mu b \sqsubseteq p} \qquad\qquad \frac{\exists x,\ i \sqsubseteq x \sqsubseteq b(x)}{i \sqsubseteq \nu b} \tag{KT}$$

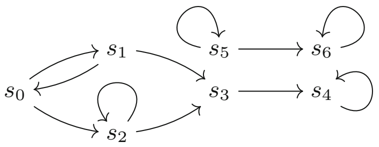


Fig. 1. The transition system of Example 1, with $S = \{s_0, \dots s_6\}$ and $I = \{s_0\}$.

By means of (KT), one can prove $\mu b \sqsubseteq p$ by finding some pre-fixed point x, often called *invariant*, such that $x \sqsubseteq p$. However, automatically finding invariants might be rather complicated, so most of the algorithms rely on another fixed-point theorem, usually attributed to Kleene. It characterises μb and νb as the least upper bound and the greatest lower bound, of the *initial* and *final chains*:

$$\bot \sqsubseteq b(\bot) \sqsubseteq b^2(\bot) \sqsubseteq \cdots \quad \text{and} \quad \cdots \sqsubseteq b^2(\top) \sqsubseteq b(\top) \sqsubseteq \top. \quad \text{That is,} \tag{Kl}$$
$$\mu b = \bigsqcup_{n \in \mathbb{N}} b^n(\bot), \qquad \nu b = \bigsqcap_{n \in \mathbb{N}} b^n(\top).$$

The assumptions are stronger than for Knaster-Tarski: for the leftmost statement, it requires the map b to be ω*-continuous* (i.e., it preserves $\bigsqcup$ of ω-chains) and, for the rightmost ω*-co-continuous* (similar but for $\bigsqcap$). Observe that every left adjoint is continuous and every right adjoint is co-continuous (see e.g. [23]).

As explained in [19], property directed reachability (PDR) algorithms [5] exploits (KT) to try to prove the inequation and (Kl) to refute it. In the algorithm we introduce in the next section, we further assume that b is of the form $f \sqcup i$ for some element $i \in L$ and map $f \colon L \to L$, namely $b(x) = f(x) \sqcup i$ for all $x \in L$. Moreover we require f to have a right adjoint $g \colon L \to L$. In this case

$$\mu(f \sqcup i) \sqsubseteq p \qquad \text{iff} \qquad i \sqsubseteq \nu(g \sqcap p) \tag{1}$$

(which is easily shown using the Knaster-Tarski theorem) and $(f \sqcup i)$ and $(g \sqcap p)$ are guaranteed to be (co)continuous. Since $f \dashv g$ and left and right adjoints preserve, resp., arbitrary joins and meets, then for all $n \in \mathbb{N}$

$$(f \sqcup i)^n(\bot) = \textstyle\bigsqcup_{j<n} f^j(i) \qquad (g \sqcap p)^n(\top) = \textstyle\bigsqcap_{j<n} g^j(p) \tag{2}$$

which by (Kl) provide useful characterisations of least and greatest fixed points.

$$\mu(f \sqcup i) = \textstyle\bigsqcup_{n \in \mathbb{N}} f^n(i) \qquad \nu(g \sqcap p) = \textstyle\bigsqcap_{n \in \mathbb{N}} g^n(p) \tag{Kl$\dashv$}$$

We conclude this section with an example that we will often revisit. It also provides a justification for the intuitive terminology that we sporadically use.

Example 1 (Safety problem for transition systems). A *transition system* consists of a triple (S, I, δ) where S is a set of states, $I \subseteq S$ is a set of initial states, and $\delta \colon S \to \mathcal{P}S$ is a transition relation. Here $\mathcal{P}S$ denotes the powerset of S, which

$$x_0 = \bot \tag{I0}$$
$$1 \leq k \leq n \tag{I1}$$
$$\forall j \in [0, n-2],\ x_j \sqsubseteq x_{j+1} \tag{I2}$$
$$i \sqsubseteq x_1 \tag{P1}$$
$$x_{n-2} \sqsubseteq p \tag{P2}$$
$$\forall j \in [0, n-2],\ f(x_j) \sqsubseteq x_{j+1} \tag{P3}$$
$$\forall j \in [0, n-2],\ x_j \sqsubseteq g(x_{j+1}) \tag{P3a}$$

$$\text{If } \boldsymbol{y} \neq \varepsilon \text{ then } p \sqsubseteq y_{n-1} \tag{N1}$$
$$\forall j \in [k, n-2],\ g(y_{j+1}) \sqsubseteq y_j \tag{N2}$$
$$\forall j \in [k, n-1],\ x_j \not\sqsubseteq y_j \tag{PN}$$
$$\forall j \in [0, n-1],\ (f \sqcup i)^j(\bot) \sqsubseteq x_j \sqsubseteq (g \sqcap p)^{n-1-j}(\top) \tag{A1}$$
$$\forall j \in [1, n-1],\ x_{j-1} \sqsubseteq g^{n-1-j}(p) \tag{A2}$$
$$\forall j \in [k, n-1],\ g^{n-1-j}(p) \sqsubseteq y_j \tag{A3}$$

Fig. 2. Invariants of `AdjointPDR`.

forms a complete lattice ordered by inclusion $\subseteq$. By defining $F\colon \mathcal{P}S \to \mathcal{P}S$ as $F(X) \overset{\text{def}}{=} \bigcup_{s \in X} \delta(s)$ for all $X \in \mathcal{P}S$, one has that $\mu(F \cup I)$ is the set of all states reachable from I. Therefore, for any $P \in \mathcal{P}S$, representing some safety property, $\mu(F \cup I) \subseteq P$ holds iff all reachable states are safe. It is worth to remark that F has a right adjoint $G\colon \mathcal{P}S \to \mathcal{P}S$ defined for all $X \in \mathcal{P}S$ as $G(X) \overset{\text{def}}{=} \{s \mid \delta(s) \subseteq X\}$. Thus by (1), $\mu(F \cup I) \subseteq P$ iff $I \subseteq \nu(G \cap P)$.

Consider the transition system in Fig. 1. Hereafter we write S_j for the set of states $\{s_0, s_1, \ldots, s_j\}$ and we fix the set of safe states to be $P = S_5$. It is immediate to see that $\mu(F \cup I) = S_4 \subseteq P$. Automatically, this can be checked with the initial chains of $(F \cup I)$ or with the final chain of $(G \cap P)$ displayed below on the left and on the right, respectively.

$$\emptyset \subseteq I \subseteq S_2 \subseteq S_3 \subseteq S_4 \subseteq S_4 \subseteq \cdots \qquad\qquad \cdots \subseteq S_4 \subseteq S_4 \subseteq P \subseteq S$$

The $(j+1)$-th element of the initial chain contains all the states that can be reached by I in at most j transitions, while $(j+1)$-th element of the final chain contains all the states that in at most j transitions reach safe states only.

3 Adjoint PDR

In this section we present `AdjointPDR`, an algorithm that takes in input a tuple (i, f, g, p) with $i, p \in L$ and $f \dashv g\colon L \to L$ and, if it terminates, it returns true whenever $\mu(f \sqcup i) \sqsubseteq p$ and false otherwise.

The algorithm manipulates two sequences of elements of L: $\boldsymbol{x} \overset{\text{def}}{=} x_0, \ldots, x_{n-1}$ of length n and $\boldsymbol{y} \overset{\text{def}}{=} y_k, \ldots y_{n-1}$ of length $n - k$. These satisfy, through the executions of `AdjointPDR`, the invariants in Fig. 2. Observe that, by (A1), x_j over-approximates the j-th element of the initial chain, namely $(f \sqcup i)^j(\bot) \sqsubseteq x_j$, while, by (A3), the j-indexed element y_j of $\boldsymbol{y}$ over-approximates $g^{n-j-1}(p)$ that, borrowing the terminology of Example 1, is the set of states which are safe in $n - j - 1$ transitions. Moreover, by (PN), the element y_j witnesses that x_j is unsafe, i.e., that $x_j \not\sqsubseteq g^{n-1-j}(p)$ or equivalently $f^{n-j-1}(x_j) \not\sqsubseteq p$. Notably, $\boldsymbol{x}$ is a positive chain and $\boldsymbol{y}$ a negative sequence, according to the definitions below.

```
                          AdjointPDR $(i, f, g, p)$

<INITIALISATION>
  $(\boldsymbol{x}\|\boldsymbol{y})_{n,k}$ := $(\bot, \top\|\varepsilon)_{2,2}$
<ITERATION>                                         % $\boldsymbol{x}, \boldsymbol{y}$ not conclusive
  case $(\boldsymbol{x}\|\boldsymbol{y})_{n,k}$ of
      $\boldsymbol{y} = \varepsilon$ and $x_{n-1} \sqsubseteq p$ :                     %(Unfold)
         $(\boldsymbol{x}\|\boldsymbol{y})_{n,k}$ := $(\boldsymbol{x}, \top\|\varepsilon)_{n+1,n+1}$
      $\boldsymbol{y} = \varepsilon$ and $x_{n-1} \not\sqsubseteq p$ :                     %(Candidate)
         choose $z \in L$ such that $x_{n-1} \not\sqsubseteq z$ and $p \sqsubseteq z$;
         $(\boldsymbol{x}\|\boldsymbol{y})_{n,k}$ := $(\boldsymbol{x}\|z)_{n,n-1}$
      $\boldsymbol{y} \neq \varepsilon$ and $f(x_{k-1}) \not\sqsubseteq y_k$ :                 %(Decide)
         choose $z \in L$ such that $x_{k-1} \not\sqsubseteq z$ and $g(y_k) \sqsubseteq z$;
         $(\boldsymbol{x}\|\boldsymbol{y})_{n,k}$ := $(\boldsymbol{x}\|z, \boldsymbol{y})_{n,k-1}$
      $\boldsymbol{y} \neq \varepsilon$ and $f(x_{k-1}) \sqsubseteq y_k$ :                 %(Conflict)
         choose $z \in L$ such that $z \sqsubseteq y_k$ and $(f \sqcup i)(x_{k-1} \sqcap z) \sqsubseteq z$;
         $(\boldsymbol{x}\|\boldsymbol{y})_{n,k}$ := $(\boldsymbol{x} \sqcap_k z\|\mathsf{tail}(\boldsymbol{y}))_{n,k+1}$
  endcase
<TERMINATION>
   if $\exists j \in [0, n-2] \,.\, x_{j+1} \sqsubseteq x_j$ then return true   % $\boldsymbol{x}$ conclusive
   if $i \not\sqsubseteq y_1$ then return false                      % $\boldsymbol{y}$ conclusive
```

Fig. 3. `AdjointPDR` algorithm checking $\mu(f \sqcup i) \sqsubseteq p$.

Definition 2 (positive chain). *A* positive chain *for $\mu(f \sqcup i) \sqsubseteq p$ is a finite chain $x_0 \sqsubseteq \cdots \sqsubseteq x_{n-1}$ in L of length $n \geq 2$ which satisfies* (P1)*,* (P2)*,* (P3) *in Fig. 2. It is* conclusive *if $x_{j+1} \sqsubseteq x_j$ for some $j \leq n-2$.*

In a conclusive positive chain, x_{j+1} provides an invariant for $f \sqcup i$ and thus, by (KT), $\mu(f \sqcup i) \sqsubseteq p$ holds. So, when $\boldsymbol{x}$ is conclusive, `AdjointPDR` returns true.

Definition 3 (negative sequence). *A* negative sequence *for $\mu(f \sqcup i) \sqsubseteq p$ is a finite sequence $y_k, \ldots, y_{n-1}$ in L with $1 \leq k \leq n$ which satisfies* (N1) *and* (N2) *in Fig. 2. It is* conclusive *if $k = 1$ and $i \not\sqsubseteq y_1$.*

When $\boldsymbol{y}$ is conclusive, `AdjointPDR` returns false as y_1 provides a counterexample: (N1) and (N2) entail (A3) and thus $i \not\sqsubseteq y_1 \sqsupseteq g^{n-2}(p)$. By (Kl⊣), $g^{n-2}(p) \sqsupseteq \nu(g \sqcap p)$ and thus $i \not\sqsubseteq \nu(g \sqcap p)$. By (1), $\mu(f \sqcup i) \not\sqsubseteq p$.

The pseudocode of the algorithm is displayed in Fig. 3, where we write $(\boldsymbol{x}\|\boldsymbol{y})_{n,k}$ to compactly represents the state of the algorithm: the pair (n, k) is called the *index* of the state, with $\boldsymbol{x}$ of length n and $\boldsymbol{y}$ of length $n - k$. When $k = n$, $\boldsymbol{y}$ is the empty sequence ε. For any $z \in L$, we write $\boldsymbol{x}, z$ for the chain $x_0, \ldots, x_{n-1}, z$ of length $n+1$ and $z, \boldsymbol{y}$ for the sequence $z, y_k, \ldots y_{n-1}$ of length $n-(k-1)$. Moreover, we write $\boldsymbol{x} \sqcap_j z$ for the chain $x_0 \sqcap z, \ldots, x_j \sqcap z, x_{j+1}, \ldots, x_{n-1}$. Finally, $\mathsf{tail}(\boldsymbol{y})$ stands for the tail of $\boldsymbol{y}$, namely $y_{k+1}, \ldots y_{n-1}$ of length $n-(k+1)$.

The algorithm starts in the initial state $s_0 \overset{\text{def}}{=} (\bot, \top\|\varepsilon)_{2,2}$ and, unless one of $\boldsymbol{x}$ and $\boldsymbol{y}$ is conclusive, iteratively applies one of the four mutually exclusive rules: (Unfold), (Candidate), (Decide) and (Conflict). The rule (Unfold) extends the positive chain by one element when the negative sequence is empty and the positive chain is under p; since the element introduced by (Unfold) is $\top$, its application typically triggers rule (Candidate) that starts the negative sequence with an over-approximation of p. Recall that the role of y_j is to witness that x_j is unsafe. After (Candidate) either (Decide) or (Conflict) are possible: if y_k witnesses that, besides x_k, also $f(x_{k-1})$ is unsafe, then (Decide) is used to further extend the negative sequence to witness that x_{k-1} is unsafe; otherwise, the rule (Conflict) improves the precision of the positive chain in such a way that y_k no longer witnesses $x_k \sqcap z$ unsafe and, thus, the negative sequence is shortened.

Note that, in (Candidate), (Decide) and (Conflict), the element $z \in L$ is chosen among a set of possibilities, thus `AdjointPDR` is nondeterministic.

To illustrate the executions of the algorithm, we adopt a labeled transition system notation. Let $\mathcal{S} \overset{\mathrm{def}}{=} \{(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \mid n \geq 2,\ k \leq n,\ \boldsymbol{x} \in L^n \text{ and } \boldsymbol{y} \in L^{n-k}\}$ be the set of all possible states of `AdjointPDR`. We call $(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \in \mathcal{S}$ *conclusive* if $\boldsymbol{x}$ or $\boldsymbol{y}$ are such. When $s \in \mathcal{S}$ is not conclusive, we write $s \overset{D}{\rightarrow}$ to mean that s satisfies the guards in the rule (Decide), and $s \overset{D}{\rightarrow}_z s'$ to mean that, being (Decide) applicable, `AdjointPDR` moves from state s to s' by choosing z. Similarly for the other rules: the labels Ca, Co and U stands for (Candidate), (Conflict) and (Unfold), respectively. When irrelevant we omit to specify labels and choices and we just write $s \rightarrow s'$. As usual $\rightarrow^+$ stands for the transitive closure of $\rightarrow$ while $\rightarrow^*$ stands for the reflexive and transitive closure of $\rightarrow$.

Example 4. Consider the safety problem in Example 1. Below we illustrate two possible computations of `AdjointPDR` that differ for the choice of z in (Conflict). The first run is conveniently represented as the following series of transitions.

$$\begin{aligned}&(\emptyset, S\|\varepsilon)_{2,2} \overset{Ca}{\rightarrow}_P (\emptyset, S\|P)_{2,1} \overset{Co}{\rightarrow}_I (\emptyset, I\|\varepsilon)_{2,2} \overset{U}{\rightarrow} (\emptyset, I, S\|\varepsilon)_{3,3} \overset{Ca}{\rightarrow}_P (\emptyset, I, S\|P)_{3,2}\\ &\overset{Co}{\rightarrow}_{S_2} (\emptyset, I, S_2\|\varepsilon)_{3,3} \overset{U}{\rightarrow}\overset{Ca}{\rightarrow}_P (\emptyset, I, S_2, S\|P)_{4,3} \overset{Co}{\rightarrow}_{S_3} (\emptyset, I, S_2, S_3\|\varepsilon)_{4,4} \overset{U}{\rightarrow}\overset{Ca}{\rightarrow}_P (\emptyset, I, S_2, S_3, S\|P)_{5,4}\\ &\overset{Co}{\rightarrow}_{S_4} (\emptyset, I, S_2, S_3, S_4\|\varepsilon)_{5,5} \overset{U}{\rightarrow}\overset{Ca}{\rightarrow}_P (\emptyset, I, S_2, S_3, S_4, S\|P)_{6,5} \overset{Co}{\rightarrow}_{S_4} (\emptyset, I, S_2, S_3, S_4, S_4\|\varepsilon)_{6,6}\end{aligned}$$

The last state returns true since $x_4 = x_5 = S_4$. Observe that the elements of $\boldsymbol{x}$, with the exception of the last element x_{n-1}, are those of the initial chain of $(F \cup I)$, namely, x_j is the set of states reachable in at most $j-1$ steps. In the second computation, the elements of $\boldsymbol{x}$ are roughly those of the final chain of $(G \cap P)$. More precisely, after (Unfold) or (Candidate), x_{n-j} for $j < n-1$ is the set of states which only reach safe states within j steps.

$$\begin{aligned}&(\emptyset, S\|\varepsilon)_{2,2} \overset{Ca}{\rightarrow}_P (\emptyset, S\|P)_{2,1} \overset{Co}{\rightarrow}_P (\emptyset, P\|\varepsilon)_{2,2}\\ &\overset{U}{\rightarrow}\overset{Ca}{\rightarrow}_P (\emptyset, P, S\|P)_{3,2} \overset{D}{\rightarrow}_{S_4} (\emptyset, P, S\|S_4, P)_{3,1} \overset{Co}{\rightarrow}_{S_4} (\emptyset, S_4, S\|P)_{3,2} \overset{Co}{\rightarrow}_P (\emptyset, S_4, P\|\varepsilon)_{3,3}\\ &\overset{U}{\rightarrow}\overset{Ca}{\rightarrow}_P (\emptyset, S_4, P, S\|P)_{4,3} \overset{D}{\rightarrow}_{S_4} (\emptyset, S_4, P, S\|S_4, P)_{4,2} \overset{Co}{\rightarrow}_{S_4} (\emptyset, S_4, S_4, S\|P)_{4,3}\end{aligned}$$

Observe that, by invariant (A1), the values of $\boldsymbol{x}$ in the two runs are, respectively, the least and the greatest values for all possible computations of `AdjointPDR`.

Theorem 5.1 follows by invariants (I2), (P1), (P3) and (KT); Theorem 5.2 by (N1), (N2) and (Kl⊣). Note that both results hold for any choice of z.

Theorem 5 (Soundness). `AdjointPDR` *is sound. Namely,*

1. *If* `AdjointPDR` *returns true then* $\mu(f \sqcup i) \sqsubseteq p$.
2. *If* `AdjointPDR` *returns false then* $\mu(f \sqcup i) \not\sqsubseteq p$.

3.1 Progression

It is necessary to prove that in any step of the execution, if the algorithm does not return true or false, then it can progress to a new state, not yet visited. To this aim we must deal with the subtleties of the non-deterministic choice of the element z in (Candidate), (Decide) and (Conflict). The following proposition ensures that, for any of these three rules, there is always a possible choice.

Proposition 6 (Canonical choices). *The following are always possible:*
1. in (Candidate) $z = p$; *3. in (Conflict) $z = y_k$;*
2. in (Decide) $z = g(y_k)$; *4. in (Conflict) $z = (f \sqcup i)(x_{k-1})$.*
Thus, for all non-conclusive $s \in \mathcal{S}$, if $s_0 \to^* s$ then $s \to$.

Then, Proposition 7 ensures that `AdjointPDR` always traverses new states.

Proposition 7 (Impossibility of loops). *If $s_0 \to^* s \to^+ s'$, then $s \neq s'$.*

Observe that the above propositions entail that `AdjointPDR` terminates whenever the lattice L is finite, since the set of reachable states is finite in this case.

Example 8. For (I, F, G, P) as in Example 1, `AdjointPDR` behaves essentially as IC3/PDR [5], solving reachability problems for transition systems with finite state space S. Since the lattice $\mathcal{P}S$ is also finite, `AdjointPDR` always terminates.

3.2 Heuristics

The nondeterministic choices of the algorithm can be resolved by using heuristics. Intuitively, a heuristic chooses for any states $s \in \mathcal{S}$ an element $z \in L$ to be possibly used in (Candidate), (Decide) or (Conflict), so it is just a function $h \colon \mathcal{S} \to L$. When defining a heuristic, we will avoid to specify its values on conclusive states or in those performing (Unfold), as they are clearly irrelevant.

With a heuristic, one can instantiate `AdjointPDR` by making the choice of z as prescribed by h. Syntactically, this means to erase from the code of Fig. 3 the three lines of `choose` and replace them by z`:=` $h(\,(\boldsymbol{x}\|\boldsymbol{c})_{n,k}\,)$. We call `AdjointPDR`$_h$ the resulting deterministic algorithm and write $s \to_h s'$ to mean that `AdjointPDR`$_h$ moves from state s to s'. We let $\mathcal{S}^h \overset{\text{def}}{=} \{s \in \mathcal{S} \mid s_0 \to_h^* s\}$ be the sets of all states reachable by `AdjointPDR`$_h$.

Definition 9 (legit heuristic). *A heuristic $h \colon \mathcal{S} \to L$ is called* legit *whenever for all $s, s' \in \mathcal{S}^h$, if $s \to_h s'$ then $s \to s'$.*

When h is legit, the only execution of the deterministic algorithm `AdjointPDR`$_h$ is one of the possible executions of the non-deterministic algorithm `AdjointPDR`.

The canonical choices provide two legit heuristics: first, we call *simple* any legit heuristic h that chooses z in (Candidate) and (Decide) as in Proposition 6:

$$(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \mapsto \begin{cases} p & \text{if } (\boldsymbol{x}\|\boldsymbol{y})_{n,k} \xrightarrow{Ca} \\ g(y_k) & \text{if } (\boldsymbol{x}\|\boldsymbol{y})_{n,k} \xrightarrow{D} \end{cases} \tag{3}$$

Then, if the choice in (Conflict) is like in Proposition 6.4, we call h *initial*; if it is like in Proposition 6.3, we call h *final*. Shortly, the two legit heuristics are:

simple initial	(3) and $(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \mapsto (f \sqcup i)(x_{k-1})$	if $(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \in Co$
simple final	(3) and $(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \mapsto y_k$	if $(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \in Co$

Interestingly, with any simple heuristic, the sequence $\boldsymbol{y}$ takes a familiar shape:

Proposition 10. *Let $h \colon \mathcal{S} \to L$ be any simple heuristic. For all $(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \in \mathcal{S}^h$, invariant (A3) holds as an equality, namely for all $j \in [k, n-1]$, $y_j = g^{n-1-j}(p)$.*

By the above proposition and (A3), the negative sequence $\boldsymbol{y}$ occurring in the execution of $\texttt{AdjointPDR}_h$, for a simple heuristic h, is the least amongst all the negative sequences occurring in any execution of `AdjointPDR`.

Instead, invariant (A1) informs us that the positive chain $\boldsymbol{x}$ is always in between the initial chain of $f \sqcup i$ and the final chain of $g \sqcap p$. Such values of $\boldsymbol{x}$ are obtained by, respectively, simple initial and simple final heuristic.

Example 11. Consider the two runs of `AdjointPDR` in Example 4. The first one exploits the simple initial heuristic and indeed, the positive chain $\boldsymbol{x}$ coincides with the initial chain. Analogously, the second run uses the simple final heuristic.

3.3 Negative Termination

When the lattice L is not finite, `AdjointPDR` may not terminate, since checking $\mu(f \sqcup i) \sqsubseteq p$ is not always decidable. In this section, we show that the use of certain heuristics can guarantee termination whenever $\mu(f \sqcup i) \not\sqsubseteq p$.

The key insight is the following: if $\mu(f \sqcup i) \not\sqsubseteq p$ then by (Kl), there should exist some $\tilde{n} \in \mathbb{N}$ such that $(f \sqcup i)^{\tilde{n}}(\bot) \not\sqsubseteq p$. By (A1), the rule (Unfold) can be applied only when $(f \sqcup i)^{n-1}(\bot) \sqsubseteq x_{n-1} \sqsubseteq p$. Since (Unfold) increases n and n is never decreased by other rules, then (Unfold) can be applied at most $\tilde{n}$ times.

The elements of negative sequences are introduced by rules (Candidate) and (Decide). If we guarantee that for any index (n, k) the heuristic in such cases returns a finite number of values for z, then one can prove termination. To make this formal, we fix $CaD^h_{n,k} \overset{\text{def}}{=} \{(\boldsymbol{x}\|\boldsymbol{y})_{n,k} \in \mathcal{S}^h \mid (\boldsymbol{x}\|\boldsymbol{y})_{n,k} \overset{Ca}{\to} \text{ or } (\boldsymbol{x}\|\boldsymbol{y})_{n,k} \overset{D}{\to}\}$, i.e., the set of all (n, k)-indexed states reachable by $\texttt{AdjointPDR}_h$ that trigger (Candidate) or (Decide), and $h(CaD^h_{n,k}) \overset{\text{def}}{=} \{h(s) \mid s \in CaD^h_{n,k}\}$, i.e., the set of all possible values returned by h in such states.

Theorem 12 (Negative termination). *Let h be a legit heuristic. If $h(CaD^h_{n,k})$ is finite for all n, k and $\mu(f \sqcup i) \not\sqsubseteq p$, then $\texttt{AdjointPDR}_h$ terminates.*

Corollary 13. *Let h be a simple heuristic. If $\mu(f \sqcup i) \not\sqsubseteq p$, then $\texttt{AdjointPDR}_h$ terminates.*

Note that this corollary ensures negative termination whenever we use the canonical choices in (Candidate) and (Decide) *irrespective of the choice for* (Conflict), therefore it holds for both simple initial and simple final heuristics.

4 Recovering Adjoints with Lower Sets

In the previous section, we have introduced an algorithm for checking $\mu b \sqsubseteq p$ whenever b is of the form $f \sqcup i$ for an element $i \in L$ and a left-adjoint $f\colon L \to L$. This, unfortunately, is not the case for several interesting problems, like the max reachability problem [1] that we will illustrate in Sect. 5.

The next result informs us that, under standard assumptions, one can transfer the problem of checking $\mu b \sqsubseteq p$ to lower sets, where adjoints can always be defined. Recall that, for a lattice $(L, \sqsubseteq)$, a *lower set* is a subset $X \subseteq L$ such that if $x \in X$ and $x' \sqsubseteq x$ then $x' \in X$; the set of lower sets of L forms a complete lattice $(L^{\downarrow}, \subseteq)$ with joins and meets given by union and intersection; as expected $\bot$ is $\emptyset$ and $\top$ is L. Given $b\colon L \to L$, one can define two functions $b^{\downarrow}, b^{\downarrow}_r\colon L^{\downarrow} \to L^{\downarrow}$ as $b^{\downarrow}(X) \overset{\text{def}}{=} b(X)^{\downarrow}$ and $b^{\downarrow}_r(X) \overset{\text{def}}{=} \{x \mid b(x) \in X\}$. It holds that $b^{\downarrow} \dashv b^{\downarrow}_r$.

$$\begin{tikzcd}
(L, \sqsubseteq) \arrow[loop left, "b"] \arrow[r, bend right, "(-)^{\downarrow}"'] \arrow[r, phantom, "\bot"] & (L^{\downarrow}, \subseteq) \arrow[l, bend right, "\bigsqcup"'] \arrow[loop right, "b^{\downarrow} \dashv b^{\downarrow}_r"]
\end{tikzcd} \tag{4}$$

In the diagram above, $(-)^{\downarrow}\colon x \mapsto \{x' \mid x' \sqsubseteq x\}$ and $\bigsqcup\colon L^{\downarrow} \to L$ maps a lower set X into $\bigsqcup\{x \mid x \in X\}$. The maps $\bigsqcup$ and $(-)^{\downarrow}$ form a *Galois insertion*, namely $\bigsqcup \dashv (-)^{\downarrow}$ and $\bigsqcup(-)^{\downarrow} = id$, and thus one can think of (4) in terms of *abstract interpretation* [8,9]: $L^{\downarrow}$ represents the concrete domain, L the abstract domain and b is a sound abstraction of $b^{\downarrow}$. Most importantly, it turns out that b is *forward-complete* [4,14] w.r.t. $b^{\downarrow}$, namely the following equation holds.

$$(-)^{\downarrow} \circ b = b^{\downarrow} \circ (-)^{\downarrow} \tag{5}$$

Proposition 14. *Let $(L, \sqsubseteq)$ be a complete lattice, $p \in L$ and $b\colon L \to L$ be a ω-continuous map. Then $\mu b \sqsubseteq p$ iff $\mu(b^{\downarrow} \cup \bot^{\downarrow}) \subseteq p^{\downarrow}$.*

By means of Proposition 14, we can thus solve $\mu b \sqsubseteq p$ in L by running `AdjointPDR` on $(\bot^{\downarrow}, b^{\downarrow}, b^{\downarrow}_r, p^{\downarrow})$. Hereafter, we tacitly assume that b is ω-continuous.

4.1 `AdjointPDR`$^{\downarrow}$: Positive Chain in L, Negative Sequence in $L^{\downarrow}$

While `AdjointPDR` on $(\bot^{\downarrow}, b^{\downarrow}, b^{\downarrow}_r, p^{\downarrow})$ might be computationally expensive, it is the first step toward the definition of an efficient algorithm that exploits a convenient form of the positive chain.

A lower set $X \in L^{\downarrow}$ is said to be a *principal* if $X = x^{\downarrow}$ for some $x \in L$. Observe that the top of the lattice $(L^{\downarrow}, \subseteq)$ is a principal, namely $\top^{\downarrow}$, and that the meet (intersection) of two principals $x^{\downarrow}$ and $y^{\downarrow}$ is the principal $(x \sqcap y)^{\downarrow}$.

Suppose now that, in (Conflict), `AdjointPDR`$(\bot^{\downarrow}, b^{\downarrow}, b^{\downarrow}_r, p^{\downarrow})$ always chooses principals rather than arbitrary lower sets. This suffices to guarantee that all the elements of $\boldsymbol{x}$ are principals (with the only exception of x_0 which is constantly the bottom element of $L^{\downarrow}$ that, note, is $\emptyset$ and not $\bot^{\downarrow}$). In fact, the elements of

```
                         AdjointPDR$^{\downarrow}$ $(b, p)$

<INITIALISATION>
  $(\boldsymbol{x}\|\boldsymbol{Y})_{n,k}$ := $(\emptyset, \bot, \top\|\varepsilon)_{3,3}$
<ITERATION>
  case $(\boldsymbol{x}\|\boldsymbol{Y})_{n,k}$ of                    % $\boldsymbol{x}, \boldsymbol{Y}$ not conclusive
      $\boldsymbol{Y} = \varepsilon$ and $x_{n-1} \sqsubseteq p$ :                  %(Unfold)
         $(\boldsymbol{x}\|\boldsymbol{Y})_{n,k}$ := $(\boldsymbol{x}, \top\|\varepsilon)_{n+1,n+1}$
      $\boldsymbol{Y} = \varepsilon$ and $x_{n-1} \not\sqsubseteq p$ :                  %(Candidate)
         choose $Z \in L^{\downarrow}$ such that $x_{n-1} \notin Z$ and $p \in Z$;
         $(\boldsymbol{x}\|\boldsymbol{Y})_{n,k}$ := $(\boldsymbol{x}\|Z)_{n,n-1}$
      $\boldsymbol{Y} \neq \varepsilon$ and $b(x_{k-1}) \notin Y_k$ :               %(Decide)
         choose $Z \in L^{\downarrow}$ such that $x_{k-1} \notin Z$ and $b_r^{\downarrow}(Y_k) \subseteq Z$;
         $(\boldsymbol{x}\|\boldsymbol{Y})_{n,k}$ := $(\boldsymbol{x}\|Z, \boldsymbol{Y})_{n,k-1}$
      $\boldsymbol{Y} \neq \varepsilon$ and $b(x_{k-1}) \in Y_k$ :               %(Conflict)
         choose $z \in L$ such that $z \in Y_k$ and $b(x_{k-1} \sqcap z) \sqsubseteq z$;
         $(\boldsymbol{x}\|\boldsymbol{Y})_{n,k}$ := $(\boldsymbol{x} \sqcap_k z\|\mathsf{tail}(\boldsymbol{Y}))_{n,k+1}$
  endcase
<TERMINATION>
   if $\exists j \in [0, n-2] \,.\, x_{j+1} \sqsubseteq x_j$ then return true   % $\boldsymbol{x}$ conclusive
   if $Y_1 = \emptyset$ then return false                   % $\boldsymbol{Y}$ conclusive
```

Fig. 4. The algorithm AdjointPDR$^{\downarrow}$ for checking $\mu b \sqsubseteq p$: the elements of negative sequence are in $L^{\downarrow}$, while those of the positive chain are in L, with the only exception of x_0 which is constantly the bottom lower set $\emptyset$. For x_0, we fix $b(x_0) = \bot$.

$\boldsymbol{x}$ are all obtained by (Unfold), that adds the principal $\top^{\downarrow}$, and by (Conflict), that takes their meets with the chosen principal.

Since principals are in bijective correspondence with the elements of L, by imposing to AdjointPDR$(\bot^{\downarrow}, b^{\downarrow}, b_r^{\downarrow}, p^{\downarrow})$ to choose a principal in (Conflict), we obtain an algorithm, named AdjointPDR$^{\downarrow}$, where the elements of the positive chain are drawn from L, while the negative sequence is taken in $L^{\downarrow}$. The algorithm is reported in Fig. 4 where we use the notation $(\boldsymbol{x}\|\boldsymbol{Y})_{n,k}$ to emphasize that the elements of the negative sequence are lower sets of elements in L.

All definitions and results illustrated in Sect. 3 for AdjointPDR are inherited[1] by AdjointPDR$^{\downarrow}$, with the only exception of Proposition 6.3. The latter does not hold, as it prescribes a choice for (Conflict) that may not be a principal. In contrast, the choice in Proposition 6.4 is, thanks to (5), a principal. This means in particular that the simple initial heuristic is always applicable.

Theorem 15. *All results in Sect. 3, but Proposition 6.3, hold for AdjointPDR$^{\downarrow}$.*

4.2 AdjointPDR$^{\downarrow}$ Simulates LT-PDR

The closest approach to AdjointPDR and AdjointPDR$^{\downarrow}$ is the lattice-theoretic extension of the original PDR, called LT-PDR [19]. While these algorithms exploit essentially the same positive chain to find an invariant, the main difference lies in the sequence used to witness the existence of some counterexamples.

[1] Up to a suitable renaming: the domain is $(L^{\downarrow}, \subseteq)$ instead of $(L, \sqsubseteq)$, the parameters are $\bot^{\downarrow}, b^{\downarrow}, b_r^{\downarrow}, p^{\downarrow}$ instead of i, f, g, p and the negative sequence is $\boldsymbol{Y}$ instead of $\boldsymbol{y}$.

Definition 16 (Kleene sequence, from [19]). *A sequence $\boldsymbol{c} = c_k, \ldots, c_{n-1}$ of elements of L is a* Kleene sequence *if the conditions* (C1) *and* (C2) *below hold. It is* conclusive *if also condition* (C0) *holds.*

$$\text{(C0) } c_1 \sqsubseteq b(\bot), \qquad \text{(C1) } c_{n-1} \not\sqsubseteq p, \qquad \text{(C2) } \forall j \in [k, n-2].\ c_{j+1} \sqsubseteq b(c_j).$$

LT-PDR tries to construct an under-approximation c_{n-1} of $b^{n-2}(\bot)$ that violates the property p. The Kleene sequence is constructed by trial and error, starting by some arbitrary choice of c_{n-1}.

AdjointPDR crucially differs from LT-PDR in the search for counterexamples: LT-PDR under-approximates the final chain while AdjointPDR over-approximates it. The algorithms are thus incomparable. However, we can draw a formal correspondence between AdjointPDR$^{\downarrow}$ and LT-PDR by showing that AdjointPDR$^{\downarrow}$ simulates LT-PDR, but cannot be simulated by LT-PDR. In fact, AdjointPDR$^{\downarrow}$ exploits the existence of the adjoint to start from an over-approximation Y_{n-1} of $p^{\downarrow}$ and computes backward an over-approximation of the set of safe states. Thus, the key difference comes from the strategy to look for a counterexample: to prove $\mu b \not\sqsubseteq p$, AdjointPDR$^{\downarrow}$ tries to find Y_{n-1} satisfying $p \in Y_{n-1}$ and $\mu b \notin Y_{n-1}$ while LT-PDR tries to find c_{n-1} s.t. $c_{n-1} \not\sqsubseteq p$ and $c_{n-1} \sqsubseteq \mu b$.

Theorem 17 below states that any execution of LT-PDR can be mimicked by AdjointPDR$^{\downarrow}$. The proof exploits a map from LT-PDR's Kleene sequences $\boldsymbol{c}$ to AdjointPDR$^{\downarrow}$'s negative sequences $\boldsymbol{neg(c)}$ of a particular form. Let $(L^{\uparrow}, \supseteq)$ be the complete lattice of upper sets, namely subsets $X \subseteq L$ such that $X = X^{\uparrow} \overset{\text{def}}{=} \{x' \in L \mid \exists x \in X \,.\, x \sqsubseteq x'\}$. There is an isomorphism $\neg\colon (L^{\uparrow}, \supseteq) \overset{\cong}{\longleftrightarrow} (L^{\downarrow}, \subseteq)$ mapping each $X \subseteq S$ into its complement. For a Kleene sequence $\boldsymbol{c} = c_k, \ldots, c_{n-1}$ of LT-PDR, the sequence $\boldsymbol{neg(c)} \overset{\text{def}}{=} \neg(\{c_k\}^{\uparrow}), \ldots, \neg(\{c_{n-1}\}^{\uparrow})$ is a negative sequence, in the sense of Definition 3, for AdjointPDR$^{\downarrow}$. Most importantly, the assignment $\boldsymbol{c} \mapsto \boldsymbol{neg(c)}$ extends to a function, from the states of LT-PDR to those of AdjointPDR$^{\downarrow}$, that is proved to be a *strong simulation* [24].

Theorem 17. AdjointPDR$^{\downarrow}$ *simulates LT-PDR.*

Remarkably, AdjointPDR$^{\downarrow}$'s negative sequences are not limited to the images of LT-PDR's Kleene sequences: they are more general than the complement of the upper closure of a singleton. In fact, a single negative sequence of AdjointPDR$^{\downarrow}$ can represent *multiple* Kleene sequences of LT-PDR at once. Intuitively, this means that a single execution of AdjointPDR$^{\downarrow}$ can correspond to multiple runs of LT-PDR. We can make this formal by means of the following result.

Proposition 18. *Let $\{\boldsymbol{c^m}\}_{m \in M}$ be a family of Kleene sequences. Then its pointwise intersection $\bigcap_{m \in M} \boldsymbol{neg(c^m)}$ is a negative sequence.*

The above intersection is pointwise in the sense that, for all $j \in [k, n-1]$, it holds $(\bigcap_{m \in M} \boldsymbol{neg(c^m)})_j \overset{\text{def}}{=} \bigcap_{m \in M} (\boldsymbol{neg(c^m)})_j = \neg(\{c_j^m \mid m \in M\}^{\uparrow})$: intuitively, this is (up to $\boldsymbol{neg(\cdot)}$) a set containing all the M counterexamples. Note that, if the negative sequence of `AdjointPDR`$^{\downarrow}$ makes (A3) hold as an equality, as it is possible with any simple heuristic (see Proposition 10), then its complement contains *all* Kleene sequences possibly computed by LT-PDR.

Proposition 19. *Let* $\boldsymbol{c}$ *be a Kleene sequence and* $\boldsymbol{Y}$ *be the negative sequence s.t.* $Y_j = (b_r^{\downarrow})^{n-1-j}(p^{\downarrow})$ *for all* $j \in [k, n-1]$*. Then* $c_j \in \neg(Y_j)$ *for all* $j \in [k, n-1]$*.*

While the previous result suggests that simple heuristics are always the best in theory, as they can carry all counterexamples, this is often not the case in practice, since they might be computationally hard and outperformed by some smart over-approximations. An example is given by (6) in the next section.

5 Instantiating `AdjointPDR`$^{\downarrow}$ for MDPs

In this section we illustrate how to use `AdjointPDR`$^{\downarrow}$ to address the max reachability problem [1] for Markov Decision Processes.

A *Markov Decision Process* (MDP) is a tuple (A, S, s_ι, δ) where A is a set of labels, S is a set of states, $s_\iota \in S$ is an initial state, and $\delta \colon S \times A \to \mathcal{D}S + 1$ is a transition function. Here $\mathcal{D}S$ is the set of probability distributions over S, namely functions $d \colon S \to [0,1]$ such that $\sum_{s \in S} d(s) = 1$, and $\mathcal{D}S + 1$ is the disjoint union of $\mathcal{D}S$ and $1 = \{*\}$. The transition function δ assigns to every label $a \in A$ and to every state $s \in S$ either a distribution of states or $* \in 1$. We assume that both S and A are finite sets and that the set $Act(s) \overset{\text{def}}{=} \{a \in A \mid \delta(s,a) \neq *\}$ of actions enabled at s is non-empty for all states.

Intuitively, the *max reachability problem* requires to check whether the probability of reaching some bad states $\beta \subseteq S$ is less than or equal to a given threshold $\lambda \in [0,1]$. Formally, it can be expressed in lattice theoretic terms, by considering the lattice $([0,1]^S, \leq)$ of all functions $d \colon S \to [0,1]$, often called frames, ordered pointwise. The max reachability problem consists in checking $\mu b \leq p$ for $p \in [0,1]^S$ and $b \colon [0,1]^S \to [0,1]^S$, defined for all $d \in [0,1]^S$ and $s \in S$, as

$$p(s) \overset{\text{def}}{=} \begin{cases} \lambda & \text{if } s = s_\iota, \\ 1 & \text{if } s \neq s_\iota, \end{cases} \qquad b(d)(s) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } s \in \beta, \\ \max\limits_{a \in Act(s)} \sum\limits_{s' \in S} d(s') \cdot \delta(s,a)(s') & \text{if } s \notin \beta. \end{cases}$$

The reader is referred to [1] for all details.

Since b is not of the form $f \sqcup i$ for a left adjoint f (see e.g. [19]), rather than using `AdjointPDR`, one can exploit `AdjointPDR`$^{\downarrow}$. Beyond the simple initial heuristic, which is always applicable and enjoys negative termination, we illustrate now two additional heuristics that are experimentally tested in Sect. 6.

The two novel heuristics make the same choices in (Candidate) and (Decide). They exploit functions $\alpha \colon S \to A$, also known as memoryless schedulers, and the function $b_\alpha \colon [0,1]^S \to [0,1]^S$ defined for all $d \in [0,1]^S$ and $s \in S$ as follows:

$$b_\alpha(d)(s) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } s \in \beta, \\ \sum_{s' \in S} d(s') \cdot \delta(s, \alpha(s))(s') & \text{otherwise.} \end{cases}$$

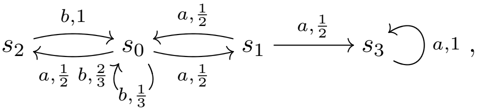
Since for all $D \in ([0,1]^S)^{\downarrow}$, $b_r^{\downarrow}(D) = \{d \mid b(d) \in D\} = \bigcap_{\alpha}\{d \mid b_{\alpha}(d) \in D\}$ and since $\texttt{AdjointPDR}^{\downarrow}$ executes (Decide) only when $b(x_{k-1}) \notin Y_k$, there should exist some α such that $b_{\alpha}(x_{k-1}) \notin Y_k$. One can thus fix

$$(\boldsymbol{x}\|\boldsymbol{Y})_{n,k} \mapsto \begin{cases} p^{\downarrow} & \text{if } (\boldsymbol{x}\|\boldsymbol{Y})_{n,k} \overset{Ca}{\rightarrow} \\ \{d \mid b_{\alpha}(d) \in Y_k\} & \text{if } (\boldsymbol{x}\|\boldsymbol{Y})_{n,k} \overset{D}{\rightarrow} \end{cases} \tag{6}$$

Intuitively, such choices are smart refinements of those in (3): for (Candidate) they are exactly the same; for (Decide) rather than taking $b_r^{\downarrow}(Y_k)$, we consider a larger lower-set determined by the labels chosen by α. This allows to represent each Y_j as a set of $d \in [0,1]^S$ satisfying a *single* linear inequality, while using $b_r^{\downarrow}(Y_k)$ would yield a systems of possibly exponentially many inequalities (see Example 21 below). Moreover, from Theorem 12, it follows that such choices ensures negative termination.

Corollary 20. *Let h be a legit heuristic defined for (Candidate) and (Decide) as in (6). If $\mu b \not\leq p$, then $\texttt{AdjointPDR}^{\downarrow}{}_h$ terminates.*

Example 21. Consider the maximum reachability problem with threshold $\lambda = \frac{1}{4}$ and $\beta = \{s_3\}$ for the following MDP on alphabet $A = \{a, b\}$ and $s_{\iota} = s_0$.



Hereafter we write $d \in [0,1]^S$ as column vectors with four entries $v_0 \ldots v_3$ and we will use $\cdot$ for the usual matrix multiplication. With this notation, the lower set $p^{\downarrow} \in ([0,1]^S)^{\downarrow}$ and $b\colon [0,1]^S \to [0,1]^S$ can be written as

$$p^{\downarrow} = \{\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \mid \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \leq \begin{bmatrix} \frac{1}{4} \end{bmatrix}\} \quad \text{and} \quad b(\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}) = \begin{bmatrix} \max(\frac{v_1+v_2}{2}, \frac{v_0+2v_2}{3}) \\ \frac{v_0+v_3}{2} \\ v_0 \\ 1 \end{bmatrix}.$$

Amongst the several memoryless schedulers, only two are relevant for us: $\zeta \overset{\text{def}}{=} (s_0 \mapsto a,\ s_1 \mapsto a,\ s_2 \mapsto b,\ s_3 \mapsto a)$ and $\xi \overset{\text{def}}{=} (s_0 \mapsto b,\ s_1 \mapsto a,\ s_2 \mapsto b,\ s_3 \mapsto a)$. By using the definition of $b_{\alpha}\colon [0,1]^S \to [0,1]^S$, we have that

$$b_{\zeta}(\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}) = \begin{bmatrix} \frac{v_1+v_2}{2} \\ \frac{v_0+v_3}{2} \\ v_0 \\ 1 \end{bmatrix} \qquad \text{and} \qquad b_{\xi}(\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}) = \begin{bmatrix} \frac{v_0+2v_2}{3} \\ \frac{v_0+v_3}{2} \\ v_0 \\ 1 \end{bmatrix}.$$

It is immediate to see that the problem has negative answer, since using ζ in 4 steps or less, s_0 can reach s_3 already with probability $\frac{1}{4} + \frac{1}{8}$.

To illustrate the advantages of (6), we run $\texttt{AdjointPDR}^{\downarrow}$ with the simple initial heuristic and with the heuristic that only differs for the choice in (Decide), taken as in (6). For both heuristics, the first iterations are the same: several

$$
\begin{array}{l|c|c}
\mathcal{F}^0 \overset{\text{def}}{=} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}1&0&0&0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{4}\end{bmatrix}\right\} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}1&0&0&0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{4}\end{bmatrix}\right\} \\
\mathcal{F}^1 \overset{\text{def}}{=} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0&1&1&0\\1&0&2&0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{2}\\\frac{3}{4}\end{bmatrix}\right\} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0&\frac{1}{2}&\frac{1}{2}&0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{4}\end{bmatrix}\right\} \\
\mathcal{F}^2 \overset{\text{def}}{=} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}3&0&0&1\\2&1&1&0\\4&0&2&0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}1\\\frac{3}{2}\\\frac{9}{4}\end{bmatrix}\right\} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}\frac{3}{4}&0&0&\frac{1}{4}\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{4}\end{bmatrix}\right\} \\
\mathcal{F}^3 \overset{\text{def}}{=} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0&\frac{3}{2}&\frac{3}{2}&0\\1&0&2&0\\\frac{3}{2}&1&1&\frac{1}{2}\\\frac{13}{6}&0&\frac{4}{3}&\frac{1}{2}\\2&2&2&0\\\frac{10}{3}&0&\frac{8}{3}&0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}0\\0\\\frac{3}{2}\\\frac{3}{2}\\\frac{9}{4}\\\frac{9}{4}\end{bmatrix}\right\} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0&\frac{3}{8}&\frac{3}{8}&0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}0\end{bmatrix}\right\} \\
\mathcal{F}^4 \overset{\text{def}}{=} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}0\\0\\0\\0\end{bmatrix}\right\} = \left\{\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\right\} & \left\{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}\frac{9}{16}&0&0&\frac{3}{16}\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}0\end{bmatrix}\right\} \\
\mathcal{F}^5 \overset{\text{def}}{=} & \emptyset & \emptyset
\end{array}
$$

Fig. 5. The elements of the negative sequences computed by $\mathtt{AdjointPDR}^{\downarrow}$ for the MDP in Example 21. In the central column, these elements are computed by means of the simple initial heuristics, that is $\mathcal{F}^i = (b_r^{\downarrow})^i(p^{\downarrow})$. In the rightmost column, these elements are computed using the heuristic in (6). In particular $\mathcal{F}^i = \{d \mid b_\zeta(d) \in \mathcal{F}^{i-1}\}$ for $i \leq 3$, while for $i \geq 4$ these are computed as $\mathcal{F}^i = \{d \mid b_\xi(d) \in \mathcal{F}^{i-1}\}$.

repetitions of (Candidate), (Conflict) and (Unfold) exploiting elements of the positive chain that form the initial chain (except for the last element x_{n-1}).

$$
\left(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|\varepsilon\right)_{3,3} \xrightarrow{Ca}\xrightarrow{Co} \left(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\|\varepsilon\right)_{3,3} \xrightarrow{U}\xrightarrow{Ca}\xrightarrow{Co}\xrightarrow{U}\xrightarrow{Ca}\xrightarrow{Co}\xrightarrow{U}\xrightarrow{Ca}\xrightarrow{Co}\xrightarrow{U}\xrightarrow{Ca} \left(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{1}{2}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{4}\\\frac{1}{2}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{4}\\\frac{5}{8}\\\frac{1}{4}\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow}\right)_{7,6}.
$$

In the latter state the algorithm has to perform (Decide), since $b(x_5) \notin p^{\downarrow}$. Now the choice of z in (Decide) is different for the two heuristics: the former uses $b_r^{\downarrow}(p^{\downarrow}) = \{d \mid b(d) \in p^{\downarrow}\}$, the latter uses $\{d \mid b_\zeta(d) \in p^{\downarrow}\}$. Despite the different choices, both the heuristics proceed with 6 steps of (Decide):

$$
\left(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{1}{2}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{4}\\\frac{1}{2}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{4}\\\frac{5}{8}\\\frac{1}{4}\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|\mathcal{F}^0\right)_{7,6} \xrightarrow{D}\xrightarrow{D}\xrightarrow{D}\xrightarrow{D}\xrightarrow{D} \left(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{1}{2}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{4}\\\frac{1}{2}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{4}\\\frac{5}{8}\\\frac{1}{4}\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|\mathcal{F}^5,\mathcal{F}^4,\mathcal{F}^3,\mathcal{F}^2,\mathcal{F}^1,\mathcal{F}^0\right)_{7,1}
$$

The element of the negative sequence $\mathcal{F}^i$ are illustrated in Fig. 5 for both the heuristics. In both cases, $\mathcal{F}^5 = \emptyset$ and thus $\mathtt{AdjointPDR}^{\downarrow}$ returns false.

To appreciate the advantages provided by (6), it is enough to compare the two columns for the $\mathcal{F}^i$ in Fig. 5: in the central column, the number of inequalities defining $\mathcal{F}^i$ significantly grows, while in the rightmost column is always 1.

Whenever Y_k is generated by a single linear inequality, we observe that $Y_k = \{d \in [0,1]^S \mid \sum_{s \in S}(r_s \cdot d(s)) \leq r\}$ for suitable non-negative real numbers r and r_s for all $s \in S$. The convex set Y_k is generated by finitely many $d \in [0,1]^S$ enjoying a convenient property: $d(s)$ is different from 0 and 1 only for at most one $s \in S$. The set of its generators, denoted by $\mathcal{G}_k$, can thus be easily computed.

We exploit this property to resolve the choice for (Conflict). We consider its sub set $\mathcal{Z}_k \overset{\text{def}}{=} \{d \in \mathcal{G}_k \mid b(x_{k-1}) \leq d\}$ and define $z_B, z_{01} \in [0,1]^S$ for all $s \in S$ as

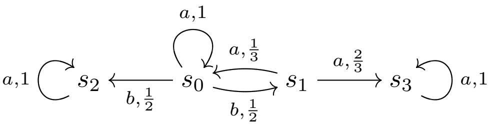
$$z_B(s) \overset{\text{def}}{=} \begin{cases} (\bigwedge \mathcal{Z}_k)(s) & \text{if } r_s \neq 0, \mathcal{Z}_k \neq \emptyset \\ b(x_{k-1})(s) & \text{otherwise} \end{cases} \qquad z_{01}(s) \overset{\text{def}}{=} \begin{cases} \lceil z_B(s) \rceil & \text{if } r_s = 0, \mathcal{Z}_k \neq \emptyset \\ z_B(s) & \text{otherwise} \end{cases} \tag{7}$$

where, for $u \in [0,1]$, $\lceil u \rceil$ denotes 0 if $u = 0$ and 1 otherwise. We call `hCoB` and `hCo01` the heuristics defined as in (6) for (Candidate) and (Decide) and as z_B, respectively z_{01}, for (Conflict). The heuristics `hCo01` can be seen as a Boolean modification of `hCoB`, rounding up positive values to 1 to accelerate convergence.

Proposition 22. *The heuristics* `hCoB` *and* `hCo01` *are legit.*

By Corollary 20, $\texttt{AdjointPDR}^{\downarrow}$ terminates for negative answers with both `hCoB` and `hCo01`. We conclude this section with a last example.

Example 23. Consider the following MDP with alphabet $A = \{a, b\}$ and $s_\iota = s_0$



and the max reachability problem with threshold $\lambda = \frac{2}{5}$ and $\beta = \{s_3\}$. The lower set $p^{\downarrow} \in ([0,1]^S)^{\downarrow}$ and $b\colon [0,1]^S \to [0,1]^S$ can be written as

$$p^{\downarrow} = \{\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \mid \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \leq \begin{bmatrix} \frac{2}{5} \end{bmatrix}\} \quad \text{and} \quad b(\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}) = \begin{bmatrix} \max(v_0, \frac{v_1+v_2}{2}) \\ \frac{v_0+2\cdot v_3}{3} \\ v_2 \\ 1 \end{bmatrix}$$

With the simple initial heuristic, $\texttt{AdjointPDR}^{\downarrow}$ does not terminate. With the heuristic `hCo01`, it returns true in 14 steps, while with `hCoB` in 8. The first 4 steps, common to both `hCoB` and `hCo01`, are illustrated below.

$$(\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \| \varepsilon)_{3,3} \xrightarrow{Ca} (\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \| p^{\downarrow})_{3,2} \xrightarrow{Co} (\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} \frac{2}{5} \\ 0 \\ 0 \\ 1 \end{bmatrix} \| \varepsilon)_{3,3} \qquad b(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \mathcal{Z}_2 = \{\begin{bmatrix} \frac{2}{5} \\ 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} \frac{2}{5} \\ 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} \frac{2}{5} \\ 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} \frac{2}{5} \\ 1 \\ 1 \\ 1 \end{bmatrix}\}$$

$$\xrightarrow{U} \xrightarrow{Ca} (\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} \frac{2}{5} \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \| p^{\downarrow})_{4,3} \xrightarrow{Co} \Big| (\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} \frac{2}{5} \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} \frac{2}{5} \\ 1 \\ 0 \\ 1 \end{bmatrix} \| \varepsilon)_{4,4} \Big| (\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} \frac{2}{5} \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} \frac{2}{5} \\ \frac{4}{5} \\ 0 \\ 1 \end{bmatrix} \| \varepsilon)_{4,4} \qquad b(\begin{bmatrix} \frac{2}{5} \\ 0 \\ 0 \\ 1 \end{bmatrix}) = \begin{bmatrix} \frac{2}{5} \\ \frac{4}{5} \\ 0 \\ 1 \end{bmatrix} \quad \mathcal{Z}_3 = \{\begin{bmatrix} \frac{2}{5} \\ 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} \frac{2}{5} \\ 1 \\ 1 \\ 1 \end{bmatrix}\}$$

Observe that in the first (Conflict) $z_B = z_{01}$, while in the second $z_{01}(s_1) = 1$ and $z_B(s_1) = \frac{4}{5}$, leading to the two different states prefixed by vertical lines.

6 Implementation and Experiments

We first developed, using Haskell and exploiting its abstraction features, a common template that accommodates both `AdjointPDR` and $\texttt{AdjointPDR}^{\downarrow}$. It is a program parametrized by two lattices—used for positive chains and negative sequences, respectively—and by a heuristic.

For our experiments, we instantiated the template to $\texttt{AdjointPDR}^{\downarrow}$ for MDPs (letting $L = [0,1]^S$), with three different heuristics: `hCoB` and `hCo01` from Proposition 22; and `hCoS` introduced below. Besides the template (∼100 lines), we needed ∼140 lines to account for `hCoB` and `hCo01`, and additional ∼100 lines to further obtain `hCoS`. All this indicates a clear benefit of our abstract theory: a general template can itself be coded succinctly; instantiation to concrete problems is easy, too, thanks to an explicitly specified interface of heuristics.

Our implementation accepts MDPs expressed in a symbolic format inspired by Prism models [20], in which states are variable valuations and transitions are described by symbolic functions (they can be segmented with symbolic guards $\{\mathrm{guard}_i\}_i$). We use rational arithmetic (`Rational` in Haskell) for probabilities to limit the impact of rounding errors.

Heuristics. The three heuristics (`hCoB`, `hCo01`, `hCoS`) use the same choices in (Candidate) and (Decide), as defined in (6), but different ones in (Conflict).

The third heuristics `hCoS` is a *symbolic* variant of `hCoB`; it relies on our symbolic model format. It uses z_S for z in (Conflict), where $z_S(s) = z_B(s)$ if $r_s \neq 0$ or $\mathcal{Z}_k = \emptyset$. The definition of $z_S(s)$ otherwise is notable: we use a piecewise affine function $(t_i \cdot s + u_i)_i$ for $z_S(s)$, where the affine functions $(t_i \cdot s + u_i)_i$ are guarded by the same guards $\{\mathrm{guard}_i\}_i$ of the MDP's transition function. We let the SMT solver Z3 [25] search for the values of the coefficients t_i, u_i, so that z_S satisfies the requirements of (Conflict) (namely $b(x_{k-1})(s) \leq z_S(s) \leq 1$ for each $s \in S$ with $r_s = 0$), together with the condition $b(z_S) \leq z_S$ for faster convergence. If the search is unsuccessful, we give up `hCoS` and fall back on the heuristic `hCoB`.

As a task common to the three heuristics, we need to calculate $\mathcal{Z}_k = \{d \in \mathcal{G}_k \mid b(x_{k-1}) \leq d\}$ in (Conflict) (see (7)). Rather than computing the whole set $\mathcal{G}_k$ of generating points of the linear inequality that defines Y_k, we implemented an ad-hoc algorithm that crucially exploits the condition $b(x_{k-1}) \leq d$ for pruning.

Experiment Settings. We conducted the experiments on Ubuntu 18.04 and AWS t2.xlarge (4 CPUs, 16 GB memory, up to 3.0 GHz Intel Scalable Processor). We used several Markov chain (MC) benchmarks and a couple of MDP ones.

Research Questions. We wish to address the following questions.

RQ1 Does $\texttt{AdjointPDR}^{\downarrow}$ advance the state-of-the-art performance of *PDR* algorithms for probabilistic model checking?

RQ2 How does $\texttt{AdjointPDR}^{\downarrow}$'s performance compare against *non-PDR* algorithms for probabilistic model checking?

RQ3 Does the theoretical framework of $\texttt{AdjointPDR}^{\downarrow}$ successfully guide the discovery of various heuristics with practical performance?

RQ4 Does $\texttt{AdjointPDR}^{\downarrow}$ successfully manage nondeterminism in MDPs (that is absent in MCs)?

Experiments on MCs (Table 1). We used six benchmarks: Haddad-Monmege is from [17]; the others are from [3,19]. We compared $\texttt{AdjointPDR}^{\downarrow}$ (with three

Table 1. Experimental results on MC benchmarks. $|S|$ is the number of states, P is the reachability probability (calculated by manual inspection), λ is the threshold in the problem $P \leq_? \lambda$ (shaded if the answer is no). The other columns show the average execution time in seconds; TO is timeout (900 s); MO is out-of-memory. For `AdjointPDR`$^\downarrow$ and LT-PDR we used the `tasty-bench` Haskell package and repeated executions until std. dev. is $< 5\%$ (at least three execs). For PrIC3 and Storm, we made five executions. Storm's execution does not depend on λ: it seems to answer queries of the form $P \leq_? \lambda$ by calculating P. We observed a wrong answer for the entry with (†) (Storm, sp.-num., Haddad-Monmege); see the discussion of RQ2.

Benchmark	$\lvert S\rvert$	P	λ	`AdjointPDR`$^\downarrow$			LT-PDR	PrIC3				Storm		
				`hCoB`	`hCo01`	`hCoS`		none	lin.	pol.	hyb.	sp.-num.	sp.-rat.	sp.-sd.
Grid	10^2	0.033	0.3	0.013	0.022	0.659	0.343	1.383	23.301	MO	MO	0.010	0.010	0.010
			0.2	0.013	0.031	0.657	0.519	1.571	26.668	TO	MO			
	10^3	<0.001	0.3	1.156	2.187	5.633	126.441	TO	TO	TO	MO	0.010	0.017	0.011
			0.2	1.146	2.133	5.632	161.667	TO	TO	TO	MO			
BRP	10^3	0.035	0.1	12.909	7.969	55.788	TO	TO	TO	MO	MO	0.012	0.018	0.011
			0.01	1.977	8.111	5.645	21.078	60.738	626.052	524.373	823.082			
			0.005	0.604	2.261	2.709	1.429	12.171	254.000	197.940	318.840			
Zero-Conf	10^2	0.5	0.9	1.217	68.937	0.196	TO	19.765	136.491	0.630	0.468	0.010	0.018	0.011
			0.75	1.223	68.394	0.636	TO	19.782	132.780	0.602	0.467			
			0.52	1.228	60.024	0.739	TO	19.852	136.533	0.608	0.474			
			0.45	<0.001	0.001	0.001	<0.001	0.035	0.043	0.043	0.043			
	10^4	0.5	0.9	MO	TO	7.443	TO	TO	TO	0.602	0.465	0.037	262.193	0.031
			0.75	MO	TO	15.223	TO	TO	TO	0.599	0.470			
			0.52	MO	TO	TO	TO	TO	TO	0.488	0.475			
			0.45	0.108	0.119	0.169	0.016	0.035	0.040	0.040	0.040			
Chain	10^3	0.394	0.9	36.083	TO	0.478	TO	269.801	TO	0.938	0.686	0.010	0.014	0.011
			0.4	35.961	TO	394.955	TO	271.885	TO	0.920	TO			
			0.35	101.351	TO	454.892	435.199	238.613	TO	TO	TO			
			0.3	62.036	463.981	120.557	209.346	124.829	746.595	TO	TO			
Double-Chain	10^3	0.215	0.9	12.122	7.318	TO	TO	TO	TO	1.878	2.053	0.011	0.018	0.010
			0.3	12.120	20.424	TO	TO	TO	TO	1.953	2.058			
			0.216	12.096	19.540	TO	TO	TO	TO	172.170	TO			
			0.15	12.344	16.172	TO	16.963	TO	TO	TO	TO			
Haddad-Monmege	41	0.7	0.9	0.004	0.009	8.528	TO	1.188	31.915	TO	MO	0.011	0.011	1.560
			0.75	0.004	0.011	2.357	TO	1.209	32.143	TO	712.086			
	10^3	0.7	0.9	59.721	61.777	TO	TO	TO	TO	TO	TO	0.013 (†)	0.043	TO
			0.75	60.413	63.050	TO	TO	TO	TO	TO	TO			

heuristics) against LT-PDR [19], PrIC3 (with four heuristics *none*, *lin.*, *pol.*, *hyb.*, see [3]), and Storm 1.5 [11]. Storm is a recent comprehensive toolsuite that implements different algorithms and solvers. Among them, our comparison is against *sparse-numeric*, *sparse-rational*, and *sparse-sound*. The *sparse* engine uses explicit state space representation by sparse matrices; this is unlike another representative *dd* engine that uses symbolic BDDs. (We did not use *dd* since it often reported errors, and was overall slower than *sparse*.) *Sparse-numeric* is a value-iteration (VI) algorithm; *sparse-rational* solves linear (in)equations using rational arithmetic; *sparse-sound* is a sound VI algorithm [26].[2]

[2] There are another two sound algorithms in Storm: one that utilizes interval iteration [2] and the other does optimistic VI [16]. We have excluded them from the results since we observed that they returned incorrect answers.

Table 2. Experimental results on MDP benchmarks. The legend is the same as Table 1, except that P is now the maximum reachability probability.

Benchmark	$\lvert S\rvert$	P	λ	`AdjointPDR`$^{\downarrow}$			Storm		
				`hCoB`	`hCo01`	`hCoS`	sp.-num	sp.-rat.	sp.-sd.
CDrive2	38	0.865	0.9	MO	0.172	TO	0.019	0.019	0.018
			0.75	MO	0.058	TO			
			0.5	0.015	0.029	86.798			
TireWorld	8670	0.233	0.9	MO	3.346	TO	0.070	0.164	0.069
			0.75	MO	3.337	TO			
			0.5	MO	6.928	TO			
			0.2	4.246	24.538	TO			

Experiments on MDPs (Table 2). We used two benchmarks from [17]. We compared `AdjointPDR`$^{\downarrow}$ only against Storm, since RQ1 is already addressed using MCs (besides, PrIC3 did not run for MDPs).

Discussion. The experimental results suggest the following answers to the RQs.

RQ1. The performance advantage of `AdjointPDR`$^{\downarrow}$, over both LT-PDR and PrIC3, was clearly observed throughout the benchmarks. `AdjointPDR`$^{\downarrow}$ outperformed LT-PDR, thus confirming empirically the theoretical observation in Sect. 4.2. The profit is particularly evident in those instances whose answer is positive. `AdjointPDR`$^{\downarrow}$ generally outperformed PrIC3, too. Exceptions are in ZeroConf, Chain and DoubleChain, where PrIC3 with polynomial (pol.) and hybrid (hyb.) heuristics performs well. This seems to be thanks to the expressivity of the polynomial template in PrIC3, which is a possible enhancement we are yet to implement (currently our symbolic heuristic `hCoS` uses only the affine template).

RQ2. The comparison with Storm is interesting. Note first that Storm's *sparse-numeric* algorithm is a VI algorithm that gives a guaranteed lower bound *without guaranteed convergence*. Therefore its positive answer to $P \leq_? \lambda$ may not be correct. Indeed, for Haddad-Monmege with $|S| \sim 10^3$, it answered $P = 0.5$ which is wrong ((†) in Table 1). This is in contrast with PDR algorithms that discovers an explicit witness for $P \leq \lambda$ via their positive chain.

Storm's *sparse-rational* algorithm is precise. It was faster than PDR algorithms in many benchmarks, although `AdjointPDR`$^{\downarrow}$ was better or comparable in ZeroConf (10^4) and Haddad-Monmege (41), for λ such that $P \leq \lambda$ is true. We believe this suggests a general advantage of PDR algorithms, namely to accelerate the search for an invariant-like witness for safety.

Storm's *sparse-sound* algorithm is a sound VI algorithm that returns correct answers aside numerical errors. Its performance was similar to that of sparse-numeric, except for the two instances of Haddad-Monmege: sparse-sound returned correct answers but was much slower than sparse-numeric. For these two instances, AdjointPDR$^\downarrow$ outperformed sparse-sound.

It seems that a big part of Storm's good performance is attributed to the sparsity of state representation. This is notable in the comparison of the two instances of Haddad-Monmege (41 vs. 10^3): while Storm handles both of them easily, AdjointPDR$^\downarrow$ struggles a bit in the bigger instance. Our implementation can be extended to use sparse representation, too; this is future work.

RQ3. We derived the three heuristics (hCoB, hCo01, hCoS) exploiting the theory of AdjointPDR$^\downarrow$. The experiments show that each heuristic has its own strength. For example, hCo01 is slower than hCoB for MCs, but it is much better for MDPs. In general, there is no silver bullet heuristic, so coming up with a variety of them is important. The experiments suggest that our theory of AdjointPDR$^\downarrow$ provides great help in doing so.

RQ4. Table 2 shows that AdjointPDR$^\downarrow$ can handle nondeterminism well: once a suitable heuristic is chosen, its performances on MDPs and on MCs of similar size are comparable. It is also interesting that better-performing heuristics vary, as we discussed above.

Summary. AdjointPDR$^\downarrow$ clearly outperforms existing probabilistic PDR algorithms in many benchmarks. It also compares well with Storm—a highly sophisticated toolsuite—in a couple of benchmarks. These are notable especially given that AdjointPDR$^\downarrow$ currently lacks enhancing features such as richer symbolic templates and sparse representation (adding which is future work). Overall, we believe that AdjointPDR$^\downarrow$ *confirms the potential of PDR algorithms in probabilistic model checking*. Through the three heuristics, we also observed the value of an abstract general theory in devising heuristics in PDR, which is probably true of verification algorithms in general besides PDR.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
2. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for Markov decision processes. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 160–180. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_8
3. Batz, K., et al.: PrIC3: property directed reachability for MDPs. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 512–538. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_27
4. Bonchi, F., Ganty, P., Giacobazzi, R., Pavlovic, D.: Sound up-to techniques and complete abstract domains. In: Dawar, A., Grädel, E. (eds.) Proceedings of LICS 2018, pp. 175–184. ACM (2018). https://doi.org/10.1145/3209108.3209169
5. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
6. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_23

7. Cousot, P.: Partial completeness of abstract fixpoint checking. In: Choueiry, B.Y., Walsh, T. (eds.) SARA 2000. LNCS (LNAI), vol. 1864, pp. 1–25. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44914-0_1
8. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL 1977, pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
10. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order, 2nd Edn. Cambridge University Press (2002)
11. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
12. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Bjesse, P., Slobodová, A. (eds.) Proc. of FMCAD 2011. pp. 125–134. FMCAD Inc. (2011). http://dl.acm.org/citation.cfm?id=2157675
13. Feldman, Y.M.Y., Sagiv, M., Shoham, S., Wilcox, J.R.: Property-directed reachability as abstract interpretation in the monotone theory. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498676
14. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47**(2), 361–416 (2000). https://doi.org/10.1145/333979.333989
15. Gurfinkel, A.: IC3, PDR, and friends (2015). https://arieg.bitbucket.io/pdf/gurfinkel_ssft15.pdf
16. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 488–511. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_26
17. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 344–350. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_20
18. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
19. Kori, M., Urabe, N., Katsumata, S., Suenaga, K., Hasuo, I.: The lattice-theoretic essence of property directed reachability analysis. In: Shoham, S., Vizel, Y. (eds.) Proceedings of CAV 2022, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 235–256. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-13185-1_12
20. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
21. Lange, T., Neuhäußer, M.R., Noll, T., Katoen, J.-P.: IC3 software model checking. Int. J. Softw. Tools Technol. Trans. **22**(2), 135–161 (2019). https://doi.org/10.1007/s10009-019-00547-x
22. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis, Semantics Structures in Computation, vol. 2. Springer, Dordrecht (2004). https://doi.org/10.1007/978-94-007-0954-6
23. MacLane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, vol. 5. Springer-Verlag, New York (1971)

24. Milner, R.: Communication and Concurrency. Prentice-Hall Inc, USA (1989)
25. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. Quatmann, T., Katoen, J.-P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 643–661. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_37
27. Seufert, T., Scholl, C.: Sequential verification using reverse PDR. In: Große, D., Drechsler, R. (eds.) Proceedings of MBMV 2017, pp. 79–90. Shaker Verlag (2017)
28. Seufert, T., Scholl, C.: Combining PDR and reverse PDR for hardware model checking. In: Madsen, J., Coskun, A.K. (eds.) Proceedings of DATE 2018, pp. 49–54. IEEE (2018). https://doi.org/10.23919/DATE.2018.8341978
29. Seufert, T., Scholl, C.: fbPDR: In-depth combination of forward and backward analysis in property directed reachability. In: Teich, J., Fummi, F. (eds.) Proceedings of DATE 2019, pp. 456–461. IEEE (2019). https://doi.org/10.23919/DATE.2019.8714819
30. Suda, M.: Property directed reachability for automated planning. In: Chien, S.A., Do, M.B., Fern, A., Ruml, W. (eds.) Proceedings of ICAPS 2014. AAAI (2014). https://doi.org/10.1613/jair.4231

Fast Approximations of Quantifier Elimination

Isabel Garcia-Contreras[1](✉), V. K. Hari Govind[1], Sharon Shoham[2], and Arie Gurfinkel[1]

[1] University of Waterloo, Waterloo, Canada
{igarciac,hgvedira,agurfink}@uwaterloo.ca
[2] Tel-Aviv University, Tel Aviv, Israel
sharon.shoham@cs.tau.ac.il

Abstract. Quantifier elimination (qelim) is used in many automated reasoning tasks including program synthesis, exist-forall solving, quantified SMT, Model Checking, and solving Constrained Horn Clauses (CHCs). Exact qelim is computationally expensive. Hence, it is often approximated. For example, Z3 uses "light" pre-processing to reduce the number of quantified variables. CHC-solver Spacer uses model-based projection (MBP) to under-approximate qelim relative to a given model, and over-approximations of qelim can be used as abstractions.

In this paper, we present the QEL framework for fast approximations of qelim. QEL provides a uniform interface for both quantifier reduction and model-based projection. QEL builds on the egraph data structure – the core of the EUF decision procedure in SMT – by casting quantifier reduction as a problem of choosing *ground* (i.e., variable-free) representatives for equivalence classes. We have used QEL to implement MBP for the theories of Arrays and Algebraic Data Types (ADTs). We integrated QEL and our new MBP in Z3 and evaluated it within several tasks that rely on quantifier approximations, outperforming state-of-the-art.

1 Introduction

Quantifier Elimination (qelim) is used in many automated reasoning tasks including program synthesis [18], exist-forall solving [8,9], quantified SMT [5], and Model Checking [17]. Complete qelim, even when possible, is computationally expensive, and solvers often approximate it. We call these approximations *quantifier reductions*, to separate them from qelim. The difference is that quantifier reduction might leave some free variables in the formula.

For example, Z3 [19] performs quantifier reduction, called QeLite, by greedily substituting variables by definitions syntactically appearing in the formulas. While it is very useful, it is necessarily sensitive to the order in which variables are substituted and depends on definitions appearing explicitly in the formula. Even though it may seem that these shortcomings need to be tolerated to keep QeLite fast, in this paper we show that it is not actually the case; we propose an egraph-based algorithm, QEL, to perform fast quantifier reduction that is complete relative to some semantic properties of the formula.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 64–86, 2023.
https://doi.org/10.1007/978-3-031-37703-7_4

Egraph [20] is a data structure that compactly represents infinitely many terms and their equivalence classes. It was initially proposed as a decision procedure for EUF [20] and used for theorem proving (e.g., SIMPLIFY [7]). Since then, the applications of egraphs have grown. Egraphs are now used as term rewrite systems in equality saturation [15,23], for theory combination in SMT solvers [7,21], and for term abstract domains in Abstract Interpretation [6,10,12].

Using egraphs for rewriting or other formula manipulations (like qelim) requires a special operation, called *extract*, that converts nodes in the egraph back into terms. Term extraction was not considered when egraphs were first designed [20]. As far as we know, extraction was first studied in the application of egraphs for compiler optimization. Specifically, equality saturation [15,22] is an optimization technique over egraphs that consists in populating an egraph with many equivalent terms inferred by applying rules. When the egraph is saturated, i.e., applying the rules has no effect, the equivalent term that is most desired, e.g., smallest in size, is *extracted*. This is a recursive process that extracts each sub-term by choosing one representative among its equivalents.

Application of egraphs to rewriting have recently resurged driven by the `egg` library [24] and the associated workshop[1]. In [24], the authors show, once again, the power and versatility of this data structure. Motivated by applications of equality saturation, they provide a generic and efficient framework equipped with term extraction, based on an extensible class analysis.

Egraphs seem to be the perfect data-structure to address the challenges of quantifier reduction: they allow reasoning about infinitely many equivalent terms and consider all available variable definitions and orderings at once. However, things are not always what they appear. The key to quantifier reduction is finding ground (i.e., variable-free) representatives for equivalence classes with free variables. This goes against existing techniques for term extraction since it requires selecting larger, rather than smaller, terms to be representatives. Selecting representatives carelessly makes term extraction diverge. To our surprise, this problem has not been studied so far. In fact, `egg` [24] incorrectly claims that any representative function can be used with its term extraction, while the implementation diverges. In this paper, we bridge this gap by providing necessary and sufficient conditions for a representative function to be admissible for term extraction as defined in [15,24]. Furthermore, we extend extraction from terms to formulas to enable extracting a formula of the egraph.

Our main contribution is a new quantifier reduction algorithm, called QEL. Building on the term extraction described above, it is formulated as finding a representative function that maximizes the number of ground terms as representatives. Furthermore, it greedily attempts to represent variables without ground representatives in terms of other variables, thus further reducing the number of variables in the output. We show that QEL is complete relative to ground definitions entailed by the formula. Specifically, QEL guarantees to eliminate a variable if it is equivalent to a ground term.

[1] https://pldi22.sigplan.org/series/egraphs.

Whenever an application requires eliminating all free variables, incomplete techniques such as QELite or QEL are insufficient. In this case, qelim is under-approximated using a Model-based Projection (MBP) that uses a model M of a formula to guide under-approximation using equalities and variable definitions that are consistent with M. In this paper, we show that MBP can be implemented using our new techniques for QEL together with the machinery from equality saturation. Just like SMT solvers use egraphs as glue to combine different theory solvers, we use egraphs as glue to combine projection for different theories. In particular, we give an algorithm for MBP in the combined theory of Arrays and Algebraic DataTypes (ADTs). The algorithm uses insights from QEL to produce less under-approximate MBPs.

We implemented QEL and the new MBP using egraphs inside the state-of-art SMT solver Z3 [19]. Our implementation (referred to as Z3eg) replaces the existing QeLite and MBP. We evaluate our algorithms in two contexts. First, inside the QSAT [5] algorithm for quantified satisfiability. The performance of QSAT in Z3eg is improved, compared to QSAT in Z3, when ADTs are involved. Second, we evaluate our algorithms inside the Constrained Horn Clause (CHC) solver Spacer [17]. Our experiments show that Spacer in Z3eg solves many more benchmarks containing nested Arrays and ADTs.

Related Work. Quantifier reduction by variable substitution is widely used in quantified SMT [5,11]. To our knowledge, we are the first to look at this problem semantically and provide an algorithm that guarantees that the variable is eliminated if the formula entails that it has a ground definition.

Term extraction for egraphs comes from equality saturation [15,22]. The egg Rust library [24] is a recent implementation of equality saturation that supports rewriting and term extraction. However, we did not use egg because we integrated QEL within Z3 and built it using Z3 data structures instead.

Model-based projection was first introduced for the Spacer CHC solver for LIA and LRA [17] and extended to the theory of Arrays [16] and ADTs [5]. Until now, it was implemented by syntactic rewriting. Our egraph-based MBP implementation is less sensitive to syntax and, more importantly, allows for combining MBPs of multiple theories for MBP of the combination. As a result, our MBP is more general and less model dependent. Specifically, it requires fewer model equalities and produces more general under-approximations than [5,16].

Outline. The rest of the paper is organized as follows. Section 2 provides background. Section 3 introduces term extraction, extends it to formulas, and characterizes representative-based term extraction for egraphs. Section 4 presents QEL, our algorithm for fast quantifier reduction that is relatively complete. Section 5 shows how to compute MBP combining equality saturation and the ideas from Sect. 4 for the theories of ADTs and Arrays. All algorithms have been implemented in Z3 and evaluated in Sect. 6.

2 Background

We assume the reader is familiar with multi-sorted first-order logic (FOL) with equality and the theory of equality with uninterpreted functions (EUF) (for an introduction see, e.g. [4]). We use $\approx$ to denote the designated logical equality symbol. For simplicity of presentation, we assume that the FOL signature Σ contains only functions (i.e., no predicates) and constants (i.e., 0-ary functions). To represent predicates, we assume the FOL signature has a designated sort Bool, and two Bool constants $\top$ and $\bot$, representing true, and false respectively. We then use Bool-valued functions to represent predicates, using $P(a) \approx \top$ and $P(a) \approx \bot$ to mean that $P(a)$ is true or false, respectively. Informally, we continue to write $P(a)$ and $\neg P(a)$ as a syntactic sugar for $P(a) \approx \top$ and $P(a) \approx \bot$, respectively. We use lowercase letters like a, b for constants, and f, g for functions, and uppercase letters like P, Q for Bool functions that represent predicates. We denote by $\psi^\exists$ the existential closure of ψ.

Quantifier Elimination (qelim). Given a quantifier-free (QF) formula φ with free variables $\boldsymbol{v}$, *quantifier elimination* of $\varphi^\exists$ is the problem of finding a QF formula ψ with no free variables such that $\psi \equiv \varphi^\exists$. For example, a qelim of $\exists a \cdot (a \approx x \wedge f(a) > 3)$ is $f(x) > 3$; and, there is no qelim of $\exists x \cdot (f(x) > 3)$, because it is impossible to restrict f to have "at least one value in its range that is greater than 3" without a quantifier.

Model Based Projection (MBP). Let φ be a formula with free variables $\boldsymbol{v}$, and M a model of φ. A *model-based projection* of φ relative to M is a QF formula ψ such that $\psi \Rightarrow \varphi^\exists$ and $M \models \psi$. That is, ψ has no free variables, is an under-approximation of φ, and satisfies the designated model M, just like φ. MBP is used by many algorithms to under-approximate qelim, when the computation of qelim is too expensive or, for some reason, undesirable.

Egraphs. An egraph is a well-known data structure to compactly represent a set of terms and an equivalence relation on those terms [20]. Throughout the paper, we assume that graphs have an ordered successor relation and use $n[i]$ to denote the ith successor (child) of a node n. An out-degree of a node n, $\mathtt{deg}(n)$, is the number of edges leaving n. Given a node n, $\mathtt{parents}(n)$ denotes the set of nodes with an outgoing edge to n and $\mathtt{children}(n)$ denotes the set of nodes with an incoming edge from n.

Definition 1. *Let Σ be a first-order logic signature. An* egraph *is a tuple $G = \langle N, E, L, \mathtt{root}\rangle$, where*

(a) $\langle N, E\rangle$ is a directed acyclic graph,
(b) L maps nodes to function symbols in Σ or logical variables, and
(c) $\mathtt{root} : N \mapsto N$ maps a node to its root *such that the relation $\rho_{\mathtt{root}} \triangleq \{(n, n') \mid \mathtt{root}(n) = \mathtt{root}(n')\}$ is an equivalence relation on N that is* closed under congruence*: $(n, n') \in \rho_{\mathtt{root}}$ whenever n and n' are* congruent *under $\mathtt{root}$, i.e., whenever $L(n) = L(n')$, $\mathtt{deg}(n) = \mathtt{deg}(n') > 0$, and, $\forall 1 \leq i \leq \mathtt{deg}(n) \cdot (n[i], n'[i]) \in \rho_{\mathtt{root}}$.*

$$\varphi_1(x, y, z) \triangleq z \approx \mathit{read}(a, x) \wedge k + 1 \approx \mathit{read}(a, y) \wedge x \approx y \wedge 3 > z$$

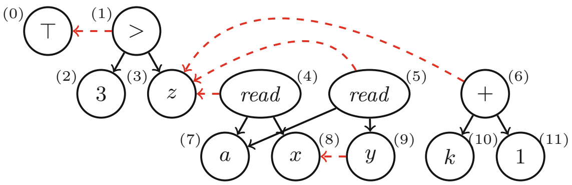


Fig. 1. Example egraph of φ_1.

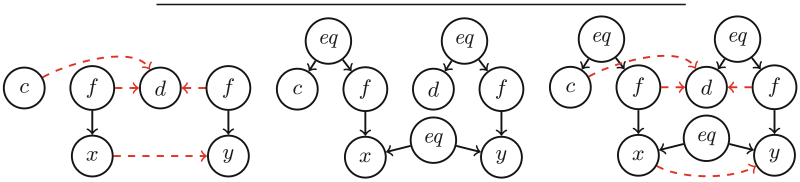
Given an egraph G, the *class* of a node $n \in G$, $\mathit{class}(n) \triangleq \rho_{\texttt{root}}(n)$, is the set of all nodes that are equivalent to n. The term of n, $\mathit{term}(n)$, with $L(n) = f$ is f if $\texttt{deg}(n) = 0$ and $f(\mathit{term}(n[1]), \ldots, \mathit{term}(n[\texttt{deg}(n)]))$, otherwise. We assume that the terms of different nodes are different, and refer to a node n by its term.

An example of an egraph $G = \langle N, E, L, \texttt{root}\rangle$ is shown in Fig. 1. A symbol f inside a circle depicts a node n with label $L(n) = f$, solid black and dashed red arrows depict E and $\texttt{root}$, respectively. The order of the black arrows from left to right defines the order of the children. In our examples, we refer to a specific node i by its number using $\mathsf{N}(i)$ or its term, e.g., $\mathsf{N}(k + 1)$. A node n without an outgoing red arrow is its own root. A set of nodes connected to the same node with red edges forms an equivalence class. In this example, $\texttt{root}$ defines the equivalence classes $\{\mathsf{N}(3), \mathsf{N}(4), \mathsf{N}(5), \mathsf{N}(6)\}$, $\{\mathsf{N}(8), \mathsf{N}(9)\}$, and a class for each of the remaining nodes. Examples of some terms in G are $\mathit{term}(\mathsf{N}(9)) = y$ and $\mathit{term}(\mathsf{N}(5)) = \mathit{read}(a, y)$.

An Egraph of a Formula. We consider formulas that are conjunctions of equality literals (recall that we represent predicate applications by equality literals). Given a formula $\varphi \triangleq (t_1 \approx u_1 \wedge \cdots \wedge t_k \approx u_k)$, an egraph from φ is built (following the standard procedure [20]) by creating nodes for each t_i and u_i, recursively creating nodes for their subexpressions, and merging the classes of each pair t_i and u_i, computing the congruence closure for $\texttt{root}$. We write $\mathit{egraph}(\varphi)$ for an egraph of φ constructed via some deterministic procedure based on the recipe above. Figure 1 shows an $\mathit{egraph}(\varphi_1)$ of φ_1. The equality $z \approx \mathit{read}(a, x)$ is captured by $\mathsf{N}(3)$ and $\mathsf{N}(4)$ belonging to the same class (i.e., red arrow from $\mathsf{N}(4)$ to $\mathsf{N}(3)$). Similarly, the equality $x \approx y$ is captured by a red arrow from $\mathsf{N}(9)$ to $\mathsf{N}(8)$. Note that by congruence, φ_1 implies $\mathit{read}(a, x) \approx \mathit{read}(a, y)$, which, by transitivity, implies that $k + 1 \approx \mathit{read}(a, x)$. In Fig. 1, this corresponds to red arrows from $\mathsf{N}(5)$ and $\mathsf{N}(6)$ to $\mathsf{N}(3)$. The predicate application $3 > z$ is captured by the red arrow from $\mathsf{N}(1)$ to $\mathsf{N}(0)$. From now on, we omit $\top$ and $\bot$ and the corresponding edges from figures to avoid clutter.

Explicit and Implicit Equality. Note that egraphs represent equality implicitly by placing nodes with equal terms in the same equivalence class. Sometimes, it is necessary to represent equality explicitly, for example, when using egraphs for

$$\varphi_2(x, y) \triangleq eq(c, f(x)) \wedge eq(d, f(y)) \wedge eq(x, y)$$



(a) G_a, interpreting *eq* as $\approx$. (b) G_b, not interpreting *eq*. (c) G_c, combining (a) and (b).

Fig. 2. Different egraph interpretations for φ_2.

equality-aware rewriting (e.g., in egg [24]). To represent equality explicitly, we introduce a binary Bool function *eq* and write $eq(a, b)$ for an equality that has to be represented explicitly. We change the *egraph* algorithm to treat $eq(a, b)$ as both a function application, and as a logical equality $a \approx b$: when processing term $eq(a, b)$, the algorithm both adds $eq(a, b)$ to the egraph, and merges the nodes for a and b into one class. For example, Fig. 2 shows three different interpretations of a formula φ_2 with equality interpreted: implicitly (as in [20]), explicitly (as in [24]), and both implicitly and explicitly (as in this paper).

3 Extracting Formulas from Egraphs

Egraphs were proposed as a decision procedure for EUF [20] – a setting in which converting an egraph back to a formula, or *extracting*, is irrelevant. Term extraction has been studied in the context of equality saturation and term rewriting [15,24]. However, existing literature presents extraction as a heuristic, and, to the best of our knowledge, has not been exhaustively explored. In this section, we fill these gaps in the literature and extend extraction from terms to formulas.

Term Extraction. We begin by recalling how to extract the term of a node. The function ntt (node-to-term) in Fig. 3 does an extraction parametrized by a representative function $\mathtt{repr} : N \mapsto N$ (same as in [24]). A function repr assigns each class a unique representative node (i.e., nodes in the same class are mapped to the same representative) so that $\rho_{\mathtt{root}} = \rho_{\mathtt{repr}}$. The function ntt extracts a term of a node recursively, similarly to *term*, except that the representatives of the children of a node are used instead of the actual children. We refer to terms built in this way by $\mathtt{ntt}(n, \mathtt{repr})$ and omit repr when it is clear from the context.

As an example, consider $\mathtt{repr}_1 \triangleq \{\mathsf{N}(3), \mathsf{N}(8)\}$ for Fig. 1. For readability, we denote representative functions by sets of nodes that are the class representatives, omitting $\mathsf{N}(\top)$ that always represents its class, and omitting all singleton classes. Thus, $\mathtt{repr}_1$ maps all nodes in $class(\mathsf{N}(3))$ to $\mathsf{N}(3)$, nodes in $class(\mathsf{N}(8))$ to $\mathsf{N}(8)$, nodes in $class(\mathsf{N}(\top))$ to $\mathsf{N}(\top)$, and all singleton classes to themselves. For example, $\mathtt{ntt}(\mathsf{N}(5))$ extracts $read(a, x)$, since $\mathsf{N}(9)$ has as representative $\mathsf{N}(8)$.

```
$egraph$ :: to_formula(repr, $S$)
  $Lits := \emptyset$
  for $r = \texttt{repr}(r) \in N$ do
    $t := \texttt{ntt}(r, \texttt{repr})$
    for $n \in (class(r) \setminus r)$ do
      if $n \notin S$ then
        $Lits := Lits \cup \{t \approx \texttt{ntt}(n, \texttt{repr})\}$
  ret $\bigwedge Lits$
```

```
$egraph$ :: ntt($n$, repr)
  $f := L[n]$
  if $\texttt{deg}(n) = 0$ then
    ret $f$
  else
    for $i \in [1, \texttt{deg}(n)]$ do
      $Args[i] := \texttt{ntt}(\texttt{repr}(n[i]), \texttt{repr})$
    ret $f(Args)$
```

Fig. 3. Producing formulas from an egraph.

Formula Extraction. Let $G = egraph(\varphi)$ be an egraph of some formula φ. A formula ψ is a *formula of* G, written $isFormula(G, \psi)$, if $\psi^{\exists} \equiv \varphi^{\exists}$.

Figure 3 shows an algorithm `to_formula`(`repr`, S) to compute a formula ψ that satisfies $isFormula(G, \psi)$ for a given egraph G. In addition to `repr`, `to_formula` is parameterized by a set of nodes $S \subseteq N$ to exclude[2]. To produce the equalities corresponding to the classes, for each representative r, for each $n \in (class(r) \setminus \{r\})$ the output formula has a literal $\texttt{ntt}(r) \approx \texttt{ntt}(n)$. For example, using $\texttt{repr}_1$ for the egraph in Fig. 1, we obtain for $class(\mathsf{N}(8))$, $(x \approx y)$; for $class(\mathsf{N}(3))$, $(z \approx read(a, x) \wedge z \approx read(a, x) \wedge z \approx k + 1)$; and for $class(\mathsf{N}(0))$, $(\top \approx 3 > z)$. The final result (slightly simplified) is: $x \approx y \wedge z \approx read(a, x) \wedge z \approx k + 1 \wedge 3 > z$.

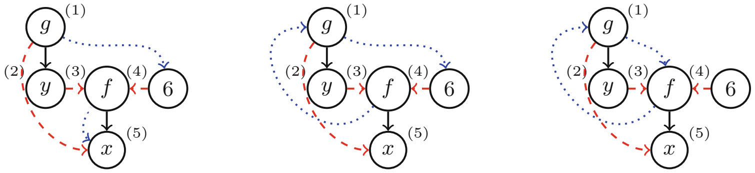
Let $G = egraph(\varphi)$ for some formula φ. Note that, ψ computed by `to_formula` is not syntactically the same as φ. That is, `to_formula` is not an inverse of *egraph*. Furthermore, since `to_formula` commits to one representative per class, it is limited in what formulas it can generate. For example, since $x \approx y$ is in φ_1, for any `repr`, φ_1 cannot be the result of `to_formula`, because the output can contain only one of $read(a, x)$ or $read(a, y)$.

Representative Functions. The representative function is instrumental for determining the terms that appear in the extracted formula. To illustrate the importance of representative choice, consider the formula φ_4 of Fig. 4 and its egraph $G_4 = egraph(\varphi_4)$. For now, ignore the blue dotted lines. For $\texttt{repr}_{4a}$, `to_formula` obtains $\psi_a \triangleq (x \approx g(6) \wedge f(x) \approx 6 \wedge y \approx 6)$. For $\texttt{repr}_{4b}$, `to_formula` produces $\psi_b \triangleq (g(6) \approx x \wedge f(g(6)) \approx 6 \wedge y \approx 6)$. In some applications (like qelim considered in this paper) ψ_b is preferred to ψ_a: simply removing the literals $g(6) \approx x$ and $y \approx 6$ from ψ_b results in a formula equivalent to $\exists x, y \cdot \varphi_4$ that does not contain variables. Consider a third representative choice $\texttt{repr}_{4c}$, for node $\mathsf{N}(1)$, `ntt` does not terminate: to produce a term for $\mathsf{N}(1)$, a term for $\mathsf{N}(3)$, the representative of its child, $\mathsf{N}(2)$, is required. Similarly to produce a term for $\mathsf{N}(3)$, a term for the representative of its child node $\mathsf{N}(5)$, $\mathsf{N}(1)$, is necessary. Thus, none of the terms can be extracted with $\texttt{repr}_{4c}$.

For extraction, representative functions `repr` are either provided explicitly or implicitly (as in [24]), the latter by associating a cost to nodes and/or terms and

[2] The set S affects the result, but for this section, we restrict to the case of $S \triangleq \emptyset$.

$\varphi_4(x,y) \triangleq y \approx f(x) \wedge x \approx g(y) \wedge f(x) \approx 6$



(a) $\texttt{repr}_{4a} \triangleq \{\mathsf{N}(4), \mathsf{N}(5)\}$. (b) $\texttt{repr}_{4b} \triangleq \{\mathsf{N}(4), \mathsf{N}(1)\}$. (c) $\texttt{repr}_{4c} \triangleq \{\mathsf{N}(3), \mathsf{N}(1)\}$.

Fig. 4. Egraphs of φ_4 with $G_{\texttt{repr}}$ (Color figure online).

letting the representative be a node with minimal cost. However, observe that not all costs guarantee that the chosen $\texttt{repr}$ can be used (the computation does not terminate). For example, the ill-defined $\texttt{repr}_{4c}$ from above is a representative function that satisfies the cost function that assigns function applications cost 0 and variables and constants cost 1. A commonly used cost function is term AST size, which is sufficient to ensure termination of $\texttt{ntt}(n, \texttt{repr})$.

We are thus interested in characterizing representative functions motivated by two observations: not every cost function guarantees that $\texttt{ntt}(n)$ terminates; and the kind of representative choices that are most suitable for qelim ($\texttt{repr}_{4b}$) cannot be expressed over term AST size.

Definition 2. *Given an egraph $G = \langle N, E, L, \texttt{root}\rangle$, a representative function $\texttt{repr} : N \to N$ is* admissible for G *if*

(a) $\texttt{repr}$ assigns a unique representative per class,
(b) $\rho_{\texttt{root}} = \rho_{\texttt{repr}}$, and
(c) the graph $G_{\texttt{repr}}$ is acyclic, where $G_{\texttt{repr}} = \langle N, E_{\texttt{repr}}\rangle$ and $E_{\texttt{repr}} \triangleq \{(n, \texttt{repr}(c)) \mid c \in \textit{children}(n), n \in N\}$.

Dotted blue edges in the graphs of Fig. 4 show the corresponding $G_{\texttt{repr}}$. Intuitively, for each node n, all reachable nodes in $G_{\texttt{repr}}$ are the nodes whose $\texttt{ntt}$ term is necessary to produce the $\texttt{ntt}(n)$. Observe that $G_{\texttt{repr}_{4c}}$ has a cycle, thus, $\texttt{repr}_{4c}$ is not admissible.

Theorem 1. *Given an egraph G and a representative function $\texttt{repr}$, the function $G.\texttt{to\_formula}(\texttt{repr}, \emptyset)$ terminates with result ψ such that isFormula(G, ψ) iff $\texttt{repr}$ is admissible for G.*

To the best of our knowledge, Theorem 1 is the first complete characterization of all terms of a node that can be obtained by extraction based on class representatives (via describing all admissible $\texttt{repr}$, note that the number is finite). This result contradicts [24], where it is claimed to be possible to extract a term of a node for any cost function. The counterexample is $\texttt{repr}_{4c}$. Importantly, this characterization allows us to explore representative functions outside those in the existing literature, which, as we show in the next section, is key for qelim.

Input: A formula φ with free variables $\boldsymbol{v}$.
Output: A quantifier reduction of φ.

```
QEL(φ, v)
1: G := egraph(φ)
2: repr := G.find_defs(v)
3: repr := G.refine_defs(repr, v)
4: core := G.find_core(repr)
5: ret G.to_formula(repr, G.Nodes() \ core)
```

Algorithm 1: QEL – Quantifier reduction using egraphs.

4 Quantifier Reduction

Quantifier reduction is a relaxation of quantifier elimination: given two formulas φ and ψ with free variables $\boldsymbol{v}$ and $\boldsymbol{u}$, respectively, ψ is a *quantifier reduction* of φ if $\boldsymbol{u} \subseteq \boldsymbol{v}$ and $\varphi^{\exists} \equiv \psi^{\exists}$. If $\boldsymbol{u}$ is empty, then ψ is a quantifier elimination of $\varphi^{\exists}$. Note that quantifier reduction is possible even when quantifier elimination is not (e.g., for EUF). We are interested in an efficient quantifier reduction algorithm (that can be used as pre-processing for qelim), even if a complete qelim is possible (e.g., for LIA). In this section, we present such an algorithm called QEL.

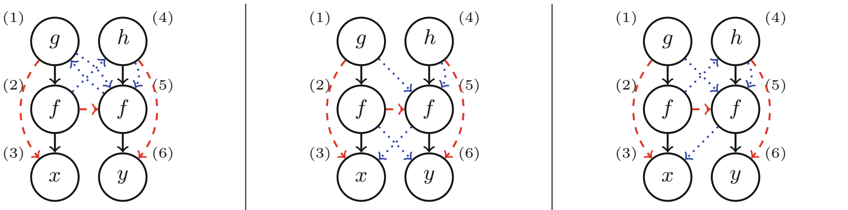
Intuitively, QEL is based on the well-known substitution rule: $(\exists x \cdot x \approx t \wedge \varphi) \equiv \varphi[x \mapsto t]$. A naive implementation of this rule, called QELite in Z3, looks for syntactic definitions of the form $x \approx t$ for a variable x and an x-free term t and substitutes x with t. While efficient, QELite is limited because of: (a) dependence on syntactic equality in the formula (specifically, it misses implicit equalities due to transitivity and congruence); (b) sensitivity to the order in which variables are eliminated (eliminating one variable may affect available syntactic equalities for another); and (c) difficulty in dealing with circular equalities such as $x \approx f(x)$.

For example, consider the formula $\varphi_4(x, y)$ in Fig. 4. Assume that y is eliminated first using $y \approx f(x)$, resulting in $x \approx g(f(x)) \wedge f(x) \approx 6$. Now, x cannot be eliminated since the only equality for x is circular. Alternatively, assume that QELite somehow noticed that by transitivity, φ_4 implies $y \approx 6$, and obtains $(\exists y \cdot \varphi_4) \triangleq x \approx g(6) \wedge f(x) \approx 6$. This time, $x \approx g(6)$ can be used to obtain $f(g(6)) \approx 6$ that is a qelim of $\varphi_4^{\exists}$. Thus, both the elimination order and implicit equalities are crucial.

In QEL, we address the above issues by using an egraph data structure to concisely capture all implicit equalities and terms. Furthermore, egraphs allow eliminating multiple variables together, ensuring that a variable is eliminated if it is equivalent (explicitly or implicitly) to a ground term in the egraph.

Pseudocode for QEL is shown in Algorithm 1. Given an input formula φ, QEL first builds its egraph G (line 1). Then, it finds a representative function `repr` that maps variables to equivalent ground terms, as much as possible (line 2). Next, it further reduces the remaining free variables by refining `repr` to map each variable x to an equivalent x-free (but not variable-free) term (line 3). At this point, QEL is committed to the variables to eliminate. To produce the output, `find_core` identifies the subset of the nodes of G, which we call *core*,

$$\varphi_5 \triangleq x \approx g(f(x)) \wedge y \approx h(f(y)) \wedge f(x) \approx f(y)$$



(a) $\texttt{repr}_{5a} = \{\mathsf{N}(1), \mathsf{N}(4), \mathsf{N}(5)\}$ (b) $\texttt{repr}_{5b} = \{\mathsf{N}(3), \mathsf{N}(6), \mathsf{N}(5)\}$ (c) $\texttt{repr}_{5c} = \{\mathsf{N}(1), \mathsf{N}(6), \mathsf{N}(5)\}$

Fig. 5. Egraphs including $G_{\texttt{repr}}$ (Color figure online) of φ_5.

that must be considered in the output (line 4). Finally, `to_formula` converts the core of G to the resulting formula (line 5). We show that the combination of these steps is even stronger than variable substitution.

To illustrate QEL, we apply it on φ_1 and its egraph G from Fig. 1. The function `find_defs` returns `repr` $= \{\mathsf{N}(6), \mathsf{N}(8)\}$[3]. Node $\mathsf{N}(6)$ is the only node with a ground term in the equivalence class $\mathit{class}(\mathsf{N}(3))$. This corresponds to the definition $z \approx k + 1$. Node $\mathsf{N}(8)$ is chosen arbitrarily since $\mathit{class}(\mathsf{N}(8))$ has no ground terms. There is no refinement possible, so `refine_defs` returns `repr`. The core is $N \setminus \{\mathsf{N}(3), \mathsf{N}(5), \mathsf{N}(9)\}$. Nodes $\mathsf{N}(3)$ and $\mathsf{N}(9)$ are omitted because they correspond to variables with definitions (under `repr`), and $\mathsf{N}(5)$ is omitted because it is congruent to $\mathsf{N}(4)$ so only one of them is needed. Finally, `to_formula` produces $k + 1 \approx \mathit{read}(a, x) \wedge 3 > k + 1$. Variables z and y are eliminated.

In the rest of this section we present QEL in detail and QEL's key properties.

Finding Ground Definitions. Ground variable definitions are found by selecting a representative function `repr` that ensures that the maximum number of terms in the formula are rewritten into ground equivalent ones, which, in turn, means finding a ground definition for all variables that have one.

Computing a representative function `repr` that is admissible and ensures finding ground definitions when they exist is not trivial. Naive approaches for identifying ground terms, such as iterating arbitrarily over the classes and selecting a representative based on $\mathit{term}(n)$ are not enough – $\mathit{term}(n)$ may not be in the output formula. It is also not possible to make a choice based on $\texttt{ntt}(n)$, since, in general, it cannot be yet computed (`repr` is not known yet).

Admissibility raises an additional challenge since choosing a node that appears to be a definition (e.g., not a leaf) may cause cycles in $G_{\texttt{repr}}$. For example, consider φ_5 of Fig. 5. Assume that $\mathsf{N}(1)$ and $\mathsf{N}(4)$ are chosen as representatives of their equivalence classes. At this point, $G_{\texttt{repr}}$ has two edges: $\langle \mathsf{N}(5), \mathsf{N}(4)\rangle$ and $\langle \mathsf{N}(2), \mathsf{N}(1)\rangle$, shown by blue dotted lines in Fig. 5a. Next, if either $\mathsf{N}(2)$ or $\mathsf{N}(5)$ are chosen as representatives (the only choices in their class), then $G_{\texttt{repr}}$

[3] Recall that we only show representatives of non-singleton classes.

```
egraph :: find_defs(v)
 1: for $n \in N$ do repr($n$) := ★
 2: $todo := \{leaf(n) \mid n \in N \wedge ground(n)\}$
 3: repr := process(repr, $todo$)
 4: $todo := \{leaf(n) \mid n \in N\}$
 5: repr := process(repr, $todo$)
 6: ret repr

egraph :: process(repr, todo)
 7: while $todo \neq \emptyset$ do
 8:   $n := todo.pop()$
 9:   if repr($n$) ≠ ★ then continue
10:   for $n' \in class(n)$ do repr($n'$) := $n$
11:   for $n' \in class(n)$ do
12:     for $p \in$ parents($n'$) do
13:       if $\forall c \in$ children($p$) · repr($c$) ≠ ★ then
14:         $todo.push(p)$
15: ret repr
```

Algorithm 2: Find definitions maximizing groundness.

becomes cyclic (shown in blue in Fig. 5a). Furthermore, backtracking on representative choices needs to be avoided if we are to find a representative function efficiently.

Algorithm 2 finds a representative function `repr` while overcoming these challenges. To ensure that the computed representative function is admissible (without backtracking), Algorithm 2 selects representatives for each class using a "bottom up" approach. Namely, leaves cannot be part of cycles in $G_{\texttt{repr}}$ because they have no outgoing edges. Thus, they can always be safely chosen as representatives. Similarly, a node whose children have already been assigned representatives in this way (leaves initially), will also never be part of a cycle in $G_{\texttt{repr}}$. Therefore, these nodes are also safe to be chosen as representatives.

This intuition is implemented in `find_defs` by initializing `repr` to be undefined (★) for all nodes, and maintaining a workset, *todo*, containing nodes that, if chosen for the remaining classes (under the current selection), maintain acyclicity of $G_{\texttt{repr}}$. The initialization of *todo* includes leaves only. The specific choice of leaves ensures that ground definitions are preferred, and we return to it later. After initialization, the function `process` extracts an element from *todo* and sets it as the representative of its class if the class has not been assigned yet (lines 9 and 10). Once a class representative has been chosen, on lines 11 to 14, the parents of all the nodes in the class such that all the children have been chosen (the condition on line 13) are added to *todo*.

So far, we discussed how admissibility of `repr` is guaranteed. To also ensure that ground definitions are found whenever possible, we observe that a similar bottom up approach identifies terms that can be rewritten into ground ones. This builds on the notion of constructively ground nodes, defined next.

A class c is *ground* if c contains a *constructively ground*, or *c-ground* for short, node n, where a node n is c-ground if either (a) $term(n)$ is ground, or (b) n is not a leaf and the class $class(n[i])$ of every child $n[i]$ is ground. Note that nodes labeled by variables are never c-ground.

In the example in Fig. 1, $class(\mathsf{N}(7))$ and $class(\mathsf{N}(8))$ are not ground, because all their nodes represent variables; $class(\mathsf{N}(6))$ is ground because $\mathsf{N}(6)$ is c-ground. Nodes $\mathsf{N}(4)$ and $\mathsf{N}(5)$ are not c-ground because the class of $\mathsf{N}(8)$ (a child of both nodes) is not ground. Interestingly, $\mathsf{N}(1)$ is c-ground, because $\mathit{class}(\mathsf{N}(3)) = \mathit{class}(\mathsf{N}(6))$ is ground, even though its term $3 > z$ is not ground.

Ground classes and c-ground nodes are of interest because whenever $\varphi \models \mathit{term}(n) \approx t$ for some node n and ground term t, then $\mathit{class}(n)$ is ground, i.e., it contains a c-ground node, where c-ground nodes can be found recursively starting from ground leaves. Furthermore, the recursive definition ensures that when the aforementioned c-ground nodes are selected as representatives, the corresponding terms w.r.t. `repr` are ground.

As a result, to maximize the ground definitions found, we are interested in finding an admissible representative function `repr` that is *maximally ground*, which means that for every node $n \in N$, if $\mathit{class}(n)$ is ground, then $\mathtt{repr}(n)$ is c-ground. That means that c-ground nodes are always chosen if they exist.

Theorem 2. *Let $G = \mathit{egraph}(\varphi)$ be an egraph and* `repr` *an admissible representative function that is maximally ground. For all $n \in N$, if $\varphi \models \mathit{term}(n) \approx t$ for some ground term t, then* $\mathtt{repr}(n)$ *is c-ground and* $\mathtt{ntt}(\mathtt{repr}(n))$ *is ground.*

We note that not every choice of c-ground nodes as representatives results in an admissible representative function. For example, consider the formula φ_4 of Fig. 4 and its egraph. All nodes except for $\mathsf{N}(5)$ and $\mathsf{N}(2)$ are c-ground. However, a `repr` with $\mathsf{N}(3)$ and $\mathsf{N}(1)$ as representatives is not admissible. Intuitively, this is because the "witness" for c-groundness of $\mathsf{N}(1)$ in $\mathit{class}(\mathsf{N}(2))$ is $\mathsf{N}(4)$ and not $\mathsf{N}(3)$. Therefore, it is important to incorporate the selection of c-ground representatives into the bottom up procedure that ensures admissibility of `repr`.

To promote c-ground nodes over non c-ground in the construction of an admissible representative function, `find_defs` chooses representatives in two steps. First, only the ground leaves are processed (line 2). This ensures that c-ground representatives are chosen while guaranteeing the absence of cycles. Then, the remaining leaves are added to *todo* (line 4). This triggers representative selection of the remaining classes (those that are not ground).

We illustrate `find_defs` with two examples. For φ_4 of Fig. 4, there is only one leaf that is ground, $\mathsf{N}(4)$, which is added to *todo* on line 2, and *todo* is processed. $\mathsf{N}(4)$ is chosen as representative and, as a consequence, its parent $\mathsf{N}(1)$ is added to *todo*. $\mathsf{N}(1)$ is chosen as representative so $\mathsf{N}(3)$, even though added to the queue later, is not chosen as representative, obtaining $\mathtt{repr}_{4b} = \{\mathsf{N}(4), \mathsf{N}(1)\}$. For φ_5 of Fig. 5, no nodes are added to *todo* on line 2. $\mathsf{N}(3)$ and $\mathsf{N}(6)$ are added on line 4. In `process`, both are chosen as representatives obtaining, $\mathtt{repr}_{5b}$.

Algorithm 2 guarantees that `repr` is maximally ground. Together with Theorem 2, this implies that all terms that can be rewritten into ground equivalent ones will be rewritten, which, in turn, means that for each variable that has a ground definition, its representative is one such definition.

Finding Additional (Non-ground) Definitions. At this point, QEL found ground definitions while avoiding cycles in $G_{\mathtt{repr}}$. However, this does not mean that as many variables as possible are eliminated. A variable can also be eliminated if it can be expressed as a function of other variables. This is not achieved by

```
$egraph$ :: refine_defs(repr, $\boldsymbol{v}$)
1:  for $n \in N$ do
2:    if $n =$ repr$(n)$ and $L(n) \in \boldsymbol{v}$ then
3:      $r := n$
4:      for $n' \in class(n) \setminus \{n\}$ do
5:        if $L(n') \notin \boldsymbol{v}$ then
6:          if not cycle$(n'$, repr$)$ then
7:            $r := n'$;
8:            break
9:      for $n' \in class(n)$ do
10:       repr$[n'] := r$
11: ret repr
```

```
$egraph$ :: find_core(repr, $\boldsymbol{v}$)
1: core $:= \emptyset$
2: for $n \in N$ s.t. $n =$ repr$(n)$ do
3:   core $:=$ core $\cup \{n\}$
4:   for $n' \in (class(n) \setminus n)$ do
5:     if $L(n') \in \boldsymbol{v}$ then continue
6:     else if $\exists m \in$ core $\cdot\, m$ congruent with $n'$
         then
7:       continue
8:     core $:=$ core $\cup \{n'\}$
9: ret core
```

Algorithm 3: Refining `repr` and building core.

`find_defs`. For example, in $\texttt{repr}_{5b}$ both variables are representatives, hence none is eliminated, even though, since $x \approx g(f(y))$, x could be eliminated in $f5$ by rewriting x as a function of y, allowing to eliminate x by rewriting it as a function of y, $g(f(y))$. Algorithm 3 shows function `refine_defs` that refines maximally ground `repr`*s* to further find such definitions while keeping admissibility and ground maximality. This is done by greedily attempting to change class representatives if they are labeled with a variable. `refine_defs` iterates over the nodes in the class checking if there is a different node that is not a variable and that does not create a cycle in $G_{\texttt{repr}}$ (line 6). The resulting `repr` remains maximally ground because representatives of ground classes are not changed.

For example, let us refine $\texttt{repr}_{5b} = \{\mathsf{N}(3), \mathsf{N}(6), \mathsf{N}(5)\}$ obtained for φ_5. Assume that x is processed first. For $class(\mathsf{N}(x))$, changing the representative to $\mathsf{N}(1)$ does not introduce a cycle (see Fig. 5c), so $\mathsf{N}(1)$ is selected. Next, for $class(\mathsf{N}(y))$, choosing $\mathsf{N}(4)$ causes $G_{\texttt{repr}}$ to be cyclic since $\mathsf{N}(1)$ was already chosen (Fig. 5a), so the representative of $class(\mathsf{N}(y))$ is not changed. The final refinement is $\texttt{repr}_{5c} = \{\mathsf{N}(1), \mathsf{N}(6), \mathsf{N}(5)\}$.

At this point, QEL found a representative function `repr` with as many ground definitions as possible and attempted to refine `repr` to have fewer variables as representatives. Next, QEL finds a core of the nodes of the egraph, based on `repr`, that will govern the translation of the egraph to a formula. While `repr` determines the semantic rewrites of terms that enable variable elimination, it is the use of the core in the translation that actually eliminates them.

Variable Elimination Based on a Core. A *core* of an egraph $G = \langle N, E, L, \texttt{root}\rangle$ and a representative function `repr`, is a subset of the nodes $N_c \subseteq N$ such that $\psi_c = G.\texttt{to\_formula}(\texttt{repr}, N \setminus N_c)$ satisfies $isFormula(G, \psi_c)$.

Algorithm 3 shows pseudocode for `find_core` that computes a core of an egraph for a given representative function. The idea is that non-representative nodes that are labeled by variables, as well as nodes congruent to nodes that are already in the core, need not be included in the core. The former are not needed since we are only interested in preserving the existential closure of the output, while the latter are not needed since congruent nodes introduce the same syntactic terms in the output. For example, for φ_1 and $\texttt{repr}_1$, `find_core` returns $\texttt{core}_1 = N_1 \setminus \{\mathsf{N}(3), \mathsf{N}(5), \mathsf{N}(9)\}$. Nodes $\mathsf{N}(3)$ and $\mathsf{N}(9)$ are excluded because they are labeled with variables; and node $\mathsf{N}(5)$ because it is congruent with $\mathsf{N}(4)$.

Finally, QEL produces a quantifier reduction by applying `to_formula` with the computed `repr` and `core`. Variables that are not in the core (they are not representatives) are eliminated – this includes variables that have a ground definition. However, QEL may eliminate a variable even if it is a representative (and thus it is in the core). As an example, consider $\psi(x, y) \triangleq f(x) \approx f(y) \wedge x \approx y$, whose egraph G contains 2 classes with 2 nodes each. The core N_c relative to any admissible `repr` contains only one representative per class: in the $\mathit{class}(\mathsf{N}(x))$ because both nodes are labeled with variables, and in the $\mathit{class}(\mathsf{N}(f(x)))$ because nodes are congruent. In this case, $\texttt{to\_formula}(\texttt{repr}, N_c)$ results in $\top$ (since singleton classes in the core produce no literals in the output formula), a quantifier elimination of ψ. More generally, the variables are eliminated because none of them is reachable in $G_{\texttt{repr}}$ from a non-singleton class in the core (only such classes contribute literals to the output).

We conclude the presentation of QEL by showing its output for our examples. For φ_1, QEL obtains $(k+1 \approx \mathit{read}(a, x) \wedge 3 > k+1)$, a quantifier reduction, using $\texttt{repr}_1 = \{\mathsf{N}(3), \mathsf{N}(8))\}$ and $\texttt{core}_1 = N_1 \setminus \{\mathsf{N}(3), \mathsf{N}(5), \mathsf{N}(9)\}$. For φ_4, QEL obtains $(6 \approx f(g(6)))$, a quantifier elimination, using $\texttt{repr}_{4b} = \{\mathsf{N}(4), \mathsf{N}(1)\}$, and $\texttt{core}_{4b} = N_4 \setminus \{\mathsf{N}(3), \mathsf{N}(2)\}$. Finally, for φ_5, QEL obtains $(y \approx h(f(y)) \wedge f(g(f(y))) \approx f(y))$, a quantifier reduction, using $\texttt{repr}_{5c} = \{\mathsf{N}(1), \mathsf{N}(6), \mathsf{N}(5)\}$ and $\texttt{core}_{5c} = N_5 \setminus \{\mathsf{N}(3)\}$.

Guarantees of QEL. Correctness of QEL is straightforward. We conclude this section by providing two conditions that ensure that a variable is eliminated by QEL. The first condition guarantees that a variable is eliminated whenever a ground definition for it exists (regardless of the specific representative function and core computed by QEL). This makes QEL *complete relative to quantifier elimination based on ground definitions.* Relative completeness is an important property since it means that QEL is unaffected by variable orderings and syntactic rewrites, unlike QeLite. The second condition, illustrated by ψ above, depends on the specific representative function and core computed by QEL.

Theorem 3. *Let φ be a QF conjunction of literals with free variables $\boldsymbol{v}$, and let $v \in \boldsymbol{v}$. Let $G = \mathit{egraph}(\varphi)$, n_v the node in G such that $L(n_v) = v$ and* `repr` *and* `core` *computed by QEL. We denote by $\mathit{NS} = \{n \in \texttt{core} \mid (\mathit{class}(n) \cap \texttt{core}) \neq \{n\}\}$ the set of nodes from classes with two or more nodes in* `core`*. If one of the following conditions hold, then v does not appear in QEL$(\varphi, \boldsymbol{v})$:*

(1) there exists a ground term t s.t. $\varphi \models v \approx t$, or
(2) n_v is not reachable from any node in NS in $G_{\texttt{repr}}$.

As a corollary, if every variable meets one of the two conditions, then QEL finds a quantifier elimination.

This concludes the presentation of our quantifier reduction algorithm. Next, we show how QEL can be used to under-approximate quantifier elimination, which allows working with formulas for which QEL does not result in a qelim.

$$\frac{\varphi[\mathit{read}(\mathit{write}(t,i,v),j)]}{\varphi[v] \wedge i \approx j}\ \textsc{ElimWrRd1}\quad M \models i \approx j$$

$$\frac{\varphi[\mathit{read}(\mathit{write}(t,i,v),j)]}{\varphi[\mathit{read}(t,j)] \wedge i \not\approx j}\ \textsc{ElimWrRd2}\quad M \models i \not\approx j$$

Fig. 6. Two MBP rules from [16]. The notation $\varphi[t]$ means that φ contains term t. The rules rewrite all occurrences of $\mathit{read}(\mathit{write}(t,i,v),j)$ with v and $\mathit{read}(t,j)$, respectively.

```
ElimWrRd
function match($t$)
  ret $t = read(write(s, i, v), j)$
function apply($t, M, G$)
  if $M \models i \approx j$ then
    $G.assert(i \approx j)$
    $G.assert(t \approx v)$
  else
    $G.assert(i \not\approx j)$
    $G.assert(t \approx read(s, j))$
```

Fig. 7. Adaptation of rules in Fig. 6 using QEL API.

5 Model Based Projection Using QEL

Applications like model checking and quantified satisfiability require efficient computation of under-approximations of quantifier elimination. They make use of model-based projection (MBP) algorithms to project variables that cannot be eliminated cheaply. Our QEL algorithm is efficient and relatively complete, but it does not guarantee to eliminate all variables. In this section, we use a model and theory-specific projection rules to implement an MBP algorithm on top of QEL.

We focus on two important theories: Arrays and Algebraic DataTypes (ADT). They are widely used to encode program verification tasks. Prior works separately develop MBP algorithms for Arrays [16] and ADTs [5]. Both MBPs were presented as a set of syntactic rewrite rules applied until fixed point.

Combining the MBP algorithms for Arrays and ADTs is non-trivial because applying projection rules for one theory may produce terms of the other theory. Therefore, separately achieving saturation in either theory is not sufficient to reach saturation in the combined setting. The MBP for the combined setting has to call both MBPs, check whether either one of them produced terms that can be processed by the other, and, if so, call the other algorithm. This is similar to theory combination in SMT solving where the core SMT solver has to keep track of different theory solvers and exchange terms between them.

Our main insight is that egraphs can be used as a glue to combine MBP algorithms for different theories, just like egraphs are used in SMT solvers to combine satisfiability checking for different theories. Implementing MBP using egraphs allows us to use the insights from QEL to combine MBP with on-the-fly quantifier reduction to produce less under-approximate formulas than what we get by syntactic application of MBP rules.

To implement MBP using egraphs, we implement all rewrite rules for MBP in Arrays [16] and ADTs [5] on top of egraphs. In the interest of space, we explain the implementation of just a couple of the MBP rules for Arrays[4].

Figure 6 shows two Array MBP rules from [16]: ElimWrRd1 and ElimWrRd2. Here, φ is a formula with arrays and M is a model for φ. Both rules rewrite terms which match the pattern $\mathit{read}(\mathit{write}(t, i, v), j)$, where t, i, j, k are all terms and t contains a variable to be projected. ElimWrRd1 is applicable when $M \models i \approx j$. It rewrites the term $\mathit{read}(\mathit{write}(t, i, v), j)$ to v. ElimWrRd2 is applicable when $M \not\models i \approx j$ and rewrites $\mathit{read}(\mathit{write}(t, i, v), j)$ to $\mathit{read}(t, j)$.

Figure 7 shows the egraph implementation of ElimWrRd1 and ElimWrRd2. The $\texttt{match}(t)$ method checks if t syntactically matches $\mathit{read}(\mathit{write}(s, i, v), j)$, where s contains a variable to be projected. The $\texttt{apply}(t)$ method assumes that t is $\mathit{read}(\mathit{write}(s, i, v), j)$. It first checks if $M \models i \approx j$, and, if so, it adds $i \approx j$ and $t \approx v$ to the egraph G. Otherwise, if $M \not\models i \approx j$, $\texttt{apply}(t)$ adds a disequality $i \not\approx j$ and an equality $t \approx \mathit{read}(s, v)$ to G. That is, the egraph implementation of the rules only adds (and does not remove) literals that capture the side condition and the conclusion of the rule.

Our algorithm for MBP based on egraphs, MBP-QEL, is shown in Alg. 4. It initializes an egraph with the input formula (line 1), applies MBP rules until saturation (line 4), and then uses the steps of QEL (lines 7–12) to generate the projected formula.

Applying rules is as straightforward as iterating over all terms t in the egraph, and for each rule r such that $r.\texttt{match}(t)$ is true, calling $r.\texttt{apply}(t, M, G)$ (lines 14–22). As opposed to the standard approach based on formula rewriting, here the terms are *not* rewritten – both remain. Therefore, it is possible to get into an infinite loop by re-applying the same rules on the same terms over and over again. To avoid this, MBP-QEL marks terms as *seen* (line 23) and avoids them in the next iteration (line 15). Some rules in MBP are applied to pairs of terms. For example, Ackermann rewrites pairs of *read* terms over the same variable. This is different from usual applications where rewrite rules are applied to individual expressions. Yet, it is easy to adapt such pairwise rewrite rules to egraphs by iterating over pairs of terms (lines 25–30).

MBP-QEL does not apply MBP rules to terms that contain variables but are already c-ground (line 16), which is sound because such terms are replaced by ground terms in the output (Theorem 3). This prevents unnecessary application of MBP rules thus allowing MBP-QEL to compute MBPs that are closer to a quantifier elimination (less model-specific).

Just like each application of a rewrite rule introduces a new term to a formula, each call to the `apply` method of a rule adds new terms to the egraph. Therefore, each call to *ApplyRules* (line 4) makes the egraph bigger. However, provided that the original MBP combination is terminating, the iterative application of *ApplyRules* terminates as well (due to marking).

Some MBP rules introduce new variables to the formula. MBP-QEL computes `repr` based on both original and newly introduced variables (line 7). This

[4] Implementation of all other rules is similar.

Input: A QF formula φ with free variables $\boldsymbol{v}$ all of sort $Array(I, V)$ or ADT, a model $M \models \varphi^{\exists}$, and sets of rules $ArrayRules$ and $ADTRules$
Output: A cube ψ s.t. $\psi^{\exists} \Rightarrow \varphi^{\exists}$, $M \models \psi^{\exists}$, and $vars(\psi)$ are not Arrays or ADTs

```
MBP-QEL($\varphi, \boldsymbol{v}, M$)
 1: $G := egraph(\varphi)$
 2: $p_1, p_2 := \top, \top$; $S, S_p := \emptyset, \emptyset$
 3: while $p_1 \lor p_2$ do
 4:   $p_1 := ApplyRules(G, M, ArrayRules, S, S_p)$
 5:   $p_2 := ApplyRules(G, M, ADTRules, S, S_p)$
 6: $\boldsymbol{v}' := G.Vars()$
 7: repr $:= G$.find_defs($\boldsymbol{v}'$)
 8: repr $:= G$.refine_defs(repr, $\boldsymbol{v}'$)
 9: core $:= G$.find_core(repr, $\boldsymbol{v}'$)
10: $\boldsymbol{v}_e := \{v \in \boldsymbol{v}' \mid is\_arr(v) \lor is\_adt(v)\}$
11: core$_e$ $:= \{n \in$ core $\mid gr(term(n), \boldsymbol{v}_e)\}$
12: ret $G$.to_formula(repr, $G.Nodes() \setminus$ core$_e$)
```

```
$ApplyRules(G, M, R, S, S_p)$
13: $progress := \bot$
14: $N := G.Nodes()$
15: $U := \{n \mid n \in N \setminus S\}$
16: $T := \{term(n) \mid n \in U \land$
      $(is\_eq(term(n)) \lor \neg c\text{-}ground(n))\}$
17: $R_p := \{r \in R \mid r.is\_for\_pairs()\}$
18: $R_u := R \setminus R_p$
19: for each $t \in T, r \in R_u$ do
20:   if $r$.match($t$) then
21:     $r$.apply($t, M, G$)
22:     $progress := \top$
23: $S := S \cup N$
24: $N_p := \{\langle n_1, n_2 \rangle \mid n_1, n_2 \in N\}$
25: $T_p := \{term(n_p) \mid n_p \in N_p \setminus S_p\}$
26: for each $t_p \in T_p, r \in R_p$ do
27:   if $r$.match($p$) then
28:     $r$.apply($p, M, G$)
29:     $progress := \top$
30: $S_p := S_p \cup N_p$
31: ret $progress$
```

Algorithm 4: MBP-QEL: an MBP using QEL. Here $gr(t, \boldsymbol{v})$ checks whether term t contains any variables in $\boldsymbol{v}$ and $is\_eq(t)$ checks if t is an equality literal.

allows MBP-QEL to eliminate all variables, including non-Array, non-ADT variables, that are equivalent to ground terms (Theorem 3).

As mentioned earlier, MBP-QEL never removes terms while rewrite rules are saturating. Therefore, after saturation, the egraph still contains all original terms and variables. From soundness of the MBP rules, it follows that after each invocation of `apply`, MBP-QEL creates an under-approximation of $\varphi^{\exists}$ based on the model M. From completeness of MBP rules, it follows that, after saturation, all terms containing Array or ADT variables can be removed from the egraph without affecting equivalence of the saturated egraph. Hence, when calling `to_formula`, MBP-QEL removes all terms containing Array or ADT variables (line 12). This includes, in particular, all the terms on which rewrite rules were applied, but potentially more.

We demonstrate our MBP algorithm on an example with nested ADTs and Arrays. Let $P \triangleq \langle A_{I \times I}, I \rangle$ be the datatype of a pair of an integer array and an integer, and let $pair : A_{I \times I} \times I \to P$ be its sole constructor with destructors $fst : P \to A_{I \times I}$ and $snd : P \to I$. In the following, let i, l, j be integers, a an integer array, p, p' pairs, and $\boldsymbol{p}_1, \boldsymbol{p}_2$ arrays of pairs ($A_{I \times P}$). Consider the formula:

$$\varphi_{mbp}(p, a) \triangleq read(a, i) \approx i \land p \approx pair(a, l) \land \boldsymbol{p}_2 \approx write(\boldsymbol{p}_1, j, p) \land p \not\approx p'$$

where p and a are free variables that we want to project and all of $i, j, l, \boldsymbol{p}_1, \boldsymbol{p}_2, p'$ are constants that we want to keep. MBP is guided by a model $M_{mbp} \models \varphi_{mbp}$. To eliminate p and a, MBP-QEL constructs the egraph of φ_{mbp} and applies the MBP rules. In particular, it uses Array MBP rules to rewrite the $\mathit{write}(\boldsymbol{p}_1, j, p)$ term by adding the equality $\mathit{read}(\boldsymbol{p}_2, j) \approx p$ and merging it with the equivalence class of $\boldsymbol{p}_2 \approx \mathit{write}(\boldsymbol{p}_1, j, p)$. It then applies ADT MBP rules to deconstruct the equality $p \approx \mathit{pair}(a, l)$ by creating two equalities $\mathit{fst}(p) \approx a$ and $\mathit{snd}(p) \approx l$. Finally, the call to `to_formula` produces

$$\begin{aligned}\mathit{read}(\mathit{fst}(\mathit{read}(\boldsymbol{p}_1, j)), i) \approx i \wedge \mathit{snd}(\mathit{read}(\boldsymbol{p}_1, j)) \approx l \wedge \\ \mathit{read}(\boldsymbol{p}_2, j) \approx \mathit{pair}(\mathit{fst}(\mathit{read}(\boldsymbol{p}_1, j)), l) \wedge \\ \boldsymbol{p}_2 \approx \mathit{write}(\boldsymbol{p}_1, j, \mathit{read}(\boldsymbol{p}_2, j)) \wedge \mathit{read}(\boldsymbol{p}_2, j) \not\approx p'\end{aligned}$$

The output is easy to understand by tracing it back to the input. For example, the first literal is a rewrite of the literal $\mathit{read}(a, i) \approx i$ where a is represented with $\mathit{fst}(p)$ and p is represented with $\mathit{read}(\boldsymbol{p}_1, j)$. While the interaction of these rules might seem straightforward in this example, the MBP implementation in Z3 fails to project a in this example because of the multilevel nesting.

Notably, in this example, the c-ground computation during projection allows MBP-QEL not splitting on the disequality $p \not\approx p'$ based on the model. While ADT MBP rules eliminate disequalities by using the model to split them, MBP-QEL benefits from the fact that, after the application of Array MBP rules, the class of p becomes ground, making $p \not\approx p'$ c-ground. Thus, the c-ground computation allows MBP-QEL to produce a formula that is less approximate than those produced by syntactic application of MBP rules. In fact, in this example, a quantifier elimination is obtained (the model M_{mbp} was not used).

In the next section, we show that our improvements to MBP translate to significant improvements in a CHC-solving procedure that relies on MBP.

6 Evaluation

We implement QEL (Alg. 1) and MBP-QEL (Alg. 4) inside Z3 [19] (version 4.12.0), a state-of-the-art SMT solver. Our implementation (referred to as Z3EG), is publicly available on GitHub[5]. Z3EG replaces QELite with QEL, and the existing MBP with MBP-QEL.

We evaluate Z3EG using two solving tasks. Our first evaluation is on the QSAT algorithm [5] for checking satisfiability of formulas with alternating quantifiers. In QSAT, Z3 uses both QELite and MBP to under-approximate quantified formulas. We compare three QSAT implementations: the existing version in Z3 with the default QELite and MBP; the existing version in Z3 in which QELite and MBP are replaced by our egraph-based algorithms, Z3EG; and the QSAT implementation in YicesQS[6], based on the Yices [8] SMT solver. During the evaluation, we found a bug in QSAT implementation of Z3 and fixed it[7].

[5] Available at https://github.com/igcontreras/z3/tree/qel-cav23.

[6] Available at https://github.com/disteph/yicesQS.

[7] Available at https://github.com/igcontreras/z3/commit/133c9e438ce.

Table 1. Instances solved within 20 min by different implementations. Benchmarks are quantified **LIA** and **LRA** formulas from SMT-LIB [2].

Cat.	Count	Z3EG		Z3		YicesQS	
		sat	unsat	sat	unsat	sat	unsat
LIA	416	150	266	150	266	107	102
LRA	2 419	795	1 589	793	1 595	808	1 610

Table 2. Instances solved within 60 s for our handcrafted benchmarks.

Cat.	Count	Z3EG		Z3	
		sat	unsat	sat	unsat
LIA-ADT	416	150	266	150	56
LRA-ADT	2 419	757	1 415	196	964

The fix resulted in Z3 solving over 40 sat instances and over 120 unsat instances more than before. In the following, we use the fixed version of Z3.

We use benchmarks in the theory of (quantified) LIA and LRA from SMT-LIB [2,3], with alternating quantifiers. LIA and LRA are the only tracks in which Z3 uses the QSAT tactic by default. To make our experiments more comprehensive, we also consider two modified variants of the LIA and LRA benchmarks, where we add some non-recursive ADT variables to the benchmarks. Specifically, we wrap all existentially quantified arithmetic variables using a record type ADT and unwrap them whenever they get used[8]. Since these benchmarks are similar to the original, we force Z3 to use the QSAT tactic on them with a `tactic.default_tactic=qsat` command line option.

Table 1 summarizes the results for the SMT-LIB benchmarks. In LIA, both Z3EG and Z3 solve all benchmarks in under a minute, while YicesQS is unable to solve many instances. In LRA, YicesQS solves all instances with very good performance. Z3 is able to solve only some benchmarks, and our Z3EG performs similarly to Z3. We found that in the LRA benchmarks, the new algorithms in Z3EG are not being used since there are not many equalities in the formula, and no equalities are inferred during the run of QSAT. Thus, any differences between Z3 and Z3EG are due to inherent randomness of the solving process.

Table 2 summarizes the results for the categories of mixed ADT and arithmetic. YicesQS is not able to compete because it does not support ADTs. As expected, Z3EG solves many more instances than Z3.

The second part of our evaluation shows the efficacy of MBP-QEL for Arrays and ADTs (Alg. 4) in the context of CHC-solving. Z3 uses both QeLite and MBP inside the CHC-solver Spacer [17]. Therefore, we compare Z3 and Z3EG on CHC problems containing Arrays and ADTs. We use two sets of benchmarks to test out the efficacy of our MBP. The benchmarks in the first set were generated for verification of Solidity smart contracts [1] (we exclude benchmarks with non-linear arithmetic, they are not supported by Spacer). These benchmarks have a very complex structure that nests ADTs and Arrays. Specifically, they contain both ADTs of Arrays, as well as Arrays of ADTs. This makes them suitable to test our MBP-QEL. Row 1 of Table 3 shows the number of instances

[8] The modified benchmarks are available at https://github.com/igcontreras/LIA-ADT and https://github.com/igcontreras/LRA-ADT.

Table 3. Instances solved within 20 min by Z3EG, Z3, and Eldarica. Benchmarks are CHCs from **Solidity** [1] and CHC competition [13]. The **abi** benchmarks are a subset of **Solidity** benchmarks.

Cat.	Count	Z3EG		Z3		Eldarica	
		SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
Solidity	3 468	2 324	1 133	2 314	1 114	**2 329**	**1 134**
↳ **abi**	127	19	108	19	88	19	108
LIA-lin-Arrays	488	**214**	72	212	**75**	147	68

solved by Z3 (Spacer) with and without MBP-QEL. Z3EG solves 29 instances more than Z3. Even though MBP is just one part of the overall Spacer algorithm, we see that for these benchmarks, MBP-QEL makes a significant impact on Spacer. Digging deeper, we find that many of these instances come from the category called *abi* (row 2 in Table 3). Z3EG solves all of these benchmarks, while Z3 fails to solve 20 of them. We traced the problem down to the MBP implementation in Z3: it fails to eliminate all variables, causing runtime exception. In contrast, MBP-QEL eliminates all variables successfully, allowing Z3EG to solve these benchmarks.

We also compare Z3EG with Eldarica [14], a state-of-the-art CHC-solver that is particularly effective on these benchmarks. Z3EG solves almost as many instances as Eldarica. Furthermore, like Z3, Z3EG is orders of magnitude faster than Eldarica. Finally, we compare the performance of Z3EG on Array benchmarks from the CHC competition [13]. Z3EG is competitive with Z3, solving 2 additional safe instances and almost as many unsafe instances as Z3 (row 3 of Table 3). Both Z3EG and Z3 solve quite a few instances more than Eldarica.

Our experiments show the effectiveness of our QEL and MBP-QEL in different settings inside the state-of-the-art SMT solver Z3. While we maintain performance on quantified arithmetic benchmarks, we improve Z3's QSAT algorithm on quantified benchmarks with ADTs. On verification tasks, QEL and MBP-QEL help Spacer solve 30 new instances, even though MBP is only a relatively small part of the overall Spacer algorithm.

7 Conclusion

Quantifier elimination, and its under-approximation, Model-Based Projection are used by many SMT-based decision procedures, including quantified SAT and Constrained Horn Clause solving. Traditionally, these are implemented by a series of syntactic rules, operating directly on the syntax of an input formula. In this paper, we argue that these procedures should be implemented directly on the egraph data-structure, already used by most SMT solvers. This results in algorithms that better handle implicit equality reasoning and result in easier to implement and faster procedures. We justify this argument by implementing quantifier reduction and MBP in Z3 using egraphs and show that the new implementation translates into significant improvements to the target decision procedures. Thus, our work provides both theoretical foundations for quantifier reduction and practical contributions to Z3 SMT-solver.

Acknowledgment. The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the Israeli Science Foundation (ISF) grant No. 1810/18. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), MathWorks Inc., and the Microsoft Research PhD Fellowship.

References

1. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity Compiler's Model Checker. 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. 325–338. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_16
2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
4. Barrett, Clark, Tinelli, Cesare: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
5. Bjørner, N.S., Janota, M.: Playing with quantified satisfaction. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24–28, 2015. EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015). https://doi.org/10.29007/vv21
6. Chang, B.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: Cousot, R. (ed.) Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17–19, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3385, pp. 147–163. Springer (2005). https://doi.org/10.1007/978-3-540-30579-8_11
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (may 2005). https://doi.org/10.1145/1066100.1066102
8. Dutertre, B.: Yices 2.2. 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. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
9. Dutertre, B.: Solving Exists/Forall Problems with Yices. In: Workshop on Satisfiability Modulo Theories (2015). https://yices.csl.sri.com/papers/smt2015.pdf

10. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: An abstract domain of uninterpreted functions. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17–19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 85–103. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_4
11. Gascón, A., Subramanyan, P., Dutertre, B., Tiwari, A., Jovanovic, D., Malik, S.: Template-based circuit understanding. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21–24, 2014, pp. 83–90. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987599
12. Gulwani, S., Tiwari, A., Necula, G.C.: Join algorithms for the theory of uninterpreted functions. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16–18, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3328, pp. 311–323. Springer (2004). https://doi.org/10.1007/978-3-540-30538-5_26
13. Gurfinkel, A., Ruemmer, P., Fedyukovich, G., Champion, A.: CHC-COMP. https://chc-comp.github.io/ (2018)
14. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: Bjørner, N.S., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, pp. 1–7. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603013
15. Joshi, R., Nelson, G., Randall, K.H.: Denali: A goal-directed superoptimizer. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17–19, 2002, pp. 304–314. ACM (2002). https://doi.org/10.1145/512529.512566
16. Komuravelli, A., Bjørner, N.S., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27–30, 2015, pp. 89–96. IEEE (2015). https://doi.org/10.5555/2893529.2893548
17. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive 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. 17–34. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_2
18. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5–10, 2010, pp. 316–329. ACM (2010). https://doi.org/10.1145/1806596.1806632
19. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24

20. Nelson, G., Oppen, D.C.: Fast decision algorithms based on union and find. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pp. 114–119. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.12
21. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979). https://doi.org/10.1145/357073.357079
22. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 264–276. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480915
23. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. Log. Methods Comput. Sci. **7**(1) (2011). https://doi.org/10.2168/LMCS-7(1:10)2011
24. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434304

Local Search for Solving Satisfiability of Polynomial Formulas

Haokun Li, Bican Xia, and Tianqi Zhao(✉)

School of Mathematical Sciences, Peking University, Beijing, China
{haokunli,zhaotq}@pku.edu.cn, xbc@math.pku.edu.cn

Abstract. Satisfiability Modulo the Theory of Nonlinear Real Arithmetic, SMT(NRA) for short, concerns the satisfiability of *polynomial formulas*, which are quantifier-free Boolean combinations of polynomial equations and inequalities with integer coefficients and real variables. In this paper, we propose a local search algorithm for a special subclass of SMT(NRA), where all constraints are strict inequalities. An important fact is that, given a polynomial formula with n variables, the zero level set of the polynomials in the formula decomposes the n-dimensional real space into finitely many components (cells) and every polynomial has constant sign in each cell. The key point of our algorithm is a new operation based on real root isolation, called *cell-jump*, which updates the current assignment along a given direction such that the assignment can 'jump' from one cell to another. One cell-jump may adjust the values of several variables while traditional local search operations, such as *flip* for SAT and *critical move* for SMT(LIA), only change that of one variable. We also design a two-level operation selection to balance the success rate and efficiency. Furthermore, our algorithm can be easily generalized to a wider subclass of SMT(NRA) where polynomial equations linear with respect to some variable are allowed. Experiments show the algorithm is competitive with state-of-the-art SMT solvers, and performs particularly well on those formulas with high-degree polynomials.

Keywords: SMT · Local search · Nonlinear real arithmetic · Cell-jump · Cylindrical Algebraic Decomposition (CAD)

1 Introduction

Satisfiability modulo theories (SMT) refers to the problem of determining whether a first-order formula is satisfiable with respect to (w.r.t.) certain theories, such as the theories of linear integer/real arithmetic, nonlinear integer/real arithmetic and strings. In this paper, we consider the theory of nonlinear real arithmetic (NRA) and restrict our attention to the problem of solving satisfiability of quantifier-free polynomial formulas.

Solving polynomial constraints has been a central problem in the development of mathematics. In 1951, Tarski's decision procedure [33] made it possible to solve polynomial constraints in an algorithmic way. However, Tarski's

The authors are listed in alphabetical order and they make equal contribution.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 87–109, 2023.
https://doi.org/10.1007/978-3-031-37703-7_5

algorithm is impractical because of its super-exponential complexity. The first relatively practical method is cylindrical algebraic decomposition (CAD) algorithm [13] proposed by Collins in 1975, followed by lots of improvements. See for example [6,14,20,22,26]. Unfortunately, those variants do not improve the complexity of the original algorithm, which is doubly-exponential. On the other hand, SMT(NRA) is important in theorem proving and program verification, since most complicated programs use real variables and perform nonlinear arithmetic operation on them. Particularly, SMT(NRA) has various applications in the formal analysis of hybrid systems, dynamical systems and probabilistic systems (see the book [12] for reference).

The most popular approach for solving SMT(NRA) is the lazy approach, also known as CDCL(T) [5]. It combines a propositional satisfiability (SAT) solver that uses a conflict-driven clause learning (CDCL) style algorithm to find assignments of the propositional abstraction of a polynomial formula and a theory solver that checks the consistency of sets of polynomial constraints. The solving effort in the approach is devoted to both the Boolean layer and the theory layer. For the theory solver, the only complete method is the CAD method, and there also exist many efficient but incomplete methods, such as linearisation [10], interval constraint propagation [34] and virtual substitution [35]. Recall that the complexity of the CAD method is doubly-exponential. In order to ease the burden of using CAD, an improved CDCL-style search framework, the model constructing satisfiability calculus (MCSAT) framework [15,21], was proposed. Further, there are many optimizations on CAD projection operation, *e.g.* [7,24,29], custom-made for this framework. Besides, an alternative algorithm for determining the satisfiability of conjunctions of non-linear polynomial constraints over the reals based on CAD is presented in [1].

The development of this approach brings us effective SMT(NRA) solvers. Almost all state-of-the-art SMT(NRA) solvers are based on the lazy approach, including Z3 [28], CVC5 [3], Yices2 [16] and MathSAT5 [11]. These solvers have made great progress in solving SMT(NRA). However, the time and memory usage of them on some hard instances may be unacceptable, particularly when the proportion of nonlinear polynomials in all polynomials appearing in the formula is high. It pushes us to design algorithms which perform well on these hard instances.

Local search plays an important role in solving satisfiability problems, which is an incomplete method since it can only determine satisfiability but not unsatisfiability. A local search algorithm moves in the space of candidate assignments (the search space) by applying local changes, until a satisfied assignment is found or a time bound is reached. It is well known that local search method has been successfully applied to SAT problems [2,4,9,23]. In recent years, some efforts trying to develop local search method for SMT solving are inspiring: Under the DPLL(T) framework, Griggio et al. [19] introduced a general procedure for integrating a local search solver of the WalkSAT family with a theory solver. Pure local search algorithms [17,30,31] were proposed to solve SMT problems with respect to the theory of bit-vectors directly on the theory level. Cai et al.

[8] developed a local search procedure for SMT on the theory of linear integer arithmetic (LIA) through the *critical move* operation, which works on the literal-level and changes the value of one variable in a false LIA literal to make it true. We also notice that there exists a local search SMT solver for the theory of NRA, called NRA-LS, performing well at the SMT Competition 2022[1]. A simple description of the solver without details about local search can be found in [25].

In this paper, we propose a local search algorithm for a special subclass of SMT(NRA), where all constraints are strict inequalities. The idea of applying the local search method to SMT(NRA) comes from CAD, which is a decomposition of the search space $\mathbb{R}^n$ into finitely many cells such that every polynomial in the formula is sign-invariant on each cell. CAD guarantees that the search space only has finitely many states. Similar to the local search method for SAT which moves between finitely many Boolean assignments, local search for SMT(NRA) should jump between finitely many cells. So, we may use a local search framework for SAT to solve SMT(NRA).

Local search algorithms require an operation to perform local changes. For SAT, a standard operation is *flip*, which modifies the current assignment by flipping the value of one Boolean variable from false to true or vice-versa. For SMT(NRA), we propose a novel operation, called *cell-jump*, updating the current assignment $x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$ ($a_i \in \mathbb{Q}$) to a solution of a false polynomial constraint '$p < 0$' or '$p > 0$', where x_i is a variable appearing in the given polynomial formula. Different from the critical move operation for linear integer constraints [8], it is difficult to determine the threshold value of some variable x_i such that the false polynomial constraint becomes true. We deal with the issue by the method of real root isolation, which isolates every real root of the univariate polynomial $p(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$ in an open interval sufficiently small with rational endpoints. If there exists at least one endpoint making the false constraint true, a cell-jump operation assigns x_i to one closest to a_i. The procedure can be viewed as searching for a solution along a line parallel to the x_i-axis. In fact, a cell-jump operation can search along any fixed straight line, and thus one cell-jump may change the values of more than one variables. Each step, the local search algorithm picks a cell-jump operation to execute according to a two-level operation selection and updates the current assignment, until a solution to the polynomial formula is found or the terminal condition is satisfied. Moreover, our algorithm can be generalized to deal with a wider subclass of SMT(NRA) where polynomial equations linear w.r.t. some variable are allowed.

The local search algorithm is implemented with `Maple2022` as a tool. Experiments are conducted to evaluate the tool on two classes of benchmarks, including selected instances from SMT-LIB[2], and some hard instances generated randomly with only nonlinear constraints. Experimental results show that our tool is competitive with state-of-the-art SMT solvers on the SMT-LIB benchmarks, and performs particularly well on the hard instances. We also combine our tool with

[1] https://smt-comp.github.io/2022.

[2] https://smtlib.cs.uiowa.edu/benchmarks.shtml.

Z3, CVC5, Yices2 and MathSAT5 respectively to obtain four sequential portfolio solvers, which show better performance.

The rest of the paper is organized as follows. The next section introduces some basic definitions and notation and a general local search framework for solving a satisfiability problem. Section 3 shows from the CAD perspective, the search space for SMT(NRA) only has finite states. In Sect. 4, we describe cell-jump operations, while in Sect. 5 we provide the scoring function which gives every operation a score. The main algorithm is presented in Sect. 6. And in Sect. 7, experimental results are provided to indicate the efficiency of the algorithm. Finally, the paper is concluded in Sect. 8.

2 Preliminaries

2.1 Notation

Let $\bar{\boldsymbol{x}} := (x_1, \ldots, x_n)$ be a vector of variables. Denote by $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{Z}$ the set of rational numbers, real numbers and integer numbers, respectively. Let $\mathbb{Q}[\bar{\boldsymbol{x}}]$ and $\mathbb{R}[\bar{\boldsymbol{x}}]$ be the ring of polynomials in the variables $x_1, \ldots, x_n$ with coefficients in $\mathbb{Q}$ and in $\mathbb{R}$, respectively.

Definition 1 (Polynomial Formula). *Suppose $\Lambda = \{P_1, \ldots, P_m\}$ where every P_i is a non-empty finite subset of $\mathbb{Q}[\bar{\boldsymbol{x}}]$. The following formula*

$$F = \bigwedge_{P_i \in \Lambda} \bigvee_{p_{ij} \in P_i} p_{ij}(x_1, \ldots, x_n) \rhd_{ij} 0, \text{ where } \rhd_{ij} \in \{<, >, =\},$$

is called a polynomial formula*. Additionally, we call $p_{ij}(x_1, \ldots, x_n) \rhd_{ij} 0$ an* atomic polynomial formula*, and $\bigvee_{p_{ij} \in P_j} p_{ij}(x_1, \ldots, x_n) \rhd_{ij} 0$ a* polynomial clause*.*

For any polynomial formula F, $\texttt{poly}(F)$ denotes the set of polynomials appearing in F. For any atomic formula ℓ, $\texttt{poly}(\ell)$ denotes the polynomial appearing in ℓ and $\texttt{rela}(\ell)$ denotes the relational operator ('<', '>' or '=') of ℓ.

For any polynomial formula F, an *assignment* is a mapping $\alpha : \bar{\boldsymbol{x}} \to \mathbb{R}^n$ such that $\alpha(\bar{\boldsymbol{x}}) = (a_1, \ldots, a_n)$ where $a_i \in \mathbb{R}$. Given an assignment α,

- an atomic polynomial formula is *true under* α if it evaluates to true under α, and otherwise it is *false under* α,
- a polynomial clause is *satisfied under* α if at least one atomic formula in the clause is true under α, and *falsified under* α otherwise.

When the context is clear, we simply say a *true* (or *false*) atomic polynomial formula and a *satisfied* (or *falsified*) polynomial clause. A polynomial formula is *satisfiable* if there exists an assignment α such that all clauses in the formula are satisfied under α, and such an assignment is a *solution* to the polynomial formula. A polynomial formula is *unsatisfiable* if any assignment is not a solution.

2.2 A General Local Search Framework

When applying local search algorithms to solve a satisfiability problem, the search space is the set of all assignments. A general local search framework begins with a complete, initial assignment. Every time, one of the operations with the highest score is picked and the assignment is updated after executing the operation until reaching the set terminal condition. Below, we give the formal definitions of *operation* and *scoring function*.

Definition 2 (Operation). *Let F be a formula. Given an assignment α which is not a solution of F, an* operation *modifies α to another assignment α'.*

Definition 3 (Scoring Function). *Let F be a formula. Suppose α is the current assignment and op is an operation. A* scoring function *is defined as* $\texttt{score}(op, \alpha) := \texttt{cost}(\alpha) - \texttt{cost}(\alpha')$*, where the real-valued function* `cost` *measures the cost of making F satisfied under an assignment according to some heuristic, and α' is the assignment after executing op.*

Example 1. In local search algorithms for SAT, a standard operation is *flip*, which modifies the current assignment by flipping the value of one Boolean variable from false to true or vice-versa. A commonly used scoring function measures the change on the number of falsified clauses by flipping a variable. Thus, operation *op* is $\texttt{flip}(b)$ for some Boolean variable b, and $\texttt{cost}(\alpha)$ is interpreted as the number of falsified clauses under the assignment α.

Actually, only when $\texttt{score}(op, \alpha)$ is a positive number does it make sense to execute operation *op*, since the operation guides the current assignment to an assignment with less cost of being a solution.

Definition 4 (Decreasing Operation). *Suppose α is the current assignment. Given a scoring function* `score`*, an operation op is a* decreasing operation *under α if* $\texttt{score}(op, \alpha) > 0$.

A general local search framework is described in Algorithm 1. The framework was used in GSAT [27] for solving SAT problems. Note that if the input formula F is satisfied, Algorithm 1 outputs either (i) a solution of F if the solution is found successfully, or (ii) "unknown" if the algorithm fails.

Algorithm 1. General Local Search Framework

```
Input  : a formula $F$ and a terminal condition $\varphi$
Output: a solution to $F$ or unknown
initialize assignment $\alpha$
while the terminal condition $\varphi$ is not satisfied do
    if $\alpha$ satisfies $F$ then
        return $\alpha$
    else
        $op \leftarrow$ one of the decreasing operations with the highest score
        perform $op$ to modify $\alpha$
return unknown
```

3 The Search Space of SMT(NRA)

The search space for SAT problems consists of finitely many assignments. So, theoretically speaking, a local search algorithm can eventually find a solution, as long as the formula indeed has a solution and there is no cycling during the search. It seems intuitive, however, that the search space of an SMT(NRA) problem, *e.g.* $\mathbb{R}^n$, is infinite and thus search algorithms may not work.

Fortunately, due to Tarski's work and the theory of CAD, SMT(NRA) is decidable. Given a polynomial formula in n variables, by the theory of CAD, $\mathbb{R}^n$ is decomposed into finitely many cells such that every polynomial in the formula is sign-invariant on each cell. Therefore, the search space of the problem is essentially finite. The cells of SMT(NRA) are very similar to the Boolean assignments of SAT, so just like traversing all Boolean assignments in SAT, there exists a basic strategy to traverse all cells.

In this section, we describe the search space of SMT(NRA) based on the CAD theory from a local search perspective, providing a theoretical foundation for the operators and heuristics we will propose in the next sections.

Example 2. Consider the polynomial formula

$$F \;=\; (f_1 > 0 \vee f_2 > 0) \wedge (f_1 < 0 \vee f_2 < 0),$$

where $f_1 = 17x^2+2xy+17y^2+48x-48y$ and $f_2 = 17x^2-2xy+17y^2-48x-48y$.

The solution set of F is shown as the shaded area in Fig. 1. Notice that $\texttt{poly}(F)$ consists of two polynomials and decomposes $\mathbb{R}^2$ into 10 areas: $C_1, \ldots, C_{10}$ (see Fig. 2). We refer to these areas as *cells*.

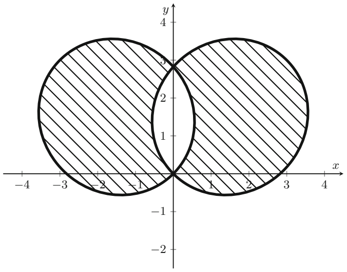


Fig. 1. The solution set of F in Example 2.

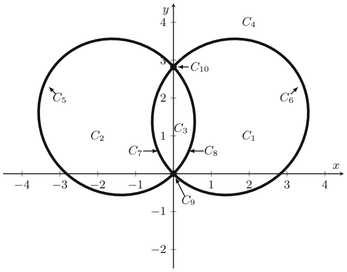


Fig. 2. The zero level set of $\texttt{poly}(F)$ decomposes $\mathbb{R}^2$ into 10 cells.

Definition 5 (Cell). *For any finite set $Q \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$, a* cell *of Q is a maximally connected set in $\mathbb{R}^n$ on which the sign of every polynomial in Q is constant. For any point $\bar{a} \in \mathbb{R}^n$, we denote by $\texttt{cell}(Q, \bar{a})$ the cell of Q containing $\bar{a}$.*

By the theory of CAD, we have

Corollary 1. *For any finite set $Q \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$, the number of cells of Q is finite.*

It is obvious that any two cells of Q are disjoint and the union of all cells of Q equals $\mathbb{R}^n$. Definition 5 shows that for a polynomial formula F with $\texttt{poly}(F) = Q$, the satisfiability of F is constant on every cell of Q, that is, either all the points in a cell are solutions to F or none of them are solutions to F.

Example 3. Consider the polynomial formula F in Example 2. As shown in Fig. 3, assume that we start from point a to search for a solution to F. Jumping from a to b makes no difference, as both points are in the same cell and thus neither are solutions to F. However, jumping from a to c or from a to d crosses different cells and we may discover a cell satisfying F. Herein, the cell containing d satisfies F.

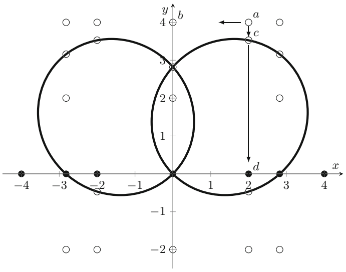


Fig. 3. Jumping from point a to search for a solution of F.

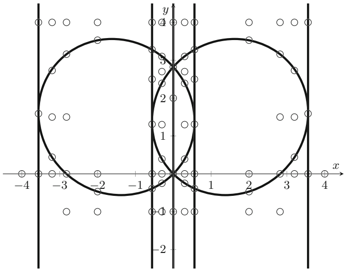


Fig. 4. A cylindrical expansion of a cylindrically complete set containing $\texttt{poly}(F)$.

For the remainder of this section, we will demonstrate how to traverse all cells through point jumps between cells. The method of traversing cell by cell in a variable by variable direction will be explained step by step from Definition 6 to Definition 8.

Definition 6 (Expansion). *Let $Q \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$ be finite and $\bar{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Given a variable x_i $(1 \leq i \leq n)$, let $r_1 < \cdots < r_s$ be all real roots of $\{q(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n) \mid q(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n) \not\equiv 0,\ q \in Q\}$, where $s \in \mathbb{Z}_{\geq 0}$. An* expansion *of $\bar{a}$ to x_i on Q is a point set $\varLambda \subseteq \mathbb{R}^n$ satisfying*

(a) $\bar{a} \in \varLambda$ and $(a_1, \ldots, a_{i-1}, r_j, a_{i+1}, \ldots, a_n) \in \varLambda$ for $1 \leq j \leq s$,
(b) for any $\bar{b} = (b_1, ..., b_n) \in \varLambda$, $b_j = a_j$ for $j \in \{1, \ldots, n\} \setminus \{i\}$, and
(c) for any interval $I \in \{(-\infty, r_1), (r_1, r_2), \ldots, (r_{s-1}, r_s), (r_s, +\infty)\}$, there exists a unique $\bar{b} = (b_1, ..., b_n) \in \varLambda$ such that $b_i \in I$.

For any point set $\{\bar{a}^{(1)}, \ldots, \bar{a}^{(m)}\} \subseteq \mathbb{R}^n$, an expansion *of the set to x_i on Q is $\bigcup_{j=1}^{m} \varLambda_j$, where $\varLambda_j$ is an expansion of $\bar{a}^{(j)}$ to x_i on Q.*

Example 4. Consider the polynomial formula F in Example 2. The set of black solid points in Fig. 3, denoted as $\varLambda$, is an expansion of point $(0, 0)$ to x on $\texttt{poly}(F)$. The set of all points (including black solid points and hollow points) is an expansion of $\varLambda$ to y on $\texttt{poly}(F)$.

As shown in Fig. 3, an expansion of a point to some variable is actually a result of the point continuously jumping to adjacent cells along that variable direction. Next, we describe the expansion of all variables in order, which is a result of jumping from cell to cell along the directions of variables w.r.t. a variable order.

Definition 7 (Cylindrical Expansion). *Let $Q \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$ be finite and $\bar{a} \in \mathbb{R}^n$. Given a variable order $x_1 \prec \cdots \prec x_n$, a* cylindrical expansion *of $\bar{a}$ w.r.t. the variable order on Q is $\bigcup_{i=1}^{n} \Lambda_i$, where Λ_1 is an expansion of $\bar{a}$ to x_1 on Q, and for $2 \leq i \leq n$, Λ_i is an expansion of Λ_{i-1} to x_i on Q. When the context is clear, we simply call $\bigcup_{i=1}^{n} \Lambda_i$ a* cylindrical expansion *of Q.*

Example 5. Consider the formula F in Example 2. It is clear that the set of all points in Fig. 3 is a cylindrical expansion of point $(0, 0)$ w.r.t. $x \prec y$ on $\texttt{poly}(F)$. The expansion actually describes the following jumping process. First, the origin $(0, 0)$ jumps along the x-axis to the black points, and then those black points jump along the y-axis direction to the white points.

Clearly, a cylindrical expansion is similar to how a Boolean vector is flipped variable by variable. Note that the points in the expansion in Fig. 3 do not cover all the cells (*e.g.* C_7 and C_8 in Fig. 2), but if we start from $(0, 2)$, all the cells can be covered. This implies that whether all the cells can be covered depends on the starting point.

Definition 8 (Cylindrically Complete). *Let $Q \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$ be finite. Given a variable order $x_1 \prec \cdots \prec x_n$, Q is said to be* cylindrically complete *w.r.t. the variable order, if for any $\bar{a} \in \mathbb{R}^n$ and for any cylindrical expansion Λ of $\bar{a}$ w.r.t. the order on Q, every cell of Q contains at least one point in Λ.*

Theorem 1. *For any finite set $Q \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$ and any variable order, there exists Q' such that $Q \subseteq Q' \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$ and Q' is cylindrically complete w.r.t. the variable order.*

Proof. Let Q' be the projection set of Q [6,13,26] obtained from the CAD projection operator w.r.t. the variable order. According to the theory of CAD, Q' is cylindrically complete. □

Corollary 2. *For any polynomial formula F and any variable order, there exists a finite set $Q \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$ such that for any cylindrical expansion Λ of Q, every cell of $\texttt{poly}(F)$ contains at least one point in Λ. Furthermore, F is satisfiable if and only if F has solutions in Λ.*

Example 6. Consider the polynomial formula F in Example 2. By the proof of Theorem 1, $Q' := \{x, -2 - 3x + x^2,\ -2 + 3x + x^2,\ 10944 + 17x^2,\ f_1,\ f_2\}$ is a cylindrically complete set w.r.t. $x \prec y$ containing $\texttt{poly}(F)$. As shown in Fig. 4, the set of all (hollow) points is a cylindrical expansion of point $(0, 0)$ w.r.t. $x \prec y$ on Q', which covers all cells of $\texttt{poly}(F)$.

Corollary 2 shows that for a polynomial formula F, there exists a finite set $Q \subseteq \mathbb{R}[\bar{\boldsymbol{x}}]$ such that we can traverse all the cells of $\texttt{poly}(F)$ through a search path containing all points in a cylindrical expansion of Q. The cost of traversing the cells is very high, and in the worst case, the number of cells will grow exponentially with the number of variables.

The key to building a local search on SMT(NRA) is to construct a heuristic search based on the operation of jumping between cells.

4 The Cell-Jump Operation

In this section, we propose a novel operation, called *cell-jump*, that performs local changes in our algorithm. The operation is determined by the means of real root isolation. We review the method of real root isolation and define *sample points* in Sect. 4.1. Section 4.2 and Sect. 4.3 present a cell-jump operation along a line parallel to a coordinate axis and along any fixed straight line, respectively.

4.1 Sample Points

Real root isolation is a symbolic way to compute the real roots of a polynomial, which is of fundamental importance in computational real algebraic geometry (*e.g.*, it is a routing sub-algorithm for CAD). There are many efficient algorithms and popular tools in computer algebra systems such as Maple and Mathematica to isolate the real roots of polynomials.

We first introduce the definition of *sequences of isolating intervals* for nonzero univariate polynomials, which can be obtained by any real root isolation tool, *e.g.* CLPoly[3].

Definition 9 (Sequence of Isolating Intervals). *For any nonzero univariate polynomial $p(x) \in \mathbb{Q}[x]$, a* sequence of isolating intervals *of $p(x)$ is a sequence of open intervals $(a_1, b_1), \ldots, (a_s, b_s)$ where $s \in \mathbb{Z}_{\geq 0}$, such that*

(i) for each i $(1 \leq i \leq s)$, $a_i, b_i \in \mathbb{Q}$, $a_i < b_i$ and $b_i < a_{i+1}$,
(ii) each interval (a_i, b_i) $(1 \leq i \leq s)$ has exactly one real root of $p(x)$, and
(iii) none of the real roots of $p(x)$ are in $\mathbb{R} \setminus \bigcup_{i=1}^{s}(a_i, b_i)$.

Specially, the sequence of isolating intervals is empty, i.e., $s = 0$, when $p(x)$ has no real roots.

By means of sequences of isolating intervals, we define *sample points* of univariate polynomials, which is the key concept of the *cell-jump* operation proposed in Sect. 4.2 and Sect. 4.3.

Definition 10 (Sample Point). *For any nonzero univariate polynomial $p(x) \in \mathbb{Q}[x]$, let $(a_1, b_1), \ldots, (a_s, b_s)$ be a sequence of isolating intervals of $p(x)$ where $s \in \mathbb{Z}_{\geq 0}$. Every point in the set $\{a_1, b_s\} \cup \bigcup_{i=1}^{s-1}\{b_i, \frac{b_i + a_{i+1}}{2}, a_{i+1}\}$ is a* sample point *of $p(x)$. If x^* is a sample point of $p(x)$ and $p(x^*) > 0$ (or $p(x^*) < 0$), then x^* is a* positive sample point *(or* negative sample point*) of $p(x)$. For the zero polynomial, it has no* sample point*, no* positive sample point *and no* negative sample point.

Remark 1. For any nonzero univariate polynomial $p(x)$ that has real roots, let $r_1, \ldots, r_s$ $(s \in \mathbb{Z}_{\geq 1})$ be all distinct real roots of $p(x)$. It is obvious that the sign of $p(x)$ is positive constantly or negative constantly on each interval I of the set $\{(-\infty, r_1), (r_1, r_2), \ldots, (r_{s-1}, r_s), (r_s, +\infty)\}$. So, we only need to take a point x^* from the interval I, and then the sign of $p(x^*)$ is the constant sign of

[3] https://github.com/lihaokun/CLPoly.

$p(x)$ on I. Specially, we take a_1 as the sample point for the interval $(-\infty, r_1)$, b_i, $\frac{b_i+a_{i+1}}{2}$ or a_{i+1} as a sample point for (r_i, r_{i+1}) where $1 \leq i \leq s-1$, and b_s as the sample point for $(r_s, +\infty)$. By Definition 10, there exists no sample point for the zero polynomial and a univariate polynomial with no real roots.

Example 7. Consider the polynomial $p(x) = x^8 - 4x^6 + 6x^4 - 4x^2 + 1$. It has two real roots -1 and 1, and a sequence of isolating intervals of it is $(-\frac{215}{128}, -\frac{19}{32})$, $(\frac{19}{32}, \frac{215}{128})$. Every point in the set $\{-\frac{215}{128}, -\frac{19}{32}, 0, \frac{19}{32}, \frac{215}{128}\}$ is a sample point of $p(x)$. Note that $p(x) > 0$ holds on the intervals $(-\infty, -1)$ and $(1, +\infty)$, and $p(x) < 0$ holds on the interval $(-1, 1)$. Thus, $-\frac{215}{128}$ and $\frac{215}{128}$ are positive sample points of $p(x)$; $-\frac{19}{32}$, 0 and $\frac{19}{32}$ are negative sample points of $p(x)$.

4.2 Cell-Jump Along a Line Parallel to a Coordinate Axis

The *critical move* operation [8, Definition 2] is a literal-level operation. For any false LIA literal, the operation changes the value of one variable in it to make the literal true. In the subsection, we propose a similar operation which adjusts the value of one variable in a false atomic polynomial formula with '<' or '>'.

Definition 11. *Suppose the current assignment is $\alpha : x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$ where $a_i \in \mathbb{Q}$. Let ℓ be a false atomic polynomial formula under α with a relational operator '<' or '>'.*

(i) Suppose ℓ is $p(\bar{\boldsymbol{x}}) < 0$. For each variable x_i such that the univariate polynomial $p(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$ has negative sample points, there exists a cell-jump *operation, denoted as* $\texttt{cjump}(x_i, \ell)$*, assigning x_i to a negative sample point closest to a_i.*

(ii) Suppose ℓ is $p(\bar{\boldsymbol{x}}) > 0$. For each variable x_i such that the univariate polynomial $p(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$ has positive sample points, there exists a cell-jump *operation, denoted as* $\texttt{cjump}(x_i, \ell)$*, assigning x_i to a positive sample point closest to a_i.*

Every assignment in the search space can be viewed as a point in $\mathbb{R}^n$. Then, performing a $\texttt{cjump}(x_i, \ell)$ operation is equivalent to moving one step from the current point $\alpha(\bar{\boldsymbol{x}})$ along the line $(a_1, \ldots, a_{i-1}, \mathbb{R}, a_{i+1}, \ldots, a_n)$. Since the line is parallel to the x_i-axis, we call $\texttt{cjump}(x_i, \ell)$ a *cell-jump along a line parallel to a coordinate axis*.

Theorem 2. *Suppose the current assignment is $\alpha : x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$ where $a_i \in \mathbb{Q}$. Let ℓ be a false atomic polynomial formula under α with a relational operator '<' or '>'. For every i $(1 \leq i \leq n)$, there exists a solution of ℓ in $\{\alpha' \mid \alpha'(\bar{\boldsymbol{x}}) \in (a_1, \ldots, a_{i-1}, \mathbb{R}, a_{i+1}, \ldots, a_n)\}$ if and only if there exists a* $\texttt{cjump}(x_i, \ell)$ *operation.*

Proof. $\Leftarrow$ It is clear by the definition of negative (or positive) sample points.

$\Rightarrow$ Let $S := \{\alpha' \mid \alpha'(\bar{\boldsymbol{x}}) \in (a_1, \ldots, a_{i-1}, \mathbb{R}, a_{i+1}, \ldots, a_n)\}$. It is equivalent to proving that if there exists no $\texttt{cjump}(x_i, \ell)$ operation, then no solution to ℓ exists in S. We only prove it for ℓ of the form $p(\bar{\boldsymbol{x}}) < 0$. Recall Definition 10 and Remark 1. There are only three cases in which $\texttt{cjump}(x_i, \ell)$ does not exist: (1) p^* is the zero polynomial, (2) p^* has no real roots, (3) p^* has a finite number of real roots, say $r_1, \ldots, r_s$ ($s \in \mathbb{Z}_{\geq 1}$), and p^* is positive on $\mathbb{R} \setminus \{r_1, \ldots, r_s\}$, where p^* denotes the polynomial $p(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$. In the first case, $p(\alpha'(\bar{\boldsymbol{x}})) = 0$ and in the third case, $p(\alpha'(\bar{\boldsymbol{x}})) \geq 0$ for any assignment $\alpha' \in S$. In the second case, the sign of p^* is positive constantly or negative constantly on the whole real axis. Since ℓ is false under α, we have $p(\alpha(\bar{\boldsymbol{x}})) \geq 0$, that is, $p^*(a_i) \geq 0$. So, $p^*(x_i) > 0$ for any $x_i \in \mathbb{R}$, which means $p(\alpha'(\bar{\boldsymbol{x}})) > 0$ for any $\alpha' \in S$. Therefore, no solution to ℓ exists in S in the three cases. That completes the proof. □

The above theorem shows that if $\texttt{cjump}(x_i, \ell)$ does not exist, then there is no need to search for a solution to ℓ along the line $(a_1, \ldots, a_{i-1}, \mathbb{R}, a_{i+1}, \ldots, a_n)$. And we can always obtain a solution to ℓ after executing a $\texttt{cjump}(x_i, \ell)$ operation.

Example 8. Assume the current assignment is $\alpha : x_1 \mapsto 1,\ x_2 \mapsto 1$. Consider two false atomic polynomial formulas $\ell_1 : 2x_1^2 + 2x_2^2 - 1 < 0$ and $\ell_2 : x_1^8 x_2^3 - 4x_1^6 + 6x_1^4 x_2 - 4x_1^2 + x_2 > 0$. Let $p_1 := \texttt{poly}(\ell_1)$ and $p_2 := \texttt{poly}(\ell_2)$.

We first consider $\texttt{cjump}(x_i, \ell_1)$. For the variable x_1, the corresponding univariate polynomial is $p_1(x_1, 1) = 2x_1^2 + 1$, and for x_2, the corresponding one is $p_1(1, x_2) = 2x_2^2 + 1$. Both of them have no real roots, and thus there exists no $\texttt{cjump}(x_1, \ell_1)$ operation and no $\texttt{cjump}(x_2, \ell_1)$ operation for ℓ_1. Applying Theorem 2, we know a solution of ℓ_1 can only locate in $\mathbb{R}^2 \setminus (1, \mathbb{R}) \cup (\mathbb{R}, 1)$ (also see Fig. 5 (a)). So, we cannot find a solution of ℓ_1 through one-step cell-jump from the assignment point $(1, 1)$ along the lines $(1, \mathbb{R})$ and $(\mathbb{R}, 1)$.

Then consider $\texttt{cjump}(x_i, \ell_2)$. For the variable x_1, the corresponding univariate polynomial is $p_2(x_1, 1) = x_1^8 - 4x_1^6 + 6x_1^4 - 4x_1^2 + 1$. Recall Example 7. There are two positive sample points of $p_2(x_1, 1) : -\frac{215}{128}, \frac{215}{128}$. And $\frac{215}{128}$ is the closest one to $\alpha(x_1)$. So, $\texttt{cjump}(x_1, \ell_2)$ assigns x_1 to $\frac{215}{128}$. After executing $\texttt{cjump}(x_1, \ell_2)$, the assignment becomes $\alpha' : x_1 \mapsto \frac{215}{128},\ x_2 \mapsto 1$ which is a solution of ℓ_2. For the variable x_2, the corresponding polynomial is $p_2(1, x_2) = x_2^3 + 7x_2 - 8$, which has one real root 1. A sequence of isolating intervals of $p_2(1, x_2)$ is $(\frac{19}{32}, \frac{215}{128})$, and $\frac{215}{128}$ is the only positive sample point. So, $\texttt{cjump}(x_2, \ell_2)$ assigns x_2 to $\frac{215}{128}$, and then the assignment becomes $\alpha'' : x_1 \mapsto 1,\ x_2 \mapsto \frac{215}{128}$ which is another solution of ℓ_2.

4.3 Cell-Jump Along a Fixed Straight Line

Given the current assignment α such that $\alpha(\bar{\boldsymbol{x}}) = (a_1, \ldots, a_n) \in \mathbb{Q}^n$, a false atomic polynomial formula ℓ of the form $p(\bar{\boldsymbol{x}}) > 0$ or $p(\bar{\boldsymbol{x}}) < 0$ and a vector $dir = (d_1, \ldots, d_n) \in \mathbb{Q}^n$, we propose Algorithm 2 to find a cell-jump operation along the straight line L specified by the point $\alpha(\bar{\boldsymbol{x}})$ and the direction dir, denoted as $\texttt{cjump}(dir, \ell)$.

In order to analyze the values of $p(\bar{\boldsymbol{x}})$ on line L, we introduce a new variable t and replace every x_i in $p(\bar{\boldsymbol{x}})$ with $a_i + d_i t$ to get $p^*(t)$. If $\texttt{rela}(\ell)$ =‘<’ and $p^*(t)$ has negative sample points, there exists a $\texttt{cjump}(dir, \ell)$ operation. Let t^* be a negative sample point of $p^*(t)$ closest to 0. The assignment becomes $\alpha' : x_1 \mapsto a_1 + d_1 t^*, \ldots, x_n \mapsto a_n + d_n t^*$ after executing the operation $\texttt{cjump}(dir, \ell)$. It is obvious that α' is a solution to ℓ. If $\texttt{rela}(\ell)$ =‘>’ and $p^*(t)$ has positive sample points, the situation is similar. Otherwise, ℓ has no cell-jump operation along line L.

Similarly, we have:

Theorem 3. *Suppose the current assignment is $\alpha : x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$ where $a_i \in \mathbb{Q}$. Let ℓ be a false atomic polynomial formula under α with a relational operator '$<$' or '$>$', $dir := (d_1, \ldots, d_n)$ a vector in $\mathbb{Q}^n$ and $L := \{(a_1 + d_1 t, \ldots, a_n + d_n t) \mid t \in \mathbb{R}\}$. There exists a solution of ℓ in L if and only if there exists a* $\texttt{cjump}(dir, \ell)$ *operation.*

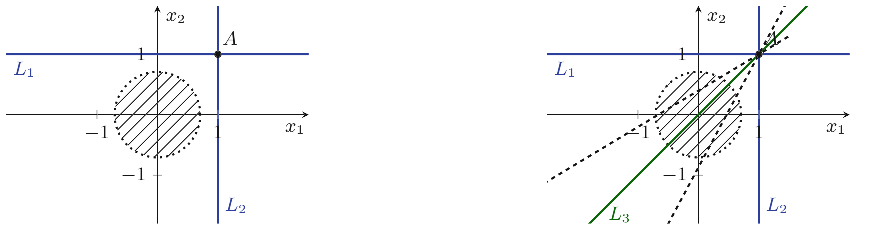
Theorem 3 implies that through one-step cell-jump from the point $\alpha(\bar{\boldsymbol{x}})$ along any line that intersects the solution set of ℓ, a solution to ℓ will be found.

Example 9. Assume the current assignment is $\alpha : x_1 \mapsto 1,\ x_2 \mapsto 1$. Consider the false atomic polynomial formula $\ell_1 : 2x_1^2 + 2x_2^2 - 1 < 0$ in Example 8. Let $p := \texttt{poly}(\ell_1)$. By Fig. 5 (b), the line (line L_3) specified by the point $\alpha(\bar{\boldsymbol{x}})$ and the direction vector $dir = (1, 1)$ intersects the solution set of ℓ_1. So, there exists a $\texttt{cjump}(dir, \ell_1)$ operation by Theorem 3. Notice that the line can be described in a parametric form, that is $\{(x_1, x_2) \mid x_1 = 1+t, x_2 = 1+t \text{ where } t \in \mathbb{R}\}$. Then, analyzing the values of $p(\bar{\boldsymbol{x}})$ on the line is equivalent to analyzing those of $p^*(t)$ on the real axis, where $p^*(t) = p(1+t, 1+t) = 4t^2+8t+3$. A sequence of isolating intervals of p^* is $(-\frac{215}{128}, -\frac{75}{64})$, $(-\frac{19}{32}, -\frac{61}{128})$, and there are two negative sample points: $-\frac{75}{64}$, $-\frac{19}{32}$. Since $-\frac{19}{32}$ is the closest one to 0, the operation $\texttt{cjump}(dir, \ell_1)$ changes the assignment to $\alpha' : x_1 \mapsto \frac{13}{32},\ x_2 \mapsto \frac{13}{32}$, which is a solution of ℓ_1. Again by Fig. 5, there are other lines (the dashed lines) that go through $\alpha(\bar{\boldsymbol{x}})$ and intersect the solution set. So, we can also find a solution to ℓ_1 along these lines. Actually, for any false atomic polynomial formula with '$<$' or '$>$' that really has solutions, there always exists some direction dir in $\mathbb{Q}^n$ such that $\texttt{cjump}(dir, \ell)$ finds one of them. Therefore, the more directions we try, the greater the probability of finding a solution of ℓ.

Algorithm 2.. Cell-Jump Along a Fixed Straight Line

```
Input  : $\alpha = (a_1, \ldots, a_n)$, the current assignment $x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$ where $a_i \in \mathbb{Q}$
         $\ell$, a false atomic polynomial formula under $\alpha$ with a relational operator '$<$' or '$>$'
         $dir = (d_1, \ldots, d_n)$, a vector in $\mathbb{Q}^n$
Output: $\alpha'$, the assignment after executing a cjump($dir, \ell$) operation, which is a solution to $\ell$;
         FAIL, if there exists no cjump($dir, \ell$) operation

$p \leftarrow$ poly($\ell$)
$p^* \leftarrow$ replace every $x_i$ in $p$ with $a_i + d_i t$, where $t$ is a new variable
if $rela(\ell) =$ '$<$' and $p^*$ has negative sample points then
    $t^* \leftarrow$ a negative sample point of $p^*$ closest to 0
    $\alpha' \leftarrow (a_1 + d_1 t^*, \ldots, a_n + d_n t^*)$
    return $\alpha'$
if $rela(\ell) =$ '$>$' and $p^*$ has positive sample points then
    $t^* \leftarrow$ a positive sample point of $p^*$ closest to 0
    $\alpha' \leftarrow (a_1 + d_1 t^*, \ldots, a_n + d_n t^*)$
    return $\alpha'$
return FAIL
```



(a) Neither L_1 nor L_2 intersects the solution set.

(b) Line L_3 and the dashed lines intersect the solution set.

Fig. 5. The figure of the cell-jump operations along the lines L_1, L_2 and L_3 for the false atomic polynomial formula $\ell_1 : 2x_1^2 + 2x_2^2 - 1 < 0$ under the assignment $\alpha : x_1 \mapsto 1, x_2 \mapsto 1$. The dashed circle denotes the circle $2x_1^2+2x_2^2-1 = 0$ and the shaded part in it represents the solution set of the atom. The coordinate of point A is $(1, 1)$. Lines L_1, L_2 and L_3 pass through A and are parallel to the x_1-axis, the x_2-axis and the vector $(1, 1)$, respectively.

Remark 2. For a false atomic polynomial formula ℓ with '$<$' or '$>$', $\texttt{cjump}(x_i, \ell)$ and $\texttt{cjump}(dir, \ell)$ make an assignment move to a new assignment, and both assignments map to an element in $\mathbb{Q}^n$. In fact, we can view $\texttt{cjump}(x_i, \ell)$ as a special case of $\texttt{cjump}(dir, \ell)$ where the i-th component of dir is 1 and all the other components are 0. The main difference between $\texttt{cjump}(x_i, \ell)$ and $\texttt{cjump}(dir, \ell)$ is that $\texttt{cjump}(x_i, \ell)$ only changes the value of one variable while $\texttt{cjump}(dir, \ell)$ may change the values of many variables. The advantage of $\texttt{cjump}(x_i, \ell)$ is to avoid that some atoms can never become true when the values of many variables are adjusted together. However, performing $\texttt{cjump}(dir, \ell)$ is more efficient in some cases, since it may happen that a solution to ℓ can be found through one-step $\texttt{cjump}(dir, \ell)$, but through many steps of $\texttt{cjump}(x_i, \ell)$.

5 Scoring Functions

Scoring functions guide local search algorithms to pick an operation at each step. In this section, we introduce a score function which measures the difference of the distances to satisfaction under the assignments before and after performing an operation.

First, we define the distance to truth of an atomic polynomial formula.

Definition 12 (Distance to Truth). *Given the current assignment α such that $\alpha(\bar{\boldsymbol{x}}) = (a_1, \dots, a_n) \in \mathbb{Q}^n$ and a positive parameter $pp \in \mathbb{Q}_{>0}$, for an atomic polynomial formula ℓ with $p := \texttt{poly}(\ell)$, its* distance to truth *is*

$$\texttt{dtt}(\ell, \alpha, pp) := \begin{cases} 0, & \text{if } \alpha \text{ is a solution to } \ell, \\ |p(a_1, \dots, a_n)| + pp, & \text{otherwise.} \end{cases}$$

For an atomic polynomial formula ℓ, the parameter pp is introduced to guarantee that the distance to truth of ℓ is 0 if and only if the current assignment

α is a solution of ℓ. Based on the definition of $\texttt{dtt}$, we use the method of [8, Definition 3 and 4] to define the distance to satisfaction of a polynomial clause and the score of an operation, respectively.

Definition 13 (Distance to Satisfaction). *Given the current assignment* α *and a parameter* $pp \in \mathbb{Q}_{>0}$*, the* distance to satisfaction *of a polynomial clause* c *is* $\texttt{dts}(c, \alpha, pp) := \min_{\ell \in c}\{\texttt{dtt}(\ell, \alpha, pp)\}$.

Definition 14 (Score). *Given a polynomial formula* F*, the current assignment* α *and a parameter* $pp \in \mathbb{Q}_{>0}$*, the* score *of an operation* op *is defined as*

$$\texttt{score}(op, \alpha, pp) := \sum_{c \in F}(\texttt{dts}(c, \alpha, pp) - \texttt{dts}(c, \alpha', pp)) \cdot w(c),$$

where $w(c)$ *denotes the weight of clause* c*, and* α' *is the assignment after performing* op.

Note that the definition of the score is associated with the weights of clauses. In our algorithm, we employ the probabilistic version of the PAWS scheme [9, 32] to update clause weights. The initial weight of every clause is 1. Given a probability sp, the clause weights are updated as follows: with probability $1 - sp$, the weight of every falsified clause is increased by one, and with probability sp, for every satisfied clause with weight greater than 1, the weight is decreased by one.

6 The Main Algorithm

Based on the proposed cell-jump operation (see Sect. 4) and scoring function (see Sect. 5), we develop a local search algorithm, called LS Algorithm, for solving satisfiability of polynomial formulas in this section. The algorithm is a refined extension of the general local search framework as described in Sect. 2.2, where we design a two-level operation selection. The section also explains the restart mechanism and an optimization strategy used in the algorithm.

Given a polynomial formula F such that every relational operator appearing in it is '$<$' or '$>$' and an initial assignment that maps to an element in $\mathbb{Q}^n$, LS Algorithm (Algorithm 3) searches for a solution of F from the initial assignment, which has the following four steps:

(i) **Test whether the current assignment is a solution if the terminal condition is not reached.** If the assignment is a solution, return the solution. If it is not, go to the next step. The algorithm terminates at once and returns "unknown" if the terminal condition is satisfied.

(ii) **Try to find a decreasing cell-jump operation along a line parallel to a coordinate axis.** We first need to check whether such an operation exists. That is, to determine whether the set D is empty, where $D = \{\texttt{cjump}(x_i, \ell) \mid \ell$ is a false atom, x_i appears in ℓ and $\texttt{cjump}(x_i, \ell)$ is decreasing$\}$. If $D = \emptyset$, go to the next step. Otherwise, we adopt the two-level heuristic in [8,

Section 4.2]. The heuristic distinguishes a special subset $S \subseteq D$ from the rest of D, where $S = \{\texttt{cjump}(x_i, \ell) \in D \mid \ell$ appears in a falsified clause$\}$, and searches for an operation with the highest score from S. If it fails to find any operation from S (*i.e.* $S = \emptyset$), then it searches for one with the highest score from $D \setminus S$. Perform the found operation and update the assignment. Go to Step (*i*).

(iii) **Update clause weights according to the PAWS scheme.**

(iv) **Generate some direction vectors and try to find a decreasing cell-jump operation along a line parallel to a generated vector.** Since it fails to execute a decreasing cell-jump operation along any line parallel to a coordinate axis, we generate some new directions and search for a decreasing cell-jump operation along one of them. The candidate set of such operations is $\{\texttt{cjump}(dir, \ell) \mid \ell$ isafalseatom, dir isageneratedirection and $\texttt{cjump}(dir, \ell)$ isdecreasing$\}$. If the set is empty, the algorithm returns "unknown". Otherwise, we use the two-level heuristic in Step (*ii*) again to choose an operation from the set. Perform the chosen operation and update the assignment. Go to Step (*i*).

We propose a two-level operation selection in LS Algorithm, which prefers to choose an operation changing the values of less variables. Concretely, only when there does not exist a decreasing $\texttt{cjump}(x_i, \ell)$ operation that changes the value of one variable, do we update clause weights and pick a $\texttt{cjump}(dir, \ell)$ operation that may change values of more variables. The strategy makes sense in experiments, since it is observed that changing too many variables together at the beginning might make some atoms never become true.

It remains to explain the restart mechanism and an optimization strategy.

Restart Mechanism. Given any initial assignment, LS Algorithm takes it as the starting point of the local search. If the algorithm returns "unknown", we restart LS Algorithm with another initial assignment. A general local search framework, like Algorithm 1, searches for a solution from only one starting point. However, the restart mechanism allows us to search from more starting points. The approach of combining the restart mechanism and a local search procedure also aids global search, which finds a solution over the entire search space.

We set the initial assignments for restarts as follows: All variables are assigned with 1 for the first time. For the second time, for a variable x_i, if there exists clause $x_i < ub \vee x_i = ub$ or $x_i > lb \vee x_i = lb$, then x_i is assigned with ub or lb; otherwise, x_i is assigned with 1. For the i-th time ($3 \leq i \leq 7$), every variable is assigned with 1 or -1 randomly. For the i-th time ($i \geq 8$), every variable is assigned with a random integer between $-50(i-6)$ and $50(i-6)$.

Forbidding Strategies. An inherent problem of the local search method is cycling, *i.e.*, revisiting assignments. Cycle phenomenon wastes time and prevents the search from getting out of local minima. So, we employ a popular forbidding strategy, called tabu strategy [18], to deal with it. The tabu strategy forbids reversing the recent changes and can be directly applied in LS Algorithm. Notice that every cell-jump operation increases or decreases the values of some variables. After executing an operation that increases/decreases the value of a variable, the tabu strategy forbids decreasing/increasing the value of the variable in the subsequent tt iterations, where $tt \in \mathbb{Z}_{\geq 0}$ is a given parameter.

Algorithm 3. LS Algorithm

Input : F, a polynomial formula such that the relational operator of every atom is '<' or '>' $init_\alpha$, an initial assignment that maps to an element in $\mathbb{Q}^n$
Output: a solution (in $\mathbb{Q}^n$) to F or unknown

```
1  $\alpha \leftarrow init_\alpha$
2  while the terminal condition is not reached do
3      if $\alpha$ satisfies $F$ then return $\alpha$
4      $fal\_cl \leftarrow$ the set of atoms in falsified clauses
5      $sat\_cl \leftarrow$ the set of false atoms in satisfied clauses
6      if $\exists$ a decreasing cjump$(x_i, \ell)$ operation where $\ell \in fal\_cl$ then
7          $op \leftarrow$ such an operation with the highest score
8          $\alpha \leftarrow \alpha$ with $op$ performed
9      else if $\exists$ a decreasing cjump$(x_i, \ell)$ operation where $\ell \in sat\_cl$ then
10         $op \leftarrow$ such an operation with the highest score
11         $\alpha \leftarrow \alpha$ with $op$ performed
12     else
13         update clause weights according to the PAWS scheme
14         generate a direction vector set $dset$
15         if $\exists$ a decreasing cjump$(dir, \ell)$ operation where $dir \in dset$ and $\ell \in fal\_cl$ then
16             $op \leftarrow$ such an operation with the highest score
17             $\alpha \leftarrow \alpha$ with $op$ performed
18         else if $\exists$ a decreasing cjump$(dir, \ell)$ operation where $dir \in dset$ and $\ell \in sat\_cl$ then
19             $op \leftarrow$ such an operation with the highest score
20             $\alpha \leftarrow \alpha$ with $op$ performed
21         else
22             return unknown
23 return unknown
```

Remark 3. If the input formula has equality constraints, then we need to define a cell-jump operation for a false atom of the form $p(\bar{\boldsymbol{x}}) = 0$. Given the current assignment $\alpha : x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$ $(a_i \in \mathbb{Q})$, the operation should assign some variable x_i to a real root of $p(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$, which may be not a rational number. Since it is time-consuming to isolate real roots of a polynomial with algebraic coefficients, we must guarantee that all assignments are rational during the search. Thus, we restrict that for every equality equation $p(\bar{\boldsymbol{x}}) = 0$ in the formula, there exists at least one variable such that the degree of p w.r.t. the variable is 1. Then, LS Algorithm also works for such a polynomial formula after some minor modifications: In Line 6 (or Line 9), for every atom $\ell \in fal\_cl$ (or $\ell \in sat\_cl$) and for every variable x_i, if ℓ has the form $p(\bar{\boldsymbol{x}}) = 0$, p is linear w.r.t. x_i and $p(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$ is not a constant polynomial, there is a candidate operation that changes the value of x_i to the (rational) solution of $p(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n) = 0$; if ℓ has the form $p(\bar{\boldsymbol{x}}) > 0$ or $p(\bar{\boldsymbol{x}}) < 0$, a candidate operation is cjump(x_i, ℓ). We perform a decreasing candidate operation with the highest score if such one exists, and update α in Line 8 (or Line 11).

In Line 15 (or Line 18), we only deal with inequality constraints from fal_cl (or sat_cl), and skip equality constraints.

7 Experiments

We carried out experiments to evaluate LS Algorithm on two classes of instances, where one class consists of selected instances from SMT-LIB while another is generated randomly, and compared our tool with state-of-the-art SMT(NRA) solvers. Furthermore, we combine our tool with Z3, CVC5, Yices2 and MathSAT5 respectively to obtain four sequential portfolio solvers, which show better performance.

7.1 Experiment Preparation

Implementation: We implemented LS Algorithm with `Maple2022` as a tool, which is also named LS. There are 3 parameters in LS Algorithm: pp for computing the score of an operation, tt for the tabu strategy and sp for the PAWS scheme, which are set as $pp = 1$, $tt = 10$ and $sp = 0.003$. The direction vectors in LS Algorithm are generated in the following way: Suppose the current assignment is $x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$ $(a_i \in \mathbb{Q})$ and the polynomial appearing in the atom to deal with is p. We generate 12 vectors. The first one is the gradient vector $(\frac{\partial p}{\partial x_1}, \ldots, \frac{\partial p}{\partial x_n})|_{(a_1,\ldots,a_n)}$. The second one is the vector $(a_1, \ldots, a_n)$. And the rest are random vectors where every component is a random integer between -1000 and 1000.

Experiment Setup: All experiments were conducted on 16-Core Intel Core i9-12900KF with 128GB of memory and ARCH LINUX SYSTEM. We compare our tool with 4 state-of-the-art SMT(NRA) solvers, namely Z3 (4.11.2), CVC5 (1.0.3), Yices2 (2.6.4) and MathSAT5 (5.6.5). Each solver is executed with a cutoff time of 1200 seconds (as in the SMT Competition) for each instance. We also combine LS with every competitor solver as a sequential portfolio solver, referred to as "LS+OtherSolver", where we first run LS with a time limit of 10 seconds, and if LS fails to solve the instance within that time, we then proceed to run OtherSolver from scratch, allotting it the remaining 1190 seconds.

7.2 Instances

We prepare two classes of instances. One class consists of in total 2736 unknown and satisfiable instances from SMT-LIB(NRA)[4], where in every equality polynomial constraint, the degree of the polynomial w.r.t. each variable is less than or equal to 1.

The rest are random instances. Before introducing the generation approach of random instances, we first define some notation. Let $\mathbf{rn}(down, up)$ denote a

[4] https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NRA.

random integer between two integers *down* and *up*, and $\mathbf{rp}(\{x_1,\ldots,x_n\},d,m)$ denote a random polynomial $\sum_{i=1}^{m} c_i M_i + c_0$, where $c_i = \mathbf{rn}(-1000, 1000)$ for $0 \leq i \leq m$, M_1 is a random monomial in $\{x_1^{a_1}\cdots x_n^{a_n} \mid a_i \in \mathbb{Z}_{\geq 0},\ a_1+\cdots+a_n = d\}$ and M_i $(2 \leq i \leq m)$ is a random monomial in $\{x_1^{a_1}\cdots x_n^{a_n} \mid a_i \in \mathbb{Z}_{\geq 0},\ a_1+\cdots+a_n \leq d\}$.

A randomly generated polynomial formula $\mathbf{rf}(\{v\_n_1, v\_n_2\}, \{p\_n_1, p\_n_2\}, \{d_-, d_+\}, \{n_-, n_+\}, \{m_-, m_+\}, \{cl\_n_1, cl\_n_2\}, \{cl\_l_1, cl\_l_2\})$, where all parameters are in $\mathbb{Z}_{\geq 0}$, is constructed as follows: First, let $n := \mathbf{rn}(v\_n_1, v\_n_2)$ and generate n variables $x_1,\ldots,x_n$. Second, let $num := \mathbf{rn}(p\_n_1, p\_n_2)$ and generate num polynomials $p_1,\ldots,p_{num}$. Every p_i is a random polynomial $\mathbf{rp}(\{x_{i_1},\ldots,x_{i_{n_i}}\},d,m)$, where $n_i = \mathbf{rn}(n_-, n_+)$, $d = \mathbf{rn}(d_-, d_+)$, $m = \mathbf{rn}(m_-, m_+)$, and $\{x_{i_1},\ldots,x_{i_{n_i}}\}$ are n_i variables randomly selected from $\{x_1,\ldots,x_n\}$. Finally, let $cl\_n := \mathbf{rn}(cl\_n_1, cl\_n_2)$ and generate cl_n clauses such that the number of atoms in a generated clause is $\mathbf{rn}(cl\_l_1, cl\_l_2)$. The $\mathbf{rn}(cl\_l_1, cl\_l_2)$ atoms are randomly picked from $\{p_i < 0, p_i > 0, p_i = 0 \mid 1 \leq i \leq num\}$. If some picked atom has the form $p_i = 0$ and there exists a variable such that the degree of p_i w.r.t. the variable is greater than 1, replace the atom with $p_i < 0$ or $p_i > 0$ with equal probability. We generate totally 500 random polynomial formulas according to $\mathbf{rf}(\{30,40\},\{60,80\},\{20,30\},\{10,20\},\{20,30\},\{40,60\},\{3,5\})$.

The two classes of instances have different characteristics. The instances selected from SMT-LIB(NRA) usually contain lots of linear constraints, and their complexity is reflected in the propositional abstraction. For a random instance, all the polynomials in it are nonlinear and of high degrees, while its propositional abstraction is relatively simple.

7.3 Experimental Results

The experimental results are presented in Table 1. The column "#inst" records the number of instances. Let us first see Column "Z3"–Column "LS". On instances from SMT-LIB(NRA), LS performs worse than all competitors except MathSAT5, but it is still comparable. It is crucial to note that our approach is much faster than both CVC5 and Z3 on 90% of the Meti-Tarski benchmarks of SMT-LIB (2194 instances in total). On random instances, only LS solved all of them, while the competitor Z3 with the best performance solved 29% of them. The results show that LS is not good at solving polynomial formulas with complex propositional abstraction and lots of linear constraints, but it has great ability to handle those with high-degree polynomials. A possible explanation is that as a local search solver, LS cannot exploit the propositional abstraction well to find a solution. However, for a formula with plenty of high-degree polynomials, cell-jump may 'jump' to a solution faster.

The data revealed in the last column "LS+CVC5" of Table 1 indicates that the combination of LS and CVC5 manages to solve the majority of the instances across both classes, suggesting a complementary performance between LS and top-tier SMT(NRA) solvers. As shown in Table 2, when evaluating combinations of different solvers with LS, it becomes evident that our method significantly enhances the capabilities of existing solvers in the portfolio configurations. The

Table 1. Results on SMT-LIB(NRA) and random instances.

	#inst	Z3	CVC5	Yices2	MathSAT5	LS	LS+CVC5
SMT-LIB(NRA)	2736	2519	**2563**	2411	1597	2246	2602
├meti-tarski	2194	**2194**	2155	2173	1185	2192	2193
├20220314-Uncu	19	**19**	**19**	**19**	16	**19**	19
└other	523	306	389	219	**396**	35	390
Random instances	500	145	0	22	0	**500**	500
Total	3236	2664	2563	2433	1597	**2746**	3102

most striking improvement can be witnessed in the "LS+MathSAT5" combination, which demonstrates superior performance and the most significant enhancement among all the combination solvers.

Table 2. Performance Comparison of Different Solver Combinations with LS.

	#inst	LS+Z3	LS+CVC5	LS+Yices2	LS+MathSAT5
SMT-LIB(NRA)	2736	2518	2602	2432	**2609**
├meti-tarski	2194	**2194**	2193	**2194**	2191
├20220314-Uncu	19	**19**	**19**	**19**	**19**
└other	523	305	390	219	**399**
Random instances	500	**500**	**500**	**500**	**500**
Total	3236	3018	3102	2932	**3109**

Besides, Fig. 6 shows the performance of LS and the competitors on all instances. The horizontal axis represents time, while the vertical axis represents the number of solved instances within the corresponding time. Figure 7 presents the run time comparisons between LS+CVC5 and CVC5. Every point in the figure represents an instance. The horizontal coordinate of the point is the computing time of LS+CVC5 while the vertical coordinate is the computing time of CVC5 (for every instance out of time, we record its computing time as 1200 seconds). The figure shows that LS+CVC5 improves the performance of CVC5. We also present the run time comparisons between LS and each competitor in Figs. 8–11.

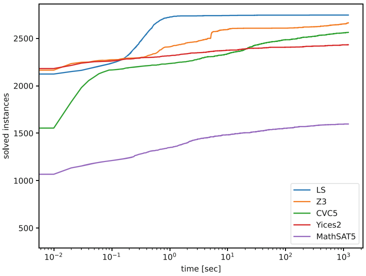


Fig. 6. Number of solved instances within given time (sec: seconds).

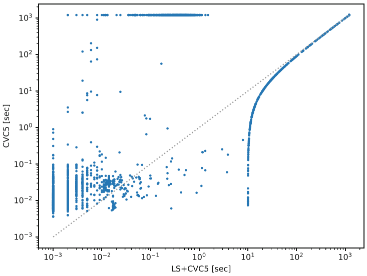


Fig. 7. Comparing LS+CVC5 with CVC5.

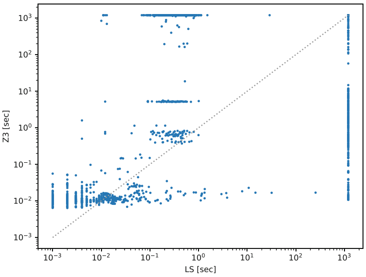


Fig. 8. Comparing LS with Z3.

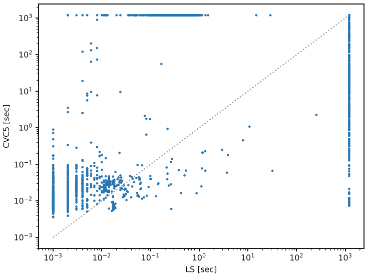


Fig. 9. Comparing LS with CVC5.

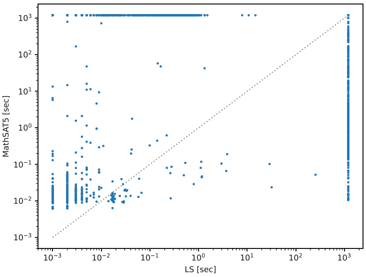


Fig. 10. Comparing LS with MathSAT5.

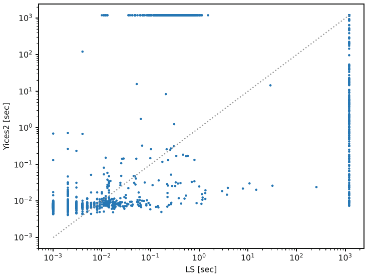


Fig. 11. Comparing LS with Yices2.

8 Conclusion

For a given SMT(NRA) formula, although the domain of variables in the formula is infinite, the satisfiability of the formula can be decided through tests on a finite number of samples in the domain. A complete search on such samples is inefficient. In this paper, we propose a local search algorithm for a special class of SMT(NRA) formulas, where every equality polynomial constraint is linear with respect to at least one variable. The novelty of our algorithm contains the cell-jump operation and a two-level operation selection which guide the algorithm to jump from one sample to another heuristically. The algorithm has been applied to two classes of benchmarks and the experimental results show that it is competitive with state-of-the-art SMT solvers and is good at solving those formulas with high-degree polynomial constraints. Tests on the solvers developed by combining this local search algorithm with Z3, CVC5, Yices2 or MathSAT5 indicate that the algorithm is complementary to these state-of-the-art SMT(NRA) solvers. For the future work, we will improve our algorithm such that it is able to handle all polynomial formulas.

Acknowledgement. This work is supported by National Key R&D Program of China (No. 2022YFA1005102) and the NSFC under grant No. 61732001. The authors are grateful to the reviewers for their valuable comments and constructive suggestions.

References

1. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. J. Logical Algebraic Methods Programm. **119**, 100633 (2021)
2. Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 16–29. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_3
3. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Biere, A.: Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. In: Proceedings of SAT Competition, pp. 44–45 (2016)
5. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS press (2009)
6. Brown, C.W.: Improved projection for cylindrical algebraic decomposition. J. Symb. Comput. **32**(5), 447–465 (2001)
7. Brown, C.W., Košta, M.: Constructing a single cell in cylindrical algebraic decomposition. J. Symb. Comput. **70**, 14–48 (2015)
8. Cai, S., Li, B., Zhang, X.: Local search for SMT on linear integer arithmetic. In: International Conference on Computer Aided Verification. pp. 227–248. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_12
9. Cai, S., Su, K.: Local search for Boolean satisfiability with configuration checking and subscore. Artif. Intell. **204**, 75–98 (2013)
10. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. ACM Trans. Comput. Log. **19**(3), 1–52 (2018)
11. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
12. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R., et al.: Handbook of Model Checking, vol. 10. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8

13. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975). https://doi.org/10.1007/3-540-07407-4_17
14. Collins, G.E., Hong, H.: Partial cylindrical algebraic decomposition for quantifier elimination. J. Symb. Comput. **12**(3), 299–328 (1991)
15. De Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 1–12. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_1
16. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
17. Fröhlich, A., Biere, A., Wintersteiger, C., Hamadi, Y.: Stochastic local search for satisfiability modulo theories. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 29 (2015)
18. Glover, F., Laguna, M.: Tabu search. In: Handbook of Combinatorial Optimization, pp. 2093–2229. Springer (1998)
19. Griggio, A., Phan, Q.-S., Sebastiani, R., Tomasi, S.: Stochastic local search for SMT: combining theory solvers with WalkSAT. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 163–178. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24364-6_12
20. Hong, H.: An improvement of the projection operator in cylindrical algebraic decomposition. In: Proceedings of the International Symposium on Symbolic and Algebraic Computation, pp. 261–264 (1990)
21. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 339–354. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_27
22. Lazard, D.: An improved projection for cylindrical algebraic decomposition. In: Algebraic Geometry and its Applications, pp. 467–476. Springer (1994). https://doi.org/10.1007/978-1-4612-2628-4_29
23. Li, C.M., Li, Yu.: Satisfying versus falsifying in local search for satisfiability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 477–478. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_43
24. Li, H., Xia, B.: Solving satisfiability of polynomial formulas by sample-cell projection. arXiv preprint arXiv:2003.00409 (2020)
25. Liu, M., et al.: NRA-LS at the SMT competition 2022. Tool description document, see https://github.com/minghao-liu/NRA-LS (2022)
26. McCallum, S.: An improved projection operation for cylindrical algebraic decomposition. In: Quantifier Elimination and Cylindrical Algebraic Decomposition, pp. 242–268. Springer (1998). https://doi.org/10.1007/978-3-7091-9459-1_12
27. Mitchell, D., Selman, B., Leveque, H.: A new method for solving hard satisfiability problems. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 440–446 (1992)
28. Moura, L.d., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
29. Nalbach, J., Ábrahám, E., Specht, P., Brown, C.W., Davenport, J.H., England, M.: Levelwise construction of a single cylindrical algebraic cell. arXiv preprint arXiv:2212.09309 (2022)
30. Niemetz, A., Preiner, M.: Ternary propagation-based local search for more bit-precise reasoning. In: 2020 Formal Methods in Computer Aided Design (FMCAD), pp. 214–224. IEEE (2020)

31. Niemetz, A., Preiner, M., Biere, A.: Precise and complete propagation based local search for satisfiability modulo theories. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 199–217. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_11
32. Talupur, M., Sinha, N., Strichman, O., Pnueli, A.: Range allocation for separation logic. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 148–161. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_12
33. Tarski, A.: A decision method for elementary algebra and geometry. University of California Press (1951)
34. Tung, V.X., Van Khanh, T., Ogawa, M.: raSAT: an SMT solver for polynomial constraints. Formal Methods Syst Design **51**(3), 462–499 (2017). https://doi.org/10.1007/s10703-017-0284-9
35. Weispfenning, V.: Quantifier elimination for real algebra-the quadratic case and beyond. Appl. Algebra Eng. Commun. Comput. **8**(2), 85–101 (1997)

Partial Quantifier Elimination and Property Generation

Eugene Goldberg(✉)

Land O Lakes, USA
eu.goldberg@gmail.com

Abstract. We study partial quantifier elimination (PQE) for propositional CNF formulas with existential quantifiers. PQE is a generalization of quantifier elimination where one can limit the set of clauses taken out of the scope of quantifiers to a small subset of clauses. The appeal of PQE is that many verification problems (e.g., equivalence checking and model checking) can be solved in terms of PQE and the latter can be dramatically simpler than full quantifier elimination. We show that PQE can be used for property generation that one can view as a generalization of testing. The objective here is to produce an *unwanted* property of a design implementation, thus exposing a bug. We introduce two PQE solvers called *EG-PQE* and *EG-PQE*$^+$. *EG-PQE* is a very simple SAT-based algorithm. *EG-PQE*$^+$ is more sophisticated and robust than *EG-PQE*. We use these PQE solvers to find an unwanted property (namely, an unwanted invariant) of a buggy FIFO buffer. We also apply them to invariant generation for sequential circuits from a HWMCC benchmark set. Finally, we use these solvers to generate properties of a combinational circuit that mimic symbolic simulation.

1 Introduction

In this paper, we consider the following problem. Let $F(X, Y)$ be a propositional formula in conjunctive normal form (CNF)[1] where X, Y are sets of variables. Let G be a subset of clauses of F. Given a formula $\exists X[F]$, find a quantifier-free formula $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$. In contrast to *full* quantifier elimination (QE), only the clauses of G are taken out of the scope of quantifiers here. So, we call this problem *partial* QE (PQE) [1]. (In this paper, we consider PQE only for formulas with *existential* quantifiers.) We will refer to H as a *solution* to PQE. Like SAT, PQE is a way to cope with the complexity of QE. But in contrast to SAT that is a *special* case of QE (where all variables are quantified), PQE *generalizes* QE. The latter is just a special case of PQE where $G = F$ and the entire formula is unquantified. Interpolation [2,3] can be viewed as a special case of PQE as well [4,5].

[1] Every formula is a propositional CNF formula unless otherwise stated. Given a CNF formula F represented as the conjunction of clauses $C_1 \wedge \cdots \wedge C_k$, we will also consider F as the *set* of clauses $\{C_1, \ldots, C_k\}$.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 110–131, 2023.
https://doi.org/10.1007/978-3-031-37703-7_6

The appeal of PQE is threefold. First, it can be much more efficient than QE if G is a *small* subset of F. Second, many verification problems like SAT, equivalence checking, model checking can be solved in terms of PQE [1,6–8]. So, PQE can be used to design new efficient methods for solving known problems. Third, one can apply PQE to solving *new* problems like property generation considered in this paper. In practice, to perform PQE, it suffices to have an algorithm that takes a single clause out of the scope of quantifiers. Namely, given a formula $\exists X[F(X,Y)]$ and a clause $C \in F$, this algorithm finds a formula $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C\}]$. To take out k clauses, one can apply this algorithm k times. Since $H \wedge \exists X[F] \equiv H \wedge \exists X[F \setminus \{C\}]$, solving the PQE above reduces to finding $H(Y)$ that makes C *redundant* in $H \wedge \exists X[F]$. So, the PQE algorithms we present here employ *redundancy based reasoning.* We describe two PQE algorithms called *EG-PQE* and *EG-PQE*$^+$ where "*EG*" stands for "Enumerate and Generalize". *EG-PQE* is a very simple SAT-based algorithm that can sometimes solve very large problems. *EG-PQE*$^+$ is a modification of *EG-PQE* that makes the algorithm more powerful and robust.

In [7], we showed the viability of an equivalence checker based on PQE. In particular, we presented instances for which this equivalence checker outperformed ABC [9], a high quality tool. In this paper, we describe and check experimentally one more important application of PQE called property generation. Our motivation here is as follows. Suppose a design implementation *Imp* meets the set of specification properties $P_1,\dots,P_m$. Typically, this set is incomplete. So, *Imp* can still be buggy even if every $P_i, i = 1,\dots,m$ holds. Let $P^*_{m+1},\dots,P^*_n$ be *desired* properties adding which makes the specification complete. If *Imp* meets the properties $P_1,\dots,P_m$ but is still buggy, a missed property P^*_i above fails. That is, *Imp* has the *unwanted* property $\overline{P^*_i}$. So, one can detect bugs by generating unspecified properties of *Imp* and checking if there is an unwanted one.

Currently, identification of unwanted properties is mostly done by massive testing. (As we show later, the input/output behavior specified by a single test can be cast as a simple property of *Imp*.) Another technique employed in practice is *guessing* unwanted properties that may hold and formally checking if this is the case. The problem with these techniques is that they can miss an unwanted property. In this paper, we describe property generation by PQE. The benefit of PQE is that it can produce much more complex properties than those corresponding to single tests. So, using PQE one can detect bugs that testing overlooks or cannot find in principle. Importantly, PQE generates properties covering different parts of *Imp*. This makes the search for unwanted properties more systematic and facilitates discovering bugs that can be missed if one simply guesses unwanted properties that may hold.

In this paper, we experimentally study generation of invariants of a sequential circuit N. An invariant of N is unwanted if a state that is supposed to be reachable in N falsifies this invariant and hence is unreachable. Note that finding a formal proof that N has no unwanted invariants is impractical. (It is hard to efficiently prove a large set of states reachable because different states are reached by different execution traces.) So developing practical methods for finding unwanted invariants if very important. We also study generation of properties mimicking symbolic simulation for a combinational circuit obtained by unrolling a sequential circuit. An unwanted property here exposes a wrong execution trace.

This paper is structured as follows. (Some additional information can be found in the supporting technical report [5].) In Sect. 2, we give basic definitions. Section 3 presents property generation for a combinational circuit. In Sect. 4, we describe invariant generation for a sequential circuit. Sections 5 and 6 present *EG-PQE* and *EG-PQE*$^+$ respectively. In Sect. 7, invariant generation is used to find a bug in a FIFO buffer. Experiments with invariant generation for HWMCC benchmarks are described in Sect. 8. Section 9 presents an experiment with property generation for combinational circuits. In Sect. 10 we give some background. Finally, in Sect. 11, we make conclusions and discuss directions for future research.

2 Basic Definitions

In this section, when we say "formula" without mentioning quantifiers, we mean "a quantifier-free formula".

Definition 1. *We assume that formulas have only Boolean variables. A* ***literal*** *of a variable v is either v or its negation. A* ***clause*** *is a disjunction of literals. A formula F is in conjunctive normal form (****CNF****) if $F = C_1 \wedge \cdots \wedge C_k$ where $C_1, \dots, C_k$ are clauses. We will also view F as the* ***set of clauses*** *$\{C_1, \dots, C_k\}$. We assume that* ***every formula is in CNF****.*

Definition 2. *Let F be a formula. Then* $\boldsymbol{Vars(F)}$ *denotes the set of variables of F and* $\boldsymbol{Vars(\exists X[F])}$ *denotes $Vars(F) \setminus X$.*

Definition 3. *Let V be a set of variables. An* ***assignment*** *$\vec{q}$ to V is a mapping $V' \rightarrow \{0,1\}$ where $V' \subseteq V$. We will denote the set of variables assigned in $\vec{q}$ as* $\boldsymbol{Vars(q)}$*. We will refer to $\vec{q}$ as a* ***full assignment*** *to V if $Vars(\boldsymbol{q}) = V$. We will denote as* $\boldsymbol{\vec{q} \subseteq \vec{r}}$ *the fact that a) $Vars(\boldsymbol{q}) \subseteq Vars(\boldsymbol{r})$ and b) every variable of $Vars(\boldsymbol{q})$ has the same value in $\vec{q}$ and $\vec{r}$.*

Definition 4. *A literal, a clause and a formula are said to be* ***satisfied*** *(respectively* ***falsified****) by an assignment $\vec{q}$ if they evaluate to 1 (respectively 0) under $\vec{q}$.*

Definition 5. *Let C be a clause. Let H be a formula that may have quantifiers, and $\vec{q}$ be an assignment to $Vars(H)$. If C is satisfied by $\vec{q}$, then* $\boldsymbol{C_q \equiv 1}$*. Otherwise,* $\boldsymbol{C_q}$ *is the clause obtained from C by removing all literals falsified by $\vec{q}$. Denote by* $\boldsymbol{H_q}$ *the formula obtained from H by removing the clauses satisfied by $\vec{q}$ and replacing every clause C unsatisfied by $\vec{q}$ with $C_{\boldsymbol{q}}$.*

Definition 6. *Given a formula $\exists X[F(X,Y)]$, a clause C of F is called a* ***quantified clause*** *if $Vars(C) \cap X \neq \emptyset$. If $Vars(C) \cap X = \emptyset$, the clause C depends only on free, i.e., unquantified variables of$F$ and is called a* ***free clause****.*

Definition 7. *Let G, H be formulas that may have existential quantifiers. We say that G, H are* ***equivalent****, written* $\boldsymbol{G \equiv H}$*, if $G_{\boldsymbol{q}} = H_{\boldsymbol{q}}$ for all full assignments $\vec{q}$ to $\mathit{Vars}(G) \cup \mathit{Vars}(H)$.*

Definition 8. *Let $F(X,Y)$ be a formula and $G \subseteq F$ and $G \neq \emptyset$. The clauses of G are said to be* ***redundant in*** $\boldsymbol{\exists X[F]}$ *if $\exists X[F] \equiv \exists X[F \setminus G]$. Note that if $F \setminus G$ implies G, the clauses of G are redundant in $\exists X[F]$.*

Definition 9. *Given a formula $\exists X[F(X,Y))]$ and G where $G \subseteq F$, the* ***Partial Quantifier Elimination*** *(****PQE****) problem is to find $H(Y)$ such that* $\boldsymbol{\exists X[F] \equiv H \wedge \exists X[F \setminus G]}$*. (So, PQE takes G out of the scope of quantifiers.) The formula H is called a* ***solution*** *to PQE. The case of PQE where $G = F$ is called* ***Quantifier Elimination*** *(****QE****).*

Example 1. Consider the formula $F = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ where $C_1 = \overline{x}_3 \vee x_4$, $C_2 = y_1 \vee x_3$, $C_3 = y_1 \vee \overline{x}_4$, $C_4 = y_2 \vee x_4$. Let Y denote $\{y_1, y_2\}$ and X denote $\{x_3, x_4\}$. Consider the PQE problem of taking C_1 out of $\exists X[F]$, i.e., finding $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C_1\}]$. As we show later, $\exists X[F] \equiv y_1 \wedge \exists X[F \setminus \{C_1\}]$. That is, $H = y_1$ is a solution to the PQE problem above.

Remark 1. Let D be a clause of a solution H to the PQE problem of Definition 9. If $F \setminus G$ implies D, then $H \setminus \{D\}$ is a solution to this PQE problem too.

Proposition 1. *Let H be a solution to the PQE problem of Definition 9. That is, $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$. Then $F \Rightarrow H$ (i.e., F implies H).*

The proofs of propositions can be found in [5].

Definition 10. *Let clauses C',C'' have opposite literals of exactly one variable $w \in \mathit{Vars}(C') \cap \mathit{Vars}(C'')$. Then C',C'' are called* ***resolvable*** *on w. Let C be a clause of a formula G and $w \in \mathit{Vars}(C)$. The clause C is said to be* ***blocked*** *[10] in G with respect to the variable w if no clause of G is resolvable with C on w.*

Proposition 2. *Let a clause C be blocked in a formula $F(X,Y)$ with respect to a variable $x \in X$. Then C is redundant in $\exists X[F]$, i.e., $\exists X[F \setminus \{C\}] \equiv \exists X[F]$.*

3 Property Generation by PQE

Many known problems can be formulated in terms of PQE, thus facilitating the design of new efficient algorithms. In [5], we give a short summary of results on solving SAT, equivalence checking and model checking by PQE presented in [1,6–8]. In this section, we describe application of PQE to *property generation* for a combinational circuit. The objective of property generation is to expose a bug via producing an *unwanted* property.

Let $M(X,V,W)$ be a combinational circuit where X, V, W specify the sets of the internal, input and output variables of M respectively. Let $F(X,V,W)$ denote a formula specifying M. As usual, this formula is obtained by Tseitin's transformations [11]. Namely, F equals $F_{G_1} \wedge \dots \wedge F_{G_k}$ where $G_1, \dots, G_k$ are the gates of M and F_{G_i} specifies the functionality of gate G_i.

Example 2. Let G be a 2-input AND gate defined as $x_3 = x_1 \wedge x_2$ where x_3 denotes the output value and x_1, x_2 denote the input values of G. Then G is specified by the formula $F_G = (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_3)$. Every clause of F_G is falsified by an inconsistent assignment (where the output value of G is not implied by its input values). For instance, $x_1 \vee \overline{x}_3$ is falsified by the inconsistent assignment $x_1 = 0, x_3 = 1$. So, every assignment *satisfying* F_G corresponds to a *consistent* assignment to G and vice versa. Similarly, every assignment satisfying the formula F above is a consistent assignment to the gates of M and vice versa.

3.1 High-Level View of Property Generation by PQE

One generates properties by PQE until an unwanted property exposing a bug is produced. (Like in testing, one runs tests until a bug-exposing test is encountered.) The benefit of property generation by PQE is fourfold. First, by property generation, one can identify bugs that are hard or simply impossible to find by testing. Second, using PQE makes property generation efficient. Third, by taking out different clauses one can generate properties covering different parts of the design. This increases the probability of discovering a bug. Fourth, every property generated by PQE specifies a large set of high-quality tests.

In this paper (Sects. 7, 9), we consider cases where identifying an unwanted property is easy. However, in general, such identification is not trivial. A discussion of this topic is beyond the scope of this paper. (An outline of a procedure for deciding if a property is unwanted is given in [5].)

3.2 Property Generation as Generalization of Testing

The behavior of M corresponding to a single test can be cast as a property. Let $w_i \in W$ be an output variable of M and $\vec{v}$ be a test, i.e., a full assignment to the input variables V of M. Let $B^{\boldsymbol{v}}$ denote the longest clause falsified by $\vec{v}$, i.e., $\mathit{Vars}(B^{\boldsymbol{v}}) = V$. Let $l(w_i)$ be the literal satisfied by the value of w_i produced by M under input $\vec{v}$. Then the clause $B^{\boldsymbol{v}} \vee l(w_i)$ is satisfied by every assignment satisfying F, i.e., $B^{\boldsymbol{v}} \vee l(w_i)$ is a property of M. We will refer to it as a **single-test property** (since it describes the behavior of M for a single test). If the input $\vec{v}$ is supposed to produce the opposite value of w_i (i.e., the one *falsifying* $l(w_i)$), then $\vec{v}$ exposes a bug in M. In this case, the single-test property above is an **unwanted** property of M exposing the same bug as the test $\vec{v}$.

A single-test property can be viewed as a weakest property of M as opposed to the strongest property specified by $\exists X[F]$. The latter is the truth table of M that can be computed explicitly by performing QE on $\exists X[F]$. One can use PQE to generate properties of M that, in terms of strength, range from the weakest ones to the strongest property inclusively. (By combining clause splitting with PQE one can generate single-test properties, see the next subsection.) Consider the PQE problem of taking a clause C out of $\exists X[F]$. Let $H(V, W)$ be a solution to this problem, i.e., $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C\}]$. Since H is implied by F, it can be viewed as a **property** of M. If H is an **unwanted** property, M has a bug.

(Here we consider the case where a property of M is obtained by taking a clause out of formula $\exists X[F]$ where only the *internal* variables of M are quantified. Later we consider cases where some external variables of M are quantified too.)

We will assume that the property H generated by PQE has no redundant clauses (see Remark 1). That is, if $D \in H$, then $F \setminus \{C\} \not\Rightarrow D$. Then one can view H as a property that holds due to the presence of the clause C in F.

3.3 Computing Properties Efficiently

If a property H is obtained by taking only one clause out of $\exists X[F]$, its computation is much easier than performing QE on $\exists X[F]$. If computing H still remains too time-consuming, one can use the two methods below that achieve better performance at the expense of generating weaker properties. The first method applies when a PQE solver forms a solution *incrementally*, clause by clause (like the algorithms described in Sects. 5 and 6). Then one can simply stop computing H as soon as the number of clauses in H exceeds a threshold. Such a formula H is still implied by F and hence specifies a property of M.

The second method employs *clause splitting*. Here we consider clause splitting on input variables $v_1,\dots,v_p$, i.e., those of V (but one can split a clause on any subset of variables from $\mathit{Vars}(F)$). Let F' denote the formula F where a clause C is replaced with $p+1$ clauses: $C_1 = C \vee \overline{l(v_1)}$,..., $C_p = C \vee \overline{l(v_p)}$, $C_{p+1} = C \vee l(v_1) \vee \dots \vee l(v_p)$, where $l(v_i)$ is a literal of v_i. The idea is to obtain a property H by taking the clause C_{p+1} out of $\exists X[F']$ rather than C out of $\exists X[F]$. The former PQE problem is simpler than the latter since it produces a weaker property H. One can show that if $\{v_1,\dots,v_p\} = V$, then a) the complexity of PQE reduces to **linear**; b) taking out C_{p+1} actually produces a **single-test property**. The latter specifies the input/output behavior of M for the test $\vec{v}$ falsifying the literals $l(v_1),\dots,l(v_p)$. (The details can be found in [5].)

3.4 Using Design Coverage for Generation of Unwanted Properties

Arguably, testing is so effective in practice because one verifies a *particular design*. Namely, one probes different parts of this design using some coverage metric rather than sampling the truth table (which would mean verifying *every possible design*). The same idea works for property generation by PQE for the following two reasons. First, by taking out a clause, PQE generates a property inherent to the *specific* circuit M. (If one replaces M with an equivalent but structurally different circuit, PQE will generate different properties.) Second, by taking out different clauses of F one generates properties corresponding to different parts of M thus "covering" the design. This increases the chance to take out a clause corresponding to the buggy part of M and generate an unwanted property.

3.5 High-Quality Tests Specified by a Property Generated by PQE

In this subsection, we show that a property H generated by PQE, in general, specifies a large set of high-quality tests. Let $H(V,W)$ be obtained by taking C

out of $\exists X[F(X,V,W)]$. Let $Q(V,W)$ be a clause of H. As mentioned above, we assume that $F \setminus \{C\} \not\Rightarrow Q$. Then there is an assignment $(\vec{x}, \vec{v},\vec{w})$ satisfying formula $(F \setminus \{C\}) \wedge \overline{Q}$ where $\vec{x}$,$\vec{v}$,$\vec{w}$ are assignments to X, V, W respectively. (Note that by definition, $(\vec{v},\vec{w})$ falsifies Q.) Let $(\vec{x}^*, \vec{v}, \vec{w}^*)$ be the execution trace of M under the input $\vec{v}$. So, $(\vec{x}^*, \vec{v}, \vec{w}^*)$ satisfies F. Note that the output assignments $\vec{w}$ and $\vec{w}^*$ must be different because $(\vec{v}, \vec{w}^*)$ has to satisfy Q. (Otherwise, $(\vec{x}^*, \vec{v}, \vec{w}^*)$ satisfies $F \wedge \overline{Q}$ and so $F \not\Rightarrow Q$ and hence $F \not\Rightarrow H$.) So, one can view $\vec{v}$ as a test "detecting" disappearance of the clause C from F. Note that different assignments satisfying $(F \setminus \{C\}) \wedge \overline{Q}$ correspond to different tests $\vec{v}$. So, the clause Q of H, in general, specifies a very large number of tests. One can show that these tests are similar to those detecting stuck-at faults and so have very high quality [5].

4 Invariant Generation by PQE

In this section, we extend property generation for combinational circuits to sequential ones. Namely, we generate *invariants*. Note that generation of *desired* auxiliary invariants is routinely used in practice to facilitate verification of a predefined property. The problem we consider here is different in that our goal is to produce an *unwanted* invariant exposing a bug. We picked generation of invariants (over that of weaker properties just claiming that a state cannot be reached in k transitions or less) because identification of an unwanted invariant is, arguably, easier.

4.1 Bugs Making States Unreachable

Let N be a sequential circuit and S denote the state variables of N. Let $I(S)$ specify the initial state $\vec{s}_{ini}$ (i.e.,$I(\vec{s}_{ini})=1$). Let $T(S',V,S'')$ denote the transition relation of N where S', S'' are the present and next state variables and V specifies the (combinational) input variables. We will say that a state $\vec{s}$ of N is reachable if there is an execution trace leading to $\vec{s}$. That is, there is a sequence of states $\vec{s_0},\dots,\vec{s_k}$ where $\vec{s_0} = \vec{s}_{ini}$, $\vec{s_k} = \vec{s}$ and there exist $\vec{v_i}$ $i = 0,\dots,k-1$ for which $T(\vec{s_i}, \vec{v_i}, \vec{s}_{i+1}) = 1$. Let N have to satisfy a set of **invariants** $P_0(S),\dots,P_m(S)$. That is, P_i holds iff $P_i(\vec{s}) = 1$ for every reachable state $\vec{s}$ of N. We will denote the **aggregate invariant** $P_0 \wedge \dots \wedge P_m$ as $\boldsymbol{P_{agg}}$. We will call $\vec{s}$ a **bad state** of N if $P_{agg}(\vec{s}) = 0$. If P_{agg} holds, no bad state is reachable. We will call $\vec{s}$ a **good state** of N if $P_{agg}(\vec{s}) = 1$.

Typically, the set of invariants $P_0,\dots,P_m$ is incomplete in the sense that it does not specify all states that must be *unreachable*. So, a good state can well be unreachable. We will call a good state **operative** (or **op-state** for short) if it is supposed to be used by N and so should be *reachable*. We introduce the term *an operative state* just to factor out "useless" good states. We will say that N has an **op-state reachability bug** if an op-state is unreachable in N. In Sect. 7, we consider such a bug in a FIFO buffer. The fact that P_{agg} holds says *nothing* about reachability of op-states. Consider, for instance, a trivial circuit

N_{triv} that simply stays in the initial state $\vec{s}_{ini}$ and $P_{agg}(\vec{s}_{ini}) = 1$. Then P_{agg} holds for N_{triv} but the latter has op-state reachability bugs (assuming that the correct circuit must reach states other than $\vec{s}_{ini}$).

Let $R_{\vec{s}}(S)$ be the predicate satisfied only by a state $\vec{s}$. In terms of CTL, identifying an op-state reachability bug means finding $\vec{s}$ for which the property $EF.R_{\vec{s}}$ must hold but it does not. The reason for assuming $\vec{s}$ to be *unknown* is that the set of op-states is typically too large to *explicitly* specify every property $ET.R_{\vec{s}}$ to hold. This makes finding op-state reachability bugs very hard. The problem is exacerbated by the fact that reachability of different states is established by *different traces*. So, in general, one cannot efficiently prove many properties $EF.R_{\vec{s}}$ (for different states) *at once*.

4.2 Proving Operative State Unreachability by Invariant Generation

In practice, there are two methods to check reachability of op-states for large circuits. The first method is testing. Of course, testing cannot prove a state unreachable, however, the examination of execution traces may point to a potential problem. (For instance, after examining execution traces of the circuit N_{triv} above one realizes that many op-states look unreachable.) The other method is to check **unwanted invariants**, i.e., those that are supposed to fail. If an unwanted invariant holds for a circuit, the latter has an op-state reachability bug. For instance, one can check if a state variable $s_i \in S$ of a circuit never changes its initial value. To break this unwanted invariant, one needs to find an op-state where the initial value of s_i is flipped. (For the circuit N_{triv} above this unwanted invariant holds for every state variable.) The potential unwanted invariants are formed manually, i.e., simply *guessed*.

The two methods above can easily overlook an op-state reachability bug. Testing cannot prove that an op-state is unreachable. To correctly guess an unwanted invariant that holds, one essentially has to know the underlying bug. Below, we describe a method for invariant generation by PQE that is based on property generation for combinational circuits. The appeal of this method is twofold. First, PQE generates invariants "inherent" to the implementation at hand, which drastically reduces the set of invariants to explore. Second, PQE is able to generate invariants related to different parts of the circuit (including the buggy one). This increases the probability of generating an unwanted invariant. We substantiate this intuition in Sect. 7.

Let formula $\boldsymbol{F_k}$ specify the combinational circuit obtained by unfolding a sequential circuit N for k time frames and adding the initial state constraint $I(S_0)$. That is, $F_k = I(S_0) \wedge T(S_0, V_0, S_1) \wedge \cdots \wedge T(S_{k-1}, V_{k-1}, S_k)$ where S_j, V_j denote the state and input variables of j-th time frame respectively. Let $H(S_k)$ be a solution to the PQE problem of taking a clause C out of $\exists X_k[F_k]$ where $X_k = S_0 \cup V_0 \cup \cdots \cup S_{k-1} \cup V_{k-1}$. That is, $\exists X_k[F_k] \equiv H \wedge \exists X_k[F_k \setminus \{C\}]$. Note that in contrast to Sect. 3, here some external variables of the combinational circuit (namely, the input variables $V_0, \ldots, V_{k-1}$) are quantified too. So, H depends only on state variables of the last time frame. H can be viewed as a **local invariant** asserting that no state falsifying H can be reached in k transitions.

One can use H to find global invariants (holding for *every* time frame) as follows. Even if H is only a local invariant, a clause Q of H can be a *global* invariant. The experiments of Sect. 8 show that, in general, this is true for many clauses of H. (To find out if Q is a global invariant, one can simply run a model checker to see if the property Q holds.) Note that by taking out different clauses of F_k one can produce global single-clause invariants Q relating to different parts of N. From now on, when we say "an invariant" without a qualifier we mean a **global invariant**.

5 Introducing *EG-PQE*

In this section, we describe a simple SAT-based algorithm for performing PQE called *EG-PQE*. Here *'EG'* stands for 'Enumerate and Generalize'. *EG-PQE* accepts a formula $\exists X[F(X,Y)]$ and a clause $C \in F$. It outputs a formula $H(Y)$ such that $\exists X[F_{ini}] \equiv H \wedge \exists X[F_{ini} \setminus \{C\}]$ where F_{ini} is the initial formula F. (This point needs clarification because *EG-PQE* changes F by adding clauses.)

5.1 An Example

Before describing the pseudocode of *EG-PQE*, we explain how it solves the PQE problem of Example 1. That is, we consider taking clause C_1 out of $\exists X[F(X,Y)]$ where $F = C_1 \wedge \cdots \wedge C_4$, $C_1 = \overline{x}_3 \vee x_4$, $C_2 = y_1 \vee x_3$, $C_3 = y_1 \vee \overline{x}_4$, $C_4 = y_2 \vee x_4$ and $Y = \{y_1, y_2\}$ and $X = \{x_3, x_4\}$.

EG-PQE iteratively generates a full assignment $\vec{y}$ to Y and checks if $(C_1)_{\boldsymbol{y}}$ is redundant in $\exists X[F_{\boldsymbol{y}}]$ (i.e., if C_1 is redundant in $\exists X[F]$ in subspace $\vec{y}$). Note that if $(F \setminus \{C_1\})_{\boldsymbol{y}}$ *implies* $(C_1)_{\boldsymbol{y}}$, then $(C_1)_{\boldsymbol{y}}$ is trivially redundant in $\exists X[F_{\boldsymbol{y}}]$. To avoid such subspaces, *EG-PQE* generates $\vec{y}$ by searching for an assignment $(\vec{y},\vec{x})$ satisfying the formula $(F \setminus \{C_1\}) \wedge \overline{C}_1$. (Here $\vec{y}$ and $\vec{x}$ are full assignments to Y and X respectively.) If such $(\vec{y}, \vec{x})$ exists, it satisfies $F \setminus \{C_1\}$ and falsifies C_1 thus proving that $(F \setminus \{C_1\})_{\boldsymbol{y}}$ *does not* imply $(C_1)_{\boldsymbol{y}}$.

Assume that *EG-PQE* found an assignment$(y_1 = 0, y_2 = 1, x_3 = 1, x_4 = 0)$ satisfying $(F \setminus \{C_1\}) \wedge \overline{C}_1$. So $\vec{y} = (y_1 = 0, y_2 = 1)$. Then *EG-PQE* checks if $F_{\boldsymbol{y}}$ is satisfiable. $F_{\boldsymbol{y}} = (\overline{x}_3 \vee x_4) \wedge x_3 \wedge \overline{x}_4$ and so it is *unsatisfiable*. This means that $(C_1)_{\boldsymbol{y}}$ *is not* redundant in $\exists X[F_{\boldsymbol{y}}]$. (Indeed, $(F \setminus \{C_1\})_{\boldsymbol{y}}$ is satisfiable. So, removing C_1 makes F satisfiable in subspace $\vec{y}$.) *EG-PQE makes* $(C_1)_{\boldsymbol{y}}$ redundant in $\exists X[F_{\boldsymbol{y}}]$ by **adding** to F a clause B falsified by $\vec{y}$. The clause B equals y_1 and is obtained by identifying the assignments to individual variables of Y that made $F_{\boldsymbol{y}}$ unsatisfiable. (In our case, this is the assignment $y_1 = 0$.) Note that derivation of clause y_1 *generalizes* the proof of unsatisfiability of F in subspace $(y_1 = 0, y_2 = 1)$ so that this proof holds for subspace $(y_1 = 0, y_2 = 0)$ too.

Now *EG-PQE* looks for a new assignment satisfying $(F \setminus \{C_1\}) \wedge \overline{C}_1$. Let the assignment $(y_1 = 1, y_2 = 1, x_3 = 1, x_4 = 0)$ be found. So, $\vec{y} = (y_1 = 1, y_2 = 1)$. Since $(y_1 = 1, y_2 = 1, x_3 = 0)$ satisfies F, the formula $F_{\boldsymbol{y}}$ is satisfiable. So, $(C_1)_{\boldsymbol{y}}$

is *already redundant* in $\exists X[F_{\boldsymbol{y}}]$. To avoid re-visiting the subspace $\vec{y}$, *EG-PQE* generates the **plugging** clause $D = \overline{y}_1 \vee \overline{y}_2$ falsified by $\vec{y}$.

EG-PQE fails to generate a new assignment $\vec{y}$ because the formula $D \wedge (F \setminus \{C_1\}) \wedge \overline{C}_1$ is unsatisfiable. Indeed, every full assignment $\vec{y}$ we have examined so far falsifies either the clause y_1 added to F or the plugging clause D. The only assignment *EG-PQE* has not explored yet is $\vec{y} = (y_1 = 1, y_2 = 0)$. Since $(F \setminus \{C_1\})_{\boldsymbol{y}} = x_4$ and $(C_1)_{\boldsymbol{y}} = \overline{x}_3 \vee x_4$, the formula $(F \setminus \{C_1\}) \wedge \overline{C}_1$ is unsatisfiable in subspace $\vec{y}$. In other words, $(C_1)_{\boldsymbol{y}}$ is implied by $(F \setminus \{C_1\})_{\boldsymbol{y}}$ and hence is redundant. Thus, C_1 is redundant in $\exists X[F_{ini} \wedge y_1]$ for every assignment to Y where F_{ini} is the initial formula F. That is, $\exists X[F_{ini}] \equiv y_1 \wedge \exists X[F_{ini} \setminus \{C_1\}]$ and so the clause y_1 is a solution H to our PQE problem.

5.2 Description of *EG-PQE*

```
EG-PQE$(F, X, Y, C)$ {
1   $Plg := \emptyset$; $F_{ini} := F$
2   while ($true$) {
3     $G := F \setminus \{C\}$
4     $\vec{y} := Sat_1(Plg \wedge G \wedge \overline{C})$
5     if ($\vec{y} = nil$)
6       return($F \setminus F_{ini}$)
7     $(\vec{x}^*, B) := Sat_2(F, \vec{y})$
8     if ($B \neq nil$) {
9       $F := F \cup \{B\}$
10      continue }
11    $D := PlugCls(\vec{y}, \vec{x}^*, F)$
12    $Plg := Plg \cup \{D\}$}}
```

Fig. 1. Pseudocode of *EG-PQE*

The pseudo-code of *EG-PQE* is shown in Fig. 1. *EG-PQE* starts with storing the initial formula F and initializing formula Plg that accumulates the plugging clauses generated by *EG-PQE* (line 1). As we mentioned in the previous subsection, plugging clauses are used to avoid re-visiting the subspaces where the formula F is proved satisfiable.

All the work is carried out in a while loop. First, *EG-PQE* checks if there is a new subspace $\vec{y}$ where $\exists X[(F \setminus \{C\})_{\boldsymbol{y}}]$ does not imply $F_{\boldsymbol{y}}$. This is done by searching for an assignment $(\vec{y}, \vec{x})$ satisfying $Plg \wedge (F \setminus \{C\}) \wedge \overline{C}$ (lines 3–4). If such an assignment does not exist, the clause C is redundant in $\exists X[F]$. (Indeed, let $\vec{y}$ be a full assignment to Y. The formula $Plg \wedge (F \setminus \{C\}) \wedge \overline{C}$ is unsatisfiable in subspace $\vec{y}$ for one of the two reasons. First, $\vec{y}$ falsifies Plg. Then $C_{\boldsymbol{y}}$ is redundant because $F_{\boldsymbol{y}}$ is satisfiable. Second, $(F \setminus \{C\})_{\boldsymbol{y}} \wedge \overline{C_{\boldsymbol{y}}}$ is unsatisfiable. In this case, $(F \setminus \{C\})_{\boldsymbol{y}}$ implies $C_{\boldsymbol{y}}$.) Then *EG-PQE* returns the set of clauses added to the initial formula F as a solution H to the PQE problem (lines 5–6).

If the satisfying assignment $(\vec{y}, \vec{x})$ above exists, *EG-PQE* checks if the formula $F_{\boldsymbol{y}}$ is satisfiable (line 7). If not, then the clause $C_{\boldsymbol{y}}$ *is not* redundant in $\exists X[F_{\boldsymbol{y}}]$ (because $(F \setminus \{C\})_{\boldsymbol{y}}$ is satisfiable). So, *EG-PQE makes* $C_{\boldsymbol{y}}$ redundant by generating a clause $B(Y)$ falsified by $\vec{y}$ and adding it to F (line 9). Note that adding B also prevents *EG-PQE* from re-visiting the subspace $\vec{y}$ again. The clause B is built by finding an *unsatisfiable* subset of $F_{\boldsymbol{y}}$ and collecting the literals of Y removed from clauses of this subset when obtaining $F_{\boldsymbol{y}}$ from F.

If $F_{\boldsymbol{y}}$ is satisfiable, *EG-PQE* generates an assignment $\vec{x}^*$ to X such that $(\vec{y}, \vec{x}^*)$ satisfies F (line 7). The satisfiability of $F_{\boldsymbol{y}}$ means that every clause of $F_{\boldsymbol{y}}$ including $C_{\boldsymbol{y}}$ is redundant in $\exists X[F_{\boldsymbol{y}}]$. At this point, *EG-PQE* uses the longest clause $D(Y)$ falsified by $\vec{y}$ as a plugging clause (line 11). The clause D is added to *Plg* to avoid re-visiting subspace $\vec{y}$. Sometimes it is possible to remove variables from $\vec{y}$ to produce a shorter assignment $\vec{y}^*$ such that $(\vec{y}^*, \vec{x}^*)$ still satisfies F. Then one can use a shorter plugging clause D that is falsified by $\vec{y}^*$ and involves only the variables assigned in $\vec{y}^*$.

5.3 Discussion

EG-PQE is similar to the QE algorithm presented at CAV-2002 [12]. We will refer to it as *CAV02-QE*. Given a formula $\exists X[F(X,Y)]$, *CAV02-QE* enumerates full assignments to Y. In subspace $\vec{y}$, if $F_{\boldsymbol{y}}$ is unsatisfiable, *CAV02-QE* adds to F a clause falsified by $\vec{y}$. Otherwise, *CAV02-QE* generates a plugging clause D. (In [12], D is called "a blocking clause". This term can be confused with the term "blocked clause" specifying a completely different kind of a clause. So, we use the term "the plugging clause" instead.) To apply the idea of *CAV02-QE* to PQE, we reformulated it in terms of redundancy based reasoning.

The main flaw of *EG-PQE* inherited from *CAV02-QE* is the necessity to use plugging clauses produced from a satisfying assignment. Consider the PQE problem of taking a clause C out of $\exists X[F(X,Y)]$. If F is proved *unsatisfiable* in subspace $\vec{y}$, typically, only a small subset of clauses of $F_{\boldsymbol{y}}$ is involved in the proof. Then the clause generated by *EG-PQE* is short and thus proves C redundant in many subspaces different from $\vec{y}$. On the contrary, to prove F *satisfiable* in subspace $\vec{y}$, every clause of F must be satisfied. So, the plugging clause built off a satisfying assignment includes almost every variable of Y. Despite this flaw of *EG-PQE*, we present it for two reasons. First, it is a very simple SAT-based algorithm that can be easily implemented. Second, *EG-PQE* has a powerful advantage over *CAV02-QE* since it solves PQE rather than QE. Namely, *EG-PQE* does not need to examine the subspaces $\vec{y}$ where C is implied by $F \setminus \{C\}$. Surprisingly, for many formulas this allows *EG-PQE* to *completely avoid* examining subspaces where F is satisfiable. In this case, *EG-PQE* is very efficient and can solve very large problems. Note that when *CAV02-QE* performs complete QE on $\exists X[F]$, it *cannot* avoid subspaces $\vec{y}$ where $F_{\boldsymbol{y}}$ is satisfiable unless F *itself* is unsatisfiable (which is very rare in practical applications).

6 Introducing *EG-PQE*$^+$

In this section, we describe *EG-PQE*$^+$, an improved version of *EG-PQE*.

6.1 Main Idea

The pseudocode of *EG-PQE*$^+$ is shown in Fig. 2. It is different from that of *EG-PQE* only in line 11 marked with an asterisk. The motivation for this change is as follows. Line 11 describes proving redundancy of C for the case where $C_{\boldsymbol{y}}$ is not implied by $(F \setminus \{C\})_{\boldsymbol{y}}$ and $F_{\boldsymbol{y}}$ is satisfiable. Then *EG-PQE* simply uses a satisfying assignment as a proof of redundancy of C in subspace $\vec{y}$. This proof is unnecessarily strong because it proves that *every* clause of F (including C) is redundant in $\exists X[F]$ in subspace $\vec{y}$. Such a strong proof is hard to generalize to other subspaces.

```
EG-PQE$^+$($F, X, Y, C$) {
1   $Plg := \emptyset$; $F_{ini} := F$
2   while ($true$) {
........
11*   $D := PrvClsRed(\vec{y}, F, C)$
12    $Plg := Plg \cup \{D\}$}}
```

Fig. 2. Pseudocode of *EG-PQE*$^+$

The idea of *EG-PQE*$^+$ is to generate a proof for a much weaker proposition namely a proof of redundancy of C (and only C). Intuitively, such a proof should be easier to generalize. So, *EG-PQE*$^+$ calls a procedure *PrvClsRed* generating such a proof. *EG-PQE*$^+$ is a generic algorithm in the sense that *any* suitable procedure can be employed as *PrvClsRed*. In our current implementation, the procedure *DS-PQE* [1] is used as *PrvClsRed*. *DS-PQE* generates a proof stating that C is redundant in $\exists X[F]$ in subspace $\vec{y}^* \subseteq \vec{y}$. Then the plugging clause D falsified by $\vec{y}^*$ is generated. Importantly, $\vec{y}^*$ can be much shorter than $\vec{y}$. (A brief description of *DS-PQE* in the context of *EG-PQE*$^+$ is given in [5].)

Example 3. Consider the example solved in Subsect. 5.1. That is, we consider taking clause C_1 out of $\exists X[F(X,Y)]$ where $F = C_1 \wedge \cdots \wedge C_4$, $C_1 = \overline{x}_3 \vee x_4$, $C_2 = y_1 \vee x_3$, $C_3 = y_1 \vee \overline{x}_4$, $C_4 = y_2 \vee x_4$ and $Y = \{y_1, y_2\}$ and $X = \{x_3, x_4\}$. Consider the step where *EG-PQE* proves redundancy of C_1 in subspace $\vec{y} = (y_1 = 1, y_2 = 1)$. *EG-PQE* shows that $(y_1 = 1, y_2 = 1, x_3 = 0)$ satisfies F, thus proving every clause of F (including C_1) redundant in $\exists X[F]$ in subspace $\vec{y}$. Then *EG-PQE* generates the plugging clause $D = \overline{y}_1 \vee \overline{y}_2$ falsified by $\vec{y}$.

In contrast to *EG-PQE*, *EG-PQE*$^+$ calls *PrvClsRed* to produce a proof of redundancy for the clause C_1 alone. Note that F has no clauses resolvable with C_1 on x_3 in subspace $\vec{y}^* = (y_1 = 1)$. (The clause C_2 containing x_3 is satisfied by $\vec{y}^*$.) This means that C_1 is blocked in subspace $\vec{y}^*$ and hence redundant there (see Proposition 2). Since $\vec{y}^* \subset \vec{y}$, *EG-PQE*$^+$ produces a more general proof of redundancy than *EG-PQE*. To avoid re-examining the subspace $\vec{y}^*$, *EG-PQE*$^+$ generates a *shorter* plugging clause $D = \overline{y}_1$.

6.2 Discussion

Consider the PQE problem of taking a clause C out of $\exists X[F(X,Y)]$. There are two features of PQE that make it easier than QE. The first feature mentioned earlier is that one can ignore the subspaces $\vec{y}$ where $F \setminus \{C\}$ implies C. The second feature is that when $F_{\boldsymbol{y}}$ is satisfiable, one only needs to prove redundancy of the clause C alone. Among the three algorithms we run in experiments, namely, *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ only the latter exploits both features. (In addition to using *DS-PQE* inside *EG-PQE*$^+$ we also run it as a stand-alone PQE solver.) *DS-PQE* does not use the first feature [1] and *EG-PQE* does not exploit the second one. As we show in Sects. 7 and 8, this affects the performance of *DS-PQE* and *EG-PQE*.

7 Experiment with FIFO Buffers

In this and the next two sections we describe some experiments with *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ (their sources are available at [13,14] and [15] respectively). We used Minisat2.0 [16] as an internal SAT-solver. The experiments were run on a computer with Intel Core i5-8265U CPU of 1.6 GHz.

```
        ...
if (write == 1 && currSize < n)
* if (dataIn != Val)
      begin
      Data[wrPnt] = dataIn;
      wrPnt = wrPnt + 1;
      end
        ...
```

Fig. 3. A buggy fragment of Verilog code describing *Fifo*

In this section, we give an example of bug detection by invariant generation for a FIFO buffer. Our objective here is threefold. First, we want to give an example of a bug that can be overlooked by testing and guessing the unwanted properties to check (see Subsect. 7.3). Second, we want to substantiate the intuition of Subsect. 3.4 that property generation by PQE (in our case, invariant generation by PQE) has the same reasons to be effective as testing. In particular, by taking out different clauses one generates invariants relating to different parts of the design. So, taking out a clause of the buggy part is likely to produce an unwanted invariant. Third, we want to give an example of an invariant that can be easily identified as unwanted[2].

7.1 Buffer Description

Consider a FIFO buffer that we will refer to as *Fifo*. Let n be the number of elements of *Fifo* and *Data* denote the data buffer of *Fifo*. Let each $\mathit{Data}[i], i = 1,\dots,n$ have p bits and be an integer where $0 \le \mathit{Data}[i] < 2^p$. A fragment of the Verilog code describing *Fifo* is shown in Fig. 3. This fragment has a buggy line marked with an asterisk. In the correct version without the marked line, a new element *dataIn* is added to *Data* if the *write* flag is on and *Fifo* has less than n elements. Since *Data* can have any combination of numbers, all *Data* states are supposed to be reachable. However, due to the bug, the number *Val* cannot appear in *Data*. (Here *Val* is some constant $0 < \mathit{Val} < 2^p$. We assume that the buffer elements are initialized to 0.) So, *Fifo* has an *op-state reachability bug* since it cannot reach operative states where an element of *Data* equals *Val*.

[2] Let $P(\hat{S})$ be an invariant for a circuit N depending only on a subset $\hat{S}$ of the state variables S. Identifying P as an unwanted invariant is much easier if $\hat{S}$ is meaningful from the high-level view of the design. Suppose, for instance, that assignments to $\hat{S}$ specify values of a high-level variable v. Then P is unwanted if it claims unreachability of a value of v that is supposed to be reachable. Another simple example is that assignments to $\hat{S}$ specify values of high-level variables v and w that are supposed to be *independent*. Then P is unwanted if it claims that some combinations of values of v and w are unreachable. (This may mean, for instance, that an assignment operator setting the value of v erroneously involves the variable w.)

7.2 Bug Detection by Invariant Generation

Let N be a circuit implementing *Fifo*. Let S be the set of state variables of N and $\boldsymbol{S_{data}} \subset \boldsymbol{S}$ be the subset corresponding to the data buffer *Data*. We used *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ to generate invariants of N as described in Sect. 4. Note that an invariant Q depending only on S_{data} is an **unwanted** one. If Q holds for N, some states of *Data* are unreachable. Then *Fifo* has an op-state reachability bug since every state of *Data* is supposed to be reachable. To generate invariants, we used the formula $F_k = I(S_0) \wedge T(S_0, V_0, S_1) \wedge \cdots \wedge T(S_{k-1}, V_{k-1}, S_k)$ introduced in Subsect. 4.2. Here I and T describe the initial state and the transition relation of N respectively and S_j and V_j denote state variables and combinational input variables of j-th time frame respectively. First, we used a PQE solver to generate a local invariant $H(S_k)$ obtained by taking a clause C out of $\exists X_k[F_k]$ where $X_k = S_0 \cup V_0 \cup \cdots \cup S_{k-1} \cup V_{k-1}$. So, $\exists X_k[F_k] \equiv H \wedge \exists X_k[F_k \setminus \{C\}]$. (Since $F_k \Rightarrow H$, no state falsifying H can be reached in k transitions.) In the experiment, we took out only clauses of F_k containing an *unquantified variable*, i.e., a state variable of the k-th time frame. The time limit for solving the PQE problem of taking out a clause was set to 10 s.

Table 1. FIFO buffer with n elements of 32 bits. Time limit is 10 s per PQE problem

buff. size n	latches	time frames	total *pqe* probs			finished *pqe* probs			unwant. invar			runtime (s.)		
			ds-pqe	*eg-pqe*	*eg-pqe*$^+$	*ds-pqe*	*eg-pqe*	*eg-pqe*$^+$	*ds-pqe*	*eg-pqe*	*eg-pqe*$^+$	*ds-pqe*	*eg-pqe*	*eg-pqe*$^+$
8	300	5	1,236	311	**8**	2%	**36%**	35%	no	**yes**	**yes**	12,141	2,138	**52**
8	300	10	560	737	**39**	2%	1%	**3%**	**yes**	**yes**	**yes**	5,551	7,681	**380**
16	560	5	2,288	2,288	**16**	1%	65%	**71%**	no	no	**yes**	22,612	9,506	**50**
16	560	10	653	2,288	**24**	1%	36%	**38%**	**yes**	no	**yes**	6,541	16,554	**153**

For each clause Q of every local invariant H generated by PQE, we checked if Q was a global invariant. Namely, we used a public version of *IC3* [17,18] to verify if the property Q held (by showing that no reachable state of N falsified Q). If so, and Q depended only on variables of S_{data}, N had an *unwanted invariant*. Then we stopped invariant generation. The results of the experiment for buffers with 32-bit elements are given in Table 1. When picking a clause to take out, i.e., a clause with a state variable of k-th time frame, one could make a good choice by pure luck. To address this issue, we picked clauses to take out *randomly* and performed 10 different runs of invariant generation and then computed the average value. So, the columns four to twelve of Table 1 actually give the average value of 10 runs.

Let us use the first line of Table 1 to explain its structure. The first two columns show the number of elements in *Fifo* implemented by N and the number of latches in N (8 and 300). The third column gives the number k of time frames (i.e., 5). The next three columns show the total number of PQE problems solved by a PQE solver before an unwanted invariant was generated. For instance, *EG-PQE*$^+$ found such an invariant after solving 8 problems. On the other hand, *DS-PQE* failed to find an unwanted invariant and had to solve *all* 1,236 PQE problems of taking out a clause of F_k with an unquantified variable. The following three columns show the share of PQE problems *finished* in the time limit of 10 s. For instance, *EG-PQE* finished 36% of 311 problems. The next three columns show if an unwanted invariant was generated by a PQE solver. (*EG-PQE* and *EG-PQE*$^+$ found one whereas *DS-PQE* did not.) The last three columns give the total run time. Table 1 shows that only *EG-PQE*$^+$ managed to generate an unwanted invariant for all four instances of *Fifo*. This invariant asserted that *Fifo* cannot reach a state where an element of *Data* equals *Val*.

7.3 Detection of the Bug by Conventional Methods

The bug above (or its modified version) can be overlooked by conventional methods. Consider, for instance, testing. It is hard to detect this bug by *random* tests because it is exposed only if one tries to add *Val* to *Fifo*. The same applies to testing using the *line coverage* metric [19]. On the other hand, a test set with 100% *branch* coverage [19] will find this bug. (To invoke the *else* branch of the *if* statement marked with '*' in Fig. 3, one must set *dataIn* to *Val*.) However, a slightly modified bug can be missed even by tests with 100% branch coverage [5].

Now consider, manual generation of unwanted properties. It is virtually impossible to guess an unwanted *invariant* of *Fifo* exposing this bug unless one knows exactly what this bug is. However, one can detect this bug by checking a property asserting that the element *dataIn* must appear in the buffer if *Fifo* is ready to accept it. Note that this is a *non-invariant* property involving states of different time frames. The more time frames are used in such a property the more guesswork is required to pick it. Let us consider a modified bug. Suppose *Fifo* does not reject the element *Val*. So, the non-invariant property above holds. However, if *dataIn* == *Val*, then *Fifo* changes the *previous* accepted element if that element was *Val* too. So, *Fifo* cannot have two consecutive elements *Val*. Our method will detect this bug via generating an unwanted invariant falsified by states with consecutive elements *Val*. One can also identify this bug by checking a property involving two consecutive elements of *Fifo*. But picking it requires a lot of guesswork and so the modified bug can be easily overlooked.

8 Experiments with HWMCC Benchmarks

In this section, we describe three experiments with 98 multi-property benchmarks of the HWMCC-13 set [20]. (We use this set because it has a multi-property track, see the explanation below.) The number of latches in those benchmarks range from 111 to 8,000. More details about the choice of benchmarks and the experiments can be found in [5]. Each benchmark consists of a sequential circuit N and invariants $P_0, \dots, P_m$ to prove. Like in Sect. 4, we call $P_{agg} = P_0 \wedge \dots \wedge P_m$ the *aggregate invariant*. In experiments 2 and 3 we used PQE to generate new invariants of N. Since every invariant P implied by P_{agg}

is a desired one, the necessary condition for P to be *unwanted* is $P_{agg} \not\Rightarrow P$. The conjunction of many invariants P_i produces a stronger invariant P_{agg}, which makes it *harder* to generate P not implied by P_{agg}. (This is the reason for using multi-property benchmarks in our experiments.) The circuits of the HWMCC-13 set are *anonymous*, so, we could not know if an unreachable state is supposed to be reachable. For that reason, we just generated invariants not implied by P_{agg} without deciding if some of them were unwanted.

Similarly to the experiment of Sect. 7, we used the formula $F_k = I(S_0) \wedge T(S_0, V_0, S_1) \wedge \dots \wedge T(S_{k-1}, V_{k-1}, S_k)$ to generate invariants. The number k of time frames was in the range of $2 \leq k \leq 10$. As in the experiment of Sect. 7, we took out only clauses containing a state variable of the k-th time frame. In all experiments, the **time limit** for solving a PQE problem was set to 10 s.

8.1 Experiment 1

In the first experiment, we generated a *local invariant* H by taking out a clause C of $\exists X_k[F_k]$ where $X_k = S_0 \cup V_0 \cup \dots \cup S_{k-1} \cup V_{k-1}$. The formula H asserts that no state falsifying H can be reached in k transitions. Our goal was to show that PQE can find H for large formulas F_k that have hundreds of thousands of clauses. We used *EG-PQE* to partition the PQE problems we tried into two groups. *The first group* consisted of 3,736 problems for which we ran *EG-PQE* with the time limit of 10 s and it never encountered a subspace $\vec{s_k}$ where F_k was satisfiable. Here $\vec{s_k}$ is a full assignment to S_k. Recall that only the variables S_k are unquantified in $\exists X_k[F_k]$. So, in every subspace $\vec{s_k}$, formula F_k was either unsatisfiable or $(F_k \setminus \{C\}) \Rightarrow C$. (The fact that so many problems meet the condition of the first group came as a big surprise.) *The second group* consisted of 3,094 problems where *EG-PQE* encountered subspaces where F_k was satisfiable.

For the first group, *DS-PQE* finished only 30% of the problems within 10 s whereas *EG-PQE* and *EG-PQE*$^+$ finished 88% and 89% respectively. The poor performance of *DS-PQE* is due to not checking if $(F_k \setminus \{C\}) \Rightarrow C$ in the current subspace. For the second group, *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ finished 15%, 2% and 27% of the problems respectively within 10 s. *EG-PQE* finished far fewer problems because it used a satisfying assignment as a proof of redundancy of C (see Subsect. 6.2).

To contrast PQE and QE, we employed a high-quality tool *CADET* [21,22] to perform QE on the 98 formulas $\exists X_k[F_k]$ (one formula per benchmark). That is, instead of taking a clause out of $\exists X_k[F_k]$ by PQE, we applied *CADET* to perform full QE on this formula. (Performing QE on $\exists X_k[F_k]$ produces a formula $H(S_k)$ specifying *all* states unreachable in k transitions.) *CADET* finished only 25% of the 98 QE problems with the time limit of 600 s. On the other hand, *EG-PQE*$^+$ finished 60% of the 6,830 problems of both groups (generated off $\exists X_k[F_k]$) within 10 s. So, PQE can be much easier than QE if only a small part of the formula gets unquantified.

8.2 Experiment 2

The second experiment was an extension of the first one. Its goal was to show that PQE can generate invariants for realistic designs. For each clause Q of a local invariant H generated by PQE we used *IC3* to verify if Q was a global invariant. If so, we checked if $P_{agg} \not\Rightarrow Q$ held. To make the experiment less time consuming, in addition to the time limit of 10 s per PQE problem we imposed a few more constraints. The PQE problem of taking a clause out of $\exists X_k[F_k]$ terminated as soon as H accumulated 5 clauses or more. Besides, processing a benchmark aborted when the summary number of clauses of all formulas H generated for this benchmark reached 100 or the total run time of all PQE problems generated off $\exists X_k[F_k]$ exceeded 2,000 s.

Table 2. Invariant generation

pqe solver	#bench marks	results		
		local invar.	glob invar.	not imp by P_{agg}
ds-pqe	98	5,556	2,678	2,309
eg-pqe	98	**9,498**	**4,839**	**4,009**
eg-pqe$^+$	98	9,303	4,773	3,940

Table 2 shows the results of the experiment. The third column gives the number of local single-clause invariants (i.e., the total number of clauses in all H over all benchmarks). The fourth column shows how many local single-clause invariants turned out to be global. (Since global invariants were extracted from H and the summary size of all H could not exceed 100, the number of global invariants per benchmark could not exceed 100.) The last column gives the number of global invariants not implied by P_{agg}. So, these invariants are candidates for checking if they are unwanted. Table 2 shows that *EG-PQE* and *EG-PQE*$^+$ performed much better than *DS-PQE*.

8.3 Experiment 3

To prove an invariant P true, *IC3* conjoins it with clauses $Q_1,\dots,Q_n$ to make $P \wedge Q_1 \wedge \dots \wedge Q_n$ inductive. If *IC3* succeeds, every Q_i is an invariant. Moreover, Q_i may be an *unwanted* invariant. The goal of the third experiment was to demonstrate that PQE and *IC3*, in general, produce different invariant clauses. The intuition here is twofold. First, *IC3* generates clauses Q_i to prove a *predefined* invariant rather than find an unwanted one. Second, the closer P to being inductive, the fewer new invariant clauses are generated by *IC3*. Consider the circuit N_{triv} that simply stays in the initial state $\vec{s}_{ini}$ (Sect. 4). Any invariant satisfied by $\vec{s}_{ini}$ is already *inductive* for N_{triv}. So, IC3 will not generate *a single new invariant* clause. On the other hand, if the correct circuit is supposed to leave the initial state, N_{triv} has unwanted invariants that our method will find.

In this experiment, we used *IC3* to generate P^*_{agg}, an *inductive* version of P_{agg}. The experiment showed that in 88% cases, an invariant clause generated by *EG-PQE*$^+$ and not implied by P_{agg} was not implied by P^*_{agg} either. (More details about this experiment can be found in [5].)

9 Properties Mimicking Symbolic Simulation

Let $M(X,V,W)$ be a combinational circuit where X,V,W are internal, input and output variables. In this section, we describe generation of properties of M that mimic symbolic simulation [23]. Every such a property $Q(V)$ specifies a cube of tests that produce the same values for a given subset of variables of W. We chose generation of such properties because deciding if Q is an unwanted property is, in general, simple. The procedure for generation of these properties is slightly different from the one presented in Sect. 3.

Let $F(X,V,W)$ be a formula specifying M. Let $B(W)$ be a clause. Let $H(V)$ be a solution to the PQE problem of taking a clause $C \in F$ out of $\exists X \exists W[F \wedge B]$. That is, $\exists X \exists W[F \wedge B] \equiv H \wedge \exists X \exists W[(F \setminus \{C\}) \wedge B]$. Let $Q(V)$ be a clause of H. Then M has the **property** that for every full assignment $\vec{v}$ to V falsifying Q, it produces an output $\vec{w}$ falsifying B (a proof of this fact can be found in [5]). Suppose, for instance, $Q = v_1 \vee \overline{v}_{10} \vee v_{30}$ and $B = w_2 \vee \overline{w}_{40}$. Then for every $\vec{v}$ where $v_1 = 0, v_{10} = 1, v_{30} = 0$, the circuit M produces an output where $w_2 = 0, w_{40} = 1$. Note that Q is implied by $F \wedge B$ rather than F. So, it is a property of M under constraint B rather than M alone. The property Q is **unwanted** if there is an input falsifying Q that *should not* produce an output falsifying B.

To generate combinational circuits, we unfolded sequential circuits of the set of 98 benchmarks used in Sect. 8 for invariant generation. Let N be a sequential circuit. (We reuse the notation of Sect. 4). Let $M_k(S_0,V_0,\dots,S_{k-1},V_{k-1},S_k)$ denote the combinational circuit obtained by unfolding N for k time frames. Here S_j, V_j are state and input variables of j-th time frame respectively. Let F_k denote the formula $I(S_0) \wedge T(S_0,V_0,S_1) \wedge \dots \wedge T(S_{k-1},V_{k-1},S_k)$ describing the unfolding of N for k time frames. Note that F_k specifies the circuit M_k above under the input constraint $I(S_0)$. Let $B(S_k)$ be a clause. Let $H(S_0,V_0,\dots,V_{k-1})$ be a solution to the PQE problem of taking a clause $C \in F_k$ out of formula $\exists S_{1,k}[F_k \wedge B]$. Here $S_{1,k} = S_1 \cup \dots \cup S_k$. That is, $\exists S_{1,k}[F_k \wedge B] \equiv H \wedge \exists S_{1,k}[(F_k \setminus \{C\}) \wedge B]$. Let Q be a clause of H. Then for every assignment $(\vec{s}_{ini},\vec{v}_0,\dots,\vec{v}_{k-1})$ falsifying Q, the circuit M_k outputs $\vec{s}_k$ falsifying B. (Here $\vec{s}_{ini}$ is the initial state of N and $\vec{s}_k$ is a state of the last time frame.)

Table 3. Property generation for combinational circuits

name	latches	time frames	size of B	subc. M'_k		results			
				gates	inp vars	min	max	time (s.)	3-val sim.
6s326	3,342	20	15	348,479	1,774	27	28	2.9	**no**
6s40m	5,608	20	15	406,474	3,450	27	29	1.1	**no**
6s250	6,185	20	15	556,562	2,456	50	54	0.8	**no**
6s395	463	30	15	36,088	569	24	26	0.7	yes
6s339	1,594	30	15	179,543	3,978	70	71	3.1	**no**
6s292	3,190	30	15	154,014	978	86	89	1.1	**no**
6s143	260	40	15	551,019	16,689	526	530	2.5	yes
6s372	1,124	40	15	295,626	2,766	513	518	1.7	**no**
6s335	1,658	40	15	207,787	2,863	120	124	6.7	**no**
6s391	2,686	40	15	240,825	7,579	340	341	8.9	**no**

In the experiment, we used *DS-PQE*,*EG-PQE* and *EG-PQE*$^+$ to solve 1,586 PQE problems described above. In Table 3, we give a sample of results by *EG-PQE*$^+$. (More details about this experiment can be found in [5].) Below, we use the first line of Table 3 to explain its structure. The first column gives the benchmark name (6s326).

The next column shows that 6s326 has 3,342 latches. The third column gives the number of time frames used to produce a combinational circuit M_k (here $k = 20$). The next column shows that the clause B introduced above consisted of 15 literals of variables from S_k. (Here and below we still use the index k assuming that $k = 20$.) The literals of B were generated *randomly*. When picking the length of B we just tried to simulate the situation where one wants to set a particular *subset* of output variables of M_k to specified values. The next two columns give the size of the subcircuit M'_k of M_k that feeds the output variables present in B. When computing a property H we took a clause out of formula $\exists S_{1,k}[F'_k \wedge B]$ where F'_k specifies M'_k instead of formula $\exists S_{1,k}[F_k \wedge B]$ where F_k specifies M_k. (The logic of M_k not feeding a variable of B is irrelevant for computing H.) The first column of the pair gives the number of gates in M'_k (i.e., 348,479). The second column provides the number of input variables feeding M'_k (i.e., 1,774). Here we count only variables of $V_0 \cup \cdots \cup V_{k-1}$ and ignore those of S_0 since the latter are already assigned values specifying the initial state $\vec{s}_{ini}$ of N.

The next four columns show the results of taking a clause out of $\exists S_{1,k}[F'_k \wedge B]$. For each PQE problem the time limit was set to 10 s. Besides, *EG-PQE*$^+$ terminated as soon as 5 clauses of property $H(S_0, V_0, \ldots, V_{k-1})$ were generated. The first three columns out of four describe the minimum and maximum sizes of clauses in H and the run time of *EG-PQE*$^+$. So, it took for *EG-PQE*$^+$ 2.9 s. to produce a formula H containing clauses of sizes from 27 to 28 variables. A clause Q of H with 27 variables, for instance, specifies 2^{1747} tests falsifying Q that produce the same output of M'_k (falsifying the clause B). Here $1747 = 1774 - 27$ is the number of input variables of M'_k not present in Q. The last column shows that at least one clause Q of H specifies a property that cannot be produced by 3-valued simulation (a version of symbolic simulation [23]). To prove this, one just needs to set the input variables of M'_k present in Q to the values falsifying Q and run 3-valued simulation. (The remaining input variables of M'_k are assigned a don't-care value.) If after 3-valued simulation some output variable of M'_k is assigned a don't-care value, the property specified by Q cannot be produced by 3-valued simulation.

Running *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ on the 1,586 PQE problems mentioned above showed that a) *EG-PQE* performed poorly producing properties only for 28% of problems; b) *DS-PQE* and *EG-PQE*$^+$ showed much better results by generating properties for 62% and 66% of problems respectively. When *DS-PQE* and *EG-PQE*$^+$ succeeded in producing properties, the latter could not be obtained by 3-valued simulation in 74% and 78% of cases respectively.

10 Some Background

In this section, we discuss some research relevant to PQE and property generation. Information on BDD based QE can be found in [24,25]. SAT based QE is described in [12,21,26–32]. Our first PQE solver called *DS-PQE* was introduced in [1]. It was based on redundancy based reasoning presented in [33] in terms of variables and in [34] in terms of clauses. The main flaw of *DS-PQE* is as follows.

Consider taking a clause C out of $\exists X[F]$. Suppose *DS-PQE* proved C redundant in a subspace where F is *satisfiable* and some *quantified* variables are assigned. The problem is that *DS-PQE* cannot simply assume that C is redundant every time it re-enters this subspace [35]. The root of the problem is that redundancy is a *structural* rather than semantic property. That is, redundancy of a clause in a formula ξ (quantified or not) does not imply such redundancy in every formula logically equivalent to ξ. Since our current implementation of *EG-PQE*$^+$ uses *DS-PQE* as a subroutine, it has the same learning problem. We showed in [36] that this problem can be addressed by the machinery of certificate clauses. So, the performance of PQE can be drastically improved via enhanced learning in subspaces where F is satisfiable.

We are unaware of research on property generation for combinational circuits. As for invariants, the existing procedures typically generate some auxiliary *desired* invariants to prove a predefined property (whereas our goal is to generate invariants that are *unwanted*). For instance, they generate loop invariants [37] or invariants relating internal points of circuits checked for equivalence [38]. Another example of auxiliary invariants are clauses generated by *IC3* to make an invariant inductive [17]. As we showed in Subsect. 8.3, the invariants produced by PQE are, in general, different from those built by *IC3*.

11 Conclusions and Directions for Future Research

We consider Partial Quantifier Elimination (PQE) on propositional CNF formulas with existential quantifiers. In contrast to *complete* quantifier elimination, PQE allows to unquantify a *part* of the formula. We show that PQE can be used to generate properties of combinational and sequential circuits. The goal of property generation is to check if a design has an *unwanted* property and thus is buggy. We used PQE to generate an unwanted invariant for a FIFO buffer exposing a non-trivial bug. We also applied PQE to invariant generation for HWMCC benchmarks. Finally, we used PQE to generate properties of combinational circuits mimicking symbolic simulation. Our experiments show that PQE can efficiently generate properties for realistic designs.

There are at least three directions for future research. The first direction is to improve the performance of PQE solving. As we mentioned in Sect. 10, the most promising idea here is to enhance the power of learning in subspaces where the formula is satisfiable. The second direction is to use the improved PQE solvers to design new, more efficient algorithms for well-known problems like SAT, model checking and equivalence checking. The third direction is to look for new problems that can be solved by PQE.

References

1. Goldberg, E., Manolios, P.: Partial quantifier elimination. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 148–164. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_12

2. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symbolic Logic **22**(3), 269–285 (1957)
3. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
4. Goldberg, E.: Property checking by logic relaxation, Technical report arXiv:1601.02742 [cs.LO] (2016)
5. Goldberg, E.: Partial quantifier elimination and property generation, Technical report arXiv:2303.13811 [cs.LO] (2023)
6. Goldberg, E., Manolios, P.: Software for quantifier elimination in propositional logic. In: ICMS-2014, Seoul, South Korea, 5–9 August 2014, pp. 291–294 (2014)
7. Goldberg, E.: Equivalence checking by logic relaxation. In: FMCAD-2016, pp. 49–56 (2016)
8. Goldberg, E.: Property checking without inductive invariant generation, Technical report arXiv:1602.05829 [cs.LO] (2016)
9. B. L. Synthesis and V. Group: ABC: a system for sequential synthesis and verification (2017). www.eecs.berkeley.edu/~alanmi/abc
10. Kullmann, O.: New methods for 3-SAT decision and worst-case analysis. Theor. Comput. Sci. **223**(1–2), 1–72 (1999)
11. Tseitin, G.: On the complexity of derivation in the propositional calculus. Zapiski nauchnykh seminarov LOMI, vol. 8, pp. 234–259 (1968). English translation of this volume: Consultants Bureau, N.Y., pp. 115–125 (1970)
12. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_19
13. The source of *DS-PQE*. http://eigold.tripod.com/software/ds-pqe.tar.gz
14. The source of *EG-PQE*. http://eigold.tripod.com/software/eg-pqe.1.0.tar.gz
15. The source of *EG-PQE*$^+$. http://eigold.tripod.com/software/eg-pqe-pl.1.0.tar.gz
16. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT, Santa Margherita Ligure, Italy, pp. 502–518 (2003)
17. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
18. An implementation of IC3 by A. Bradley. https://github.com/arbrad/IC3ref
19. Aniche, M.: Effective Software Testing: A Developer's Guide. Manning Publications (2022)
20. HardWare Model Checking Competition (HWMCC-2013) (2013). http://fmv.jku.at/hwmcc13/
21. Rabe, M.N.: Incremental determinization for quantifier elimination and functional synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 84–94. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_6
22. CADET. http://github.com/MarkusRabe/cadet
23. Bryant, R.: Symbolic simulation–techniques and applications. In: DAC-1990, pp. 517–521 (1990)
24. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
25. Chauhan, P., Clarke, E., Jha, S., Kukula, J., Veith, H., Wang, D.: Using combinatorial optimization methods for quantification scheduling. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 293–309. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44798-9_24

26. Jin, H., Somenzi, F.: Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In: DAC 2005, pp. 750–753 (2005)
27. Ganai, M., Gupta, A., Ashar, P.: Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In: ICCAD-2004, pp. 510–517 (2004)
28. Jiang, J.-H.R.: Quantifier elimination via functional composition. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 383–397. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_30
29. Brauer, J., King, A., Kriener, J.: Existential quantification as incremental SAT. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_17
30. Klieber, W., Janota, M., Marques-Silva, J., Clarke, E.: Solving QBF with free variables. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 415–431. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_33
31. Bjorner, N., Janota, M., Klieber, W.: On conflicts and strategies in QBF. In: LPAR (2015)
32. Bjorner, N., Janota, M.: Playing with quantified satisfaction. In: LPAR (2015)
33. Goldberg, E., Manolios, P.: Quantifier elimination by dependency sequents. In: FMCAD-2012, pp. 34–44 (2012)
34. Goldberg, E., Manolios, P.: Quantifier elimination via clause redundancy. In: FMCAD 2013, pp. 85–92 (2013)
35. Goldberg, E.: Quantifier elimination with structural learning, Technical report arXiv: 1810.00160 [cs.LO] (2018)
36. Goldberg, E.: Partial quantifier elimination by certificate clauses, Technical report arXiv:2003.09667 [cs.LO] (2020)
37. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference, vol. 48, pp. 443–456, October 2013
38. Baumgartner, J., Mony, H., Case, M., Sawada, J., Yorav, K.: Scalable conditional equivalence checking: an automated invariant-generation based approach. In: Formal Methods in Computer-Aided Design, pp. 120–127 (2009)

Rounding Meets Approximate Model Counting

Jiong Yang(✉) and Kuldeep S. Meel

National University of Singapore, Singapore, Singapore
jiong@comp.nus.edu.sg

Abstract. The problem of model counting, also known as #SAT, is to compute the number of models or satisfying assignments of a given Boolean formula F. Model counting is a fundamental problem in computer science with a wide range of applications. In recent years, there has been a growing interest in using hashing-based techniques for approximate model counting that provide (ε, δ)-guarantees: i.e., the count returned is within a $(1+\varepsilon)$-factor of the exact count with confidence at least $1-\delta$. While hashing-based techniques attain reasonable scalability for large enough values of δ, their scalability is severely impacted for smaller values of δ, thereby preventing their adoption in application domains that require estimates with high confidence.

The primary contribution of this paper is to address the Achilles heel of hashing-based techniques: we propose a novel approach based on *rounding* that allows us to achieve a significant reduction in runtime for smaller values of δ. The resulting counter, called ApproxMC6 (The resulting tool ApproxMC6 is available open-source at https://github.com/meelgroup/approxmc), achieves a substantial runtime performance improvement over the current state-of-the-art counter, ApproxMC. In particular, our extensive evaluation over a benchmark suite consisting of 1890 instances shows ApproxMC6 solves 204 more instances than ApproxMC, and achieves a 4× speedup over ApproxMC.

1 Introduction

Given a Boolean formula F, the problem of model counting is to compute the number of models of F. Model counting is a fundamental problem in computer science with a wide range of applications, such as control improvisation [13], network reliability [9,28], neural network verification [2], probabilistic reasoning [5,11,20,21], and the like. In addition to myriad applications, the problem of model counting is a fundamental problem in theoretical computer science. In his seminal paper, Valiant showed that #SAT is #P-complete, where #P is the set of counting problems whose decision versions lie in NP [28]. Subsequently, Toda demonstrated the theoretical hardness of the problem by showing that every problem in the entire polynomial hierarchy can be solved by just one call to a #P oracle; more formally, $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$ [27].

Given the computational intractability of #SAT, there has been sustained interest in the development of approximate techniques from theoreticians and

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 132–162, 2023.
https://doi.org/10.1007/978-3-031-37703-7_7

practitioners alike. Stockmeyer introduced a randomized hashing-based technique that provides (ε, δ)-guarantees (formally defined in Sect. 2) given access to an NP oracle [25]. Given the lack of practical solvers that could handle problems in NP satisfactorily, there were no practical implementations of Stockmeyere's hashing-based techniques until the 2000s [14]. Building on the unprecedented advancements in the development of SAT solvers, Chakraborty, Meel, and Vardi extended Stockmeyer's framework to a scalable (ε, δ)-counting algorithm, ApproxMC [7]. The subsequent years have witnessed a sustained interest in further optimizations of the hashing-based techniques for approximate counting [5,6,10,11,17–19,23,29,30]. The current state-of-the-art technique for approximate counting is a hashing-based framework called ApproxMC, which is in its fourth version, called ApproxMC4 [22,24].

The core theoretical idea behind the hashing-based framework is to use 2-universal hash functions to partition the solution space, denoted by $\mathsf{sol(F)}$ for a formula F, into *roughly equal small* cells, wherein a cell is considered *small* if it contains solutions less than or equal to a pre-computed threshold, thresh. An NP oracle (in practice, a SAT solver) is employed to check if a cell is small by enumerating solutions one-by-one until either there are no more solutions or we have already enumerated $\mathsf{thresh} + 1$ solutions. Then, we randomly pick a cell, enumerate solutions within the cell (if the cell is small), and scale the obtained count by the number of cells to obtain an estimate for $|\mathsf{sol(F)}|$. To amplify the confidence, we rely on the standard *median technique*: repeat the above process, called ApproxMCCore, multiple times and return the median. Computing the median amplifies the confidence as for the median of t repetitions to be outside the desired range (i.e., $\left[\frac{|\mathsf{sol(F)}|}{1+\varepsilon}, (1+\varepsilon)|\mathsf{sol(F)}|\right]$), it should be the case that at least half of the repetitions of ApproxMCCore returned a wrong estimate.

In practice, every subsequent repetition of ApproxMCCore takes a similar time, and the overall runtime increases linearly with the number of invocations. The number of repetitions depends logarithmically on δ^{-1}. As a particular example, for $\epsilon = 0.8$, the number of repetitions of ApproxMCCore to attain $\delta = 0.1$ is 21, which increases to 117 for $\delta = 0.001$: a significant increase in the number of repetitions (and accordingly, the time taken). Accordingly, it is no surprise that empirical analysis of tools such as ApproxMC has been presented with a high delta (such as $\delta = 0.1$). On the other hand, for several applications, such as network reliability, and quantitative verification, the end users desire estimates with high confidence. Therefore, the design of efficient counting techniques for small δ is a major challenge that one needs to address to enable the adoption of approximate counting techniques in practice.

The primary contribution of our work is to address the above challenge. We introduce a new technique called *rounding* that enables dramatic reductions in the number of repetitions required to attain a desired value of confidence. The core technical idea behind the design of the *rounding* technique is based on the following observation: Let L (resp. U) refer to the event that a given invocation of ApproxMCCore under (resp. over)-estimates $|\mathsf{sol(F)}|$. For a median estimate to be wrong, either the event L happens in half of the invocations of ApproxMCCore or the event U happens in half of the invocations of ApproxMCCore. The number of repetitions depends on $\max(\Pr[L], \Pr[U])$. The current algorithmic design (and ensuing analysis) of ApproxMCCore provides a weak upper bound on $\max\{\Pr[L], \Pr[U]\}$: in particular, the bounds on $\max\{\Pr[L], \Pr[U]\}$ and $\Pr[L \cup U]$ are almost identical. Our key technical contribution is to design a new procedure, ApproxMC6Core, based on the rounding technique that allows us to obtain significantly better bounds on $\max\{\Pr[L], \Pr[U]\}$.

The resulting algorithm, called ApproxMC6, follows a similar structure to that of ApproxMC: it repeatedly invokes the underlying core procedure ApproxMC6Core and returns the median of the estimates. Since a single invocation of ApproxMC6Core takes as much time as ApproxMCCore, the reduction in the number of repetitions is primarily responsible for the ensuing speedup. As an example, for $\varepsilon = 0.8$, the number of repetitions of ApproxMC6Core to attain $\delta = 0.1$ and $\delta = 0.001$ is just 5 and 19, respectively; the corresponding numbers for ApproxMC were 21 and 117. An extensive experimental evaluation on 1890 benchmarks shows that the rounding technique provided 4× speedup than the state-of-the-art approximate model counter, ApproxMC. Furthermore, for a given timeout of 5000 s, ApproxMC6 solves 204 more instances than ApproxMC and achieves a reduction of 1063 s in the PAR-2 score.

The rest of the paper is organized as follows. We introduce notation and preliminaries in Sect. 2. To place our contribution in context, we review related works in Sect. 3. We identify the weakness of the current technique in Sect. 4 and present the rounding technique in Sect. 5 to address this issue. Then, we present our experimental evaluation in Sect. 6. Finally, we conclude in Sect. 7.

2 Notation and Preliminaries

Let F be a Boolean formula in conjunctive normal form (CNF), and let $\mathsf{Vars}(F)$ be the set of variables appearing in F. The set $\mathsf{Vars}(F)$ is also called the *support* of F. An assignment σ of truth values to the variables in $\mathsf{Vars}(F)$ is called a *satisfying assignment* or *witness* of F if it makes F evaluate to true. We denote the set of all witnesses of F by $\mathsf{sol(F)}$. Throughout the paper, we will use n to denote $|\mathsf{Vars}(F)|$.

The *propositional model counting problem* is to compute $|\mathsf{sol(F)}|$ for a given CNF formula F. A *probably approximately correct* (or PAC) counter is a probabilistic algorithm $\mathsf{ApproxCount}(\cdot, \cdot, \cdot)$ that takes as inputs a formula F, a tolerance parameter $\varepsilon > 0$, and a confidence parameter $\delta \in (0, 1]$, and returns an (ε, δ)-estimate c, i.e., $\mathsf{Pr}\left[\frac{|\mathsf{sol(F)}|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathsf{sol(F)}|\right] \geq 1-\delta$. PAC guarantees are also sometimes referred to as (ε, δ)-guarantees.

A closely related notion is projected model counting, where we are interested in computing the cardinality of $\mathsf{sol(F)}$ projected on a subset of variables $\mathcal{P} \subseteq \mathsf{Vars}(F)$. While for clarity of exposition, we describe our algorithm in the context of model counting, the techniques developed in this paper are applicable to projected model counting as well. Our empirical evaluation indeed considers such benchmarks.

2.1 Universal Hash Functions

Let $n, m \in \mathbb{N}$ and $\mathcal{H}(n,m) \triangleq \{h : \{0,1\}^n \rightarrow \{0,1\}^m\}$ be a family of hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$. We use $h \xleftarrow{R} \mathcal{H}(n,m)$ to denote the probability space obtained by choosing a function h uniformly at random from $\mathcal{H}(n,m)$. To measure the quality of a hash function we are interested in the set of elements of $\mathsf{sol(F)}$ mapped to α by h, denoted $\mathsf{Cell}_{\langle F,h,\alpha\rangle}$ and its cardinality, i.e., $|\mathsf{Cell}_{\langle F,h,\alpha\rangle}|$. We write $\Pr[Z : \Omega]$ to denote the probability of outcome Z when sampling from a probability space Ω. For brevity, we omit Ω when it is clear from the context. The expected value of Z is denoted $\mathsf{E}[Z]$ and its variance is denoted $\sigma^2[Z]$.

Definition 1. *A family of hash functions $\mathcal{H}(n,m)$ is strongly 2-universal if $\forall x, y \in \{0,1\}^n$, $\alpha \in \{0,1\}^m$, $h \xleftarrow{R} \mathcal{H}(n,m)$,*

$$\mathsf{Pr}[h(x) = \alpha] = \frac{1}{2^m} = \mathsf{Pr}[h(x) = h(y)]$$

For $h \xleftarrow{R} \mathcal{H}(n,n)$ and $\forall m \in \{1,...,n\}$, the m^{th} prefix-slice of h, denoted $h^{(m)}$, is a map from $\{0,1\}^n$ to $\{0,1\}^m$, such that $h^{(m)}(y)[i] = h(y)[i]$, for all $y \in \{0,1\}^n$ and for all $i \in \{1,...,m\}$. Similarly, the m^{th} prefix-slice of $\alpha \in \{0,1\}^n$, denoted $\alpha^{(m)}$, is an element of $\{0,1\}^m$ such that $\alpha^{(m)}[i] = \alpha[i]$ for all $i \in \{1,...,m\}$. To avoid cumbersome terminology, we abuse notation and write $\mathsf{Cell}_{\langle F,m\rangle}$(resp. $\mathsf{Cnt}_{\langle F,m\rangle}$) as a short-hand for $\mathsf{Cell}_{\langle F,h^{(m)},\alpha^{(m)}\rangle}$ (resp. $|\mathsf{Cell}_{\langle F,h^{(m)},\alpha^{(m)}\rangle}|$). The following proposition presents two results that are frequently used throughout this paper. The proof is deferred to Appendix A.

Proposition 1. *For every $1 \leq m \leq n$, the following holds:*

$$\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] = \frac{|\mathsf{sol(F)}|}{2^m} \tag{1}$$

$$\sigma^2\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] \leq \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] \tag{2}$$

The usage of prefix-slice of h ensures monotonicity of the random variable, $\mathsf{Cnt}_{\langle F,m\rangle}$, since from the definition of prefix-slice, we have that for every $1 \leq m < n$, $h^{(m+1)}(y) = \alpha^{(m+1)} \Rightarrow h^{(m)}(y) = \alpha^{(m)}$. Formally,

Proposition 2. *For every $1 \leq m < n$, $\mathsf{Cell}_{\langle F,m+1\rangle} \subseteq \mathsf{Cell}_{\langle F,m\rangle}$*

2.2 Helpful Combinatorial Inequality

Lemma 1. *Let $\eta(t,m,p) = \sum_{k=m}^{t} \binom{t}{k} p^k (1-p)^{t-k}$ and $p < 0.5$, then*

$$\eta(t, \lceil t/2 \rceil, p) \in \Theta\left(t^{-\frac{1}{2}}\left(2\sqrt{p(1-p)}\right)^t\right)$$

Proof. We will derive both an upper and a matching lower bound for $\eta(t, \lceil t/2 \rceil, p)$. We begin by deriving an upper bound: $\eta(t, \lceil t/2 \rceil, p) = \sum_{k=\lceil \frac{t}{2} \rceil}^{t} \binom{t}{k} p^k (1-p)^{t-k} \leq \binom{t}{\lceil t/2 \rceil} \sum_{k=\lceil \frac{t}{2} \rceil}^{t} p^k (1-p)^{t-k} \leq \binom{t}{\lceil t/2 \rceil} \cdot (p(1-p))^{\lceil \frac{t}{2} \rceil} \cdot \frac{1}{1-2p}$ $\leq \frac{1}{\sqrt{2\pi}} \cdot \frac{t}{\sqrt{\left(\frac{t}{2}-0.5\right)\left(\frac{t}{2}+0.5\right)}} \cdot \left(\frac{t}{t-1}\right)^t \cdot e^{\frac{1}{12t}-\frac{1}{6t+6}-\frac{1}{6t-6}} \cdot t^{-\frac{1}{2}} 2^t \cdot (p(1-p))^{\frac{t}{2}} \cdot (p(1-p))^{\frac{1}{2}} \cdot \frac{1}{1-2p}$. The last inequality follows Stirling's approximation. As a result, $\eta(t, \lceil t/2 \rceil, p) \in \mathcal{O}\left(t^{-\frac{1}{2}} \left(2\sqrt{p(1-p)}\right)^t\right)$. Afterwards; we move on to deriving a matching lower bound: $\eta(t, \lceil t/2 \rceil, p) = \sum_{k=\lceil \frac{t}{2} \rceil}^{t} \binom{t}{k} p^k (1-p)^{t-k} \geq \binom{t}{\lceil t/2 \rceil} p^{\lceil \frac{t}{2} \rceil} (1-p)^{t-\lceil \frac{t}{2} \rceil} \geq \frac{1}{\sqrt{2\pi}} \cdot \frac{t}{\sqrt{\left(\frac{t}{2}-0.5\right)\left(\frac{t}{2}+0.5\right)}} \cdot \left(\frac{t}{t+1}\right)^t \cdot e^{\frac{1}{12t}-\frac{1}{6t+6}-\frac{1}{6t-6}} \cdot t^{-\frac{1}{2}} 2^t \cdot (p(1-p))^{\frac{t}{2}} \cdot p^{\frac{1}{2}} (1-p)^{-\frac{1}{2}} \cdot \frac{1}{1-2p}$. The last inequality again follows Stirling's approximation. Hence, $\eta(t, \lceil t/2 \rceil, p) \in \Omega\left(t^{-\frac{1}{2}} \left(2\sqrt{p(1-p)}\right)^t\right)$. Combining these two bounds, we conclude that $\eta(t, \lceil t/2 \rceil, p) \in \Theta\left(t^{-\frac{1}{2}} \left(2\sqrt{p(1-p)}\right)^t\right)$. □

3 Related Work

The seminal work of Valiant established that #SAT is #P-complete [28]. Toda later showed that every problem in the polynomial hierarchy could be solved by just a polynomial number of calls to a #P oracle [27]. Based on Carter and Wegman's seminal work on universal hash functions [4], Stockmeyer proposed a probabilistic polynomial time procedure, with access to an NP oracle, to obtain an (ε, δ)-approximation of F [25].

Built on top of Stockmeyer's work, the core theoretical idea behind the hashing-based approximate solution counting framework, as presented in Algorithm 1 (ApproxMC [7]), is to use 2-universal hash functions to partition the solution space (denoted by $\mathsf{sol(F)}$ for a given formula F) into *small* cells of *roughly equal* size. A cell is considered *small* if the number of solutions it contains is less than or equal to a pre-determined threshold, thresh. An NP oracle is used to determine if a cell is small by iteratively enumerating its solutions until either there are no more solutions or thresh + 1 solutions have been found. In practice, an SAT solver is used to implement the NP oracle. To ensure a polynomial number of calls to the oracle, the threshold, thresh, is set to be polynomial in the input parameter ε at Line 1. The subroutine ApproxMCCore takes the formula F and thresh as inputs and estimates the number of solutions at Line 7. To determine the appropriate number of cells, i.e., the value of m for $\mathcal{H}(n, m)$, ApproxMCCore uses a search procedure at Line 3 of Algorithm 2. The estimate is calculated as the number of solutions in a randomly chosen cell, scaled by the number of cells, i.e., 2^m at Line 5. To improve confidence in the estimate, ApproxMC performs multiple runs of the ApproxMCCore subroutine at Lines 5–9 of Algorithm 1. The final count is computed as the median of the estimates obtained at Line 10.

Algorithm 1. ApproxMC(F, ε, δ)

```
$\mathsf{thresh} \leftarrow 9.84\left(1+\frac{\varepsilon}{1+\varepsilon}\right)\left(1+\frac{1}{\varepsilon}\right)^2$;
$Y \leftarrow \mathsf{BoundedSAT}(F, \mathsf{thresh})$;
if $(|Y| < \mathsf{thresh})$ then return $|Y|$;
$t \leftarrow \lceil 17\log_2(3/\delta)\rceil$ ; $C \leftarrow \mathsf{emptyList}$; $\mathsf{iter} \leftarrow 0$;
repeat
    $\mathsf{iter} \leftarrow \mathsf{iter} + 1$;
    $\mathsf{nSols} \leftarrow \mathsf{ApproxMCCore}(F, \mathsf{thresh})$;
    $\mathsf{AddToList}(C, \mathsf{nSols})$;
until $(\mathsf{iter} \geq t)$;
$\mathsf{finalEstimate} \leftarrow \mathsf{FindMedian}(C)$;
return $\mathsf{finalEstimate}$;
```

Algorithm 2. ApproxMCCore(F, thresh)

```
1: Choose $h$ at random from $\mathcal{H}(n, n)$;
2: Choose $\alpha$ at random from $\{0,1\}^n$;
3: $m \leftarrow \mathsf{LogSATSearch}(F, h, \alpha, \mathsf{thresh})$;
4: $\mathsf{Cnt}_{\langle F,m\rangle} \leftarrow \mathsf{BoundedSAT}\left(F \wedge \left(h^{(m)}\right)^{-1}\left(\alpha^{(m)}\right), \mathsf{thresh}\right)$;
5: return $(2^m \times \mathsf{Cnt}_{\langle F,m\rangle})$;
```

In the second version of ApproxMC [8], two key algorithmic improvements are proposed to improve the practical performance by reducing the number of calls to the SAT solver. The first improvement is using galloping search to more efficiently find the correct number of cells, i.e., LogSATSearch at Line 3 of Algorithm 2. The second is using linear search over a small interval around the previous value of m before resorting to the galloping search. Additionally, the third and fourth versions [22,23] enhance the algorithm's performance by effectively dealing with CNF formulas conjuncted with XOR constraints, commonly used in the hashing-based counting framework. Moreover, an effective preprocessor named Arjun [24] is proposed to enhance ApproxMC's performance by constructing shorter XOR constraints. As a result, the combination of Arjun and ApproxMC4 solved almost all existing benchmarks [24], making it the current state of the art in this field.

In this work, we aim to address the main limitation of the ApproxMC algorithm by focusing on an aspect that still needs to be improved upon by previous developments. Specifically, we aim to improve the core algorithm of ApproxMC, which has remained unchanged.

4 Weakness of ApproxMC

As noted above, the core algorithm of ApproxMC has not changed since 2016, and in this work, we aim to address the core limitation of ApproxMC. To put our contribution in context, we first review ApproxMC and its core algorithm, called

$\mathsf{ApproxMCCore}$. We present the pseudocode of $\mathsf{ApproxMC}$ and $\mathsf{ApproxMCCore}$ in Algorithms 1 and 2, respectively. $\mathsf{ApproxMCCore}$ may return an estimate that falls outside the PAC range $\left[\frac{|\mathsf{sol(F)}|}{1+\varepsilon}, (1+\varepsilon)|\mathsf{sol(F)}|\right]$ with a certain probability of error. Therefore, $\mathsf{ApproxMC}$ repeatedly invokes $\mathsf{ApproxMCCore}$ (Lines 5– 9) and returns the median of the estimates returned by $\mathsf{ApproxMCCore}$ (Line 10), which reduces the error probability to the user-provided parameter δ.

Let Error_t denote the event that the median of t estimates falls outside $\left[\frac{|\mathsf{sol(F)}|}{1+\varepsilon}, (1+\varepsilon)|\mathsf{sol(F)}|\right]$. Let L denote the event that an invocation $\mathsf{ApproxMCCore}$ returns an estimate less than $\frac{|\mathsf{sol(F)}|}{1+\varepsilon}$. Similarly, let U denote the event that an individual estimate of $|\mathsf{sol(F)}|$ is greater than $(1+\varepsilon)|\mathsf{sol(F)}|$. For simplicity of exposition, we assume t is odd; the current implementation of t indeed ensures that t is odd by choosing the smallest odd t for which $\Pr[\mathsf{Error}_t] \leq \delta$.

In the remainder of the section, we will demonstrate that reducing $\max\{\Pr[L], \Pr[U]\}$ can effectively reduce the number of repetitions t, making the small-δ scenarios practical. To this end, we will first demonstrate the existing analysis technique of $\mathsf{ApproxMC}$ leads to loose bounds on $\Pr[\mathsf{Error}_t]$. We then present a new analysis that leads to tighter bounds on $\Pr[\mathsf{Error}_t]$.

The existing combinatorial analysis in [7] derives the following proposition:

Proposition 3.

$$Pr[\mathsf{Error}_t] \leq \eta(t, \lceil t/2 \rceil, Pr[L \cup U])$$

where $\eta(t,m,p) = \sum_{k=m}^{t} \binom{t}{k} p^k (1-p)^{t-k}$.

Proposition 3 follows from the observation that if the median falls outside the PAC range, at least $\lceil t/2 \rceil$ of the results must also be outside the range. Let $\eta(t, \lceil t/2 \rceil, \Pr[L \cup U]) \leq \delta$, and we can compute a valid t at Line 4 of $\mathsf{ApproxMC}$.

Proposition 3 raises a question: can we derive a tight upper bound for $\Pr[\mathsf{Error}_t]$? The following lemma provides an affirmative answer to this question.

Lemma 2. *Assuming t is odd, we have:*

$$Pr[\mathsf{Error}_t] = \eta(t, \lceil t/2 \rceil, Pr[L]) + \eta(t, \lceil t/2 \rceil, Pr[U])$$

Proof. Let I_i^L be an indicator variable that is 1 when $\mathsf{ApproxMCCore}$ returns a nSols less than $\frac{|\mathsf{sol(F)}|}{1+\varepsilon}$, indicating the occurrence of event L in the i-th repetition. Let I_i^U be an indicator variable that is 1 when $\mathsf{ApproxMCCore}$ returns a nSols greater than $(1+\varepsilon)|\mathsf{sol(F)}|$, indicating the occurrence of event U in the i-th repetition. We aim first to prove that $\mathsf{Error}_t \Leftrightarrow \left(\sum_{i=1}^{t} I_i^L \geq \left\lceil \frac{t}{2} \right\rceil\right) \vee \left(\sum_{i=1}^{t} I_i^U \geq \left\lceil \frac{t}{2} \right\rceil\right)$. We will begin by proving the right ($\Rightarrow$) implication. If the median of t estimates violates the PAC guarantee, the median is either less than $\frac{|\mathsf{sol(F)}|}{1+\varepsilon}$ or greater than $(1+\varepsilon)|\mathsf{sol(F)}|$. In the first case, since half of the estimates are less than the median, at least $\left\lceil \frac{t}{2} \right\rceil$ estimates are less than $\frac{|\mathsf{sol(F)}|}{1+\varepsilon}$. Formally, this implies $\sum_{i=1}^{t} I_i^L \geq \left\lceil \frac{t}{2} \right\rceil$. Similarly, in the case that the median is greater than $(1+\varepsilon)|\mathsf{sol}(\mathsf{F})|$, since half of the estimates are greater than the median, at least $\left\lceil \frac{t}{2} \right\rceil$ estimates are greater than $(1+\varepsilon)|\mathsf{sol}(\mathsf{F})|$, thus formally implying $\sum_{i=1}^{t} I_i^U \geq \left\lceil \frac{t}{2} \right\rceil$. On the other hand, we prove the left ($\Leftarrow$) implication. Given $\sum_{i=1}^{t} I_i^L \geq \left\lceil \frac{t}{2} \right\rceil$, more than half of the estimates are less than $\frac{|\mathsf{sol}(\mathsf{F})|}{1+\varepsilon}$, and therefore the median is less than $\frac{|\mathsf{sol}(\mathsf{F})|}{1+\varepsilon}$, violating the PAC guarantee. Similarly, given $\sum_{i=1}^{t} I_i^U \geq \left\lceil \frac{t}{2} \right\rceil$, more than half of the estimates are greater than $(1+\varepsilon)|\mathsf{sol}(\mathsf{F})|$, and therefore the median is greater than $(1+\varepsilon)|\mathsf{sol}(\mathsf{F})|$, violating the PAC guarantee. This concludes the proof of $\mathsf{Error}_t \Leftrightarrow \left(\sum_{i=1}^{t} I_i^L \geq \left\lceil \frac{t}{2} \right\rceil\right) \vee \left(\sum_{i=1}^{t} I_i^U \geq \left\lceil \frac{t}{2} \right\rceil\right)$. Then we obtain:

$$\begin{aligned}
\Pr\left[\mathsf{Error}_t\right] &= \Pr\left[\left(\sum_{i=1}^{t} I_i^L \geq \lceil t/2 \rceil\right) \vee \left(\sum_{i=1}^{t} I_i^U \geq \lceil t/2 \rceil\right)\right] \\
&= \Pr\left[\left(\sum_{i=1}^{t} I_i^L \geq \lceil t/2 \rceil\right)\right] + \Pr\left[\left(\sum_{i=1}^{t} I_i^U \geq \lceil t/2 \rceil\right)\right] \\
&- \Pr\left[\left(\sum_{i=1}^{t} I_i^L \geq \lceil t/2 \rceil\right) \wedge \left(\sum_{i=1}^{t} I_i^U \geq \lceil t/2 \rceil\right)\right]
\end{aligned}$$

Given $I_i^L + I_i^U \leq 1$ for $i = 1, 2, ..., t$, $\sum_{i=1}^{t}(I_i^L + I_i^U) \leq t$ is there, but if $\left(\sum_{i=1}^{t} I_i^L \geq \lceil t/2 \rceil\right) \wedge \left(\sum_{i=1}^{t} I_i^U \geq \lceil t/2 \rceil\right)$ is also given, we obtain $\sum_{i=1}^{t}(I_i^L + I_i^U) \geq t + 1$ contradicting $\sum_{i=1}^{t}(I_i^L + I_i^U) \leq t$; Hence, we can conclude that $\Pr\left[\left(\sum_{i=1}^{t} I_i^L \geq \lceil t/2 \rceil\right) \wedge \left(\sum_{i=1}^{t} I_i^U \geq \lceil t/2 \rceil\right)\right] = 0$. From this, we can deduce:

$$\begin{aligned}
\Pr\left[\mathsf{Error}_t\right] &= \Pr\left[\left(\sum_{i=1}^{t} I_i^L \geq \lceil t/2 \rceil\right)\right] + \Pr\left[\left(\sum_{i=1}^{t} I_i^U \geq \lceil t/2 \rceil\right)\right] \\
&= \eta(t, \lceil t/2 \rceil, \Pr\left[L\right]) + \eta(t, \lceil t/2 \rceil, \Pr\left[U\right])
\end{aligned}$$

□

Though Lemma 2 shows that reducing $\Pr[L]$ and $\Pr[U]$ can decrease the error probability, it is still uncertain to what extent $\Pr[L]$ and $\Pr[U]$ affect the error probability. To further understand this impact, the following lemma is presented to establish a correlation between the error probability and t depending on $\Pr[L]$ and $\Pr[U]$.

Lemma 3. *Let* $p_{max} = \max\left\{Pr\left[L\right], Pr\left[U\right]\right\}$ *and* $p_{max} < 0.5$*, we have*

$$Pr\left[\mathsf{Error}_t\right] \in \Theta\left(t^{-\frac{1}{2}}\left(2\sqrt{p_{max}(1-p_{max})}\right)^t\right)$$

Proof. Applying Lemmas 1 and 2, we have

$$\begin{aligned}\Pr\left[\mathsf{Error}_t\right] &\in \Theta\left(t^{-\frac{1}{2}}\left(\left(2\sqrt{\Pr\left[L\right]\left(1-\Pr\left[L\right]\right)}\right)^t+\left(2\sqrt{\Pr\left[U\right]\left(1-\Pr\left[U\right]\right)}\right)^t\right)\right)\\ &=\Theta\left(t^{-\frac{1}{2}}\left(2\sqrt{p_{max}(1-p_{max})}\right)^t\right)\end{aligned}$$

□

In summary, Lemma 3 provides a way to tighten the bound on $\Pr[\mathsf{Error}_t]$ by designing an algorithm such that we can obtain a tighter bound on p_{max} in contrast to previous approaches that relied on obtaining a tighter bound on $\Pr[L \cup U]$.

5 Rounding Model Counting

In this section, we present a *rounding*-based technique that allows us to obtain a tighter bound on p_{max}. On a high-level, instead of returning the estimate from one iteration of the underlying core algorithm as the number of solutions in a randomly chosen cell multiplied by the number of cells, we *round* each estimate of the model count to a value that is more likely to be within $(1+\varepsilon)$-bound. While counter-intuitive at first glance, we show that rounding the estimate reduces $\max\left\{\Pr\left[L\right],\Pr\left[U\right]\right\}$, thereby resulting in a smaller number of repetitions of the underlying algorithm.

We present ApproxMC6, a *rounding*-based approximate model counting algorithm, in Sect. 5.1. Section 5.2 will demonstrate how ApproxMC6 decreases $\max\left\{\Pr\left[L\right],\Pr\left[U\right]\right\}$ and the number of estimates. Lastly, in Sect. 5.3, we will provide proof of the theoretical correctness of the algorithm.

5.1 Algorithm

Algorithm 3 presents the procedure of ApproxMC6. ApproxMC6 takes as input a formula F, a tolerance parameter ε, and a confidence parameter δ. ApproxMC6 returns an (ε,δ)-estimate c of $|\mathsf{sol(F)}|$ such that $\mathsf{Pr}\left[\frac{|\mathsf{sol(F)}|}{1+\varepsilon} \le c \le (1+\varepsilon)|\mathsf{sol(F)}|\right] \ge 1-\delta$. ApproxMC6 is identical to ApproxMC in its initialization of data structures and handling of base cases (Lines 1–4).

In Line 5, we pre-compute the rounding type and rounding value to be used in ApproxMC6Core. configRound is implemented in Algorithm 5; the precise choices arise due to technical analysis, as presented in Sect. 5.2. Note that, in configRound, $\mathsf{Cnt}_{\langle F,m\rangle}$ is *rounded up* to roundValue for $\varepsilon < 3$ ($\mathsf{roundUp}=1$) but *rounded* to roundValue for $\varepsilon \ge 3$ ($\mathsf{roundUp}=0$). Rounding up means we assign roundValue to $\mathsf{Cnt}_{\langle F,m\rangle}$ if $\mathsf{Cnt}_{\langle F,m\rangle}$ is less than roundValue and, otherwise, keep $\mathsf{Cnt}_{\langle F,m\rangle}$ unchanged. Rounding means that we assign roundValue to $\mathsf{Cnt}_{\langle F,m\rangle}$ in all cases. ApproxMC6 computes the number of repetitions necessary to lower error probability down to δ at Line 6. The implementation of computeIter is presented

Algorithm 3. ApproxMC6(F, ε, δ)

```
1:  thresh ← $9.84\left(1+\frac{\varepsilon}{1+\varepsilon}\right)\left(1+\frac{1}{\varepsilon}\right)^2$;
2:  $Y$ ← BoundedSAT($F$, thresh);
3:  if ($|Y| <$ thresh) then return $|Y|$;
4:  $C$ ← emptyList; iter ← 0;
5:  (roundUp, roundValue) ← configRound($\varepsilon$)
6:  $t$ ← computeIter($\varepsilon, \delta$)
7:  repeat
8:      iter ← iter + 1;
9:      nSols ← ApproxMC6Core($F$, thresh, roundUp, roundValue);
10:     AddToList($C$, nSols);
11: until (iter $\geq t$);
12: finalEstimate ← FindMedian($C$);
13: return finalEstimate ;
```

in Algorithm 6 following Lemma 2. The iterator keeps increasing until the tight error bound is no more than δ. As we will show in Sect. 5.2, $\Pr[L]$ and $\Pr[U]$ depend on ε. In the loop of Lines 7–11, ApproxMC6Core repeatedly estimates $|\mathsf{sol}(\mathsf{F})|$. Each estimate nSols is stored in List C, and the median of C serves as the final estimate satisfying the (ε, δ)-guarantee.

Algorithm 4 shows the pseudo-code of ApproxMC6Core. A random hash function is chosen at Line 1 to partition sol(F) into *roughly equal* cells. A random hash value is chosen at Line 2 to randomly pick a cell for estimation. In Line 3, we search for a value m such that the cell picked from 2^m available cells is *small* enough to enumerate solutions one by one while providing a good estimate of $|\mathsf{sol}(\mathsf{F})|$. In Line 4, a bounded model counting is invoked to compute the size of the picked cell, i.e., $\mathsf{Cnt}_{\langle F,m\rangle}$. Finally, if roundUp equals 1, $\mathsf{Cnt}_{\langle F,m\rangle}$ is rounded up to roundValue at Line 6. Otherwise, roundUp equals 0, and $\mathsf{Cnt}_{\langle F,m\rangle}$ is rounded to roundValue at Line 8. Note that *rounding up* returns roundValue only if $\mathsf{Cnt}_{\langle F,m\rangle}$ is less than roundValue. However, in the case of *rounding*, roundValue is always returned no matter what value $\mathsf{Cnt}_{\langle F,m\rangle}$ is.

For large ε ($\varepsilon \geq 3$), ApproxMC6Core returns a value that is independent of the value returned by BoundedSAT in line 4 of Algorithm 4. However, observe the value depends on m returned by LogSATSearch [8], which in turn uses BoundedSAT to find the value of m; therefore, the algorithm's run is not independent of all the calls to BoundedSAT. The technical reason for correctness stems from the observation that for large values of ε, we can always find a value of m such that $2^m \times c$ (where c is a constant) is a $(1+\varepsilon)$-approximation of $|\mathsf{sol}(\mathsf{F})|$. An example, consider $n = 7$ and let $c = 1$, then a $(1+3)$-approximation of a number between 1 and 128 belongs to $[1, 2, 4, 8, 16, 32, 64, 128]$; therefore, returning an answer of the form $c \times 2^m$ suffices as long as we are able to search for the right value of m, which is accomplished by LogSATSearch. We could skip the final call to BoundedSAT in line 4 of ApproxMC6Core for large values of ε, but the actual computation of BoundedSAT comes with LogSATSearch.

Algorithm 4. ApproxMC6Core(F, thresh, roundUp, roundValue)

```
Choose $h$ at random from $\mathcal{H}(n,n)$;
Choose $\alpha$ at random from $\{0,1\}^n$;
$m \leftarrow \mathsf{LogSATSearch}(F, h, \alpha, \mathsf{thresh})$;
$\mathsf{Cnt}_{\langle F,m\rangle} \leftarrow \mathsf{BoundedSAT}\left(F \wedge \left(h^{(m)}\right)^{-1}\left(\alpha^{(m)}\right), \mathsf{thresh}\right)$;
if roundUp $= 1$ then
    return $(2^m \times \max\{\mathsf{Cnt}_{\langle F,m\rangle}, \mathsf{roundValue}\})$;
else
    return $(2^m \times \mathsf{roundValue})$;
```

Algorithm 5. configRound(ε)

```
if $(\varepsilon < \sqrt{2}-1)$ then return $(1, \frac{\sqrt{1+2\varepsilon}}{2}\mathsf{pivot})$;
else if $(\varepsilon < 1)$ then return $(1, \frac{\mathsf{pivot}}{\sqrt{2}})$;
else if $(\varepsilon < 3)$ then return $(1, \mathsf{pivot})$;
else if $(\varepsilon < 4\sqrt{2}-1)$ then return $(0, \mathsf{pivot})$;
else
    return $(0, \sqrt{2}\mathsf{pivot})$;
```

5.2 Repetition Reduction

We will now show that ApproxMC6Core allows us to obtain a smaller $\max\{\Pr[L], \Pr[U]\}$. Furthermore, we show the large gap between the error probability of ApproxMC6 and that of ApproxMC both analytically and visually.

The following lemma presents the upper bounds of $\Pr[L]$ and $\Pr[U]$ for ApproxMC6Core. Let $\mathsf{pivot} = 9.84\left(1+\frac{1}{\varepsilon}\right)^2$ for simplicity.

Lemma 4. *The following bounds hold for* ApproxMC6*:*

$$Pr[L] \leq \begin{cases} 0.262 & \textit{if } \varepsilon < \sqrt{2}-1 \\ 0.157 & \textit{if } \sqrt{2}-1 \leq \varepsilon < 1 \\ 0.085 & \textit{if } 1 \leq \varepsilon < 3 \\ 0.055 & \textit{if } 3 \leq \varepsilon < 4\sqrt{2}-1 \\ 0.023 & \textit{if } \varepsilon \geq 4\sqrt{2}-1 \end{cases}$$

$$Pr[U] \leq \begin{cases} 0.169 & \textit{if } \varepsilon < 3 \\ 0.044 & \textit{if } \varepsilon \geq 3 \end{cases}$$

The proof of Lemma 4 is deferred to Sect. 5.3. Observe that Lemma 4 influences the choices in the design of configRound (Algorithm 5). Recall that $\max\{\Pr[L], \Pr[U]\} \leq 0.36$ for ApproxMC (Appendix C), but Lemma 4 ensures $\max\{\Pr[L], \Pr[U]\} \leq 0.262$ for ApproxMC6. For $\varepsilon \geq 4\sqrt{2}-1$, Lemma 4 even delivers $\max\{\Pr[L], \Pr[U]\} \leq 0.044$.

Algorithm 6. computeIter(ε, δ)

```
iter ← $1$;
while ($\eta(\mathsf{iter}, \lceil \mathsf{iter}/2 \rceil, \Pr_\varepsilon[L]) + \eta(\mathsf{iter}, \lceil \mathsf{iter}/2 \rceil, \Pr_\varepsilon[U]) > \delta$) do
    iter ← iter $+ 2$;
return iter;
```

The following theorem analytically presents the gap between the error probability of ApproxMC6 and that of ApproxMC[1].

Theorem 1. *For* $\sqrt{2} - 1 \leq \varepsilon < 1$,

$$Pr\left[\mathsf{Error}_t\right] \in \begin{cases} \mathcal{O}\left(t^{-\frac{1}{2}} 0.75^t\right) & \textit{for } \mathsf{ApproxMC6} \\ \mathcal{O}\left(t^{-\frac{1}{2}} 0.96^t\right) & \textit{for } \mathsf{ApproxMC} \end{cases}$$

Proof. From Lemma 4, we obtain $p_{max} \leq 0.169$ for ApproxMC6. Applying Lemma 3, we have

$$\Pr\left[\mathsf{Error}_t\right] \in \mathcal{O}\left(t^{-\frac{1}{2}}\left(2\sqrt{0.169(1-0.169)}\right)^t\right) \subseteq \mathcal{O}\left(t^{-\frac{1}{2}} 0.75^t\right)$$

For ApproxMC, combining $p_{max} \leq 0.36$ (Appendix C) and Lemma 3, we obtain

$$\Pr\left[\mathsf{Error}_t\right] \in \mathcal{O}\left(t^{-\frac{1}{2}}\left(2\sqrt{0.36(1-0.36)}\right)^t\right) = \mathcal{O}\left(t^{-\frac{1}{2}} 0.96^t\right)$$

□

Figure 1 visualizes the large gap between the error probability of ApproxMC6 and that of ApproxMC. The x-axis represents the number of repetitions (t) in ApproxMC6 or ApproxMC. The y-axis represents the upper bound of error probability in the log scale. For example, as $t = 117$, ApproxMC guarantees that with a probability of 10^{-3}, the median over 117 estimates violates the PAC guarantee. However, ApproxMC6 allows a much smaller error probability that is at most 10^{-15} for $\sqrt{2} - 1 \leq \varepsilon < 1$. The smaller error probability enables ApproxMC6 to repeat fewer repetitions while providing the same level of theoretical guarantee. For example, given $\delta = 0.001$ to ApproxMC, i.e., $y = 0.001$ in Fig. 1, ApproxMC requests 117 repetitions to obtain the given error probability. However, ApproxMC6 claims that 37 repetitions for $\varepsilon < \sqrt{2} - 1$, 19 repetitions for $\sqrt{2} - 1 \leq \varepsilon < 1$, 17 repetitions for $1 \leq \varepsilon < 3$, 7 repetitions for $3 \leq \varepsilon < 4\sqrt{2} - 1$, and 5 repetitions for $\varepsilon \geq 4\sqrt{2} - 1$ are sufficient to obtain the same level of error probability. Consequently, ApproxMC6 can obtain 3×, 6×, 7×, 17×, and 23× speedups, respectively, than ApproxMC.

[1] We state the result for the case $\sqrt{2} - 1 \leq \varepsilon < 1$. A similar analysis can be applied to other cases, which leads to an even bigger gap between ApproxMC6 and ApproxMC.

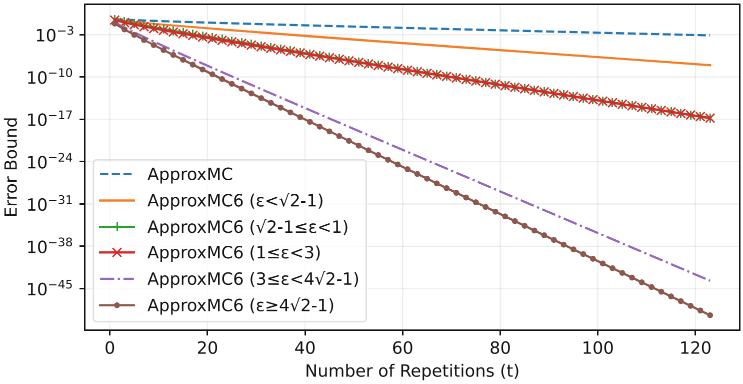


Fig. 1. Comparison of error bounds for ApproxMC6 and ApproxMC.

5.3 Proof of Lemma 4 for Case $\sqrt{2}-1 \leq \varepsilon < 1$

We provide full proof of Lemma 4 for case $\sqrt{2}-1 \leq \varepsilon < 1$. We defer the proof of other cases to Appendix D.

Let T_m denote the event $\left(\mathsf{Cnt}_{\langle F,m\rangle} < \mathsf{thresh}\right)$, and let L_m and U_m denote the events $\left(\mathsf{Cnt}_{\langle F,m\rangle} < \frac{\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]}{1+\varepsilon}\right)$ and $\left(\mathsf{Cnt}_{\langle F,m\rangle} > \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right](1+\varepsilon)\right)$, respectively. To ease the proof, let U'_m denote $\left(\mathsf{Cnt}_{\langle F,m\rangle} > \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right](1+\frac{\varepsilon}{1+\varepsilon})\right)$, and thereby $U_m \subseteq U'_m$. Let $m^* = \lfloor \log_2 |\mathsf{sol(F)}| - \log_2(\mathsf{pivot}) + 1 \rfloor$ such that m^* is the smallest m satisfying $\frac{|\mathsf{sol(F)}|}{2^m}(1+\frac{\varepsilon}{1+\varepsilon}) \leq \mathsf{thresh} - 1$.

Let us first prove the lemmas used in the proof of Lemma 4.

Lemma 5. *For every $0 < \beta < 1$, $\gamma > 1$, and $1 \leq m \leq n$, the following holds:*

1. $Pr\left[\mathsf{Cnt}_{\langle F,m\rangle} \leq \beta \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]\right] \leq \frac{1}{1+(1-\beta)^2\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]}$
2. $Pr\left[\mathsf{Cnt}_{\langle F,m\rangle} \geq \gamma \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]\right] \leq \frac{1}{1+(\gamma-1)^2\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]}$

Proof. Statement 1 can be proved following the proof of Lemma 1 in [8]. For statement 2, we rewrite the left-hand side and apply Cantelli's inequality: $\Pr\left[\mathsf{Cnt}_{\langle F,m\rangle} - \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] \geq (\gamma-1)\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]\right] \leq \frac{\sigma^2\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]}{\sigma^2\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]+((\gamma-1)\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right])^2}$. Finally, applying Eq. 2 completes the proof. □

Lemma 6. *Given $\sqrt{2}-1 \leq \varepsilon < 1$, the following bounds hold:*

1. $Pr[T_{m^*-3}] \leq \frac{1}{62.5}$
2. $Pr[L_{m^*-2}] \leq \frac{1}{20.68}$
3. $Pr[L_{m^*-1}] \leq \frac{1}{10.84}$
4. $Pr[U'_{m^*}] \leq \frac{1}{5.92}$

Proof. Following the proof of Lemma 2 in [8], we can prove statements 1, 2, and 3. To prove statement 4, replacing γ with $(1+\frac{\varepsilon}{1+\varepsilon})$ in Lemma 5 and employing $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*\rangle}\right] \geq \mathsf{pivot}/2$, we obtain $\Pr\left[U'_{m^*}\right] \leq \frac{1}{1+\left(\frac{\varepsilon}{1+\varepsilon}\right)^2 \mathsf{pivot}/2} \leq \frac{1}{5.92}$. □

Now we prove the upper bounds of $\Pr[L]$ and $\Pr[U]$ in Lemma 4 for $\sqrt{2}-1 \leq \varepsilon < 1$. The proof for other ε is deferred to Appendix D due to the page limit.

Lemma 4. *The following bounds hold for* ApproxMC6:

$$\Pr[L] \leq \begin{cases} 0.262 & \text{if } \varepsilon < \sqrt{2}-1 \\ 0.157 & \text{if } \sqrt{2}-1 \leq \varepsilon < 1 \\ 0.085 & \text{if } 1 \leq \varepsilon < 3 \\ 0.055 & \text{if } 3 \leq \varepsilon < 4\sqrt{2}-1 \\ 0.023 & \text{if } \varepsilon \geq 4\sqrt{2}-1 \end{cases}$$

$$\Pr[U] \leq \begin{cases} 0.169 & \text{if } \varepsilon < 3 \\ 0.044 & \text{if } \varepsilon \geq 3 \end{cases}$$

Proof. We prove the case of $\sqrt{2}-1 \leq \varepsilon < 1$. The proof for other ε is deferred to Appendix D. Let us first bound $\Pr[L]$. Following LogSATSearch in [8], we have

$$\Pr[L] = \left[\bigcup_{i\in\{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap L_i\right)\right] \tag{3}$$

Equation 3 can be simplified by three observations labeled $O1, O2$ and $O3$ below.

$O1$: $\forall i \leq m^*-3, T_i \subseteq T_{i+1}$. Therefore,

$$\bigcup_{i\in\{1,\ldots,m^*-3\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq \bigcup_{i\in\{1,\ldots,m^*-3\}} T_i \subseteq T_{m^*-3}$$

$O2$:]For $i \in \{m^*-2, m^*-1\}$, we have

$$\bigcup_{i\in\{m^*-2,m^*-1\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq L_{m^*-2} \cup L_{m^*-1}$$

$O3$: $\forall i \geq m^*$, since rounding $\mathsf{Cnt}_{\langle F,i\rangle}$ up to $\frac{\mathsf{pivot}}{\sqrt{2}}$ and $m^* \geq \log_2|\mathsf{sol(F)}| - \log_2(\mathsf{pivot})$, we have $2^i \times \mathsf{Cnt}_{\langle F,i\rangle} \geq 2^{m^*} \times \frac{\mathsf{pivot}}{\sqrt{2}} \geq \frac{|\mathsf{sol(F)}|}{\sqrt{2}} \geq \frac{|\mathsf{sol(F)}|}{1+\varepsilon}$. The last inequality follows from $\varepsilon \geq \sqrt{2}-1$. Then we have $\mathsf{Cnt}_{\langle F,i\rangle} \geq \frac{\mathsf{E}\left[\mathsf{Cnt}_{\langle F,i\rangle}\right]}{1+\varepsilon}$. Therefore, $L_i = \emptyset$ for $i \geq m^*$ and we have

$$\bigcup_{i\in\{m^*,\ldots,n\}} (\overline{T_{i-1}} \cap T_i \cap L_i) = \emptyset$$

Following the observations $O1$, $O2$, and $O3$, we simplify Eq. 3 and obtain

$$\Pr[L] \leq \Pr[T_{m^*-3}] + \Pr[L_{m^*-2}] + \Pr[L_{m^*-1}]$$

Employing Lemma 6 gives $\Pr[L] \leq 0.157$.

Now let us bound $\Pr[U]$. Similarly, following $\mathsf{LogSATSearch}$ in [8], we have

$$\Pr[U] = \left[\bigcup_{i\in\{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap U_i\right) \right] \tag{4}$$

We derive the following observations $O4$ and $O5$.

$O4$: $\forall i \leq m^* - 1$, since $m^* \leq \log_2|\mathsf{sol(F)}| - \log_2(\mathsf{pivot}) + 1$, we have $2^i \times \mathsf{Cnt}_{\langle F,i\rangle} \leq 2^{m^*-1} \times \mathsf{thresh} \leq |\mathsf{sol(F)}| \left(1 + \frac{\varepsilon}{1+\varepsilon}\right)$. Then we obtain $\mathsf{Cnt}_{\langle F,i\rangle} \leq \mathsf{E}\left[\mathsf{Cnt}_{\langle F,i\rangle}\right] \left(1 + \frac{\varepsilon}{1+\varepsilon}\right)$. Therefore, $T_i \cap U_i' = \emptyset$ for $i \leq m^* - 1$ and we have

$$\bigcup_{i\in\{1,\ldots,m^*-1\}} \left(\overline{T_{i-1}} \cap T_i \cap U_i\right) \subseteq \bigcup_{i\in\{1,\ldots,m^*-1\}} \left(\overline{T_{i-1}} \cap T_i \cap U_i'\right) = \emptyset$$

$O5$: $\forall i \geq m^*$, $\overline{T_i}$ implies $\mathsf{Cnt}_{\langle F,i\rangle} > \mathsf{thresh}$, and then we have $2^i \times \mathsf{Cnt}_{\langle F,i\rangle} > 2^{m^*} \times \mathsf{thresh} \geq |\mathsf{sol(F)}| \left(1 + \frac{\varepsilon}{1+\varepsilon}\right)$. The second inequality follows from $m^* \geq \log_2|\mathsf{sol(F)}| - \log_2(\mathsf{pivot})$. Then we obtain $\mathsf{Cnt}_{\langle F,i\rangle} > \mathsf{E}\left[\mathsf{Cnt}_{\langle F,i\rangle}\right] \left(1 + \frac{\varepsilon}{1+\varepsilon}\right)$. Therefore, $\overline{T_i} \subseteq U_i'$ for $i \geq m^*$. Since $\forall i, \overline{T_i} \subseteq \overline{T_{i-1}}$, we have

$$\begin{aligned}
\bigcup_{i\in\{m^*,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap U_i\right) &\subseteq \bigcup_{i\in\{m^*+1,\ldots,n\}} \overline{T_{i-1}} \cup \left(\overline{T_{m^*-1}} \cap T_{m^*} \cap U_{m^*}\right) \\
&\subseteq \overline{T_{m^*}} \cup \left(\overline{T_{m^*-1}} \cap T_{m^*} \cap U_{m^*}\right) \\
&\subseteq \overline{T_{m^*}} \cup U_{m^*} \\
&\subseteq U'_{m^*}
\end{aligned} \tag{5}$$

Remark that for $\sqrt{2} - 1 \leq \varepsilon < 1$, we round $\mathsf{Cnt}_{\langle F,m^*\rangle}$ up to $\frac{\mathsf{pivot}}{\sqrt{2}}$, and we have $2^{m^*} \times \frac{\mathsf{pivot}}{\sqrt{2}} \leq |\mathsf{sol(F)}|(1+\varepsilon)$, which means *rounding* doesn't affect the event U_{m^*}; therefore, Inequality 5 still holds.

Following the observations $O4$ and $O5$, we simplify Eq. 4 and obtain

$$\Pr[U] \leq \Pr[U'_{m^*}]$$

Employing Lemma 6 gives $\Pr[U] \leq 0.169$. □

The breakpoints in ε of Lemma 4 arise from how we use rounding to lower the error probability for events L and U. Rounding up counts can lower $\Pr[L]$ but may increase $\Pr[U]$. Therefore, we want to round up counts to a value that doesn't affect the event U. Take $\sqrt{2} - 1 \leq \varepsilon < 1$ as an example; we round up the count to a value such that L_{m^*} becomes an empty event with zero probability while U_{m^*} remains unchanged. To make L_{m^*} empty, we have

$$2^{m^*} \times \mathsf{roundValue} \geq 2^{m^*} \times \frac{1}{1+\varepsilon}\mathsf{pivot} \geq \frac{1}{1+\varepsilon}|\mathsf{sol(F)}| \tag{6}$$

where the last inequality follows from $m^* \geq \log_2 |\mathsf{sol(F)}| - \log_2 (\mathsf{pivot})$. To maintain U_{m^*} unchanged, we obtain

$$2^{m^*} \times \mathsf{roundValue} \leq 2^{m^*} \times \frac{1+\varepsilon}{2}\mathsf{pivot} \leq (1+\varepsilon)|\mathsf{sol(F)}| \tag{7}$$

where the last inequality follows from $m^* \leq \log_2 |\mathsf{sol(F)}| - \log_2 (\mathsf{pivot}) + 1$. Combining Eqs. 6 and 7 together, we obtain

$$2^{m^*} \times \frac{1}{1+\varepsilon}\mathsf{pivot} \leq 2^{m^*} \times \frac{1+\varepsilon}{2}\mathsf{pivot}$$

which gives us $\varepsilon \geq \sqrt{2} - 1$. Similarly, we can derive other breakpoints.

6 Experimental Evaluation

It is perhaps worth highlighting that both ApproxMCCore and ApproxMC6Core invoke the underlying SAT solver on identical queries; the only difference between ApproxMC6 and ApproxMC lies in what estimate to return and how often ApproxMCCore and ApproxMC6Core are invoked. From this viewpoint, one would expect that theoretical improvements would also lead to improved runtime performance. To provide further evidence, we perform extensive empirical evaluation and compare ApproxMC6's performance against the current state-of-the-art model counter, ApproxMC [22]. We use Arjun as a pre-processing tool. We used the latest version of ApproxMC, called ApproxMC4; an entry based on ApproxMC4 won the Model Counting Competition 2022.

Previous comparisons of ApproxMC have been performed on a set of 1896 instances, but the latest version of ApproxMC is able to solve almost all the instances when these instances are pre-processed by Arjun. Therefore, we sought to construct a new comprehensive set of 1890 instances derived from various sources, including Model Counting Competitions 2020–2022 [12,15,16], program synthesis [1], quantitative control improvisation [13], quantification of software properties [26], and adaptive chosen ciphertext attacks [3]. As noted earlier, our technique extends to projected model counting, and our benchmark suite indeed comprises 772 projected model counting instances.

Experiments were conducted on a high-performance computer cluster, with each node consisting of 2xE5-2690v3 CPUs featuring 2 × 12 real cores and 96GB of RAM. For each instance, a counter was run on a single core, with a time limit of 5000 s and a memory limit of 4 GB. To compare runtime performance, we use the PAR-2 score, a standard metric in the SAT community. Each instance is assigned a score that is the number of seconds it takes the corresponding tool to complete execution successfully. In the event of a timeout or memory out, the score is the doubled time limit in seconds. The PAR-2 score is then calculated as the average of all the instance scores. We also report the speedup of ApproxMC6 over ApproxMC4, calculated as the ratio of the runtime of ApproxMC4 to that of ApproxMC6 on instances solved by both counters. We set δ to 0.001 and ε to 0.8.

Specifically, we aim to address the following research questions:

RQ 1. How does the runtime performance of ApproxMC6 compare to that of ApproxMC4?

RQ 2. How does the accuracy of the counts computed by ApproxMC6 compare to that of the exact count?

Summary. In summary, ApproxMC6 consistently outperforms ApproxMC4. Specifically, it solved 204 additional instances and reduced the PAR-2 score by 1063 s in comparison to ApproxMC4. The average speedup of ApproxMC6 over ApproxMC4 was 4.68. In addition, ApproxMC6 provided a high-quality approximation with an average observed error of 0.1, much smaller than the theoretical error tolerance of 0.8.

6.1 RQ1. Overall Performance

Figure 2 compares the counting time of ApproxMC6 and ApproxMC4. The x-axis represents the index of the instances, sorted in ascending order of runtime, and the y-axis represents the runtime for each instance. A point (x, y) indicates that a counter can solve x instances within y seconds. Thus, for a given time limit y, a counter whose curve is on the right has solved more instances than a counter on the left. It can be seen in the figure that ApproxMC6 consistently outperforms ApproxMC4. In total, ApproxMC6 solved 204 more instances than ApproxMC4.

Table 1 provides a detailed comparison between ApproxMC6 and ApproxMC4. The first column lists three measures of interest: the number of solved instances, the PAR-2 score, and the speedup of ApproxMC6 over ApproxMC4. The second and third columns show the results for ApproxMC4 and ApproxMC6, respectively. The second column indicates that ApproxMC4 solved 998 of the 1890 instances and achieved a PAR-2 score of 4934. The third column shows that ApproxMC6 solved 1202 instances and achieved a PAR-2 score of 3871. In comparison, ApproxMC6 solved 204 more instances and reduced the PAR-2 score by 1063 s in comparison to ApproxMC4. The geometric mean of the speedup for ApproxMC6 over ApproxMC4 is 4.68. This speedup was calculated only for instances solved by both counters.

6.2 RQ2. Approximation Quality

We used the state-of-the-art probabilistic exact model counter Ganak to compute the exact model count and compare it to the results of ApproxMC6. We collected statistics on instances solved by both Ganak and ApproxMC6. Figure 3 presents results for a subset of instances. The x-axis represents the index of instances

Table 1. The number of solved instances and PAR-2 score for ApproxMC6 versus ApproxMC4 on 1890 instances. The geometric mean of the speedup of ApproxMC6 over ApproxMC4 is also reported.

	ApproxMC4	ApproxMC6
# Solved	998	1202
PAR-2 score	4934	3871
Speedup	—	4.68

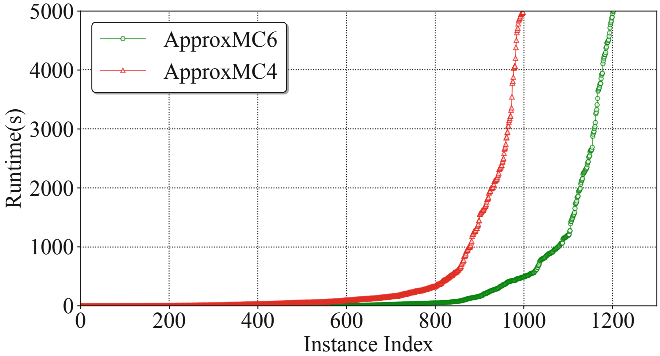


Fig. 2. Comparison of counting times for ApproxMC6 and ApproxMC4.

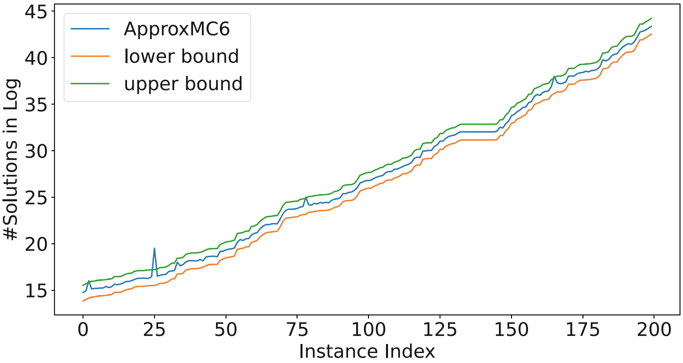


Fig. 3. Comparison of approximate counts from ApproxMC6 to exact counts from Ganak.

sorted in ascending order by the number of solutions, and the y-axis represents the number of solutions in a log scale. Theoretically, the approximate count from ApproxMC6 should be within the range of $|\mathsf{sol}(\mathsf{F})| \cdot 1.8$ and $|\mathsf{sol}(\mathsf{F})|/1.8$ with probability 0.999, where $|\mathsf{sol}(\mathsf{F})|$ denotes the exact count returned by Ganak. The range is indicated by the upper and lower bounds, represented by the curves $y = |\mathsf{sol}(\mathsf{F})| \cdot 1.8$ and $y = |\mathsf{sol}(\mathsf{F})|/1.8$, respectively. Figure 3 shows that the approximate counts from ApproxMC6 fall within the expected range $[|\mathsf{sol}(\mathsf{F})|/1.8, |\mathsf{sol}(\mathsf{F})| \cdot 1.8]$ for all instances except for four points slightly above the upper bound. These four outliers are due to a bug in the preprocessor Arjun that probably depends on the version of the C++ compiler and will be fixed in the future. We also calculated the observed error, which is the mean relative difference between the approximate and exact counts in our experiments, i.e., $\max\{\mathsf{finalEstimate}/|\mathsf{sol}(\mathsf{F})| - 1, |\mathsf{sol}(\mathsf{F})|/\mathsf{finalEstimate} - 1\}$. The overall observed error was 0.1, which is significantly smaller than the theoretical error tolerance of 0.8.

7 Conclusion

In this paper, we addressed the scalability challenges faced by ApproxMC in the smaller δ range. To this end, we proposed a *rounding*-based algorithm, ApproxMC6, which reduces the number of estimations required by 84% while providing the same (ε, δ)-guarantees. Our empirical evaluation on 1890 instances shows that ApproxMC6 solved 204 more instances and achieved a reduction in PAR-2 score of 1063 s. Furthermore, ApproxMC6 achieved a 4× speedup over ApproxMC on the instances both ApproxMC6 and ApproxMC could solve.

Acknowledgements. This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004], Ministry of Education Singapore Tier 2 Grant [MOE-T2EP20121-0011], and Ministry of Education Singapore Tier 1 Grant [R-252-000-B59-114]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg. We are thankful to Yash Pote for the insightful early discussions that helped shape the idea. We are grateful to Tim van Bremen for his detailed feedback on the early drafts of the paper. We sincerely appreciate the anonymous reviewers for their constructive comments to enhance this paper.

A Proof of Proposition 1

Proof. For $\forall y \in \{0,1\}^n, \alpha^{(m)} \in \{0,1\}^m$, let $\gamma_{y,\alpha^{(m)}}$ be an indicator variable that is 1 when $h^{(m)}(y) = \alpha^{(m)}$. According to the definition of strongly 2-universal function, we obtain $\forall x, y \in \{0,1\}^n$, $\mathsf{E}\left[\gamma_{y,\alpha^{(m)}}\right] = \frac{1}{2^m}$ and $\mathsf{E}\left[\gamma_{x,\alpha^{(m)}} \cdot \gamma_{y,\alpha^{(m)}}\right] = \frac{1}{2^{2m}}$. To prove Eq. 1, we obtain

$$\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] = \mathsf{E}\left[\sum_{y\in\mathsf{sol}(\mathsf{F})} \gamma_{y,\alpha^{(m)}}\right] = \sum_{y\in\mathsf{sol}(\mathsf{F})} \mathsf{E}\left[\gamma_{y,\alpha^{(m)}}\right] = \frac{|\mathsf{sol}(\mathsf{F})|}{2^m}$$

To prove Eq. 2, we derive

$$\begin{aligned}
\mathsf{E}\left[\mathsf{Cnt}^2_{\langle F,m\rangle}\right] &= \mathsf{E}\left[\sum_{y\in\mathsf{sol(F)}} \gamma^2_{y,\alpha^{(m)}} + \sum_{x\neq y\in\mathsf{sol(F)}} \gamma_{x,\alpha^{(m)}}\cdot\gamma_{y,\alpha^{(m)}}\right] \\
&= \mathsf{E}\left[\sum_{y\in\mathsf{sol(F)}} \gamma_{y,\alpha^{(m)}}\right] + \sum_{x\neq y\in\mathsf{sol(F)}} \mathsf{E}\left[\gamma_{x,\alpha^{(m)}}\cdot\gamma_{y,\alpha^{(m)}}\right] \\
&= \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] + \frac{|\mathsf{sol(F)}|(|\mathsf{sol(F)}|-1)}{2^{2m}}
\end{aligned}$$

Then, we obtain

$$\begin{aligned}
\sigma^2\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] &= \mathsf{E}\left[\mathsf{Cnt}^2_{\langle F,m\rangle}\right] - \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]^2 \\
&= \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] + \frac{|\mathsf{sol(F)}|(|\mathsf{sol(F)}|-1)}{2^{2m}} - \left(\frac{|\mathsf{sol(F)}|}{2^m}\right)^2 \\
&= \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right] - \frac{|\mathsf{sol(F)}|}{2^{2m}} \\
&\leq \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m\rangle}\right]
\end{aligned}$$

□

B Weakness of Proposition 3

The following proposition states that Proposition 3 provides a loose upper bound for $\Pr[\mathsf{Error}_t]$.

Proposition 4. *Assuming t is odd, we have:*

$$Pr[\mathsf{Error}_t] < \eta(t, \lceil t/2\rceil, Pr[L\cup U])$$

Proof. We will now construct a case counted by $\eta(t, \lceil t/2\rceil, \Pr[L\cup U])$ but not contained within the event Error_t. Let I_i^L be an indicator variable that is 1 when $\mathsf{ApproxMCCore}$ returns a nSols less than $\frac{|\mathsf{sol(F)}|}{1+\varepsilon}$, indicating the occurrence of event L in the i-th repetition. Let I_i^U be an indicator variable that is 1 when $\mathsf{ApproxMCCore}$ returns a nSols greater than $(1+\varepsilon)|\mathsf{sol(F)}|$, indicating the occurrence of event U in the i-th repetition. Consider a scenario where $I_i^L = 1$ for $i = 1, 2, ..., \left\lceil\frac{t}{4}\right\rceil$, $I_j^U = 1$ for $j = \left\lceil\frac{t}{4}\right\rceil + 1, ..., \left\lceil\frac{t}{2}\right\rceil$, and $I_k^L = I_k^U = 0$ for $k > \left\lceil\frac{t}{2}\right\rceil$. $\eta(t, \lceil t/2\rceil, \Pr[L\cup U])$ represents $\sum_{i=1}^t (I_i^L \vee I_i^U) \geq \left\lceil\frac{t}{2}\right\rceil$. We can see that this case is included in $\sum_{i=1}^t (I_i^L \vee I_i^U) \geq \left\lceil\frac{t}{2}\right\rceil$ and therefore counted by $\eta(t, \lceil t/2\rceil, \Pr[L\cup U])$ since there are $\left\lceil\frac{t}{2}\right\rceil$ estimates outside the PAC range. However, this case means that $\left\lceil\frac{t}{4}\right\rceil$ estimates fall within the range less than $\frac{|\mathsf{sol(F)}|}{1+\varepsilon}$ and $\left\lceil\frac{t}{2}\right\rceil - \left\lceil\frac{t}{4}\right\rceil$ estimates fall within the range greater than $(1+\varepsilon)|\mathsf{sol(F)}|$, while the remaining $\left\lfloor\frac{t}{2}\right\rfloor$ estimates correctly fall within the range $\left[\frac{|\mathsf{sol(F)}|}{1+\varepsilon}, (1+\varepsilon)|\mathsf{sol(F)}|\right]$.

Therefore, after sorting all the estimates, $\mathsf{ApproxMC6}$ returns a correct estimate since the median falls within the PAC range $\left[\frac{|\mathsf{sol(F)}|}{1+\varepsilon}, (1+\varepsilon)|\mathsf{sol(F)}|\right]$. In other words, this case is out of the event Error_t. In conclusion, there is a scenario that is out of the event Error_t, undesirably included in expression $\sum_{i=1}^{t}(I_i^L \vee I_i^U) \geq \lceil \frac{t}{2} \rceil$ and counted by $\eta(t, \lceil t/2 \rceil, \Pr[L \cup U])$, which means $\Pr[\mathsf{Error}_t]$ is strictly less than $\eta(t, \lceil t/2 \rceil, \Pr[L \cup U])$. □

C Proof of $p_{max} \leq 0.36$ for ApproxMC

Proof. We prove the case of $\sqrt{2}-1 \leq \varepsilon < 1$. Similarly to the proof in Sect. 5.3, we aim to bound $\Pr[L]$ by the following equation:

$$\Pr[L] = \left[\bigcup_{i \in \{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap L_i\right)\right] \qquad (3 \text{ revisited})$$

which can be simplified by three observations labeled $O1, O2$ and $O3$ below.

$O1 : \forall i \leq m^* - 3, T_i \subseteq T_{i+1}$. Therefore,

$$\bigcup_{i \in \{1,\ldots,m^*-3\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq \bigcup_{i \in \{1,\ldots,m^*-3\}} T_i \subseteq T_{m^*-3}$$

$O2$: For $i \in \{m^*-2, m^*-1\}$, we have

$$\bigcup_{i \in \{m^*-2,m^*-1\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq L_{m^*-2} \cup L_{m^*-1}$$

$O3$: $\forall i \geq m^*$, $\overline{T_i}$ implies $\mathsf{Cnt}_{\langle F,i \rangle} > \mathsf{thresh}$ and then we have $2^i \times \mathsf{Cnt}_{\langle F,i \rangle} > 2^{m^*} \times \mathsf{thresh} \geq |\mathsf{sol(F)}|\left(1+\frac{\varepsilon}{1+\varepsilon}\right)$. The second inequality follows from $m^* \geq \log_2 |\mathsf{sol(F)}| - \log_2(\mathsf{pivot})$. Then we obtain $\left(\mathsf{Cnt}_{\langle F,i \rangle} > \mathsf{E}\left[\mathsf{Cnt}_{\langle F,i \rangle}\right]\left(1+\frac{\varepsilon}{1+\varepsilon}\right)\right)$. Therefore, $\overline{T_i} \subseteq U'_i$ for $i \geq m^*$. Since $\forall i, \overline{T_i} \subseteq \overline{T_{i-1}}$, we have

$$\begin{aligned}
\bigcup_{i \in \{m^*,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap L_i\right) &\subseteq \bigcup_{i \in \{m^*+1,\ldots,n\}} \overline{T_{i-1}} \cup (\overline{T_{m^*-1}} \cap T_{m^*} \cap L_{m^*}) \\
&\subseteq \overline{T_{m^*}} \cup (\overline{T_{m^*-1}} \cap T_{m^*} \cap L_{m^*}) \\
&\subseteq \overline{T_{m^*}} \cup L_{m^*} \\
&\subseteq U'_{m^*} \cup L_{m^*}
\end{aligned}$$

Following the observations $O1, O2$ and $O3$, we simplify Eq. 3 and obtain

$$\Pr[L] \leq \Pr[T_{m^*-3}] + \Pr[L_{m^*-2}] + \Pr[L_{m^*-1}] + \Pr[U'_{m^*} \cup L_{m^*}]$$

Employing Lemma 2 in [8] gives $\Pr[L] \leq 0.36$. Note that U in [8] represents U' of our definition.

Then, following the $O4$ and $O5$ in Sect. 5.3, we obtain

$$\Pr[U] \leq \Pr[U'_{m^*}]$$

Employing Lemma 6 gives $\Pr[U] \leq 0.169$. As a result, $p_{max} \leq 0.36$. □

D Proof of Lemma 4

We restate the lemma below and prove the statements section by section. The proof for $\sqrt{2}-1 \le \varepsilon < 1$ has been shown in Sect. 5.3.

Lemma 4. *The following bounds hold for* ApproxMC6:

$$\Pr\left[L\right] \le \begin{cases} 0.262 & \text{if } \varepsilon < \sqrt{2}-1 \\ 0.157 & \text{if } \sqrt{2}-1 \le \varepsilon < 1 \\ 0.085 & \text{if } 1 \le \varepsilon < 3 \\ 0.055 & \text{if } 3 \le \varepsilon < 4\sqrt{2}-1 \\ 0.023 & \text{if } \varepsilon \ge 4\sqrt{2}-1 \end{cases}$$

$$\Pr\left[U\right] \le \begin{cases} 0.169 & \text{if } \varepsilon < 3 \\ 0.044 & \text{if } \varepsilon \ge 3 \end{cases}$$

D.1 Proof of $\Pr\left[L\right] \le 0.262$ for $\varepsilon < \sqrt{2}-1$

We first consider two cases: $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*\rangle}\right] < \frac{1+\varepsilon}{2}\mathsf{thresh}$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*\rangle}\right] \ge \frac{1+\varepsilon}{2}\mathsf{thresh}$, and then merge the results to complete the proof.

Case 1: $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*\rangle}\right] < \frac{1+\varepsilon}{2}$ thresh

Lemma 7. *Given $\varepsilon < \sqrt{2}-1$, the following bounds hold:*

1. *$Pr\left[T_{m^*-2}\right] \le \frac{1}{29.67}$*
2. *$Pr\left[L_{m^*-1}\right] \le \frac{1}{10.84}$*

Proof. Let's first prove the statement 1. For $\varepsilon < \sqrt{2}-1$, we have $\mathsf{thresh} < (2-\frac{\sqrt{2}}{2})\mathsf{pivot}$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-2\rangle}\right] \ge 2\mathsf{pivot}$. Therefore, $\Pr\left[T_{m^*-2}\right] \le \Pr\left[\mathsf{Cnt}_{\langle F,m^*-2\rangle} \le (1-\frac{\sqrt{2}}{4})\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-2\rangle}\right]\right]$. Finally, employing Lemma 5 with $\beta = 1-\frac{\sqrt{2}}{4}$, we obtain $\Pr\left[T_{m^*-2}\right] \le \frac{1}{1+(\frac{\sqrt{2}}{4})^2\cdot 2\mathsf{pivot}} \le \frac{1}{1+(\frac{\sqrt{2}}{4})^2\cdot 2\cdot 9.84\cdot(1+\frac{1}{\sqrt{2}-1})^2} \le \frac{1}{29.67}$. To prove the statement 2, we employ Lemma 5 with $\beta = \frac{1}{1+\varepsilon}$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-1\rangle}\right] \ge \mathsf{pivot}$ to obtain $\Pr\left[L_{m^*-1}\right] \le \frac{1}{1+(1-\frac{1}{1+\varepsilon})^2\cdot\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-1\rangle}\right]} \le \frac{1}{1+(1-\frac{1}{1+\varepsilon})^2\cdot 9.84\cdot(1+\frac{1}{\varepsilon})^2} = \frac{1}{10.84}$. □

Then, we prove that $\Pr\left[L\right] \le 0.126$ for $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*\rangle}\right] < \frac{1+\varepsilon}{2}\mathsf{thresh}$.

Proof. We aim to bound $\Pr\left[L\right]$ by the following equation:

$$\Pr\left[L\right] = \left[\bigcup_{i\in\{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap L_i\right)\right] \qquad (3 \text{ revisited})$$

which can be simplified by the three observations labeled $O1, O2$ and $O3$ below.

$O1$: $\forall i \leq m^* - 2, T_i \subseteq T_{i+1}$. Therefore,

$$\bigcup_{i \in \{1,\ldots,m^*-2\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq \bigcup_{i \in \{1,\ldots,m^*-2\}} T_i \subseteq T_{m^*-2}$$

$O2$: For $i = m^* - 1$, we have

$$\overline{T_{m^*-2}} \cap T_{m^*-1} \cap L_{m^*-1} \subseteq L_{m^*-1}$$

$O3$: $\forall i \geq m^*$, since rounding $\mathsf{Cnt}_{\langle F,i \rangle}$ up to $\frac{\sqrt{1+2\varepsilon}}{2}\mathsf{pivot}$, we have $\mathsf{Cnt}_{\langle F,i \rangle} \geq \frac{\sqrt{1+2\varepsilon}}{2}\mathsf{pivot} \geq \frac{\mathsf{thresh}}{2} > \frac{\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right]}{1+\varepsilon} \geq \frac{\mathsf{E}\left[\mathsf{Cnt}_{\langle F,i \rangle}\right]}{1+\varepsilon}$. The second last inequality follows from $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right] < \frac{1+\varepsilon}{2}\mathsf{thresh}$. Therefore, $L_i = \emptyset$ for $i \geq m^*$ and we have

$$\bigcup_{i \in \{m^*,\ldots,n\}} (\overline{T_{i-1}} \cap T_i \cap L_i) = \emptyset$$

Following the observations $O1, O2$ and $O3$, we simplify Eq. 3 and obtain

$$\Pr\left[L\right] \leq \Pr\left[T_{m^*-2}\right] + \Pr\left[L_{m^*-1}\right]$$

Employing Lemma 7 gives $\Pr\left[L\right] \leq 0.126$. □

Case 2: $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right] \geq \frac{1+\varepsilon}{2}$ thresh

Lemma 8. *Given* $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right] \geq \frac{1+\varepsilon}{2}\mathsf{thresh}$*, the following bounds hold:*

1. $Pr\left[T_{m^*-1}\right] \leq \frac{1}{10.84}$
2. $Pr\left[L_{m^*}\right] \leq \frac{1}{5.92}$

Proof. Let's first prove the statement 1. From $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right] \geq \frac{1+\varepsilon}{2}\mathsf{thresh}$, we can derive $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-1 \rangle}\right] \geq (1+\varepsilon)\mathsf{thresh}$. Therefore, $\Pr\left[T_{m^*-1}\right] \leq \Pr\left[\mathsf{Cnt}_{\langle F,m^*-1 \rangle} \leq \frac{1}{1+\varepsilon}\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-1 \rangle}\right]\right]$. Finally, employing Lemma 5 with $\beta = \frac{1}{1+\varepsilon}$, we obtain $\Pr\left[T_{m^*-1}\right] \leq \frac{1}{1+(1-\frac{1}{1+\varepsilon})^2 \cdot \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-1 \rangle}\right]} \leq \frac{1}{1+(1-\frac{1}{1+\varepsilon})^2 \cdot (1+\varepsilon)\mathsf{thresh}} = \frac{1}{1+9.84(1+2\varepsilon)} \leq \frac{1}{10.84}$. To prove the statement 2, we employ Lemma 5 with $\beta = \frac{1}{1+\varepsilon}$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right] \geq \frac{1+\varepsilon}{2}\mathsf{thresh}$ to obtain $\Pr\left[L_{m^*}\right] \leq \frac{1}{1+(1-\frac{1}{1+\varepsilon})^2 \cdot \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right]} \leq \frac{1}{1+(1-\frac{1}{1+\varepsilon})^2 \cdot \frac{1+\varepsilon}{2}\mathsf{thresh}} = \frac{1}{1+4.92(1+2\varepsilon)} \leq \frac{1}{5.92}$. □

Then, we prove that $\Pr\left[L\right] \leq 0.262$ for $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right] \geq \frac{1+\varepsilon}{2}\mathsf{thresh}$.

Proof. We aim to bound $\Pr\left[L\right]$ by the following equation:

$$\Pr\left[L\right] = \left[\bigcup_{i \in \{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap L_i\right)\right] \qquad (3 \text{ revisited})$$

which can be simplified by the three observations labeled $O1, O2$ and $O3$ below.

$O1$: $\forall i \leq m^* - 1, T_i \subseteq T_{i+1}$. Therefore,

$$\bigcup_{i \in \{1,\ldots,m^*-1\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq \bigcup_{i \in \{1,\ldots,m^*-1\}} T_i \subseteq T_{m^*-1}$$

$O2$: For $i = m^*$, we have

$$\overline{T_{m^*-1}} \cap T_{m^*} \cap L_{m^*} \subseteq L_{m^*}$$

$O3$: $\forall i \geq m^* + 1$, since rounding $\mathsf{Cnt}_{\langle F,i \rangle}$ up to $\frac{\sqrt{1+2\varepsilon}}{2}\mathsf{pivot}$ and $m^* \geq \log_2 |\mathsf{sol(F)}| - \log_2 (\mathsf{pivot})$, we have $2^i \times \mathsf{Cnt}_{\langle F,i \rangle} \geq 2^{m^*+1} \times \frac{\sqrt{1+2\varepsilon}}{2}\mathsf{pivot} \geq \sqrt{1+2\varepsilon}|\mathsf{sol(F)}| \geq \frac{|\mathsf{sol(F)}|}{1+\varepsilon}$. Then we have $\left(\mathsf{Cnt}_{\langle F,i \rangle} \geq \frac{\mathsf{E}\left[\mathsf{Cnt}_{\langle F,i \rangle}\right]}{1+\varepsilon}\right)$. Therefore, $L_i = \emptyset$ for $i \geq m^* + 1$ and we have

$$\bigcup_{i \in \{m^*+1,\ldots,n\}} (\overline{T_{i-1}} \cap T_i \cap L_i) = \emptyset$$

Following the observations $O1, O2$ and $O3$, we simplify Eq. 3 and obtain

$$\Pr[L] \leq \Pr[T_{m^*-1}] + \Pr[L_{m^*}]$$

Employing Lemma 8 gives $\Pr[L] \leq 0.262$. □

Combining the Case 1 and 2, we obtain $\Pr[L] \leq \max\{0.126, 0.262\} = 0.262$. Therefore, we prove the statement for ApproxMC6: $\Pr[L] \leq 0.262$ for $\varepsilon < \sqrt{2}-1$.

D.2 Proof of $\Pr[L] \leq 0.085$ for $1 \leq \varepsilon < 3$

Lemma 9. *Given $1 \leq \varepsilon < 3$, the following bounds hold:*

1. $Pr[T_{m^*-4}] \leq \frac{1}{86.41}$
2. $Pr[L_{m^*-3}] \leq \frac{1}{40.36}$
3. $Pr[L_{m^*-2}] \leq \frac{1}{20.68}$

Proof. Let's first prove the statement 1. For $\varepsilon < 3$, we have $\mathsf{thresh} < \frac{7}{4}\mathsf{pivot}$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-4 \rangle}\right] \geq 8\mathsf{pivot}$. Therefore, $\Pr[T_{m^*-4}] \leq \Pr\left[\mathsf{Cnt}_{\langle F,m^*-4 \rangle} \leq \frac{7}{32}\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-4 \rangle}\right]\right]$. Finally, employing Lemma 5 with $\beta = \frac{7}{32}$, we obtain $\Pr[T_{m^*-4}] \leq \frac{1}{1+(1-\frac{7}{32})^2 \cdot 8\mathsf{pivot}} \leq \frac{1}{1+(1-\frac{7}{32})^2 \cdot 8 \cdot 9.84 \cdot (1+\frac{1}{3})^2} \leq \frac{1}{86.41}$. To prove the statement 2, we employ Lemma 5 with $\beta = \frac{1}{1+\varepsilon}$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-3 \rangle}\right] \geq 4\mathsf{pivot}$ to obtain $\Pr[L_{m^*-3}] \leq \frac{1}{1+(1-\frac{1}{1+\varepsilon})^2 \cdot \mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-3 \rangle}\right]} \leq \frac{1}{1+(1-\frac{1}{1+\varepsilon})^2 \cdot 4 \cdot 9.84 \cdot (1+\frac{1}{\varepsilon})^2} = \frac{1}{40.36}$. Following the proof of Lemma 2 in [8] we can prove the statement 3. □

Now let us prove the statement for ApproxMC6: $\Pr[L] \leq 0.085$ for $1 \leq \varepsilon < 3$.

Proof. We aim to bound $\Pr[L]$ by the following equation:

$$\Pr[L] = \left[\bigcup_{i \in \{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap L_i \right) \right] \qquad \text{(3 revisited)}$$

which can be simplified by the three observations labeled $O1, O2$ and $O3$ below.

$O1 : \forall i \leq m^* - 4, T_i \subseteq T_{i+1}$. Therefore,

$$\bigcup_{i \in \{1,\ldots,m^*-4\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq \bigcup_{i \in \{1,\ldots,m^*-4\}} T_i \subseteq T_{m^*-4}$$

$O2$: For $i \in \{m^* - 3, m^* - 2\}$, we have

$$\bigcup_{i \in \{m^*-3,m^*-2\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq L_{m^*-3} \cup L_{m^*-2}$$

$O3 : \forall i \geq m^* - 1$, since rounding $\mathsf{Cnt}_{\langle F,i \rangle}$ up to pivot and $m^* \geq \log_2 |\mathsf{sol(F)}| - \log_2 (\mathsf{pivot})$, we have $2^i \times \mathsf{Cnt}_{\langle F,i \rangle} \geq 2^{m^*-1} \times \mathsf{pivot} \geq \frac{|\mathsf{sol(F)}|}{2} \geq \frac{|\mathsf{sol(F)}|}{1+\varepsilon}$. The last inequality follows from $\varepsilon \geq 1$. Then we have $\left(\mathsf{Cnt}_{\langle F,i \rangle} \geq \frac{\mathsf{E}\left[\mathsf{Cnt}_{\langle F,i \rangle}\right]}{1+\varepsilon} \right)$. Therefore, $L_i = \emptyset$ for $i \geq m^* - 1$ and we have

$$\bigcup_{i \in \{m^*-1,\ldots,n\}} (\overline{T_{i-1}} \cap T_i \cap L_i) = \emptyset$$

Following the observations $O1, O2$ and $O3$, we simplify Eq. 3 and obtain

$$\Pr[L] \leq \Pr[T_{m^*-4}] + \Pr[L_{m^*-3}] + \Pr[L_{m^*-2}]$$

Employing Lemma 9 gives $\Pr[L] \leq 0.085$. □

D.3 Proof of $\Pr[L] \leq 0.055$ for $3 \leq \varepsilon < 4\sqrt{2} - 1$

Lemma 10. *Given* $3 \leq \varepsilon < 4\sqrt{2} - 1$*, the following bound hold:*

$$Pr[T_{m^*-3}] \leq \frac{1}{18.19}$$

Proof. For $\varepsilon < 4\sqrt{2} - 1$, we have $\mathsf{thresh} < (2 - \frac{\sqrt{2}}{8})\mathsf{pivot}$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-3 \rangle}\right] \geq 4\mathsf{pivot}$. Therefore, $\Pr[T_{m^*-3}] \leq \Pr\left[\mathsf{Cnt}_{\langle F,m^*-3 \rangle} \leq (\frac{1}{2} - \frac{\sqrt{2}}{32})\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-3 \rangle}\right]\right]$. Finally, employing Lemma 5 with $\beta = \frac{1}{2} - \frac{\sqrt{2}}{32}$, we obtain $\Pr[T_{m^*-3}] \leq \frac{1}{1+(1-(\frac{1}{2}-\frac{\sqrt{2}}{32}))^2 \cdot 4\mathsf{pivot}} \leq \frac{1}{1+(1-(\frac{1}{2}-\frac{\sqrt{2}}{32}))^2 \cdot 4 \cdot 9.84 \cdot (1+\frac{1}{4\sqrt{2}-1})^2} \leq \frac{1}{18.19}$. □

Now let us prove the statement for $\mathsf{ApproxMC6}$: $\Pr[L] \leq 0.055$ for $3 \leq \varepsilon < 4\sqrt{2} - 1$.

Proof. We aim to bound $\Pr[L]$ by the following equation:

$$\Pr[L] = \left[\bigcup_{i\in\{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap L_i\right)\right] \tag{3 revisited}$$

which can be simplified by the two observations labeled $O1$ and $O2$ below.

$O1 : \forall i \leq m^* - 3, T_i \subseteq T_{i+1}$. Therefore,

$$\bigcup_{i\in\{1,\ldots,m^*-3\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq \bigcup_{i\in\{1,\ldots,m^*-3\}} T_i \subseteq T_{m^*-3}$$

$O2 : \forall i \geq m^* - 2$, since rounding $\mathsf{Cnt}_{\langle F,i\rangle}$ to pivot and $m^* \geq \log_2 |\mathsf{sol(F)}| - \log_2(\mathsf{pivot})$, we have $2^i \times \mathsf{Cnt}_{\langle F,i\rangle} \geq 2^{m^*-2} \times \mathsf{pivot} \geq \frac{|\mathsf{sol(F)}|}{4} \geq \frac{|\mathsf{sol(F)}|}{1+\varepsilon}$. The last inequality follows from $\varepsilon \geq 3$. Then we have $\left(\mathsf{Cnt}_{\langle F,i\rangle} \geq \frac{\mathsf{E}\left[\mathsf{Cnt}_{\langle F,i\rangle}\right]}{1+\varepsilon}\right)$. Therefore, $L_i = \emptyset$ for $i \geq m^* - 2$ and we have

$$\bigcup_{i\in\{m^*-2,\ldots,n\}} (\overline{T_{i-1}} \cap T_i \cap L_i) = \emptyset$$

Following the observations $O1$ and $O2$, we simplify Eq. 3 and obtain

$$\Pr[L] \leq \Pr[T_{m^*-3}]$$

Employing Lemma 10 gives $\Pr[L] \leq 0.055$. □

D.4 Proof of $\Pr[L] \leq 0.023$ for $\varepsilon \geq 4\sqrt{2} - 1$

Lemma 11. *Given $\varepsilon \geq 4\sqrt{2} - 1$, the following bound hold:*

$$Pr[T_{m^*-4}] \leq \frac{1}{45.28}$$

Proof. We have $\mathsf{thresh} < 2\mathsf{pivot}$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-4\rangle}\right] \geq 8\mathsf{pivot}$. Therefore, $\Pr[T_{m^*-4}] \leq \Pr\left[\mathsf{Cnt}_{\langle F,m^*-4\rangle} \leq \frac{1}{4}\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*-4\rangle}\right]\right]$. Finally, employing Lemma 5 with $\beta = \frac{1}{4}$, we obtain $\Pr[T_{m^*-4}] \leq \frac{1}{1+(1-\frac{1}{4})^2\cdot 8\mathsf{pivot}} \leq \frac{1}{1+(1-\frac{1}{4})^2\cdot 8\cdot 9.84} \leq \frac{1}{45.28}$. □

Now let us prove the statement for $\mathsf{ApproxMC6}$: $\Pr[L] \leq 0.023$ for $\varepsilon \geq 4\sqrt{2}-1$.

Proof. We aim to bound $\Pr[L]$ by the following equation:

$$\Pr[L] = \left[\bigcup_{i\in\{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap L_i\right)\right] \tag{3 revisited}$$

which can be simplified by the two observations labeled $O1$ and $O2$ below.

$O1: \forall i \leq m^* - 4, T_i \subseteq T_{i+1}$. Therefore,

$$\bigcup_{i \in \{1,\ldots,m^*-4\}} (\overline{T_{i-1}} \cap T_i \cap L_i) \subseteq \bigcup_{i \in \{1,\ldots,m^*-4\}} T_i \subseteq T_{m^*-4}$$

$O2: \forall i \geq m^* - 3$, since rounding $\mathsf{Cnt}_{\langle F,i \rangle}$ to $\sqrt{2}\mathsf{pivot}$ and $m^* \geq \log_2 |\mathsf{sol(F)}| - \log_2(\mathsf{pivot})$, we have $2^i \times \mathsf{Cnt}_{\langle F,i \rangle} \geq 2^{m^*-3} \times \sqrt{2}\mathsf{pivot} \geq \frac{\sqrt{2}|\mathsf{sol(F)}|}{8} \geq \frac{|\mathsf{sol(F)}|}{1+\varepsilon}$. The last inequality follows from $\varepsilon \geq 4\sqrt{2} - 1$. Then we have $\left(\mathsf{Cnt}_{\langle F,i \rangle} \geq \frac{\mathsf{E}\left[\mathsf{Cnt}_{\langle F,i \rangle}\right]}{1+\varepsilon}\right)$. Therefore, $L_i = \emptyset$ for $i \geq m^* - 3$ and we have

$$\bigcup_{i \in \{m^*-3,\ldots,n\}} (\overline{T_{i-1}} \cap T_i \cap L_i) = \emptyset$$

Following the observations $O1$ and $O2$, we simplify Eq. 3 and obtain

$$\Pr[L] \leq \Pr[T_{m^*-4}]$$

Employing Lemma 11 gives $\Pr[L] \leq 0.023$. □

D.5 Proof of $\Pr[U] \leq 0.169$ for $\varepsilon < 3$

Lemma 12

$$Pr[U'_{m^*}] \leq \frac{1}{5.92}$$

Proof. Employing Lemma 5 with $\gamma = (1 + \frac{\varepsilon}{1+\varepsilon})$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^* \rangle}\right] \geq \mathsf{pivot}/2$, we obtain $\Pr[U'_{m^*}] \leq \frac{1}{1+\left(\frac{\varepsilon}{1+\varepsilon}\right)^2 \mathsf{pivot}/2} \leq \frac{1}{1+9.84/2} \leq \frac{1}{5.92}$. □

Now let us prove the statement for ApproxMC6: $\Pr[U] \leq 0.169$ for $\varepsilon < 3$.

Proof. We aim to bound $\Pr[U]$ by the following equation:

$$\Pr[U] = \left[\bigcup_{i \in \{1,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap U_i\right)\right] \qquad \text{(4 revisited)}$$

We derive the following observations $O1$ and $O2$.

$O1: \forall i \leq m^* - 1$, since $m^* \leq \log_2 |\mathsf{sol(F)}| - \log_2(\mathsf{pivot}) + 1$, we have $2^i \times \mathsf{Cnt}_{\langle F,i \rangle} \leq 2^{m^*-1} \times \mathsf{thresh} \leq |\mathsf{sol(F)}| \left(1 + \frac{\varepsilon}{1+\varepsilon}\right)$. Then we obtain $\left(\mathsf{Cnt}_{\langle F,i \rangle} \leq \mathsf{E}\left[\mathsf{Cnt}_{\langle F,i \rangle}\right] \left(1 + \frac{\varepsilon}{1+\varepsilon}\right)\right)$. Therefore, $T_i \cap U'_i = \emptyset$ for $i \leq m^* - 1$ and we have

$$\bigcup_{i \in \{1,\ldots,m^*-1\}} \left(\overline{T_{i-1}} \cap T_i \cap U_i\right) \subseteq \bigcup_{i \in \{1,\ldots,m^*-1\}} \left(\overline{T_{i-1}} \cap T_i \cap U'_i\right) = \emptyset$$

$O2$: $\forall i \geq m^*$, $\overline{T_i}$ implies $\mathsf{Cnt}_{\langle F,i\rangle} > \mathsf{thresh}$ and then we have $2^i \times \mathsf{Cnt}_{\langle F,i\rangle} > 2^{m^*} \times \mathsf{thresh} \geq |\mathsf{sol(F)}|\left(1+\frac{\varepsilon}{1+\varepsilon}\right)$. The second inequality follows from $m^* \geq \log_2|\mathsf{sol(F)}| - \log_2(\mathsf{pivot})$. Then we obtain $\left(\mathsf{Cnt}_{\langle F,i\rangle} > \mathsf{E}\left[\mathsf{Cnt}_{\langle F,i\rangle}\right]\left(1+\frac{\varepsilon}{1+\varepsilon}\right)\right)$. Therefore, $\overline{T_i} \subseteq U'_i$ for $i \geq m^*$. Since $\forall i, \overline{T_i} \subseteq \overline{T_{i-1}}$, we have

$$\begin{aligned}
\bigcup_{i\in\{m^*,\ldots,n\}}\left(\overline{T_{i-1}} \cap T_i \cap U_i\right) &\subseteq \bigcup_{i\in\{m^*+1,\ldots,n\}} \overline{T_{i-1}} \cup (\overline{T_{m^*-1}} \cap T_{m^*} \cap U_{m^*})\\
&\subseteq \overline{T_{m^*}} \cup (\overline{T_{m^*-1}} \cap T_{m^*} \cap U_{m^*})\\
&\subseteq \overline{T_{m^*}} \cup U_{m^*}\\
&\subseteq U'_{m^*} \qquad (8)
\end{aligned}$$

Remark that for $\varepsilon < \sqrt{2}-1$, we round $\mathsf{Cnt}_{\langle F,m^*\rangle}$ up to $\frac{\sqrt{1+2\varepsilon}}{2}\mathsf{pivot}$ and we have $2^{m^*} \times \frac{\sqrt{1+2\varepsilon}}{2}\mathsf{pivot} \leq |\mathsf{sol(F)}|(1+\varepsilon)$. For $\sqrt{2}-1 \leq \varepsilon < 1$, we round $\mathsf{Cnt}_{\langle F,m^*\rangle}$ up to $\frac{\mathsf{pivot}}{\sqrt{2}}$ and we have $2^{m^*} \times \frac{\mathsf{pivot}}{\sqrt{2}} \leq |\mathsf{sol(F)}|(1+\varepsilon)$. For $1 \leq \varepsilon < 3$, we round $\mathsf{Cnt}_{\langle F,m^*\rangle}$ up to pivot and we have $2^{m^*} \times \mathsf{pivot} \leq |\mathsf{sol(F)}|(1+\varepsilon)$. The analysis means *rounding* doesn't affect the event U_{m^*} and therefore Inequality 8 still holds.

Following the observations $O1$ and $O2$, we simplify Eq. 4 and obtain

$$\Pr[U] \leq \Pr[U'_{m^*}]$$

Employing Lemma 12 gives $\Pr[U] \leq 0.169$. □

D.6 Proof of $\Pr[U] \leq 0.044$ for $\varepsilon \geq 3$

Lemma 13

$$Pr\left[\overline{T_{m^*+1}}\right] \leq \frac{1}{23.14}$$

Proof. Since $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*+1\rangle}\right] \leq \frac{\mathsf{pivot}}{2}$, we have $\Pr\left[\overline{T_{m^*+1}}\right] \leq \Pr\left[\mathsf{Cnt}_{\langle F,m^*+1\rangle} > 2(1+\frac{\varepsilon}{1+\varepsilon})\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*+1\rangle}\right]\right]$. Employing Lemma 5 with $\gamma = 2(1+\frac{\varepsilon}{1+\varepsilon})$ and $\mathsf{E}\left[\mathsf{Cnt}_{\langle F,m^*+1\rangle}\right] \geq \frac{\mathsf{pivot}}{4}$, we obtain $\Pr\left[\overline{T_{m^*+1}}\right] \leq \frac{1}{1+\left(1+\frac{2\varepsilon}{1+\varepsilon}\right)^2\mathsf{pivot}/4} = \frac{1}{1+2.46\cdot\left(3+\frac{1}{\varepsilon}\right)^2} \leq \frac{1}{1+2.46\cdot 3^2} \leq \frac{1}{23.14}$. □

Now let us prove the statement for $\mathsf{ApproxMC6}$: $\Pr[U] \leq 0.044$ for $\varepsilon \geq 3$.

Proof. We aim to bound $\Pr[U]$ by the following equation:

$$\Pr[U] = \left[\bigcup_{i\in\{1,\ldots,n\}}\left(\overline{T_{i-1}} \cap T_i \cap U_i\right)\right] \qquad (4 \text{ revisited})$$

We derive the following observations $O1$ and $O2$.

$O1$: $\forall i \leq m^*+1$, for $3 \leq \varepsilon < 4\sqrt{2}-1$, because we round $\mathsf{Cnt}_{\langle F,i \rangle}$ to pivot and have $m^* \leq \log_2 |\mathsf{sol(F)}| - \log_2(\mathsf{pivot}) + 1$, we obtain $2^i \times \mathsf{Cnt}_{\langle F,i \rangle} \leq 2^{m^*+1} \times \mathsf{pivot} \leq 4 \cdot |\mathsf{sol(F)}| \leq (1+\varepsilon)|\mathsf{sol(F)}|$. For $\varepsilon \geq 4\sqrt{2}-1$, we round $\mathsf{Cnt}_{\langle F,i \rangle}$ to $\sqrt{2}\mathsf{pivot}$ and obtain $2^i \times \mathsf{Cnt}_{\langle F,i \rangle} \leq 2^{m^*+1} \times \sqrt{2}\mathsf{pivot} \leq 4\sqrt{2} \cdot |\mathsf{sol(F)}| \leq (1+\varepsilon)|\mathsf{sol(F)}|$. Then, we obtain $\mathsf{Cnt}_{\langle F,i \rangle} \leq \mathsf{E}\left[\mathsf{Cnt}_{\langle F,i \rangle}\right](1+\varepsilon)$. Therefore, $U_i = \emptyset$ for $i \leq m^*+1$ and we have

$$\bigcup_{i \in \{1,\ldots,m^*+1\}} \left(\overline{T_{i-1}} \cap T_i \cap U_i\right) = \emptyset$$

$O2$: $\forall i \geq m^*+2$, since $\forall i, \overline{T_i} \subseteq \overline{T_{i-1}}$, we have

$$\bigcup_{i \in \{m^*+2,\ldots,n\}} \left(\overline{T_{i-1}} \cap T_i \cap U_i\right) \subseteq \bigcup_{i \in \{m^*+2,\ldots,n\}} \overline{T_{i-1}} \subseteq \overline{T_{m^*+1}}$$

Following the observations $O1$ and $O2$, we simplify Eq. 4 and obtain

$$\Pr[U] \leq \Pr\left[\overline{T_{m^*+1}}\right]$$

Employing Lemma 13 gives $\Pr[U] \leq 0.044$. □

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Proceedings of FMCAD (2013)
2. Baluta, T., Shen, S., Shine, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of CCS (2019)
3. Beck, G., Zinkus, M., Green, M.: Automating the development of chosen ciphertext attacks. In: Proceedings of USENIX Security (2020)
4. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. J. Comput. Syst. Sci. (1977)
5. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for SAT. In: Proceedings of AAAI (2014)
6. Chakraborty, S., Meel, K.S., Mistry, R., Vardi, M.Y.: Approximate probabilistic inference via word-level counting. In: Proceedings of AAAI (2016)
7. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Proceedings of CP (2013)
8. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic SAT calls. In: Proceedings of IJCAI (2016)
9. Duenas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Proceedings of AAAI (2017)
10. Ermon, S., Gomes, C.P., Sabharwal, A., Selman, B.: Embed and project: discrete sampling with universal hashing. In: Proceedings of NeurIPS (2013)
11. Ermon, S., Gomes, C.P., Sabharwal, A., Selman, B.: Taming the curse of dimensionality: discrete integration by hashing and optimization. In: Proceedings of ICML (2013)

12. Fichte, J.K., Hecher, M., Hamiti, F.: The model counting competition 2020. ACM J. Exp. Algorithmics (2021)
13. Gittis, A., Vin, E., Fremont, D.J.: Randomized synthesis for diversity and cost constraints with control improvisation. In: Proceedings of CAV (2022)
14. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting: a new strategy for obtaining good bounds. In: Proceedings of AAAI (2006)
15. Hecher, M., Fichte, J.K.: Model counting competition 2021 (2021). https://www.mccompetition.org/2021/mc_description
16. Hecher, M., Fichte, J.K.: Model counting competition 2022 (2022). https://mccompetition.org/2022/mc_description
17. Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. Constraints (2016)
18. Meel, K.S., Akshay, S.: Sparse hashing for scalable approximate model counting: theory and practice. In: Proceedings of LICS (2020)
19. Meel, K.S., et al.: Constrained sampling and counting: universal hashing meets sat solving. In: Proceedings of Workshop on Beyond NP(BNP) (2016)
20. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. (1996)
21. Sang, T., Bearne, P., Kautz, H.: Performing Bayesian inference by weighted model counting. In: Proceedings of AAAI (2005)
22. Soos, M., Gocht, S., Meel, K.S.: Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In: Proceedings of CAV (2020)
23. Soos, M., Meel, K.S.: Bird: engineering an efficient CNF-XOR sat solver and its applications to approximate model counting. In: Proceedings of AAAI (2019)
24. Soos, M., Meel, K.S.: Arjun: an efficient independent support computation technique and its applications to counting and sampling. In: Proceedings of ICCAD (2022)
25. Stockmeyer, L.: The complexity of approximate counting. In: Proceedings of STOC (1983)
26. Teuber, S., Weigl, A.: Quantifying software reliability via model-counting. In: Proceedings of QEST (2021)
27. Toda, S.: On the computational power of PP and (+)P. In: Proceedings of FOCS (1989)
28. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. (1979)
29. Yang, J., Chakraborty, S., Meel, K.S.: Projected model counting: beyond independent support. In: Proceedings of ATVA (2022)
30. Yang, J., Meel, K.S.: Engineering an efficient PB-XOR solver. In: Proceedings of CP (2021)

Satisfiability Modulo Finite Fields

Alex Ozdemir[1(✉)], Gereon Kremer[1,2], Cesare Tinelli[3], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
aozdemir@stanford.edu
[2] Certora, Tel Aviv-Yafo, Israel
[3] University of Iowa, Iowa, USA

Abstract. We study satisfiability modulo the theory of finite fields and give a decision procedure for this theory. We implement our procedure for prime fields inside the cvc5 SMT solver. Using this theory, we construct SMT queries that encode translation validation for various zero knowledge proof compilers applied to Boolean computations. We evaluate our procedure on these benchmarks. Our experiments show that our implementation is superior to previous approaches (which encode field arithmetic using integers or bit-vectors).

1 Introduction

Finite fields are critical to the design of recent cryptosystems. For instance, elliptic curve operations are defined in terms of operations in a finite field. Also, Zero-Knowledge Proofs (ZKPs) and Multi-Party Computations (MPCs), powerful tools for building secure and private systems, often require key properties of the system to be expressed as operations in a finite field.

Field-based cryptosystems already safeguard everything from our money to our privacy. Over 80% of our TLS connections, for example, use elliptic curves [4,66]. Private cryptocurrencies [32,59,89] built on ZKPs have billion-dollar market capitalizations [44,45]. And MPC protocols have been used to operate auctions [17], facilitate sensitive cross-agency collaboration in the US federal government [5], and compute cross-company pay gaps [8]. These systems safeguard our privacy, assets, and government data. Their importance justifies spending considerable effort to ensure that the systems are free of bugs that could compromise the resources they are trying to protect; thus, they are prime targets for formal verification.

However, verifying field-based cryptosystems is challenging, in part because current automated verification tools do not reason directly about finite fields. Many tools use Satisfiability Modulo Theories (SMT) solvers as a back-end [9,27,33,93,95]. SMT solvers [7,10,12,20,26,35,73,76,77] are automated reasoners that determine the satisfiability of formulas in first-order logic with respect to one or more *background theories*. They combine propositional search with specialized reasoning procedures for these theories, which model common data types such as Booleans, integers, reals, bit-vectors, arrays, algebraic datatypes, and more.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 163–186, 2023.
https://doi.org/10.1007/978-3-031-37703-7_8

Since SMT solvers do not currently support a theory of finite fields, SMT-based tools must encode field operations using another theory.

There are two natural ways to represent finite fields using commonly supported theories in SMT, but both are ultimately inefficient. Recall that a finite field of prime order can be represented as the integers with addition and multiplication performed modulo a prime p. Thus, field operations can be represented using integers or bit-vectors: both support addition, multiplication, and modular reduction. However, both approaches fall short. Non-linear integer reasoning is notoriously challenging for SMT solvers, and bit-vector solvers perform abysmally on fields of cryptographic size (hundreds of bits).

In this paper, we develop for the first time a direct solver for finite fields within an SMT solver. We use well-known ideas from computer algebra (specifically, Gröbner bases [21] and triangular decomposition [6,99]) to form the basis of our decision procedure. However, we improve on this baseline in two important ways. First, our decision procedure does not manipulate *field polynomials* (i.e., those of form $X^p - X$). As expected, this results in a loss of completeness at the Gröbner basis stage. However, surprisingly, this often does not matter. Furthermore, completeness is recovered during the model construction algorithm (albeit in a rather rudimentary way). This modification turns out to be crucial for obtaining reasonable performance. Second, we implement a proof-tracing mechanism in the Gröbner basis engine, thereby enabling it to compute unsatisfiable cores, which is also very beneficial in the context of SMT solving. Finally, we implement all of this as a theory solver for prime-order fields inside the cvc5 SMT solver.

To guide research in this area, we also give a first set of `QF_FF` (quantifier-free, finite field) benchmarks, obtained from the domain of ZKP compiler correctness. ZKP compilers translate from high-level computations (e.g., over Booleans, bit-vectors, arrays, etc.) to systems of finite field constraints that are usable by ZKPs. We instrument existing ZKP compilers to produce translation validation [86] verification conditions, i.e. conditions that represent desirable correctness properties of a specific compilation. We give these compilers concrete Boolean computations (which we sample at random), and construct SMT formulas capturing the correctness of the ZKP compilers' translations of those computations into field constraints. We represent the formulas using both our new theory of finite fields and also the alternative theory encodings mentioned above.

We evaluate our tool on these benchmarks and compare it to the approaches based on bit-vectors, integers, and pure computer algebra (without SMT). We find that our tool significantly outperforms the other solutions. Compared to the best previous solution (we list prior alternatives in Sect. 7), it is 6× faster and it solves 2× more benchmarks.

In sum, our contributions are:

1. a definition of the theory of finite fields in the context of SMT;
2. a decision procedure for this theory that avoids field polynomials and produces unsatisfiable cores;
3. the first public theory solver for this theory (implemented in cvc5); and

4. the first set of `QF_FF` benchmarks, which encode translation validation queries for ZKP compilers on Boolean computations.

In the rest of the paper, we discuss related work (§1.1), cover background and notation (§2), define the theory of finite fields (§3), give a decision procedure (§4), describe our implementation (§5), explain the benchmarks (§6), and report on experiments (§7).

1.1 Related Work

There is a large body of work on computer algebra, with many algorithms implemented in various tools [1,18,31,37,49,52,58,72,100,101]. However, the focus in this work is on quickly constructing useful algebraic objects (e.g., a Gröbner basis), rather than on searching for a solution to a set of field constraints.

One line of recent work [54,55] by Hader and Kovács considers SMT-oriented field reasoning. One difference with our work is that it scales poorly with field size because it uses field polynomials to achieve completeness. Furthermore, their solver is not public.

Others consider verifying field constraints used in ZKPs. One paper surveys possible approaches [97], and another considers proof-producing ZKP compilation [24]. However, neither develops automated, general-purpose tools.

Still other works study automated reasoning for non-linear arithmetic over reals and integers [3,23,25,29,47,60–62,70,74,96,98]. A key challenge is reasoning about *comparisons*. We work over finite fields and do not consider comparisons because they are used for neither elliptic curves nor most ZKPs.

Further afield, researchers have developed techniques for verified algebraic reasoning in proof assistants [15,64,75,79], with applications to mathematics [19,28,51,65] and cryptography [39,40,85,91]. In contrast, our focus is on *fully automated* reasoning about finite fields.

2 Background

2.1 Algebra

Here, we summarize algebraic definitions and facts that we will use; see [71, Chapters 1 through 8] or [34, Part IV] for a full presentation.

Finite Fields. A *finite field* is a finite set equipped with binary operations $+$ and $\times$ that have identities (0 and 1 respectively), have inverses (save that there is no multiplicative inverse for 0), and satisfy associativity, commutativity, and distributivity. The *order* of a finite field is the size of the set. All finite fields have order $q = p^e$ for some prime p (called the *characteristic*) and positive integer e. Such an integer q is called a *prime power*.

Up to isomorphism, the field of order q is unique and is denoted $\mathbb{F}_q$, or $\mathbb{F}$ when the order is clear from context. The fields $\mathbb{F}_{q^d}$ for $d > 1$ are called *extension fields* of $\mathbb{F}_q$. In contrast, $\mathbb{F}_q$ may be called the *base field*. We write $\mathbb{F} \subset \mathbb{G}$ to indicate that $\mathbb{F}$ is a field that is isomorphic to the result of restricting field $\mathbb{G}$ to some subset of its elements (but with the same operations). We note in particular that $\mathbb{F}_q \subset \mathbb{F}_{q^d}$. A field of prime order p is called a *prime field.*

Polynomials. For a finite field $\mathbb{F}$ and formal variables $X_1, \dots, X_k$, $\mathbb{F}[X_1, \dots, X_k]$ denotes the set of polynomials in $X_1, \dots, X_k$ with coefficients in $\mathbb{F}$. By taking the variables to be in $\mathbb{F}$, a polynomial $f \in \mathbb{F}[X_1, \dots, X_k]$ can be viewed as a function from $\mathbb{F}^k \to \mathbb{F}$. However, by taking the variables to be in an extension $\mathbb{G}$ of $\mathbb{F}$, f can also be viewed as function from $\mathbb{G}^k \to \mathbb{G}$.

For a set of polynomials $S = \{f_1, \dots, f_m\} \subset \mathbb{F}_q[X_1, \dots, X_k]$, the set $I = \{g_1 f_1 + \cdots + g_m f_m : g_i \in \mathbb{F}_q[X_1, \dots, X_k]\}$ is called the *ideal* generated by S and is denoted $\langle f_1, \dots, f_m \rangle$ or $\langle S \rangle$. In turn, S is called a *basis* for the ideal I.

The *variety* of an ideal I in field $\mathbb{G} \supset \mathbb{F}$ is denoted $\mathcal{V}_{\mathbb{G}}(I)$, and is the set $\{\mathbf{x} \in \mathbb{G}^k : \forall f \in I, f(\mathbf{x}) = 0\}$. That is, $\mathcal{V}_{\mathbb{G}}(I)$ contains the common zeros of polynomials in I, viewed as functions over $\mathbb{G}$. Note that for any set of polynomials S that generates I, $\mathcal{V}_{\mathbb{G}}(I)$ contains exactly the common zeros of S in $\mathbb{G}$. When the space $\mathbb{G}$ is just $\mathbb{F}$, we denote the variety as $\mathcal{V}(I)$. An ideal I that contains 1 contains all polynomials and is called *trivial.*

One can show that if I is trivial, then $\mathcal{V}(I) = \emptyset$. However, the converse does not hold. For instance, $X^2 + 1 \in \mathbb{F}_3[X]$ has no zeros in $\mathbb{F}_3$, but $1 \notin \langle X^2 + 1 \rangle$. But, one can also show that I is trivial iff for all extensions $\mathbb{G}$ of $\mathbb{F}$, $\mathcal{V}_{\mathbb{G}}(I) = \emptyset$.

The *field polynomial* for field $\mathbb{F}_q$ in variable X is $X^q - X$. Its zeros are all of $\mathbb{F}_q$ and it has no additional zeros in any extension of $\mathbb{F}_q$. Thus, for an ideal I of polynomials in $\mathbb{F}[X_1, \dots, X_k]$ that contains field polynomials for each variable X_i, I is trivial iff $\mathcal{V}(I) = \emptyset$. For this reason, field polynomials are a common tool for ensuring the completeness of ideal-based reasoning techniques [48,54,97].

Representation. We represent $\mathbb{F}_p$ as the set of integers $\{0, 1, \dots, p-1\}$, with the operations + and × performed modulo p. The representation of $\mathbb{F}_{p^e}$ with $e > 1$ is more complex. Unfortunately, the set $\{0, 1, \dots, p^e - 1\}$ with + and × performed modulo p^e **is not** a field because multiples of p do not have multiplicative inverses. Instead, we represent $\mathbb{F}_{p^e}$ as the set of polynomials in $\mathbb{F}[X]$ of degree less than e. The operations + and × are performed modulo $q(X)$, an irreducible polynomial[1] of degree e [71, Chapter 6]. There are p^e such polynomials, and so long as $q(X)$ is irreducible, all (save 0) have inverses. Note that this definition of $\mathbb{F}_{p^e}$ generalizes $\mathbb{F}_p$, and captures the fact that $\mathbb{F}_p \subset \mathbb{F}_{p^e}$.

2.2 Ideal Membership

The ideal membership problem is to determine whether a given polynomial p is in the ideal generated by a given set of polynomials D. We summarize definitions and facts relevant to algorithms for this problem; see [30] for a full presentation.

Monomial Ordering. In $\mathbb{F}[X_1, \dots, X_k]$, a *monomial* is a polynomial of form $X_1^{e_1} \cdots X_k^{e_k}$ with non-negative integers e_i. A *monomial ordering* is a total ordering on monomials such that for all monomials p, q, r, if $p < q$, then $pr < qr$.

[1] Recall that an irreducible polynomial cannot be factored into two or more non-constant polynomials.

The *lexicographical* ordering for monomials $X_1^{e_1} \cdots X_k^{e_k}$ orders them lexicographically by the tuple $(e_1, \ldots, e_k)$. The *graded-reverse lexicographical* (grevlex) ordering is lexicographical by the tuple $(e_1 + \cdots + e_k, e_1, \ldots, e_k)$. With respect to an ordering, $\mathsf{lm}(f)$ denotes the greatest monomial of a polynomial f.

Reduction. For polynomials p and d, if $\mathsf{lm}(d)$ divides a term t of p, then we say that p *reduces* to r *modulo* d (written $p \rightarrow_d r$) for $r = p - \frac{t}{\mathsf{lm}(d)} d$. For a set of polynomials D, we write $p \rightarrow_D r$ if $p \rightarrow_d r$ for some $d \in D$. Let $\rightarrow_D^*$ be the transitive closure of $\rightarrow_D$. We define $p \Rightarrow_D r$ to hold when $p \rightarrow_D^* r$ and there is no r' such that $r \rightarrow_D r'$.

Reduction is a sound—but incomplete—algorithm for ideal membership. That is, one can show that $p \Rightarrow_D 0$ implies $p \in \langle D \rangle$, but the converse does not hold in general.

Gröbner Bases. Define the *s-polynomial* for polynomials p and q, by $\mathsf{spoly}(p, q) = p \cdot \mathsf{lm}(q) - q \cdot \mathsf{lm}(p)$. A Gröbner basis (GB) [21] is a set of polynomials P characterized by the following equivalent conditions:

1. $\forall p, p' \in P$, $\mathsf{spoly}(p, p') \Rightarrow_P 0$ (*closure under the reduction of s-polynomials*)
2. $\forall p \in \langle P \rangle$, $p \Rightarrow_P 0$ (*reduction is a complete test for ideal membership*)

Gröbner bases are useful for deciding ideal membership. From the first characterization, one can build algorithms for constructing a Gröbner basis for any ideal [21]. Then, the second characterization gives an ideal membership test. When P is a GB, the relation $\Rightarrow_P$ is a function (i.e., $\rightarrow_P$ is confluent), and it can be efficiently computed [1,21]; thus, this test is efficient.

A *Gröbner basis engine* takes a set of generators G for some ideal I and computes a Gröbner basis for I. We describe the high-level design of such engines here. An engine constructs a sequence of bases $G_0, G_1, G_2, \ldots$ (with $G_0 = G$) until some G_i is a Gröbner basis. Each G_i is constructed from G_{i-1} according to one of three types of steps. First, for some $p, q \in G_{i-1}$ such that $\mathsf{spoly}(p, q) \Rightarrow_{G_{i-1}} r \neq 0$, the engine can set $G_i = G_{i-1} \cup \{r\}$. Second, for some $p \in G_{i-1}$ such that $p \Rightarrow_{G_{i-1} \setminus \{p\}} r \neq p$, the engine can set $G_i = (G_{i-1} \setminus \{p\}) \cup \{r\}$. Third, for some $p \in G_{i-1}$ such that $p \Rightarrow_{G_{i-1} \setminus \{p\}} 0$, the engine can set $G_i = G_{i-1} \setminus \{p\}$. Notice that all rules depend on the current basis; some add polynomials, and some remove them. In general, it is unclear which sequence of steps will construct a Gröbner basis most quickly: this is an active area of research [1,18,41,43].

2.3 Zero Knowledge Proofs

Zero-knowledge proofs allow one to prove that some secret data satisfies a public property, without revealing the data itself. See [94] for a full presentation; we give a brief overview here. There are two parties: a *verifier* $\mathcal{V}$ and a *prover* $\mathcal{P}$. $\mathcal{V}$ knows a public *instance* x and asks $\mathcal{P}$ to show that it has knowledge of a secret *witness* w satisfying a public *predicate* $\phi(x, w)$. To do so, $\mathcal{P}$ runs an efficient (i.e., polytime in a security parameter λ) proving algorithm $\mathsf{Prove}(\phi, x, w) \rightarrow \pi$ and sends the resulting *proof* π to $\mathcal{V}$. Then, $\mathcal{V}$ runs an efficient verification algorithm $\mathsf{Verify}(\phi, x, \pi) \to \{0, 1\}$ that accepts or rejects the proof. A system for Zero-Knowledge Proofs of knowledge (ZKPs) is a $(\mathsf{Prove}, \mathsf{Verify})$ pair with:

- *completeness*: If $\phi(x, w)$, then $\Pr[\mathsf{Verify}(\phi, x, \mathsf{Prove}(\phi, x, w)) = 0] \leq \mathsf{negl}(\lambda)$,[2]
- *computational knowledge soundness* [16]: (informal) a polytime adversary that does not know w satisfying ϕ can produce an acceptable π with probability at most $\mathsf{negl}(\lambda)$.
- *zero-knowledge* [50]: (informal) π reveals nothing about w, other than its existence.

ZKP applications are manifold. ZKPs are the basis of private cryptocurrencies such as Zcash and Monero, which have a combined market capitalization of $2.80B as of 30 June 2022 [44,45]. They've also been proposed for auditing sealed court orders [46], operating private gun registries [63], designing privacy-preserving middleboxes [53] and more [22,56].

This breadth of applications is possible because implemented ZKPs are very general: they support any ϕ checkable in polytime. However, ϕ must be first compiled to a cryptosystem-compatible computation language. The most common language is a *rank-1 constraint system* (R1CS). In an R1CS $\mathcal{C}$, x and w are together encoded as a vector $\mathbf{z} \in \mathbb{F}^m$. The system $\mathcal{C}$ is defined by three matrices $A, B, C \in \mathbb{F}^{n \times m}$; it is satisfied when $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$, where $\circ$ is the element-wise product. Thus, the predicate can be viewed as n distinct *constraints*, where constraint i has form $(\sum_j A_{ij} z_j)(\sum_j B_{ij} z_j) - (\sum_j C_{ij} z_j) = 0$. Note that each constraint is a degree ≤ 2 polynomial in m variables that $\mathbf{z}$ must be a zero of. For security reasons, $\mathbb{F}$ must be large: its prime must have ≈255 bits.

Encoding. The efficiency of the ZKP scales quasi-linearly with n. Thus, it's useful to encode ϕ as an R1CS with a minimal number of constraints. Since equisatifiability—not logical equivalence—is needed, encodings may introduce new variables.

As an example, consider the Boolean computation $a \leftarrow c_1 \vee \cdots \vee c_k$. Assume that $c'_1, \ldots, c'_k \in \mathbb{F}$ are elements in $\mathbf{z}$ that are 0 or 1 such that $c_i \leftrightarrow (c'_i = 1)$. How can one ensure that $a' \in \mathbb{F}$ (also in $\mathbf{z}$) is 0 or 1 and $a \leftrightarrow (a' = 1)$? Given that there are $k - 1$ ORs, natural approaches use $\Theta(k)$ constraints. One clever approach is to introduce variable x' and enforce constraints $x'(\sum_i c'_i) = a'$ and $(1 - a')(\sum_i c'_i) = 0$. If any c_i is true, a' must be 1 to satisfy the second constraint; setting x' to the sum's inverse satisfies the first. If all c_i are false, the first constraint ensures a' is 0. This encoding is correct when the sum does not overflow; thus, k must be smaller than $\mathbb{F}$'s characteristic.

Optimizations like this can be quite complex. Thus, ZKP programmers use constraint synthesis libraries [14,69] or compilers [13,24,80,81,84,92,102] to generate an R1CS from a high-level description. Such tools support objects like Booleans, fixed-width integers, arrays, and user-defined data-types. The correctness of these tools is critical to the correctness of any system built with them.

[2] $f(\lambda) \leq \mathsf{negl}(\lambda)$ if for all $c \in \mathbb{N}$, $f(\lambda) = o(\lambda^{-c})$.

2.4 SMT

We assume usual terminology for many-sorted first order logic with equality ([38] gives a complete presentation). Let Σ be a many-sorted signature including a sort Bool and symbol family $\approx_\sigma$ (abbreviated $\approx$) with sort $\sigma \times \sigma \rightarrow \mathsf{Bool}$ for all σ in Σ. A *theory* is a pair $T = (\Sigma, \mathbf{I})$, where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations. A Σ-formula ϕ is *satisfiable* (resp., *unsatisfiable*) in T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. Given a (set of) formula(s) S, we write $S \models_T \phi$ if every interpretation $\mathcal{M} \in \mathbf{I}$ that satisfies S also satisfies ϕ.

When using the CDCL(T) framework for SMT, the reasoning engine for each theory is encapsulated inside a *theory solver*. Here, we mention the fragment of CDCL(T) that is relevant for our purposes ([78] gives a complete presentation)).

The goal of CDCL(T) is to check a formula ϕ for satisfiability. A *core* module manages a propositional search over the propositional abstraction of ϕ and communicates with the theory solver. As the core constructs partial propositional assignments for the abstract formula, the theory solver is given the literals that correspond to the current propositional assignment. When the propositional assignment is completed (or, optionally, before), the theory solver must determine whether its literals are jointly satisfiable. If so, it must be able to provide an interpretation in $\mathbf{I}$ (which includes an assignment to theory variables) that satisfies them. If not, it may indicate a strict subset of the literals which are unsatisfiable: an unsatisfiable core. Smaller unsatisfiable cores usually accelerate the propositional search.

3 The Theory of Finite Fields

We define the theory $T_{\mathbb{F}_q}$ of the finite field $\mathbb{F}_q$, for any order q. Its sort and symbols are indexed by the parameter q; we omit q when clear from context.

The signature of the theory is given in Fig. 1. It includes sort F, which intuitively denotes the sort of elements of $\mathbb{F}_q$ and is represented in our proposed SMT-LIB format as `(_ FiniteField` q`)`. There is a constant symbol for each element of $\mathbb{F}_q$, and function symbols for addition and multiplication. Other finite field operations (e.g., negation, subtraction, and inverses) naturally reduce to this signature.

An interpretation $\mathcal{M}$ of $T_{\mathbb{F}_q}$ must interpret: F as $\mathbb{F}_q$, $n \in \{0, \ldots, q-1\}$ as the n^{th} element of $\mathbb{F}_q$ in lexicographical order,[3] $+$ as addition in $\mathbb{F}_q$, $\times$ as multiplication in $\mathbb{F}_q$, and $\approx$ as equality in $\mathbb{F}_q$.

Note that in order to avoid ambiguity, we require that the sort of any constant `ff`n must be ascribed. For instance, the n^{th} element of $\mathbb{F}_q$ would be `(as ff`n `(_ FiniteField` q`))`. The sorts of non-nullary function symbols need not be ascribed: they can be inferred from their arguments.

[3] For non-prime $\mathbb{F}_{p^e}$, we use the lexicographical ordering of elements represented as polynomials in $\mathbb{F}_p[X]$ modulo the Conway polynomial [83,90] $C_{p,e}(X)$. This representation is standard [57].

Symbol	**Arity**	**SMT-LIB**	**Description**
$n \in \{0, \ldots, q-1\}$	F	`ff`n	The n^{th} element of $\mathbb{F}_q$
$+$	$\mathsf{F} \times \mathsf{F} \to \mathsf{F}$	`ff.add`	Addition in $\mathbb{F}_q$
$\times$	$\mathsf{F} \times \mathsf{F} \to \mathsf{F}$	`ff.mul`	Multiplication in $\mathbb{F}_q$

Fig. 1. Signature of the theory of $\mathbb{F}_q$

```
Function $DecisionProcedure$:
    Input: A set of $\mathbb{F}$-literals $L$ in variables $\mathbf{X}$
    Output: UNSAT and a core $C \subseteq L$, or
    Output: SAT and a model $M : \mathbf{X} \to \mathbb{F}$
    $P \leftarrow$ empty set; $W_i \leftarrow$ fresh, $\forall i$;
    for $s_i \bowtie_i t_i \in L$ do
        if $\bowtie_i = \approx$ then $P \leftarrow P \cup \{[\![s_i]\!] - [\![t_i]\!]\}$ ;
        else if $\bowtie_i = \not\approx$ then $P \leftarrow P \cup \{W_i([\![s_i]\!] - [\![t_i]\!]) - 1\}$ ;
    $B \leftarrow GB(P)$;
    if $1 \Rightarrow_B 0$ then return UNSAT, $CoreFromTree()$ ;
    $m \leftarrow FindZero(P)$;
    if $m = \bot$ then return UNSAT, $L$ ;
    else return SAT, $\{X \mapsto z : (X \mapsto z) \in m, X \in \mathbf{X}\}$ ;
```

Fig. 2. The decision procedure for $\mathbb{F}_q$.

4 Decision Procedure

Recall (§2.4) that a CDCL(T) theory solver for $\mathbb{F}$ must decide the satisfiability of a set of $\mathbb{F}$-literals. At a high level, our decision procedure comprises three steps. First, we reduce to a problem concerning a single algebraic variety. Second, we use a GB-based test for unsatisfiability that is fast and sound, but incomplete. Third, we attempt model construction. Figure 2 shows pseudocode for the decision procedure; we will explain it incrementally.

4.1 Algebraic Reduction

Let $L = \{\ell_1, \ldots, \ell_{|L|}\}$ be a set of literals. Each $\mathbb{F}$-literal has the form $\ell_i = s_i \bowtie t_i$ where s and t are $\mathbb{F}$-terms and $\bowtie \in \{\approx, \not\approx\}$. Let $\mathbf{X} = \{X_1, \ldots, X_k\}$ denote the free variables in L. Let $E, D \subseteq \{1, \ldots, |L|\}$ be the sets of indices corresponding to equalities and disequalities in L, respectively. Let $[\![t]\!] \in \mathbb{F}[\mathbf{X}]$ denote the natural interpretation of $\mathbb{F}$-terms as polynomials in $\mathbb{F}[\mathbf{X}]$ (Fig. 3). Let $P_E \subset \mathbb{F}[\mathbf{X}]$ be the set of interpretations of the equalities; i.e., $P_E = \{[\![s_i]\!] - [\![t_i]\!]\}_{i \in E}$. Let $P_D \subset \mathbb{F}[\mathbf{X}]$ be the interpretations of the *dis*equalities; i.e., $P_D = \{[\![s_i]\!] - [\![t_i]\!]\}_{i \in D}$. The satisfiability of L reduces to whether $\mathcal{V}(\langle P_E \rangle) \setminus \left[\bigcup_{p \in P_D} \mathcal{V}(\langle p \rangle)\right]$ is non-empty.

To simplify, we reduce disequalities to equalities using a classic technique [88]: we introduce a fresh variable W_i for each $i \in D$ and define P'_D as

$$P'_D = \{W_i([\![s_i]\!] - [\![t_i]\!]) - 1\}_{i \in D}$$

$$\text{Const}\,\frac{t \in \mathbb{F}}{[\![t]\!] = t} \qquad \text{Var}\,\frac{}{[\![X_i]\!] = X_i} \qquad \text{Add}\,\frac{[\![s]\!] = s' \quad [\![t]\!] = t'}{[\![s + t]\!] = s' + t'} \qquad \text{Mul}\,\frac{[\![s]\!] = s' \quad [\![t]\!] = t'}{[\![s \times t]\!] = s' \times t'}$$

Fig. 3. Interpreting $\mathbb{F}$-terms as polynomials

Note that each $p \in P'_D$ has zeros for exactly the values of $\mathbf{X}$ where its analog in P_D is *not* zero. Also note that $P'_D \subset \mathbb{F}_q[\mathbf{X}']$, with $\mathbf{X}' = \mathbf{X} \cup \{W_i\}_{i \in D}$.

We define P to be $P_E \cup P'_D$ (constructed in lines 2 to 6, Fig. 2) and note three useful properties of P. First, L is satisfiable if and only if $\mathcal{V}(\langle P \rangle)$ is non-empty. Second, for any $P' \subset P$, if $\mathcal{V}(\langle P' \rangle) = \emptyset$, then $\{\pi(p) : p \in P'\}$ is an unsatisfiable core, where π maps a polynomial to the literal it is derived from. Third, from any $\mathbf{x} \in \mathcal{V}(\langle P \rangle)$ one can immediately construct a model. Thus, our theory solver reduces to understanding properties of the variety $\mathcal{V}(\langle P \rangle)$.

4.2 Incomplete Unsatisfiability and Cores

Recall (§2.2) that if $1 \in \langle P \rangle$, then $\mathcal{V}(\langle P \rangle)$ is empty. We can answer this ideal membership query using a Gröbner basis engine (line 7, Fig. 2). Let *GB* be a subroutine that takes a list of polynomials and computes a Gröbner basis for the ideal that they generate, according to some monomial ordering. We use grevlex: the ordering for which GB engines are typically most efficient [42]. We compute $GB(P)$ and check whether $1 \Rightarrow_{GB(P)} 0$. If so, we report that $\mathcal{V}(\langle P \rangle)$ is empty. If not, recall (§2.2) that $\mathcal{V}(\langle P \rangle)$ may still be empty; we proceed to attempt model construction (lines 9 to 11, Fig. 2, described in the next subsection).

If 1 *does* reduce by the Gröbner basis, then identifying a subset of P which is sufficient to reduce 1 yields an unsatisfiable core. To construct such a subset, we formalize the inferences performed by the Gröbner basis engine as a calculus for proving ideal membership.

Figure 4 presents IdealCalc: our ideal membership calculus. IdealCalc proves facts of the form $p \in \langle P \rangle$, where p is a polynomial and P is the set of generators for an ideal. The G rule states that the generators are in the ideal. The Z rule states that 0 is in the ideal. The S rule states that for any two polynomials in the ideal, their s-polynomial is in the ideal too. The $\mathsf{R}_\uparrow$ and $\mathsf{R}_\downarrow$ rules state that if $p \rightarrow_q r$ with q in the ideal, then p is in the ideal if and only if r is.

The soundness of IdealCalc follows immediately from the definition of an ideal. Completeness relies on the existence of algorithms for computing Gröbner bases using only s-polynomials and reduction [21,41,43]. We prove both properties in Appendix A.

Theorem 1 (IdealCalcSoundness). *If there exists an* IdealCalc *proof tree with conclusion $p \in \langle P \rangle$, then $p \in \langle P \rangle$.*

Theorem 2 (IdealCalcCompleteness). *If $p \in \langle P \rangle$, then there exists an* IdealCalc *proof tree with conclusion $p \in \langle P \rangle$.*

$$\mathsf{Z}\,\frac{}{0 \in \langle P \rangle} \qquad \mathsf{G}\,\frac{p \in P}{p \in \langle P \rangle} \qquad \mathsf{R}_{\uparrow}\,\frac{r \in \langle P \rangle \quad q \in \langle P \rangle \quad p \rightarrow_q r}{p \in \langle P \rangle}$$

$$\mathsf{S}\,\frac{p \in \langle P \rangle \quad q \in \langle P \rangle}{\mathsf{spoly}(p, q) \in \langle P \rangle} \qquad \mathsf{R}_{\downarrow}\,\frac{p \in \langle P \rangle \quad q \in \langle P \rangle \quad p \rightarrow_q r}{r \in \langle P \rangle}$$

Fig. 4. IdealCalc: a calculus for ideal membership

```
Function FindZero:
    Input: A Gröbner basis $B \subset \mathbb{F}[\mathbf{X}']$
    Input: A partial map $M : \mathbf{X}' \to \mathbb{F}$ (empty by default)
    Output: A total map $M : \mathbf{X}' \to \mathbb{F}$ or $\bot$
    if $1 \in \langle B \rangle$ then return $\bot$ ;
    if $|M| = |\mathbf{X}'|$ then return $M$ ;
    for $(X'_i \mapsto z) \in ApplyRule(B, M)$ do
        $r \leftarrow FindZero(GB(B \cup \{X'_i - z\}), M \cup \{X'_i \mapsto z\})$;
        if $r \neq \bot$ then return $r$;
    return $\bot$
```

Fig. 5. Finding common zeros for a Gröbner basis. After handling trivial cases, *FindZero* uses *ApplyRule* to apply the first applicable rule from Fig. 6.

By instrumenting a Gröbner basis engine and reduction engine, one can construct IdealCalc proof trees. Then, for a conclusion $1 \in \langle P \rangle$, traversing the proof tree to its leaves gives a subset $P' \subseteq P$ such that $1 \in \langle P' \rangle$. The procedure *CoreFromTree* (called in line 8, Fig. 2) performs this traversal, by accessing a proof tree recorded by the *GB* procedure and the reductions. The proof of Theorem 2 explains our instrumentation in more detail (Appendix A).

4.3 Completeness Through Model Construction

As discussed, we still need a *complete* decision procedure for determining if $\mathcal{V}(\langle P \rangle)$ is empty. We call this procedure *FindZero*; it is a backtracking search for an element of $\mathcal{V}(\langle P \rangle)$. It also serves as our model construction procedure.

Figure 5 presents *FindZero* as a recursive search. It maintains two data structures: a Gröbner basis B and partial map $M : \mathbf{X}' \to \mathbb{F}$ from variables to field elements. By applying a branching rule (which we will discuss in the next paragraph), *FindZero* obtains a disjunction of single-variable assignments $X'_i \mapsto z$, which it branches on. *FindZero* branches on an assignment $X'_i \mapsto z$ by adding it to M and updating B to $GB(B \cup \{X'_i - z\})$.

Figure 6 shows the branching rules of *FindZero*. Each rule comprises *antecedents* (conditions that must be met for the rule to apply) and a *conclusion* (a disjunction of single-variable assignments to branch on). The Univariate rule applies when B contains a polynomial p that is univariate in some variable X'_i that M does not have a value for. The rule branches on the univariate roots of p. The Triangular rule comes from work on triangular decomposition [68]. It

$$\textsf{Univariate}\ \frac{p \in B \quad p \in \mathbb{F}[X_i'] \quad X_i' \notin M \quad Z \leftarrow \mathit{UnivariateZeros}(p)}{\bigvee_{z \in Z}(X_i' \mapsto z)}$$

$$\textsf{Triangular}\ \frac{\mathit{Dim}(\langle B \rangle) = 0 \quad X_i' \notin M \quad p \leftarrow \mathit{MinPoly}(B, X_i') \quad Z \leftarrow \mathit{UnivariateZeros}(p)}{\bigvee_{z \in Z}(X_i' \mapsto z)}$$

$$\textsf{Exhaust}\ \frac{}{\bigvee_{z \in \mathbb{F}} \bigvee_{X_i' \notin M}(X_i' \mapsto z)}$$

Fig. 6. Branching rules for *FindZero*.

applies when B is zero-dimensional.[4] It computes a univariate *minimal polynomial* $p(X_i')$ in some unassigned variables X_i', and branches on the univariate roots of p. The final rule Exhaust has no conditions and simply branches on all possible values for all unassigned variables.

FindZero's *ApplyRule* sub-routine applies the first rule in Fig. 6 whose conditions are met. The other subroutines (*GB* [21,41,43], *Dim* [11], *MinPoly* [2], and *UnivariateZeros* [87]) are commonly implemented in computer algebra libraries. *Dim*, *MinPoly*, and *UnivariateZeros* run in (randomized) polytime.

Theorem 3 (*FindZero***Correctness).** *If $\mathcal{V}(\langle B \rangle) = \emptyset$ then FindZero returns $\perp$; otherwise, it returns a member of $\mathcal{V}(\langle B \rangle)$. (Proof: Appendix B)*

Correctness and Efficiency. The branching rules achieve a careful balance between correctness and efficiency. The Exhaust rule is always applicable, but a full exhaustive search over a large field is unreasonable (recall: ZKPs operate of ≈255-bit fields). The Triangular and Univariate rules are important alternatives to exhaustion. They create a far smaller set of branches, but apply only when the variety has dimension zero or the basis has a univariate polynomial.

As an example of the importance of Univariate, consider the univariate system $X^2 = 2$, in a field where 2 is not a perfect square (e.g., $\mathbb{F}_7$). $X^2 - 2$ is already a (reduced) Gröbner basis, and it does not contain 1, so *FindZero* applies. With the Univariate rule, *FindZero* computes the univariate zeros of $X^2 - 2$ (there are none) and exits. Without it, the Exhaust rule creates $|\mathbb{F}|$ branches.

As an example of when Triangular is critical, consider

$$\begin{gathered}
X_1 + X_2 + X_3 + X_4 + X_5 = 0 \\
X_1X_2 + X_2X_3 + X_3X_4 + X_4X_5 + X_5X_1 = 0 \\
X_1X_2X_3 + X_2X_3X_4 + X_3X_4X_5 + X_4X_5X_1 + X_5X_1X_2 = 0 \\
X_1X_2X_3X_4 + X_2X_3X_4X_5 + X_3X_4X_5X_1 + X_4X_5X_1X_2 + X_5X_1X_2X_3 = 0 \\
X_1X_2X_3X_4X_5 = 1
\end{gathered}$$

[4] The *dimension* of an ideal is a natural number that can be efficiently computed from a Gröbner basis. If the dimension is zero, then one can efficiently compute a minimal polynomial in any variable X, given a Gröbner basis [2,68].

in $\mathbb{F}_{394357}$ [68]. The system is unsatisfiable, it has dimension 0, and its ideal does not contain 1. Moreover, our solver computes a (reduced) Gröbner basis for it that does not contain any univariate polynomials. Thus, Univariate does not apply. However, Triangular does, and with it, *FindZero* quickly terminates. Without Triangular, Exhaust would create at least $|\mathbb{F}|$ branches.

In the above examples, Exhaust performs very poorly. However, that is not always the case. For example, in the system $X_1 + X_2 = 0$, using Exhaust to guess X_1, and then using the univariate rule to determine X_2 is quite reasonable. In general, Exhaust is a powerful tool for solving *underconstrained* systems. Our experiments will show that despite including Exhaust, our procedure performs quite well on our benchmarks. We reflect on its performance in Sect. 8.

Field Polynomials: A Road not Taken. By guaranteeing completeness through (potential) exhaustion, we depart from prior work. Typically, one ensures completeness by including *field polynomials* in the ideal (§2.2). Indeed, this is the approach suggested [97] and taken [55] by prior work. However, field polynomials induce enormous overhead in the Gröbner basis engine because their degree is so large. The result is a procedure that is only efficient for tiny fields [55]. In our experiments, we compare our system's performance to what it would be if it used field polynomials.[5] The results confirm that deferring completeness to *FindZero* is far superior for our benchmarks.

5 Implementation

We have implemented our decision procedure for prime fields in the cvc5 SMT solver [7] as a theory solver. It is exposed through cvc5's SMT-LIB, C++, Java, and Python interfaces. Our implementation comprises ≈2k lines of C++. For the algebraic sub-routines of our decision procedure (§4), it uses CoCoALib [1]. To compute unsatisfiable cores (§4.2), we inserted hooks into CoCoALib's Gröbner basis engine (17 lines of C++).

Our theory solver makes sparse use of the interface between it and the rest of the SMT solver. It acts only once a full propositional assignment has been constructed. It then runs the decision procedure, reporting either satisfiability (with a model) or unsatisfiability (with an unsatisfiable core).

6 Benchmark Generation

Recall that one motivation for this work is to enable translation validation for compilers to field constraint systems (R1CSs) used in zero-knowledge proofs (ZKPs). Our benchmarks are SMT formulas that encode translation validation queries for compilers from *Boolean* computations to R1CS. At a high level, each benchmark is generated as follows.

[5] We add field polynomials to our procedure on line 2, Fig. 2. This renders our ideal triviality test (lines 7 and 8) complete, so we can eliminate the fallback to *FindZero*.

1. Sample a Boolean formula Ψ in v variables with t non-variable terms.
2. Compile Ψ to R1CS using ZoKrates [36], CirC [81], or ZoK-CirC [81].
3. Optionally remove some constraints from the R1CS.
4. Construct a formula ϕ in `QF_FF` that tests the soundness (all assignments satisfying the R1CS agree with Ψ) or determinism (the inputs uniquely determine the output) of the R1CS.
5. Optionally encode ϕ in `QF_BV`, in `QF_NIA`, or as (Boolean-free) $\mathbb{F}$-equations.

Through step 3, we construct SMT queries that are satisfiable, unsatisfiable, and of unknown status. Through step 5, we construct queries solvable using bit-vector reasoning, integer reasoning, or a stand-alone computer algebra system.

6.1 Examples

We describe our benchmark generator in full and give the definitions of soundness and determinism in Appendix C. Here, we give three example benchmarks. Our examples are based on the Boolean formula $\Psi(x_1, x_2, x_3, x_4) = x_1 \lor x_2 \lor x_3 \lor x_4$. Our convention is to mark field variables with a prime, but not Boolean variables. Using the technique from Sect. 2.3, CirC compiles this formula to the two-constraint system: $i's' = r' \land (1-r')s' = 0$ where $s' \triangleq \sum_{i=0}^{3} x'_i$. Each Boolean input x_i corresponds to field element x'_i and r' corresponds to the result of Ψ.

Soundness. An R1CS is sound if it ensures the output r' corresponds to the value of Ψ (when given valid inputs). Concretely, our system is sound if the following formula is valid:

$$\underbrace{\forall i.(x'_i = 0 \lor x'_i = 1) \land (x'_i = 1 \iff x_i)}_{\text{inputs are correct}} \land \underbrace{i's' = r' \land (1-r')s' = 0}_{\text{constraints hold}}$$
$$\implies$$
$$\underbrace{(r'_i = 0 \lor r'_i = 1) \land (r'_i = 1 \iff \Psi)}_{\text{output is correct}}$$

where Ψ and s' are defined as above. This is an UNSAT benchmark, because the formula is valid.

Determinism. An R1CS is deterministic if the values of the inputs uniquely determine the value of the output. To represent this in a formula, we use two copies of the constraint system: one with primed variables, and one with double-primed variables. Our example is deterministic if the following formula is valid:

$$\underbrace{\forall i.(x'_i = x''_i)}_{\text{inputs agree}} \land \underbrace{i's' = r' \land (1-r')s' = 0 \land i''s'' = r'' \land (1-r'')s'' = 0}_{\text{constraints hold for both systems}}$$
$$\implies$$
$$\underbrace{r' = r''}_{\text{outputs agree}}$$

Unsoundness. Removing constraints from the system can give a formula that is not valid (a SAT benchmark). For example, if we remove $(1 - r')s' = 0$, then the soundness formula is falsified by $\{x_i \mapsto \top, x'_i \mapsto 1, r' \mapsto 0, i' \mapsto 0\}$.

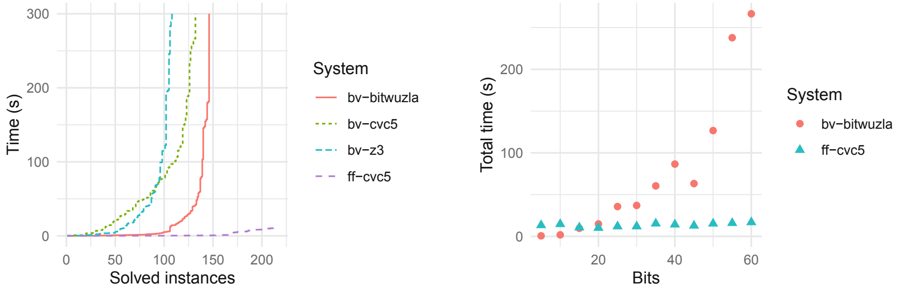
7 Experiments

Our experiments show that our approach:

1. scales well with the size of $\mathbb{F}$ (unlike a BV-based approach),
2. would scale poorly with the size of $\mathbb{F}$ if field polynomials were used,
3. benefits from unsatisfiable cores, and
4. substantially outperforms all reasonable alternatives.

Our test bed is a cluster with Intel Xeon E5-2637 v4 CPUs. Each run is limited to one physical core, 8GB memory, and 300s.

Throughout, we generate benchmarks for two correctness properties (soundness and determinism), three different ZKP compilers, and three different statuses (sat, unsat, and unknown). We vary the field size, encoding, number of inputs, and number of terms, depending on the experiment. We evaluate our cvc5 extension, Bitwuzla (commit `27f6291`), and z3 (version 4.11.2).



(a) Instances solved

(b) Total solve time for (field-based) cvc5 and (BV-based) Bitwuzla on commonly solved instances at all bit-widths.

Fig. 7. The performance of field-based and BV-based approaches (with various BV solvers) when the field size ranges from 5 to 60 bits.

7.1 Comparison with Bit-Vectors

Since bit-vector solvers scale poorly with bit-width, one would expect the effectiveness of a BV encoding of our properties to degrade as the field size grows. To validate this, we generate BV-encoded benchmarks for varying bit-widths and evaluate state-of-the-art bit-vector solvers on them. Though our applications of interest use $b = 255$, we will see that the BV-based approach does not scale to

Table 1. Solved small-field benchmarks by tool, property, and status.

	determinism			soundness			total		
system	unsat	unk.	sat	unsat	unk.	sat	timeout	memout	solved
bv-bitwuzla	4	16	29	28	32	36	71	0	145
bv-cvc5	5	11	36	25	25	29	78	7	131
bv-z3	5	9	14	25	25	29	100	9	107
ff-cvc5	36	36	36	36	36	36	0	0	216
all benchmarks	36	36	36	36	36	36			216

fields this large. Thus, for this set of experiments we use $b \in \{5, 10, \ldots, 60\}$, and we sample formulas with 4 inputs and 8 intermediate terms.

Figure 7a shows performance of three bit-vector solvers (cvc5 [7], Bitwuzla [76], and z3 [73]) and our $\mathbb{F}$ solver as a cactus plot; Table 1 splits the solved instances by property and status. We see that even for these small bit-widths, the field-based approach is already superior. The bit-vector solvers are more competitive on the soundness benchmarks, since these benchmarks include only half as many field operations as the determinism benchmarks.

For our benchmarks, Bitwuzla is the most efficient BV solver. We further examine the time that it and our solver take to solve the 9 benchmarks they can both solve at all bit-widths. Figure 7b plots the total solve time against b. While the field-based solver's runtime is nearly independent of field size, the bit-vector solvers slow down substantially as the field grows.

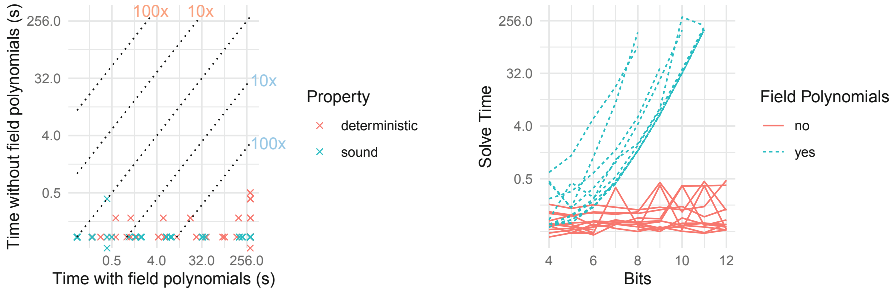
In sum, the BV approach scales poorly with field size and is already inferior on fields of size at least 2^{40}.

7.2 The Cost of Field Polynomials

Recall that our decision procedure does not use field polynomials (§4.3), but our implementation optionally includes them (§5). In this experiment, we measure the cost they incur. We use propositional formulas in 2 variables with 4 terms, and we take $b \in \{4, \ldots, 12\}$, and include SAT and unknown benchmarks.

Figure 8a compares the performance of our tool with and without field polynomials. For many benchmarks, field polynomials cause a slowdown greater than 100×. To better show the effect of the field size, we consider the solve time for the SAT benchmarks, at varying values of b. Figure 8b shows how solve times change as b grows: using field polynomials causes exponential growth. For UNSAT benchmarks, both configurations complete within 1s. This is because (for these benchmarks) the GB is just $\{1\}$ and CoCoA's GB engine is good at discovering that (and exiting) without considering the field polynomials.

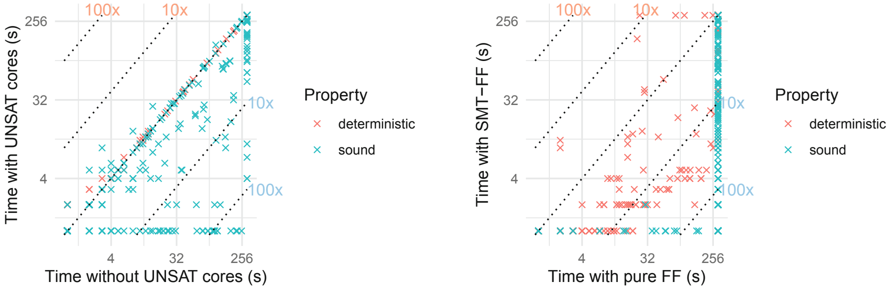
This growth is predictable. GB engines can take time exponential (or worse) in the degree of their inputs. A simple example illustrates this fact: consider computing a Gröbner basis with $X^{2^b} - X$ and $X^2 - X$. The former reduces to 0 modulo the latter, but the reduction takes $2^b - 1$ steps.



(a) All benchmarks, both configurations.

(b) Each series is one property at different numbers of bits.

Fig. 8. Solve times, with and without field polynomials. The field size varies from 4 to 12 bits. The benchmarks are all SAT or unknown.



(a) Our SMT solver with and without UNSAT cores.

(b) Our SMT solver compared with a pure computer algebra system.

Fig. 9. The performance of alternative algebra-based approaches.

7.3 The Benefit of UNSAT Cores

Section 4.2 describes how we compute unsatisfiable (UNSAT) cores in the $\mathbb{F}$ solver by instrumenting our Gröbner basis engine. In this experiment, we measure the benefit of doing so. We generate Boolean formulas with 2, 4, 6, 8, 10, and 12 variables; and 2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, and 2^7 intermediate terms, for a 255-bit field. We vary the number of intermediate terms widely in order to generate benchmarks of widely variable difficulty. We configure our solver with and without GB instrumentation.

Figure 9a shows the results. For many soundness benchmarks, the cores cause a speedup of more than 10×. As expected, only the soundness benchmarks benefit. Soundness benchmarks have non-trivial boolean structure, so the SMT core makes many queries to the theory solver. Returning good UNSAT cores shrinks the propositional search space, reduces the number of theory queries, and thus reduces solve time. However, determinism benchmarks are just a conjunction

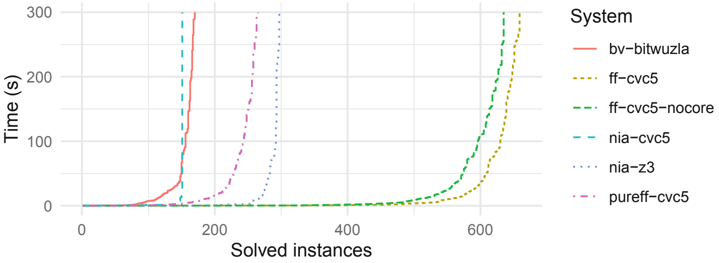


Fig. 10. A comparison of all approaches.

of theory literals, so the SMT core makes only one theory query. For them, returning a good UNSAT core has no benefit—but also induces little overhead.

7.4 Comparison to Pure Computer Algebra

In this experiment, we compare our SMT-based approach (which integrates computer-algebra techniques into SMT) against a stand-alone use of computer-algebra. We encode the Boolean structure of our formulas in $\mathbb{F}_p$ (see Appendix C). When run on such an encoding, our SMT solver makes just one query to its field solver, so it cannot benefit from the search optimizations present in CDCL(T). For this experiment, we use the same benchmark set as the last.

Figure 9b compares the pure $\mathbb{F}$ approach with our SMT-based approach. For benchmarks that encode soundness properties, the SMT-based approach is clearly dominant. The intuition here is is that computer algebra systems are not optimized for Boolean reasoning. If a problem has non-trivial Boolean structure, a cooperative approach like SMT has clear advantages. SMT's advantage is less pronounced for determinism benchmarks, as these manifest as a single query to the finite field solver; still, in this case, our encoding seems to have some benefit much of the time.

7.5 Main Experiment

In our main experiment, we compare our approach against all reasonable alternatives: a pure computer-algebra approach (§7.4), a BV approach with Bitwuzla (the best BV solver on our benchmarks, §7.1), an NIA approach with cvc5 and z3, and our own tool without UNSAT cores (§7.3). We use the same benchmark set as the last experiment; this uses a 255-bit field.

Figure 10 shows the results as a cactus plot. Table 2 shows the number of solved instances for each system, split by property and status. Bitwuzla quickly runs out of memory on most of the benchmarks. A pure computer-algebra approach outperforms Bitwuzla and cvc5's NIA solver. The NIA solver of z3 does a bit better, but our field-aware SMT solver is the best by far. Moreover, its best configuration uses UNSAT cores. Comparing the total solve time of ff-cvc5 and

Table 2. Solved benchmarks by tool, property, and status.

	determinism			soundness			total		
system	unsat	unk.	sat	unsat	unk.	sat	timeout	memout	solved
bv-bitwuzla	7	8	16	34	52	52	127	568	169
ff-cvc5	94	78	78	135	137	137	168	37	**659**
ff-cvc5-nocore	94	78	78	123	125	136	193	37	634
nia-cvc5	1	29	41	8	25	46	714	0	150
nia-z3	2	30	55	66	70	73	568	0	296
pureff-cvc5	84	74	75	6	15	10	532	68	264
all benchmarks	144	144	144	144	144	144			864

nia-z3 on commonly solved benchmarks, we find that ff-cvc5 reduces total solve time by 6×. In sum, the techniques we describe in this paper yield a tool that substantially outperforms all alternatives on our benchmarks.

8 Discussion and Future Work

We've presented a basic study of the potential of an SMT theory solver for finite fields based on computer algebra. Our experiments have focused on translation validation for ZKP compilers, as applied to Boolean input computations. The solver shows promise, but much work remains.

As discussed (Sect. 5), our implementation makes limited use of the interface exposed to a theory solver for CDCL(T). It does no work until a full propositional assignment is available. It also submits no lemmas to the core solver. Exploring which lightweight reasoning should be performed during propositional search and what kinds of lemmas are useful is a promising direction for future work.

Our model construction (Sect. 4.3) is another weakness. Without univariate polynomials or a zero-dimensional ideal, it falls back to exhaustive search. If a solution over an extension field is acceptable, then there are $\Theta(|\mathbb{F}|^d)$ solutions, so an exhaustive search seems likely to quickly succeed. Of course, we need a solution in the base field. If the base field is closed, then every solution is in the base field. Our fields are finite (and thus, not closed), but for our benchmarks, they seem to bear some empirical resemblance to closed fields (e.g., the GB-based test for an empty variety never fails, even though it is theoretically incomplete). For this reason, exhaustive search may not be completely unreasonable for our benchmarks. Indeed, our experiments show that our procedure is effective on our benchmarks, including for SAT instances. However, the worst-case performance of this kind of model construction is clearly abysmal. We think that a more intelligent search procedure and better use of ideas from computer algebra [6,67] would both yield improvement.

Theory combination is also a promising direction for future work. The benchmarks we present here are in the `QF_FF` logic: they involve only Booleans and finite fields. Reasoning about different fields in combination with one another would have natural applications to the representation of elliptic curve operations inside ZKPs. Reasoning about datatypes, arrays, and bit-vectors in combination with fields would also have natural applications to the verification of ZKP compilers.

Acknowledgements. We appreciate the help and guidance of Andres Nötzli, Andy Reynolds, Anna Bigatti, Dan Boneh, Erika Ábrahám, Fraser Brown, Gregory Sankaran, Jacob Van Geffen, James Davenport, John Abbott, Leonardo Alt, Lucas Vella, Maya Sankar, Riad Wahby, Shankara Pailoor, and Thomas Hader.

This material is in part based upon work supported by the DARPA SIEVE program and the Simons foundation. Any opinions, findings, and conclusions or recommendations expressed in this report are those of the author(s) and do not necessarily reflect the views of DARPA. It is also funded in part by NSF grant number 2110397.

A Proofs of IdealCalc Properties

This appendix is available in the full version of the paper [82].

B Proof of Correctness for *FindZero*

We prove that *FindZero* is correct (Theorem 3).

Proof. It suffices to show that for each branching rule that results in $\bigvee_j (X_{i_j} - r_j)$,

$$\mathcal{V}(\langle B \rangle) \subset \bigcup_j \mathcal{V}(\langle B \cup \{X_{i_j} - r_j\} \rangle)$$

First, consider an application of Univariate with univariate $p(X_i)$. Fix $z \in \mathcal{V}(\langle B \rangle)$. z is a zero of p, so for some j, $r_j = z$ and $z \in \mathcal{V}(\langle B \cup \{X_i - z\} \rangle)$.

Next, consider an application of Triangular to variable X_i with minimal polynomial $p(X_i)$. By the definition of minimal polynomial, any zero z of $\langle B \rangle$ has a value for X_i that is a root of p. Let that root be r. Then, $z \in \mathcal{V}(\langle B \cup \{X_i - z\} \rangle)$.

Finally, consider an application of Exhaust. The desired property is immediate.

C Benchmark Generation

This appendix is available in the full version of the paper [82].

References

1. Abbott, J., Bigatti, A.M.: CoCoALib: A C++ library for computations in commutative algebra... and beyond. In: International Congress on Mathematical Software (2010)

2. Abbott, J., Bigatti, A.M., Palezzato, E., Robbiano, L.: Computing and using minimal polynomials. J. Symbolic Comput. **100** (2020)
3. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. J. Logical Algebraic Methods in Programm. **119** (2021)
4. Anderson, B., McGrew, D.: Tls beyond the browser: Combining end host and network data to understand application behavior. In: IMC (2019)
5. Archer, D., O'Hara, A., Issa, R., Strauss, S.: Sharing sensitive department of education data across organizational boundaries using secure multiparty computation (2021)
6. Aubry, P., Lazard, D., Maza, M.M.: On the theories of triangular sets. J. Symbolic Comput. **28**(1) (1999)
7. Barbosa, H., et al.: cvc5: A versatile and industrial-strength SMT solver. In: TACAS (2022)
8. Barlow, R.: Computational thinking breaks a logjam. https://www.bu.edu/cise/computational-thinking-breaks-a-logjam/ (2015)
9. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO (2005)
10. Barrett, C., et al.: CVC4. In: CAV (2011)
11. Bayer, D., Stillman, M.: Computation of hilbert functions. J. Symb. Comput. **14**(1), 31–50 (1992)
12. Bayless, S., Bayless, N., Hoos, H., Hu, A.: SAT modulo monotonic theories. In: AAAI (2015)
13. Baylina, J.: Circom. https://github.com/iden3/circom
14. bellman. https://github.com/zkcrypto/bellman
15. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
16. Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, pp. 103–112 (1988)
17. Bogetoft, P., et al.: Secure multiparty computation goes live. In: FC (2009)
18. Bosma, W., Cannon, J., Playoust, C.: The magma algebra system i: the user language. J. Symb. Comput. **24**(3–4), 235–265 (1997)
19. Braun, D., Magaud, N., Schreck, P.: Formalizing some "small" finite models of projective geometry in coq. In: International Conference on Artificial Intelligence and Symbolic Computation (2018)
20. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: TACAS (2010)
21. Buchberger, B.: A theoretical basis for the reduction of polynomials to canonical forms. SIGSAM Bulletin (1976)
22. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS (2017)
23. Caviness, B.F., Johnson, J.R.: Quantifier elimination and cylindrical algebraic decomposition. Springer Science & Business Media (2012)
24. Chin, C., Wu, H., Chu, R., Coglio, A., McCarthy, E., Smith, E.: Leo: A programming language for formally verified, zero-knowledge applications. Cryptology ePrint Archive (2021)

25. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. In: ACM TOCL **19**(3) (2018)
26. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS (2013)
27. Cimatti, A., Mover, S., Tonetta, S.: Smt-based verification of hybrid systems. In: AAAI (2012)
28. Cohen, C.: Pragmatic quotient types in coq. In: ITP (2013)
29. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: `SMT-RAT`: an open source `C++` toolbox for strategic and parallel SMT solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 360–368. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_26
30. Cox, D., Little, J., OShea, D.: Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra. Springer Science & Business Media (2013)
31. Davenport, J.: The axiom system (1992)
32. developers, M.: Monero technical specs. https://monerodocs.org/technical-specs/ (2022)
33. D'silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. Comput.-Aided Design Integr. Circ. Syst. **27**(7) (2008)
34. Dummit, D.S., Foote, R.M.: Abstract algebra, vol. 3. Wiley Hoboken (2004)
35. Dutertre, B.: Yices 2.2. In: CAV (2014)
36. Eberhardt, J., Tai, S.: ZoKrates–scalable privacy-preserving off-chain computations. In: IEEE Blockchain (2018)
37. Eisenbud, D., Grayson, D.R., Stillman, M., Sturmfels, B.: Computations in algebraic geometry with Macaulay 2, vol. 8. Springer Science & Business Media (2001)
38. Enderton, H.B.: A mathematical introduction to logic. Elsevier (2001)
39. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Systematic Generation Of Fast Elliptic Curve Cryptography Implementations. Tech. rep, MIT (2018)
40. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic: With proofs, without compromises. ACM SIGOPS Oper. Syst. Rev. **54**(1) (2020)
41. Faugère, J.C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In: ISSAC. ACM (2002)
42. Faugere, J.C., Gianni, P., Lazard, D., Mora, T.: Efficient computation of zero-dimensional Gröbner bases by change of ordering. J. Symb. Comput. **16**(4) (1993)
43. Faugére, J.C.: A new efficient algorithm for computing gröbner bases (f4). J. Pure Appl. Algebra **139**(1), 61–88 (1999)
44. Finance, Y.: Monero quote. https://finance.yahoo.com/quote/XMR-USD/ (2022) Accessed 30 June 2022
45. Finance, Y.: Zcash quote. https://finance.yahoo.com/quote/ZEC-USD/ (2022). Accessed 30 June 2022
46. Frankle, J., Park, S., Shaar, D., Goldwasser, S., Weitzner, D.: Practical accountability of secret processes. In: USENIX Security (2018)
47. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. J. Satisfiability, Boolean Modeling and Comput. 1(3–4) (2006)
48. Gao, S.: Counting zeros over finite fields with Gröbner bases. Ph.D. thesis, Master's thesis, Carnegie Mellon University (2009)

49. GAP – Groups, Algorithms, and Programming, Version 4.13dev. www.gap-system.org (this year)
50. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems. In: STOC (1985)
51. Gonthier, G., et al.: A machine-checked proof of the odd order theorem. In: ITP, pp. 163–179 (2013)
52. Greuel, G.M., Pfister, G., Schönemann, H.: Singular-a computer algebra system for polynomial computations. In: Symbolic computation and automated reasoning, pp. 227–233. AK Peters/CRC Press (2001)
53. Grubbs, P., Arun, A., Zhang, Y., Bonneau, J., Walfish, M.: {Zero-Knowledge} middleboxes. In: USENIX Security (2022)
54. Hader, T.: Non-Linear SMT-Reasoning over Finite Fields. Ph.D. thesis, TU Wien (2022), mS Thesis
55. Hader, T., Kovács, L.: Non-linear SMT-reasoning over finite fields. In: SMT (2022). http://ceur-ws.org/Vol-3185/extended3245.pdf extended Abstract
56. Heath, D., Kolesnikov, V.: Stacked garbling for disjunctive zero-knowledge proofs. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques (2020)
57. Heath, L.S., Loehr, N.A.: New algorithms for generating conway polynomials over finite fields. J. Symb. Comput. (2004)
58. Heck, A., Koepf, W.: Introduction to MAPLE, vol. 1993 (1993)
59. Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash protocol specification. https://raw.githubusercontent.com/zcash/zips/master/protocol/protocol.pdf (2016)
60. Jovanović, D.: Solving nonlinear integer arithmetic with MCSAT. In: VMCAI (2017)
61. Jovanović, D., De Moura, L.: Solving non-linear arithmetic. ACM Commun. Comput. Algebra 46(3/4) (2013)
62. Jovanović, D., Moura, L.d.: Cutting to the chase solving linear integer arithmetic. In: CADE (2011)
63. Kamara, S., Moataz, T., Park, A., Qin, L.: A decentralized and encrypted national gun registry. In: IEEE S&P (2021)
64. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-aided reasoning: ACL2 case studies, vol. 4. Springer Science & Business Media (2013)
65. Komendantsky, V., Konovalov, A., Linton, S.: View of computer algebra data from coq. In: International Conference on Intelligent Computer Mathematics (2011)
66. Kotzias, P., Razaghpanah, A., Amann, J., Paterson, K.G., Vallina-Rodriguez, N., Caballero, J.: Coming of age: A longitudinal study of tls deployment. In: IMC (2018)
67. Lazard, D.: A new method for solving algebraic systems of positive dimension. Discr. Appl. Math. **33**, 1–3 (1991)
68. Lazard, D.: Solving zero-dimensional algebraic systems. J. Symb. Comput. **13**(2), 117–131 (1992)
69. libsnark. https://github.com/scipr-lab/libsnark
70. Maréchal, A., Fouilhé, A., King, T., Monniaux, D., Périn, M.: Polyhedral approximation of multivariate polynomials using handelman's theorem. In: VMCAI (2016)
71. McEliece, R.J.: Finite fields for computer scientists and engineers, vol. 23. Springer Science & Business Media (2012)
72. Meurer, A., et al.: Sympy: symbolic computing in python. PeerJ Comput. Sci. **3**, e103 (2017)

73. Moura, L.d., Bjørner, N.: Z3: An efficient smt solver. In: TACAS (2008)
74. Moura, L.d., Jovanović, D.: A model-constructing satisfiability calculus. In: VMCAI (2013)
75. Moura, L.d., Kong, S., Avigad, J., Doorn, F.v., Raumer, J.v.: The lean theorem prover (system description). In: CADE (2015)
76. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. arXiv:2006.01621 (2020)
77. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: CAV (2018)
78. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract davis-putnam-logemann-loveland procedure to DPLL(T). J. ACM (2006)
79. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
80. Noir. https://noir-lang.github.io/book/index.html
81. Ozdemir, A., Brown, F., Wahby, R.S.: Circ: Compiler infrastructure for proof systems, software verification, and more. In: IEEE S&P (2022)
82. Ozdemir, A., Kremer, G., Tinelli, C., Barrett, C.: Satisfiability modulo finite fields (2023), https://eprint.iacr.org/2023/091, (Full version)
83. Parker, R.: Finite fields and conway polynomials (1990), talk at the IBM Heidelberg Scientific Center. Cited by Scheerhorn
84. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. Commun. ACM **59**(2), 103–112 (2016)
85. Philipoom, J.: Correct-by-construction finite field arithmetic in Coq. Ph.D. thesis, Massachusetts Institute of Technology (2018)
86. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS (1998)
87. Rabin, M.O.: Probabilistic algorithms in finite fields. SIAM Journal on computing 9(2) (1980)
88. Rabinowitsch, J.L.: Zum hilbertschen nullstellensatz. Mathematische Annalen 102 (1930), https://doi.org/10.1007/BF01782361
89. Sasson, E.B., et al.: Zerocash: Decentralized anonymous payments from bitcoin. In: IEEE S&P (2014)
90. Scheerhorn, A.: Trace-and Norm-compatible Extensions of Finite Fields. Applicable Algebra in Engineering, Communication and Computing (1992)
91. Schwabe, P., Viguier, B., Weerwag, T., Wiedijk, F.: A coq proof of the correctness of x25519 in tweetnacl. In: CSF (2021)
92. Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M.: Resolving the conflict between generality and plausibility in verified computation. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp. 71–84 (2013)
93. Shankar, N.: Automated deduction for verification. CSUR **41**(4) (2009)
94. Thaler, J.: Proofs, Arguments, and Zero-Knowledge (2022)
95. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: PLDI (2014)
96. Tung, V.X., Khanh, T.V., Ogawa, M.: raSAT: An smt solver for polynomial constraints. In: IJCAR (2016)
97. Vella, L., Alt, L.: On satisfiability of polynomial equations over large prime field. In: SMT (2022). http://ceur-ws.org/Vol-3185/extended9913.pdf extended Abstract
98. Weispfenning, V.: Quantifier elimination for real algebra-the quadratic case and beyond. Appl. Algebra Eng., Commun. Comput. **8**(2) (1997)

99. Wen-Tsún, W.: A zero structure theorem for polynomial-equations-solving and its applications. In: European Conference on Computer Algebra (1987)
100. Wolfram, S.: Mathematica: a system for doing mathematics by computer. Addison Wesley Longman Publishing Co., Inc. (1991)
101. Zimmermann, P., et al.: Computational mathematics with SageMath. SIAM (2018)
102. Zinc. https://zinc.matterlabs.dev/

Solving String Constraints Using SAT

Kevin Lotz[1](✉), Amit Goel[2], Bruno Dutertre[2], Benjamin Kiesl-Reiter[2], Soonho Kong[2], Rupak Majumdar[2], and Dirk Nowotka[1]

[1] Department of Computer Science, Kiel University, Kiel, Germany
{kel,dn}@informatik.uni-kiel.de
[2] Amazon Web Services, Seattle, USA
{amgoel,dutebrun,benkiesl,soonho,rumajumd}@amazon.com

Abstract. String solvers are automated-reasoning tools that can solve combinatorial problems over formal languages. They typically operate on restricted first-order logic formulas that include operations such as string concatenation, substring relationship, and regular expression matching. String solving thus amounts to deciding the satisfiability of such formulas. While there exists a variety of different string solvers, many string problems cannot be solved efficiently by any of them. We present a new approach to string solving that encodes input problems into propositional logic and leverages incremental SAT solving. We evaluate our approach on a broad set of benchmarks. On the logical fragment that our tool supports, it is competitive with state-of-the-art solvers. Our experiments also demonstrate that an eager SAT-based approach complements existing approaches to string solving in this specific fragment.

1 Introduction

Many problems in software verification require reasoning about strings. To tackle these problems, numerous *string solvers*—automated decision procedures for quantifier-free first-order theories of strings and string operations—have been developed over the last years. These solvers form the workhorse of automated-reasoning tools in several domains, including web-application security [19,31,33], software model checking [15], and conformance checking for cloud-access-control policies [2,30].

The general theory of strings relies on deep results in combinatorics on words [5,16,23,29]; unfortunately, the related decision procedures remain intractable in practice. Practical string solvers achieve scalability through a judicious mix of heuristics and restrictions on the language of constraints.

We present a new approach to string solving that relies on an *eager* reduction to the Boolean satisfiability problem (SAT), using incremental solving and unsatisfiable-core analysis for completeness and scalability. Our approach supports a theory that contains Boolean combinations of regular membership constraints and equality constraints on string variables, and captures a large set of practical queries [6].

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 187–208, 2023.
https://doi.org/10.1007/978-3-031-37703-7_9

Our solving method iteratively searches for satisfying assignments up to a length bound on each string variable; it stops and reports unsatisfiability when the search reaches computed upper bounds without finding a solution. Similar to the solver Woorpje [12], we formulate regular membership constraints as reachability problems in nondeterministic finite automata. By bounding the number of transitions allowed by each automaton, we obtain a finite problem that we encode into propositional logic. To cut down the search space of the underlying SAT problem, we perform an *alphabet reduction* step (SMT-LIB string constraints are defined over an alphabet of $3 \cdot 2^{16}$ letters and a naive reduction to SAT does not scale). Inspired by bounded model checking [8], we iteratively increase bounds and utilize an incremental SAT solver to solve the resulting series of propositional formulas. We perform an unsatisfiable-core analysis after each unsatisfiable incremental call to increase only the bounds of a minimal subset of variables until a theoretical upper bound is reached.

We have evaluated our solver on a large set of benchmarks. The results show that our SAT-based approach is competitive with state-of-the-art SMT solvers in the logical fragment that we support. It is particularly effective on satisfiable instances.

Closest to our work is the Woorpje solver [12], which also employs an eager reduction to SAT. Woorpje reduces systems of word equations with linear constraints to a single Boolean formula and calls a SAT solver. An extension can also handle regular membership constraints [21]. However, Woorpje does not handle the full language of constraints considered here and does not employ the reduction and incremental solving techniques that make our tool scale in practice. More importantly, in contrast to our solver, Woorpje is not complete—it does not terminate on unsatisfiable instances.

Other solvers such as Hampi [19] and Kaluza [31] encode string problems into constraints on fixed-size bit-vector, which can be solved by reduction to SAT. These tools support expressive constraints but they require a user-provided bound on the length of string variables.

Further from our work are approaches based on the *lazy* SMT paradigm, which tightly integrates dedicated, heuristic, theory solvers for strings using the CDCL(T) architecture (also called DPLL(T) in early papers). Solvers that follow this paradigm include Ostrich [11], Z3 [25], Z3str4 [24], cvc5 [3], Z3str3RE [7], Trau [1], and CertiStr [17]. Our evaluation shows that our eager approach is competitive with lazy solvers overall, but it also shows that combining both types of solvers in a portfolio is most effective. Our eager approach tends to perform best on satisfiable instances while lazy approaches work better on unsatisfiable problems.

2 Preliminaries

We assume a fixed alphabet Σ and a fixed set of variables Γ. Words of Σ^* are denoted by w, w', w'', etc. Variables are denoted by $\mathsf{x}, \mathsf{y}, \mathsf{z}$. Our decision procedure supports the theory described in Fig. 1.

$$\begin{aligned} F &:= F \lor F \mid F \land F \mid \neg F \mid \text{Atom} \\ \text{Atom} &:= \mathsf{x} \mathrel{\dot\in} RE \mid \mathsf{x} \doteq \mathsf{y} \\ RE &:= RE \cup RE \mid RE \cdot RE \mid RE^* \mid RE \cap RE \mid ? \mid w \end{aligned}$$

Fig. 1. Syntax: x and y denote string variables and w denotes a word of Σ^*. The symbol ? is the wildcard character.

Atoms in this theory include *regular membership constraints* (or *regular constraints* for short) of the form $\mathsf{x} \mathrel{\dot\in} RE$, where RE is a regular expression, and *variable equations* of the form $\mathsf{x} \doteq \mathsf{y}$. Concatenation is not allowed in equations.

Regular expressions are defined inductively using union, concatenation, intersection, and the Kleene star. Atomic regular expressions are constant words $w \in \Sigma^*$ and the wildcard character ?, which is a placeholder for an arbitrary symbol $c \in \Sigma$. All regular expressions are grounded, meaning that they do not contain variables. We use the symbols $\mathrel{\dot\notin}$ and $\mathrel{\dot\neq}$ as a shorthand notation for negations of atoms using the respective predicate symbols. The following is an example formula in our language: $\neg(\mathsf{x} \mathrel{\dot\in} \mathrm{a} \cdot ?^* \land \mathsf{y} \mathrel{\dot\in} ?^* \cdot \mathrm{b}) \lor \mathsf{x} \mathrel{\dot\neq} \mathsf{y} \lor \mathsf{x} \mathrel{\dot\in} \mathrm{a} \cdot \mathrm{b}$.

Using our basic syntax, we can define additional relations, such as *constant equations* $\mathsf{x} \doteq w$, and *prefix and suffix constraints*, written $w \mathrel{\dot\sqsubseteq} \mathsf{x}$ and $w \mathrel{\dot\sqsupseteq} \mathsf{x}$, respectively. Even though these relations can be expressed as regular constraints (e.g., the prefix constraint $\mathrm{ab} \mathrel{\dot\sqsubseteq} \mathsf{x}$ can be expressed as $\mathsf{x} \mathrel{\dot\in} \mathrm{a} \cdot \mathrm{b} \cdot ?^*$), we can generate more efficient reductions to SAT by encoding them explicitly.

This string theory is not as expressive as others, since it does not include string concatenation, but it still has important practical applications. It is used in the Zelkova tool described by Backes, et al. [2] to support analysis of AWS security policies. Zelkova is a major industrial application of SMT solvers [30].

Given a formula ψ, we denote by *atoms*(ψ) the set of atoms occurring in ψ, by $V(\psi)$ the set of variables occurring in ψ, and by $\Sigma(\psi)$ the set of constant symbols occurring in ψ. We call $\Sigma(\psi)$ the *alphabet of* ψ. Similarly, given a regular expression R, we denote by $\Sigma(R)$ the set of characters occurring in R. In particular, we have $\Sigma(?) = \emptyset$.

We call a formula *conjunctive* if it is a conjunction of literals and we call it a *clause* if it is a disjunction of literals. We say that a formula is in *normal form* if it is a conjunctive formula without unnegated variable equations. Every conjunctive formula can be turned into normal form by substitution, i.e., by repeatedly rewriting $\psi \land \mathsf{x} \doteq \mathsf{y}$ to $\psi[\mathsf{x} := \mathsf{y}]$. If ψ is in negation normal form (NNF), meaning that the negation symbol occurs only directly in front of atoms, we denote by *lits*(ψ) the set of literals occurring in ψ. We say that an atom a occurs with *positive polarity* in ψ if $a \in$ *lits*(ψ) and that it occurs with *negative polarity* in ψ if $\neg a \in$ *lits*(ψ); we denote the respective sets of atoms of ψ by *atoms*$^+(\psi)$ and *atoms*$^-(\psi)$. The notion of polarity can be extended to arbitrary formulas (not necessarily in NNF), intuitively by considering polarity in a formula's corresponding NNF (see [26] for details).

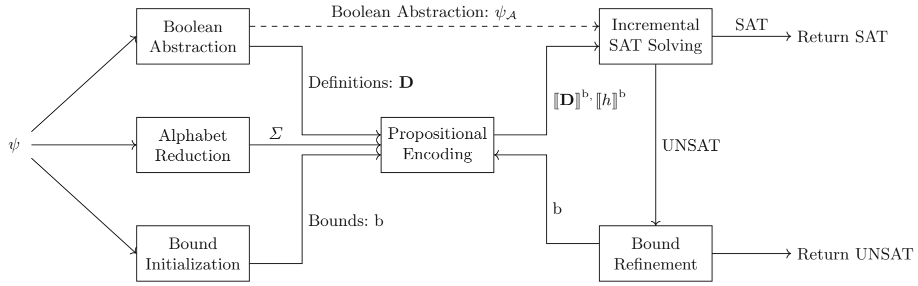


Fig. 2. Overview of the solving process.

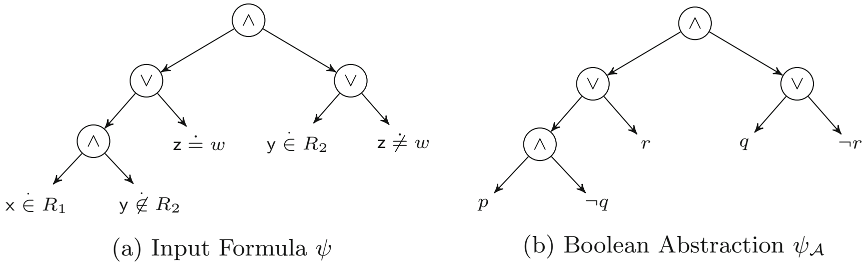
The semantics of our language is standard. A regular expression R defines a regular language $\mathcal{L}(R)$ over Σ in the usual way. An *interpretation* is a mapping (also called a *substitution*) $h\colon \Gamma \to \Sigma^*$ from string variables to words. Atoms are interpreted as usual, and a *model* (also called a *solution*) is an interpretation that makes a formula evaluate to true under the usual semantics of the Boolean connectives.

3 Overview

Our solving method is illustrated in Fig. 2. It first performs three preprocessing steps that generate a Boolean abstraction of the input formula, reduce the size of the input alphabet, and initialize bounds on the lengths of all string variables. After preprocessing, we enter an encode-solve-and-refine loop that iteratively queries a SAT solver with a problem encoding based on the current bounds and refines the bounds after each unsatisfiable solver call. We repeat this loop until either the propositional encoding is satisfiable, in which case we conclude satisfiability of the input formula, or each bound has reached a theoretical upper bound, in which case we conclude unsatisfiability.

Generating the Boolean Abstraction. We abstract the input formula ψ by replacing each theory atom $a \in \mathit{atoms}(\psi)$ with a new Boolean variable $\mathbf{d}(a)$, and keep track of the mapping between a and $\mathbf{d}(a)$. This gives us a *Boolean abstraction* $\psi_{\mathcal{A}}$ of ψ and a set $\mathbf{D}$ of definitions, where each definition expresses the relationship between an atom a and its corresponding Boolean variable $\mathbf{d}(a)$. If a occurs with only one polarity in ψ, we encode the corresponding definition as an implication, i.e., as $\mathbf{d}(a) \rightarrow a$ or as $\neg\,\mathbf{d}(a) \rightarrow \neg a$, depending on the polarity of a. Otherwise, if a occurs with both polarities, we encode it as an equivalence consisting of both implications. This encoding, which is based on ideas behind the well-known *Plaisted-Greenbaum transformation* [28], ensures that the formulas ψ and $\psi_{\mathcal{A}} \wedge \bigwedge_{d \in \mathbf{D}} d$ are equisatisfiable. An example is shown in Fig. 3.

Reducing the Alphabet. In the SMT-LIB theory of strings [4], the alphabet Σ comprises $3 \cdot 2^{16}$ letters, but we can typically use a much smaller alphabet without affecting satisfiability. In Sect. 4, we show that using $\Sigma(\psi)$ and one extra


∧
∨
∨
∧
$\mathsf{z} \doteq w$
$\mathsf{y} \mathrel{\dot\in} R_2$
$\mathsf{z} \not\doteq w$
$\mathsf{x} \mathrel{\dot\in} R_1$
$\mathsf{y} \mathrel{\dot\notin} R_2$
∧
∨
∨
∧
r
q
$\neg r$
p
$\neg q$

(a) Input Formula ψ

(b) Boolean Abstraction $\psi_{\mathcal{A}}$

Fig. 3. Example of Boolean abstraction. The formula ψ, whose expression tree is shown on the left, results in the Boolean abstraction illustrated on the right, where p, q, and r are fresh Boolean variables. We additionally get the definitions $p \rightarrow \mathsf{x} \mathrel{\dot\in} R_1$, $q \leftrightarrow \mathsf{y} \mathrel{\dot\in} R_2$, and $r \leftrightarrow \mathsf{z} \doteq w$. We use an implication (instead of an equivalence) for atom $\mathsf{x} \mathrel{\dot\in} R_1$ since it occurs only with positive polarity within ψ.

character per string variable is sufficient. Reducing the alphabet is critical for our SAT encoding to be practical.

Initializing Bounds. A model for the original first-order formula ψ is a substitution $h : \Gamma \rightarrow \Sigma^*$ that maps each string variable to a word of arbitrary length such that ψ evaluates to true. As we use a SAT solver to find such substitutions, we need to bound the lengths of strings, which we do by defining a bound function $\mathrm{b} : \Gamma \rightarrow \mathbb{N}$ that assigns an upper bound to each string variable. We initialize a small upper bound for each variable, relying on simple heuristics. If the bounds are too small, we increase them in a later refinement step.

Encoding, Solving, and Refining Bounds. Given a bound function b, we build a propositional formula $\llbracket\psi\rrbracket^{\mathrm{b}}$ that is satisfiable if and only if the original formula ψ has a solution h such that $|h(\mathsf{x})| \leq \mathrm{b}(\mathsf{x})$ for all $\mathsf{x} \in \Gamma$. We encode $\llbracket\psi\rrbracket^{\mathrm{b}}$ as the conjunction $\psi_{\mathcal{A}} \wedge \llbracket\mathbf{D}\rrbracket^{\mathrm{b}} \wedge \llbracket h\rrbracket^{\mathrm{b}}$, where $\psi_{\mathcal{A}}$ is the Boolean abstraction of ψ, $\llbracket\mathbf{D}\rrbracket^{\mathrm{b}}$ is an encoding of the definitions $\mathbf{D}$, and $\llbracket h\rrbracket^{\mathrm{b}}$ is an encoding of the set of possible substitutions. We discuss details of the encoding in Sect. 5. A key property is that it relies on *incremental SAT solving under assumptions* [13]. Increasing bounds amounts to adding new clauses to the formula $\llbracket\psi\rrbracket^{\mathrm{b}}$ and fixing a set of assumptions, i.e., temporarily fixing the truth values of a set of Boolean variables. If $\llbracket\psi\rrbracket^{\mathrm{b}}$ is satisfiable, we can construct a substitution h from a Boolean model ω of $\llbracket\psi\rrbracket^{\mathrm{b}}$. Otherwise, we examine an unsatisfiable core (i.e., an unsatisfiable subformula) of $\llbracket\psi\rrbracket^{\mathrm{b}}$ to determine whether increasing the bounds may give a solution and, if so, to identify the variables whose bounds must be increased. In Sect. 6, we explain in detail how we analyze unsatisfiable cores, increase bounds, and conclude unsatisfiability.

4 Reducing the Alphabet

In many applications, the alphabet Σ is large—typically Unicode or an approximation of Unicode as defined in the SMT-LIB standard—but formulas use much fewer symbols (less than 100 symbols is common in our experiments). In order to check the satisfiability of a formula ψ, we can restrict the alphabet to the symbols that occur in ψ and add one extra character per variable. This allows us to produce compact propositional encodings that can be solved efficiently in practice.

To prove that such a reduced alphabet A is sufficient, we show that a model $h\colon \Gamma \to \Sigma^*$ of ψ can be transformed into a model $h'\colon \Gamma \to A^*$ of ψ by replacing characters of Σ that do not occur in ψ by new symbols—one new symbol per variable of ψ. For example, suppose $V(\psi) = \{\mathsf{x}_1, \mathsf{x}_2\}$, $\Sigma(\psi) = \{\mathrm{a}, \mathrm{c}, \mathrm{d}\}$, and h is a model of ψ such that $h(\mathsf{x}_1) = \mathrm{abcdef}$ and $h(\mathsf{x}_2) = \mathrm{abbd}$. We introduce two new symbols $\alpha_1, \alpha_2 \in \Sigma \setminus \Sigma(\psi)$, define $h'(\mathsf{x}_1) = \mathrm{a}\alpha_1\mathrm{cd}\alpha_1\alpha_1$ and $h'(\mathsf{x}_2) = \mathrm{a}\alpha_2\alpha_2\mathrm{d}$, and argue that h' is a model as well.

More generally, assume B is a subset of Σ and n is a positive integer such that $|B| \leq |\Sigma| - n$. We can then pick n distinct symbols $\alpha_1, \dots, \alpha_n$ from $\Sigma \setminus B$. Let A be the set $B \cup \{\alpha_1, \dots, \alpha_n\}$. We construct n functions $f_1, \dots, f_n$ from Σ to A by setting $f_i(a) = a$ if $a \in B$, and $f_i(a) = \alpha_i$ otherwise. We extend f_i to words of Σ^* in the natural way: $f_i(\varepsilon) = \varepsilon$ and $f_i(a \cdot w) = f_i(a) \cdot f_i(w)$. This construction satisfies the following property:

Lemma 4.1. *Let $f_1, \dots, f_n$ be mappings as defined above, and let $i, j \in 1, \dots, n$ such that $i \neq j$. Then, the following holds:*

1. *If a and b are distinct symbols of Σ, then $f_i(a) \neq f_j(b)$.*
2. *If w and w' are distinct words of Σ^*, then $f_i(w) \neq f_j(w')$.*

Proof. The first part is an easy case analysis. For the second part, we have that $|f_i(w)| = |w|$ and $|f_j(w')| = |w'|$, so the statement holds if w and w' have different lengths. Assume now that w and w' have the same length and let v be the longest common prefix of w and w'. Since w and w' are distinct, we have that $w = v \cdot a \cdot u$ and $w' = v \cdot b \cdot u'$, where $a \neq b$ are symbols of Σ and u and u' are words of Σ^*. By the first part, we have $f_i(a) \neq f_j(b)$, so $f_i(w)$ and $f_j(w')$ must be distinct. □

The following lemma can be proved by induction on R.

Lemma 4.2. *Let $f_1, \dots, f_n$ be mappings as defined above and let R be a regular expression with $\Sigma(R) \subseteq B$. Then, for all words $w \in \Sigma^*$ and all $i \in 1, \dots, n$, $w \in \mathcal{L}(R)$ if and only if $f_i(w) \in \mathcal{L}(R)$.*

Given a subset A of Σ, we say that ψ is satisfiable in A if there is a model $h\colon V(\psi) \to A^*$ of ψ. We can now prove the main theorem of this section, which shows how to reduce the alphabet while maintaining satisfiability.

Theorem 4.3. *Let ψ be a formula with at most n string variables $\mathsf{x}_1, \dots, \mathsf{x}_n$ such that $|\Sigma(\psi)| + n \leq |\Sigma|$. Then, ψ is satisfiable if and only if it is satisfiable in an alphabet $A \subseteq \Sigma$ of cardinality $|A| = |\Sigma(\psi)| + n$.*

Proof. We set $B = \Sigma(\psi)$ and use the previous construction. So the alphabet $A = B \cup \{\alpha_1, \dots, \alpha_n\}$ has cardinality $|\Sigma(\psi)| + n$, where $\alpha_1, \dots \alpha_n$ are distinct symbols of $\Sigma \setminus B$. We can assume that ψ is in disjunctive normal form, meaning that it is a disjunction of the form $\psi = \psi_1 \vee \cdots \vee \psi_m$, where each ψ_t is a conjunctive formula. If ψ is satisfiable, then one of the disjuncts ψ_k is satisfiable and we have $\Sigma(\psi_k) \subseteq B$. We can turn ψ_k into normal form by eliminating all variable equalities of the form $\mathsf{x}_i \doteq \mathsf{x}_j$ from ψ_k, resulting in a conjunction φ_k of literals of the form $\mathsf{x}_i \mathrel{\dot\in} R$, $\mathsf{x}_i \mathrel{\dot\notin} R$, or $\mathsf{x}_i \mathrel{\dot\neq} \mathsf{x}_j$. Clearly, for any $A \subseteq \Sigma$, φ_k is satisfiable in A if and only if ψ_k is satisfiable in A.

Let $h\colon V(\varphi_k) \to \Sigma^*$ be a model of φ_k and define the mapping $h'\colon V(\varphi_k) \to A^*$ as $h'(\mathsf{x}_i) = f_i(h(\mathsf{x}_i))$. We show that h' is a model of φ_k. Consider a literal l of φ_k. We have three cases:

- l is of the form $\mathsf{x}_i \mathrel{\dot\in} R$ where $\Sigma(R) \subseteq \Sigma(\psi) = B$. Since h satisfies φ_k, we must have $h(\mathsf{x}_i) \in \mathcal{L}(R)$ so $h'(\mathsf{x}_i) = f_i(h(\mathsf{x}_i))$ is also in $\mathcal{L}(\mathcal{R})$ by Lemma 4.2.
- l is of the form $\mathsf{x}_i \mathrel{\dot\notin} R$ with $\Sigma(R) \subseteq B$. Then, $h'(\mathsf{x}_i) \notin \mathcal{L}(R)$ and we can conclude $h'(\mathsf{x}_i) \notin \mathcal{L}(R)$ again by Lemma 4.2.
- l is of the form $\mathsf{x}_i \mathrel{\dot\neq} \mathsf{x}_j$. Since h satisfies φ_k, we must have $i \neq j$ and $h(\mathsf{x}_i) \neq h(\mathsf{x}_j)$, which implies $h'(\mathsf{x}_i) = f_i(h(\mathsf{x}_i)) \neq f_j(h(\mathsf{x}_j)) = h'(\mathsf{x}_j)$ by Lemma 4.1.

All literals of φ_k are then satisfied by h', hence φ_k is satisfiable in A and thus so is ψ_k. It follows that ψ is satisfiable in A. □

The reduction presented here can be improved and generalized. For example, it can be worthwhile to use different alphabets for different variables or to reduce large character intervals to smaller sets.

5 Propositional Encodings

Our algorithm performs a series of calls to a SAT solver. Each call determines the satisfiability of the propositional encoding $[\![\psi]\!]^{\mathrm{b}}$ of ψ for some upper bounds b. Recall that $[\![\psi]\!]^{\mathrm{b}} = \psi_{\mathcal{A}} \wedge [\![h]\!]^{\mathrm{b}} \wedge [\![\mathbf{D}]\!]^{\mathrm{b}}$, where $\psi_{\mathcal{A}}$ is the Boolean abstraction of ψ, $[\![h]\!]^{\mathrm{b}}$ is an encoding of the set of possible substitutions, and $[\![\mathbf{D}]\!]^{\mathrm{b}}$ is an encoding of the theory-literal definitions, both bounded by b. Intuitively, $[\![h]\!]^{\mathrm{b}}$ tells the SAT solver to "guess" a substitution, $[\![\mathbf{D}]\!]^{\mathrm{b}}$ makes sure that all theory literals are assigned proper truth values according to the substitution, and $\psi_{\mathcal{A}}$ forces the evaluation of the whole formula under these truth values.

Suppose the algorithm performs n calls and let $\mathrm{b}_k : \Gamma \to \mathbb{N}$ for $k \in 1, \dots, n$ denote the upper bounds used in the k-th call to the SAT solver. For convenience, we additionally define $\mathrm{b}_0(\mathsf{x}) = 0$ for all $\mathsf{x} \in \Gamma$. In the k-th call, the SAT solver decides whether $[\![\psi]\!]^{\mathrm{b}_k}$ is satisfiable. The Boolean abstraction $\psi_{\mathcal{A}}$, which we already discussed in Sect. 3, stays the same for each call. In the following, we thus discuss the encodings of the substitutions $[\![h]\!]^{\mathrm{b}_k}$ and of the various theory literals $[\![a]\!]^{\mathrm{b}_k}$ and $[\![\neg a]\!]^{\mathrm{b}_k}$ that are part of $[\![\mathbf{D}]\!]^{\mathrm{b}_k}$. Even though SAT solvers expect their input in CNF, we do not present the encodings in CNF to simplify the presentation, but they can be converted to CNF using simple equivalence transformations.

Most of our encodings are *incremental* in the sense that the formula for call k is constructed by only adding clauses to the formula for call $k-1$. In other words, for substitution encodings we have $[\![h]\!]^{\mathrm{b}_k} = [\![h]\!]^{\mathrm{b}_{k-1}} \wedge [\![h]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$ and for literals we have $[\![l]\!]^{\mathrm{b}_k} = [\![l]\!]^{\mathrm{b}_{k-1}} \wedge [\![l]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$, with the base case $[\![h]\!]^{\mathrm{b}_0} = [\![l]\!]^{\mathrm{b}_0} = \top$. In these cases, it is thus enough to encode the incremental additions $[\![l]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$ and $[\![h]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$ for each call to the SAT solver. Some of our encodings, however, introduce clauses that are valid only for a specific bound b_k and thus become invalid for larger bounds. We handle the deactivation of these encodings with *selector variables* as is common in incremental SAT solving.

Our encodings are correct in the following sense.[1]

Theorem 5.1. *Let l be a literal and let $\mathrm{b} : \Gamma \to \mathbb{N}$ be a bound function. Then, l has a model that is bounded by b if and only if $[\![h]\!]^{\mathrm{b}} \wedge [\![l]\!]^{\mathrm{b}}$ is satisfiable.*

5.1 Substitutions

We encode substitutions by defining for each variable $\mathsf{x} \in \Gamma$ the characters to which each of x's positions is mapped. Specifically, given x and its corresponding upper bound $\mathrm{b}(\mathsf{x})$, we represent the substitution $h(\mathsf{x})$ by introducing new variables $\mathsf{x}[1], \dots, \mathsf{x}[\mathrm{b}(\mathsf{x})]$, one for each symbol $h(\mathsf{x})[i]$ of the word $h(\mathsf{x})$. We call these variables *filler variables* and we denote the set of all filler variables by $\check{\Gamma}$. By introducing a new symbol $\lambda \notin \Sigma$, which stands for an unused filler variable, we can define h based on a substitution $\check{h}\colon \check{\Gamma} \to \Sigma_\lambda$ over the filler variables, where $\Sigma_\lambda = \Sigma \cup \{\lambda\}$:

$$h(\mathsf{x})[i] = \begin{cases} \varepsilon & \text{if } \check{h}(\mathsf{x}[i]) = \lambda \\ \check{h}(\mathsf{x}[i]) & \text{otherwise} \end{cases}$$

We use this representation of substitutions (known as "filling the positions" [18]) because it has a straightforward propositional encoding: For each variable $\mathsf{x} \in \Gamma$ and each position $i \in 1, \dots, \mathrm{b}(\mathsf{x})$, we create a set $\{h^a_{\mathsf{x}[i]} \mid a \in \Sigma_\lambda\}$ of Boolean variables, where $h^a_{\mathsf{x}[i]}$ is true if $\check{h}(\mathsf{x}[i]) = a$. We then use a propositional encoding of an *exactly-one* (EO) constraint (e.g., [20]) to assert that exactly one variable in this set must be true:

$$[\![h]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \bigwedge_{\mathsf{x}\in\Gamma} \bigwedge_{i=\mathrm{b}_{k-1}(\mathsf{x})+1}^{\mathrm{b}_k(\mathsf{x})} \mathrm{EO}(\{h^a_{\mathsf{x}[i]} \mid a \in \Sigma_\lambda\}) \tag{1}$$

$$\wedge \bigwedge_{\mathsf{x}\in\Gamma} \bigwedge_{i=\mathrm{b}_{k-1}(\mathsf{x})}^{\mathrm{b}_k(\mathsf{x})-1} h^\lambda_{\mathsf{x}[i]} \to h^\lambda_{\mathsf{x}[i+1]} \tag{2}$$

[1] Proof is omitted due to space constraints but made available for review purposes.

Constraint (2) prevents the SAT solver from considering filled substitutions that are equivalent modulo λ-substitutions—it enforces that if a position i is mapped to λ, all following positions are mapped to λ too. For instance, $ab\lambda\lambda$, $a\lambda b\lambda$, and $\lambda\lambda ab$ all correspond to the same word ab, but our encoding allows only $ab\lambda\lambda$. Thus, every Boolean assignment ω that satisfies $[\![h]\!]^{\mathrm{b}}$ encodes exactly one substitution h_ω, and for every substitution h (bounded by b) there exists a corresponding assignment ω_h that satisfies $[\![h]\!]^{\mathrm{b}}$.

5.2 Theory Literals

The only theory literals of our core language are regular constraints ($\mathsf{x} \mathrel{\dot\in} R$) and variable equations ($\mathsf{x} \doteq \mathsf{y}$) with their negations. Constant equations ($\mathsf{x} \doteq w$) as well as prefix and suffix constraints ($w \mathrel{\dot\sqsubseteq} \mathsf{x}$ and $w \mathrel{\dot\sqsupseteq} \mathsf{x}$) could be expressed as regular constraints, but we encode them explicitly to improve performance.

Regular Constraints. We encode a regular constraint $\mathsf{x} \mathrel{\dot\in} R$ by constructing a propositional formula that is true if and only if the word $h(\mathsf{x})$ is accepted by a specific nondeterministic finite automaton that accepts the language $\mathcal{L}(R)$. Let $\mathsf{x} \mathrel{\dot\in} R$ be a regular constraint and let $M = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton (with states Q, alphabet Σ, transition relation δ, initial state q_0, and accepting states F) that accepts $\mathcal{L}(R)$ and that additionally allows λ-self-transitions on every state. Given that λ is a placeholder for the empty symbol, λ-transitions do not change the language accepted by M. We allow them so that M performs exactly $\mathrm{b}(\mathsf{x})$ transitions, even for substitutions of length less than $\mathrm{b}(\mathsf{x})$. This reduces checking whether the automaton accepts a word to only evaluating the states reached after exactly $\mathrm{b}(\mathsf{x})$ transitions.

Given a model $\omega \models [\![h]\!]^{\mathrm{b}}$, we express the semantics of M in propositional logic by encoding which states are reachable after reading $h_\omega(\mathsf{x})$. To this end, we assign $\mathrm{b}(\mathsf{x}) + 1$ Boolean variables $\{S^0_q, S^1_q, \ldots, S^{\mathrm{b}(\mathsf{x})}_q\}$ to each state $q \in Q$ and assert that $\omega_h(S^i_q) = 1$ if and only if q can be reached by reading prefix $h_\omega(\mathsf{x})[1..i]$. We encode this as a conjunction $[\![(M; \mathsf{x})]\!] = [\![\mathrm{I}_{(M;\mathsf{x})}]\!] \wedge [\![\mathrm{T}_{(M;\mathsf{x})}]\!] \wedge [\![\mathrm{P}_{(M;\mathsf{x})}]\!]$ of three formulas, modelling the semantics of the initial state, the transition relation, and the predecessor relation of M. We assert that the initial state q_0 is the only state reachable after reading the prefix of length 0, i.e., $[\![\mathrm{I}_{(M;\mathsf{x})}]\!]^{b_1} = S^0_{q_0} \wedge \bigwedge_{q \in Q \setminus \{q_0\}} \neg S^0_q$. The condition is independent of the bound on x, thus we set $[\![\mathrm{I}_{(M;\mathsf{x})}]\!]^{b_k}_{b_{k-1}} = \top$ for all $k > 1$.

We encode the transition relation of M by stating that if M is in some state q after reading $h_\omega(\mathsf{x})[1..i]$, and if there exists a transition from q to q' labelled with an a, then M can reach state q' after $i+1$ transitions if $h_\omega(\mathsf{x})[i+1] = a$. This is expressed in the following formula:

$$[\![\mathrm{T}_{(M;\mathsf{x})}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \bigwedge_{i=\mathrm{b}_{k-1}(\mathsf{x})}^{\mathrm{b}_k(\mathsf{x})-1} \; \bigwedge_{(q,a)\in\mathrm{dom}(\delta)} \; \bigwedge_{q'\in\delta(q,a)} (S^i_q \wedge h^a_{\mathsf{x}[i+1]}) \rightarrow S^{i+1}_{q'}$$

The formula captures all possible forward moves from each state. We must also ensure that a state is reachable only if it has a reachable predecessor, which we encode with the following formula, where $\mathrm{pred}(q') = \{(q, a) \mid q' \in \delta(q, a)\}$:

$$[\![\mathrm{P}_{(M;\mathsf{x})}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \bigwedge_{i=\mathrm{b}_{k-1}(\mathsf{x})+1}^{\mathrm{b}_k(\mathsf{x})} \bigwedge_{q' \in Q} (S^i_{q'} \rightarrow \bigvee_{(q,a)\in \mathrm{pred}(q')} (S^{i-1}_q \wedge h^a_{\mathsf{x}[i]}))$$

The formula states that if state q' is reachable after $i \geq 1$ transitions, then there must be a reachable predecessor state $q \in \hat{\delta}(\{q_0\}, h_\omega(\mathsf{x})[1..i-1])$ such that $q' \in \delta(q, h_\omega(\mathsf{x})[i])$.

To decide whether the automaton accepts $h_\omega(\mathsf{x})$, we encode that it must reach an accepting state after $\mathrm{b}_k(\mathsf{x})$ transitions. Our corresponding encoding is only valid for the particular bound $\mathrm{b}_k(\mathsf{x})$. To account for this, we introduce a fresh selector variable s_k and define $[\![\mathrm{accept}_{\mathsf{x} \dot\in M}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = s_k \rightarrow \bigvee_{q_f \in F} S^{\mathrm{b}_k(\mathsf{x})}_{q_f}$. Analogously, we define $[\![\mathrm{reject}_{\mathsf{x} \dot\in M}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = s_k \rightarrow \bigwedge_{q_f \in F} \neg S^{\mathrm{b}_k(\mathsf{x})}_{q_f}$. In the k-th call to the SAT solver and all following calls with the same bound on x, we solve under the assumption that s_k is true. In the first call k' with $\mathrm{b}_k(\mathsf{x}) < \mathrm{b}_{k'}(\mathsf{x})$, we re-encode the condition using a new selector variable $s_{k'}$ and solve under the assumption that s_k is false and s'_k is true. The full encoding of the regular constraint $x \dot\in R$ is thus given by

$$[\![\mathsf{x} \dot\in R]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = [\![(M; \mathsf{x})]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} \wedge [\![\mathrm{accept}_{\mathsf{x} \dot\in M}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$$

and its negation $x \dot\notin R$ is encoded as

$$[\![\mathsf{x} \dot\notin R]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = [\![(M; \mathsf{x})]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} \wedge [\![\mathrm{reject}_{\mathsf{x} \dot\in M}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}.$$

Variable Equations. Let $\mathsf{x}, \mathsf{y} \in \Gamma$ be two string variables, let $l = \min(\mathrm{b}_{k-1}(\mathsf{x}), \mathrm{b}_{k-1}(\mathsf{y}))$, and let $u = \min(\mathrm{b}_k(\mathsf{x}), \mathrm{b}_k(\mathsf{y}))$. We encode equality between x and y with respect to b_k position-wise up to u:

$$[\![\mathsf{x} \doteq \mathsf{y}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \bigwedge_{i=l+1}^{u} \bigwedge_{a \in \Sigma_\lambda} (h^a_{\mathsf{x}[i]} \rightarrow h^a_{\mathsf{y}[i]}).$$

The formula asserts that for each position $i \in l+1, \ldots, u$, if $\mathsf{x}[i]$ is mapped to a symbol, then $\mathsf{y}[i]$ is mapped to the same symbol (including λ). Since our encoding of substitutions ensures that every position in a string variable is mapped to exactly one character, $[\![\mathsf{x} \doteq \mathsf{y}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$ ensures $\mathsf{x}[i] = \mathsf{y}[i]$ for $i \in l+1, \ldots, u$. In conjunction with $[\![\mathsf{x} \doteq \mathsf{y}]\!]^{\mathrm{b}_{k-1}}$, which encodes equality up to the l-th position, we have symbol-wise equality of x and y up to bound u. Thus, if $\mathrm{b}_k(\mathsf{x}) = \mathrm{b}_k(\mathsf{y})$, then the formula ensures the equality of both variables. If $\mathrm{b}_k(\mathsf{x}) > \mathrm{b}_k(\mathsf{y})$, we add $h^\lambda_{\mathsf{x}[u+1]}$ as an assumption to the solver to ensure $\mathsf{x}[i] = \lambda$ for $i \in u+1, \ldots, \mathrm{b}_k(\mathsf{x})$ and, symmetrically, we add the assumption $h^\lambda_{\mathsf{y}[u+1]}$ if $\mathrm{b}_k(\mathsf{y}) > \mathrm{b}_k(\mathsf{x})$.

For the negation $\mathsf{x} \not\doteq \mathsf{y}$, we encode that $h(\mathsf{x})$ and $h(\mathsf{y})$ must disagree on at least one position, which can happen either because they map to different symbols or because the variable with the higher bound is mapped to a longer word. As for the regular constraints, we again use selector variable s_k to deactivate the encoding for all later bounds, for which it will be re-encoded:

$$[\![\mathsf{x} \not\doteq \mathsf{y}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \begin{cases} s_k \to (\bigvee_{i=1}^{u} \bigvee_{a\in\Sigma_\lambda} (\neg h^a_{\mathsf{x}[i]} \wedge h^a_{\mathsf{y}[i]})) & \text{if } \mathrm{b}_k(\mathsf{x}) = \mathrm{b}_k(\mathsf{y}) \\ s_k \to (\bigvee_{i=1}^{u} \bigvee_{a\in\Sigma_\lambda} (\neg h^a_{\mathsf{x}[i]} \wedge h^a_{\mathsf{y}[i]})) \vee \neg h^\lambda_{\mathsf{y}[u+1]} & \text{if } \mathrm{b}_k(\mathsf{x}) < \mathrm{b}_k(\mathsf{y}) \\ s_k \to (\bigvee_{i=1}^{u} \bigvee_{a\in\Sigma_\lambda} (\neg h^a_{\mathsf{x}[i]} \wedge h^a_{\mathsf{y}[i]})) \vee \neg h^\lambda_{\mathsf{x}[u+1]} & \text{if } \mathrm{b}_k(\mathsf{x}) > \mathrm{b}_k(\mathsf{y}) \end{cases}$$

Constant Equations. Given a constant equation $\mathsf{x} \doteq w$, if the upper bound of x is less than $|w|$, the atom is trivially unsatisfiable. Thus, for all i such that $b_i(\mathsf{x}) < |w|$, we encode $\mathsf{x} \doteq w$ with a simple literal $\neg s_{\mathsf{x},w}$ and add $s_{\mathsf{x},w}$ to the assumptions. For $\mathrm{b}_k(\mathsf{x}) \geq |w|$, the encoding is based on the value of $\mathrm{b}_{k-1}(\mathsf{x})$:

$$[\![\mathsf{x} \doteq w]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \begin{cases} \bigwedge_{i=1}^{|w|} h^{w[i]}_{\mathsf{x}[i]} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) < |w| = \mathrm{b}_k(\mathsf{x}) \\ \bigwedge_{i=1}^{|w|} h^{w[i]}_{\mathsf{x}[i]} \wedge h^\lambda_{\mathsf{x}[|w|+1]} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) < |w| < \mathrm{b}_k(\mathsf{x}) \\ h^\lambda_{\mathsf{x}[|w|+1]} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) = |w| < \mathrm{b}_k(\mathsf{x}) \\ \top & \text{if } |w| < \mathrm{b}_{k-1}(\mathsf{x}) \end{cases}$$

If $\mathrm{b}_{k-1}(\mathsf{x}) < |w|$, then equality is encoded for all positions $1, \ldots, |w|$. Additionally, if $\mathrm{b}_k(\mathsf{x}) > |w|$, we ensure that the suffix of x is empty starting from position $|w| + 1$. If $\mathrm{b}_{k-1}(\mathsf{x}) = |w| < \mathrm{b}_k(\mathsf{x})$, then only the empty suffix has to be ensured. Lastly, if $|w| < \mathrm{b}_{k-1}(\mathsf{x})$, then $[\![\mathsf{x} \doteq w]\!]^{\mathrm{b}_{k-1}} \Leftrightarrow [\![\mathsf{x} \doteq w]\!]^{\mathrm{b}_k}$.

Conversely, for an inequality $\mathsf{x} \not\doteq w$, if $\mathrm{b}_k(\mathsf{x}) < |w|$, then any substitution trivially is a solution, which we simply encode with $\top$. Otherwise, we introduce a selector variable $s'_{\mathsf{x},w}$ and define

$$[\![\mathsf{x} \not\doteq w]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \begin{cases} s'_{\mathsf{x},w} \to \bigvee_{i=1}^{|w|} \neg h^{w[i]}_{\mathsf{x}[i]} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) < |w| = \mathrm{b}_k(\mathsf{x}) \\ \bigvee_{i=1}^{|w|} \neg h^{w[i]}_{\mathsf{x}[i]} \vee \neg h^\lambda_{\mathsf{x}[|w|+1]} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) < |w| < \mathrm{b}_k(\mathsf{x}) \\ \top & \text{if } |w| < \mathrm{b}_{k-1}(\mathsf{x}) \leq \mathrm{b}_k(\mathsf{x}) \end{cases}$$

If $\mathrm{b}_k(\mathsf{x}) = |w|$, then a substitution h satisfies the constraint if and only if $h(\mathsf{x})[i] \neq w[i]$ for some $i \in 1, \ldots, |w|$. If $\mathrm{b}_k(\mathsf{x}) > |w|$, in addition, h satisfies the constraint if $|h(\mathsf{x})| > |w|$. Thus, if $\mathrm{b}_k(\mathsf{x}) = |w|$, we perform solver call k under the assumption $s'_{\mathsf{x},w}$, and if $\mathrm{b}_k(\mathsf{x}) > |w|$, we perform it under the assumption $\neg s'_{\mathsf{x},w}$. Again, if $|w| < \mathrm{b}_{k-1}(\mathsf{x})$, then $[\![\mathsf{x} \not\doteq w]\!]^{\mathrm{b}_{k-1}} \Leftrightarrow [\![\mathsf{x} \not\doteq w]\!]^{\mathrm{b}_k}$.

Prefix and Suffix Constraints. A prefix constraint $w \mathrel{\dot{\sqsubseteq}} \mathsf{x}$ expresses that the first $|w|$ positions of x must be mapped exactly onto w. As with equations between a variable x and a constant word w, we could express this as a regular constraint of the form $\mathsf{x} \mathrel{\dot{\in}} w\cdot?^*$. However, we achieve a more efficient encoding simply by dropping from the encoding of $[\![\mathsf{x} \doteq w]\!]$ the assertion that the suffix of x starting at $|w+1|$ be empty. Accordingly, a negated prefix constraint $w \mathrel{\dot{\not\sqsubseteq}} \mathsf{x}$ expresses that there is an index $i \in 1, \dots, |w|$ such that the i-th position of x is mapped onto a symbol different from $w[i]$, which we encode by repurposing $[\![\mathsf{x} \mathrel{\dot{\neq}} w]\!]$ in a similar manner. Suffix constraints $w \mathrel{\dot{\sqsupseteq}} \mathsf{x}$ and $w \mathrel{\dot{\not\sqsupseteq}} \mathsf{x}$ can be encoded by analogous modifications to the encodings of $\mathsf{x} \doteq w$ and $\mathsf{x} \mathrel{\dot{\neq}} w$.

6 Refining Upper Bounds

Our procedure solves a series of SAT problems where the length bounds on string variables increase after each unsatisfiable solver call. The procedure terminates once the bounds are large enough so that further increasing them would be futile. To determine when this is the case, we rely on the upper bounds of a *shortest solution* to a formula ψ. We call a model h of ψ a shortest solution of ψ if ψ has no model h' such that $\sum_{\mathsf{x}\in\Gamma}|h'(\mathsf{x})| < \sum_{\mathsf{x}\in\Gamma}|h(\mathsf{x})|$. We first establish this bound for conjunctive formulas in normal form, where all literals are of the form $\mathsf{x} \mathrel{\dot{\neq}} \mathsf{y}$, $\mathsf{x} \mathrel{\dot{\in}} R$, or $\mathsf{x} \mathrel{\dot{\notin}} R$. Once established, we show how the bound can be generalized to arbitrary formulas.

Let φ be a formula in normal form and let $\mathsf{x}_1, \dots, \mathsf{x}_n$ be the variables of φ. For each variable x_i, we can collect all the regular constraints on x_i, that is, all the literals of the form $\mathsf{x}_i \mathrel{\dot{\in}} R$ or $\mathsf{x}_i \mathrel{\dot{\notin}} R$ that occur in φ. We can characterize the solutions to all these constraints by a single nondeterministic finite automaton M_i. If the constraints on x_i are $\mathsf{x}_i \mathrel{\dot{\in}} R_1, \dots, \mathsf{x}_i \mathrel{\dot{\in}} R_k, \mathsf{x}_i \mathrel{\dot{\notin}} R'_1 \dots, \mathsf{x}_i \mathrel{\dot{\notin}} R'_l$, then M_i is an NFA that accepts the regular language $\bigcap_{t=1}^{k} \mathcal{L}(R_t) \cap \bigcap_{t=1}^{l} \overline{\mathcal{L}(R'_t)}$, where $\overline{\mathcal{L}(R)}$ denotes the complement of $\mathcal{L}(R)$. We say that M_i accepts the regular constraints on x_i in φ. If there are no such constraints on x_i, then M_i is the one-state NFA that accepts the full language Σ^*. Let Q_i denote the set of states of M_i. If we do not take inequalities into account and if the regular constraints on x_i are satisfiable, then a shortest solution h has length $|h(\mathsf{x}_i)| \leq |Q_i|$.

Theorem 6.1 gives a bound for the general case with variable inequalities. Intuitively, we prove the theorem by constructing a single automaton $\mathcal{P}$ that takes as input a vector of words $W = (w_1, ..., w_n)^T$ and accepts W iff the substitution h_W with $h_W(\mathsf{x}_i) = w_i$ satisfies φ. To construct $\mathcal{P}$, we introduce one two-state NFA for each inequality and we then form the product of these NFAs with (slightly modified versions of) the NFAs $M_1, \dots, M_n$. We can then derive the bound of a shortest solution from the number of states of $\mathcal{P}$.

Theorem 6.1. *Let φ be a conjunctive formula in normal form over variables $\mathsf{x}_1, \dots, \mathsf{x}_n$. Let $M_i = (Q_i, \Sigma, \delta_i, q_{0,i}, F_i)$ be an NFA that accepts the regular constraints on x_i in φ and let k be the number of inequalities occurring in φ. If φ is satisfiable, then it has a model h such that*

$$|h(\mathsf{x}_i)| \leq 2^k \times |Q_1| \times \dots \times |Q_n|.$$

Proof. Let λ be a symbol that does not belong to Σ and define $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. As previously, we use λ to extend words of Σ^* by padding. Given a word $w \in \Sigma_\lambda^*$, we denote by $\hat{w}$ the word of Σ^* obtained by removing all occurrences of λ from w. We say that w is well-formed if it can be written as $w = v \cdot \lambda^t$ with $v \in \Sigma^*$ and $t \geq 0$. In this case, we have $\hat{w} = v$. Thus a well-formed word w consists of a prefix in Σ^* followed by a sequence of λs.

Let Δ be the alphabet Σ_λ^n, i.e., the letters of Δ are the n-letter words over Σ_λ. We can then represent a letter u of Δ as an n-element vector $(u_1, \ldots, u_n)$, and a word W of Δ^t can be written as an $n \times t$ matrix

$$W = \begin{pmatrix} u_{11} & \ldots & u_{t1} \\ \vdots & & \vdots \\ u_{1n} & \ldots & u_{tn} \end{pmatrix}$$

where $u_{ij} \in \Sigma_\lambda$. Each column of this matrix is a letter in Δ and each row is a word in Σ_λ^t. We denote by $p_i(W)$ the i-th row of this matrix and by $\hat{p_i}(W) = \widehat{p_i(W)}$ the word $p_i(W)$ with all occurrences of λ removed. We say that W is well-formed if the words $p_1(W), \ldots, p_n(W)$ are all well-formed. Given a well-formed word W, we can construct a mapping $h_W : \{\mathsf{x}_1, \ldots, \mathsf{x}_n\} \to \Sigma^*$ by setting $h_W(\mathsf{x}_i) = \hat{p_i}(W)$ and we have $|h_W(\mathsf{x}_i)| \leq |W| = t$.

To prove the theorem, we build an NFA $\mathcal{P}$ with alphabet Δ such that a well-formed word W is accepted by $\mathcal{P}$ iff h_W satisfies φ. The shortest well-formed W accepted by $\mathcal{P}$ has length no more than the number of states of $\mathcal{P}$ and the bound will follow.

We first extend the NFA $M_i = (Q_i, \Sigma, \delta_i, q_{0,i}, F_i)$ to an automaton M_i' with alphabet Δ. M_i' has the same set of states, initial state, and final states as M_i. Its transition relation δ_i' is defined by

$$\delta_i'(q, u) = \begin{cases} \delta_i(q, u_i) & \text{if } u_i \in \Sigma \\ \{q\} & \text{if } u_i = \lambda \end{cases}$$

One can easily check that M_i' accepts a word W iff M_i accepts $\hat{p_i}(W)$.

For an inequality $\mathsf{x}_i \ne \mathsf{x}_j$, we construct an NFA $D_{i,j} = (\{e, d\}, \Delta, \delta, e, \{d\})$ with transition function defined as follows:

$$\begin{aligned} \delta(e, u) &= \{e\} \quad \text{if } u_i = u_j \\ \delta(e, u) &= \{d\} \quad \text{if } u_i \neq u_j \\ \delta(d, u) &= \{d\}. \end{aligned}$$

This NFA has two states. It starts in state e (for "equal") and stays in e as long as the characters u_i and u_j are equal. It transitions to state d (for "different") on the first u where $u_i \neq u_j$ and stays in state d from that point. Since d is the final state, a word W is accepted by $D_{i,j}$ iff $p_i(W) \neq p_j(W)$. If W is well-formed, we also have that W is accepted by $D_{i,j}$ iff $\hat{p_i}(W) \neq \hat{p_j}(W)$.

Let $\mathsf{x}_{i_1} \ne \mathsf{x}_{j_1}, \ldots, \mathsf{x}_{i_k} \ne \mathsf{x}_{j_k}$ denote the k inequalities of φ. We define $\mathcal{P}$ to be the product of the NFAs $M_1', \ldots, M_n'$ and $D_{i_1,j_1}, \ldots, D_{i_k,j_k}$. A well-formed word

W is accepted by $\mathcal{P}$ if it is accepted by all M_i' and all D_{i_t,j_t}, which means that $\mathcal{P}$ accepts a well-formed word W iff h_W satisfies φ.

Let P be the set of states of $\mathcal{P}$. We then have $|P| \leq 2^k \times |Q_1| \times \ldots \times |Q_n|$. Assume φ is satisfiable, so $\mathcal{P}$ accepts a well-formed word W. The shortest well-formed word accepted by $\mathcal{P}$ has an accepting run that does not visit the same state twice. So the length of this well-formed word W is no more than $|P|$. The mapping h_W satisfies φ and for every x_i, it satisfies $|h_W(\mathsf{x}_i)| = |\hat{p_i}(W)| \leq |W| \leq |P| \leq 2^k \times |Q_1| \times \ldots \times |Q_n|$. □

The bound given by Theorem 6.1 holds if φ is in normal form but it also holds for a general conjunctive formula ψ. This follows from the observation that converting conjunctive formulas to normal form preserves the length of solutions. In particular, we convert $\psi \wedge \mathsf{x} \doteq \mathsf{y}$ to formula $\psi' = \psi[x := y]$ so x does not occur in ψ', but clearly, a bound for y in ψ' gives us the same bound for x in ψ.

In practice, before we apply the theorem we decompose the conjunctive formula φ into subformulas that have disjoint sets of variables. We write φ as $\varphi_1 \wedge \ldots \wedge \varphi_m$ where the conjuncts have no common variables. Then, φ is satisfiable if each conjunct φ_t is satisfiable and we derive upper bounds on the shortest solution for the variables of φ_t, which gives more precise bounds than deriving bounds from φ directly. In particular, if a variable x_i of ψ does not occur in any inequality, then the bound on $|h(\mathsf{x}_i)|$ is $|Q_i|$.

Theorem 6.1 only holds for conjunctive formulas. For an arbitrary (non-conjunctive) formula ψ, a generalization is to convert ψ into disjunctive normal form. Alternatively, it is sufficient to enumerate the subsets of $\mathit{lits}(\psi)$. Given a subset A of $\mathit{lits}(\psi)$, let us denote by d_A a mapping that bounds the length of solutions to A, i.e., any solution h to A satisfies $|h(x)| \leq d_A(x)$. This mapping d_A can be computed from Theorem 6.1. The following property gives a bound for ψ.

Proposition 6.2. *If ψ is satisfiable, then it has a model h such that for all $\mathsf{x} \in \Gamma$, it holds that $|h(\mathsf{x})| \leq \max\{d_A(x) \mid A \subseteq \mathit{lits}(\psi)\}$.*

Proof. We can assume that ψ is in negation normal form. We can then convert ψ to disjunctive normal form $\psi \Leftrightarrow \psi_1 \vee \cdots \vee \psi_n$ and we have $\mathit{lits}(\psi_i) \subseteq \mathit{lits}(\psi)$. Also, ψ is satisfiable if and only if at least one ψ_i is satisfiable and the proposition follows. □

Since there are $2^{|\mathit{lits}(\psi)|}$ subsets of $\mathit{lits}(\psi)$, a direct application of Proposition 6.2 is rarely feasible in practice. Fortunately, we can use unsatisfiable cores to reduce the number of subsets to consider.

6.1 Unsatisfiable-Core Analysis

Instead of calculating the bounds upfront, we use the unsatisfiable core produced by the SAT solver after each incremental call to evaluate whether the upper bounds on the variables exceed the upper bounds of the shortest solution. If $[\![\psi]\!]^{\mathrm{b}}$ is unsatisfiable for bounds b, then it has an unsatisfiable core

$$C = C_{\mathcal{A}} \wedge C_h \wedge \bigwedge_{a \in atoms^+(\psi)} C_a \wedge \bigwedge_{a \in atoms^-(\psi)} C_{\bar{a}}$$

with (possibly empty) subsets of clauses $C_{\mathcal{A}} \subseteq \psi_{\mathcal{A}}$, $C_h \subseteq [\![h]\!]^{\mathrm{b}}$, $C_a \subseteq (\mathbf{d}(a) \rightarrow [\![a]\!]^{\mathrm{b}})$, and $C_{\bar{a}} \subseteq (\neg\, \mathbf{d}(a) \rightarrow [\![\neg a]\!]^{\mathrm{b}})$. Here we implicitly assume $\psi_{\mathcal{A}}$, $\mathbf{d}(a) \rightarrow [\![a]\!]^{\mathrm{b}}$, and $\neg\, \mathbf{d}(a) \rightarrow [\![\neg a]\!]^{\mathrm{b}}$ to be in CNF. Let $\mathcal{C}^+ = \{a \mid C_a \neq \emptyset\}$ and $\mathcal{C}^- = \{\neg a \mid C_{\bar{a}} \neq \emptyset\}$ be the sets of literals whose encodings contain at least one clause of the core C. Using these sets, we construct the formula

$$\psi^{\mathcal{C}} = \psi_{\mathcal{A}} \wedge \bigwedge_{a \in \mathcal{C}^+} \mathbf{d}(a) \rightarrow a \wedge \bigwedge_{\neg a \in \mathcal{C}^-} \neg\, \mathbf{d}(a) \rightarrow \neg a,$$

which consists of the conjunction of the abstraction and the definitions of the literals that are contained in $\mathcal{C}^+$, respectively $\mathcal{C}^-$. Recall that ψ is equisatisfiable to the conjunction $\psi_{\mathcal{A}} \wedge \bigwedge_{d \in \mathbf{D}} d$ of the abstraction and all definitions in $\mathbf{D}$. Let ψ' denote this formula, i.e.,

$$\psi' = \psi_{\mathcal{A}} \wedge \bigwedge_{a \in atoms^+(\psi)} \mathbf{d}(a) \rightarrow a \wedge \bigwedge_{\neg a \in atoms^-(\psi)} \neg\, \mathbf{d}(a) \rightarrow \neg a.$$

The following proposition shows that it suffices to refine the bounds according to $\psi^{\mathcal{C}}$.

Proposition 6.3. *Let ψ be unsatisfiable with respect to* b *and let C be an unsatisfiable core of $[\![\psi]\!]^{\mathrm{b}}$. Then, $\psi^{\mathcal{C}}$ is unsatisfiable with respect to* b *and $\psi' \models \psi^{\mathcal{C}}$.*

Proof. By definition, we have $[\![\psi^{\mathcal{C}}]\!]^{\mathrm{b}} = \psi_{\mathcal{A}} \wedge [\![h]\!]^{\mathrm{b}} \wedge \bigwedge_{a \in \mathcal{C}^+} \mathbf{d}(a) \rightarrow [\![a]\!]^{\mathrm{b}} \wedge \bigwedge_{\neg a \in \mathcal{C}^-} \neg\, \mathbf{d}(a) \rightarrow \neg[\![\neg a]\!]^{\mathrm{b}}$. This implies $C \subseteq [\![\psi^{\mathcal{C}}]\!]^{\mathrm{b}}$ and, since C is an unsatisfiable core, $[\![\psi^{\mathcal{C}}]\!]^{\mathrm{b}}$ is unsatisfiable. That is, $\psi^{\mathcal{C}}$ is unsatisfiable with respect to b. We also have $\psi' \models \psi^{\mathcal{C}}$ since $\mathcal{C}^+ \subseteq atoms^+(\psi)$ and $\mathcal{C}^- \subseteq atoms^-(\psi)$. □

Applying Proposition 6.2 to $\psi^{\mathcal{C}}$ results in the upper bounds of the shortest solution h_C for $\psi^{\mathcal{C}}$. If $|h_C(\mathsf{x})| \leq \mathrm{b}(\mathsf{x})$ holds for all $\mathsf{x} \in \Gamma$, then ψ^C has no solution and unsatisfiability of ψ' follows from Proposition 6.3. Because ψ and ψ' are equisatisfiable, we can conclude that ψ is unsatisfiable.

Otherwise, we increase the bounds on the variables that occur in ψ^C while keeping bounds on the other variables unchanged: We construct b_{k+1} with $\mathrm{b}_k(\mathsf{x}) \leq \mathrm{b}_{k+1}(\mathsf{x}) \leq |h_C(\mathsf{x})|$ for all $\mathsf{x} \in \Gamma$, such that $\mathrm{b}_k(\mathsf{y}) < \mathrm{b}_{k+1}(\mathsf{y})$ holds for at least one $\mathsf{y} \in V(\psi^{\mathcal{C}})$. By strictly increasing at least one variable's bound, we eventually either reach the upper bounds of $\psi^{\mathcal{C}}$ and return unsatisfiability, or we eliminate it as an unsatisfiable implication of ψ. As there are only finitely many possibilities for $\mathcal{C}$ and thus for $\psi^{\mathcal{C}}$, our procedure is guaranteed to terminate.

We do not explicitly construct formula $\psi^{\mathcal{C}}$ to compute bounds on h_C as we know the set $\mathit{lits}(\psi^{\mathcal{C}}) = \mathcal{C}^+ \cup \mathcal{C}^-$. Finding upper bounds still requires enumerating all subsets of $\mathit{lits}(\psi^{\mathcal{C}})$, but we have $|\mathit{lits}(\psi^{\mathcal{C}})| \leq |\mathit{lits}(\psi)|$ and usually $\mathit{lits}(\psi^{\mathcal{C}})$ is much smaller than $\mathit{lits}(\psi)$. For example, consider the formula

$$\psi = \mathsf{z} \dot{\neq} abd \wedge (\mathsf{x} \doteq a \vee \mathsf{x} \mathrel{\dot{\in}} ab^*) \wedge \mathsf{x} \doteq \mathsf{y} \wedge (\mathsf{y} \doteq bbc \vee \mathsf{z} \in a(b|c)^*d) \wedge \mathsf{y} \mathrel{\dot{\in}} ab\cdot?^*$$

which is unsatisfiable for the bounds $\mathrm{b}(\mathsf{x}) = \mathrm{b}(\mathsf{y}) = 1$ and $\mathrm{b}(\mathsf{z}) = 4$. The unsatisfiable core C returned after solving $[\![\psi]\!]^{\mathrm{b}}$ results in the formula $\psi^{\mathcal{C}} = (\mathsf{x} \doteq a \vee \mathsf{x} \mathrel{\dot{\in}} ab^*) \wedge \mathsf{x} \doteq \mathsf{y} \wedge \mathsf{y} \mathrel{\dot{\in}} ab\cdot?^*$ containing four literals. Finding upper bounds for $\psi^{\mathcal{C}}$ thus amounts to enumerating just 2^4 subsets, which is substantially less than considering all 2^7 subsets of $\mathit{lits}(\psi)$ upfront. The conjunction of a subset of $\mathit{lits}(\psi^{\mathcal{C}})$ yielding the largest upper bounds is $\mathsf{x} \mathrel{\dot{\in}} ab^* \wedge \mathsf{x} \doteq \mathsf{y} \wedge \mathsf{y} \mathrel{\dot{\in}} ab\cdot?^*$, which simplifies to $\mathsf{x} \mathrel{\dot{\in}} ab^* \cap ab\cdot?^*$ and has a solution of length at most 2 for x and y. With bounds $\mathrm{b}(\mathsf{x}) = \mathrm{b}(\mathsf{y}) = 2$ and $\mathrm{b}(\mathsf{z}) = 4$, the formula is satisfiable.

7 Implementation

We have implemented our approach in a solver called NFA2SAT. NFA2SAT is written in RUST and uses CADICAL [9] as the backend SAT solver. We use the incremental API provided by CADICAL to solve problems under assumptions. Soundness of NFA2SAT follows from Theorem 5.1. For completeness, we rely on CADICAL's *failed* function to efficiently determine *failed assumptions*, i.e., assumption literals that were used to conclude unsatisfiability.

The procedure works as follows. Given a formula ψ, we first introduce one fresh Boolean selector variable s_l for each theory literal $l \in \mathit{lits}(\psi)$. Then, instead of adding the encoded definitions of the theory literals directly to the SAT solver, we precede them with their corresponding selector variables: for a positive literal a, we add $s_a \rightarrow (\mathbf{d}(a) \rightarrow [\![a]\!])$, and for a negative literal $\neg a$, we add $s_{\neg a} \rightarrow (\neg\, \mathbf{d}(a) \rightarrow [\![\neg a]\!])$ (considering assumptions introduced by $[\![a]\!]$ as unit clauses). In the resulting CNF formula, the new selector variables are present in all clauses that encode their corresponding definition, and we use them as assumptions for every incremental call to the SAT solver, which does not affect satisfiability. If such an assumption failed, then we know that at least one of the corresponding clauses in the propositional formula was part of an unsatisfiable core, which enables us to efficiently construct the sets $\mathcal{C}^+$ and $\mathcal{C}^-$ of positive and negative atoms present in the unsatisfiable core. As noted previously, we have $\mathit{lits}(\psi^C) = \mathcal{C}^+ \cup \mathcal{C}^-$ and hence the sets are sufficient to find bounds on a shortest model for ψ^C.

This approach is efficient for obtaining $\mathit{lits}(\psi^C)$ but since CADICAL does not guarantee that the set of failed assumptions is minimal, $\mathit{lits}(\psi^C)$ is not minimal in general. Moreover, even a minimal $\mathit{lits}(\psi^C)$ can contain too many elements for processing all subsets. To address this issue, we enumerate the subsets only if $\mathit{lits}(\psi^C)$ is small (by default, we use a limit of ten literals). In this case, we construct the automata M_i used in Theorem 6.1 for each subset, facilitating the techniques described in [7] for quickly ruling out unsatisfiable ones. Otherwise, instead of enumerating the subsets, we resort to sound approximations of upper bounds, which amounts to over-approximating the number of states without explicitly constructing the automata (c.f. [14]).

Once we have obtained upper bounds on the length of the solution of ψ^C, we increment bounds on all variables involved, except those that have reached their maximum. Our default heuristics computes a new bound that is either double the current bound of a variable or its maximum, whichever is smaller.

8 Experimental Evaluation

We have evaluated our solver on a large set of benchmarks from the ZaligVinder [22] repository[2]. The repository contains 120,287 benchmarks stemming from both academic and industrial applications. In particular, all the string problems from the SMT-LIB repository,[3] are included in the ZaligVinder repository. We converted the ZaligVinder problems to the SMT-LIB 2.6 syntax and removed duplicates. This resulted in 82,632 unique problems out of which 29,599 are in the logical fragment we support.

We compare nfa2sat with the state-of-the-art solvers cvc5 (version 1.0.3) and Z3 (version 4.12.0). The comparison is limited to these two solvers because they are widely adopted and because they had the best performance in our evaluation. Other string solvers either don't support our logical fragment (CertiStr, Woorpje) or gave incorrect answers on the benchmark problems considered here. Older, no-longer maintained, solvers have known soundness problems, as reported in [7] and [27].

We ran our experiment on a Linux server, with a timeout of 1200 s seconds CPU time and a memory limit of 16 GB. Table 1 shows the results. As a single tool, nfa2sat solves more problems than cvc5 but not as many as Z3. All three tools solve more than 98% of the problems.

The table also shows results of portfolios that combine two solvers. In a portfolio configuration, the best setting is to use both Z3 and nfa2sat. This combination solves all but 20 problems within the timeout. It also reduces the total run-time from 283,942 s for Z3 (about 79 h) to 28,914 s for the portfolio (about 8 h), that is, a 90% reduction in total solve time. The other two portfolios—namely, Z3 with cvc5 and nfa2sat with cvc5—also have better performance than a single solver, but the improvement in runtime and number of timeouts is not as large.

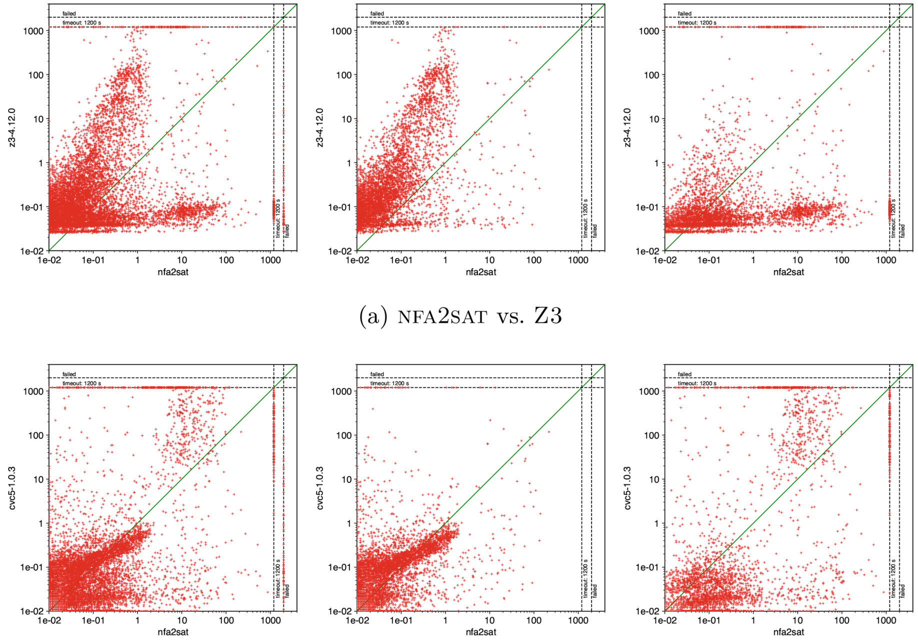
Figure 4a illustrates why nfa2sat and Z3 complement each other well. The figure shows three scatter plots that compare the runtime of nfa2sat and Z3 on our problems. The plot on the left compares the two solvers on *all* problems, the one in the middle compares them on *satisfiable* problems, and the one on the right compares them on *unsatisfiable* problems. Points in the left plot are concentrated close to the axes, with a smaller number of points near the diagonal, meaning that Z3 and nfa2sat have different runtime on most problems. The other two

[2] https://github.com/zaligvinder/zaligvinder.

[3] https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S.

Table 1. Evaluation on ZaligVinder benchmarks. The three left columns show results of individual solvers. The other three columns show results of portfolios combining two solvers.

	cvc5	Z3	nfa2sat	cvc5 Z3	nfa2sat cvc5	nfa2sat Z3
SAT	22895	22927	22922	22934	22934	22934
UNSAT	6259	6486	6405	6526	6598	6645
Timeout	445	185	206	139	67	20
Out-of-memory	0	1	66	n/a	n/a	n/a
Total Solved	29154	29413	29327	29460	29532	29579
Total Runtime (s)	655877	283942	275420	169553	126655	28914



(a) nfa2sat vs. Z3

(b) nfa2sat vs. cvc5

Fig. 4. Comparison of runtime (in seconds) with Z3 and cvc5. The left plots include all problems, the middle plots include only satisfiable problems, and the right plots include only unsatisfiable problems. The lines marked "failed" correspond to problems that are not solved because a solver ran out of memory. The lines marked "timeout" correspond to problems not solved because of a timeout (1200 s).

plots show this even more clearly: nfa2sat is faster on satisfiable problems while Z3 is faster on unsatisfiable problems. Figure 4b shows analogous scatter plots comparing nfa2sat and cvc5. The two solvers show similar performance on a large set of easy benchmarks although cvc5 is faster on problems that both solvers can solve in less than 1 s. However, CVC5 times out on 38 problems that NFA2SAT solves in less than 2 s. On unsatisfiable problems, CVC5 tends to be faster than NFA2SAT, but there is a class of problems for which NFA2SAT takes between 10 and 100 s whereas CVC5 is slower.

Overall, the comparison shows that NFA2SAT is competitive with CVC5 and Z3 on these benchmarks. We also observe that NFA2SAT tends to work better on satisfiable problems. For best overall performance, our experiments show that a portfolio of Z3 and NFA2SAT would solve all but 20 problems within the timeout, and reduce the total solve time by 90%.

9 Conclusion

We have presented the first eager SAT-based approach to string solving that is both sound and complete for a reasonably expressive fragment of string theory. Our experimental evaluation shows that our approach is competitive with the state-of-the-art lazy SMT solvers Z3 and CVC5, outperforming them on satisfiable problems but falling behind on unsatisfiable ones. A portfolio that combines our approach with these solvers—particularly with Z3—would thus yield strong performance across both types of problems.

In future work, we plan to extend our approach to a more expressive logical fragment, including more general word equations. Other avenues of research include the adaption of model checking techniques such as IC3 [10] to string problems, which we hope would lead to better performance on unsatisfiable instances. A particular benefit of the eager approach is that it enables the use of mature techniques from the SAT world, especially for proof generation and parallel solving. Producing proofs of unsatisfiability is complex for traditional CDCL(T) solvers because of the complex rewriting and deduction rules they employ. In contrast, efficiently generating and checking proofs produced by SAT solvers (using the DRAT format [32]) is well-established and practicable. A challenge in this respect would be to combine unsatisfiability proofs from a SAT solver with proof that our reduction to SAT is sound. For parallel solving, we plan to explore the use of a parallel incremental solver (such as ILINGELING [9]) as well as other possible ways to solve multiple bounds in parallel.

References

1. Abdulla, P.A., et al.: Trau: SMT solver for string constraints. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–5 (2018). https://doi.org/10.23919/FMCAD.2018.8602997
2. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8602994

3. Barbosa, H., et al.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017). www.smt-lib.org
5. Berzish, M., et al.: String theories involving regular membership predicates: From practice to theory and back. In: Lecroq, T., Puzynina, S. (eds.) Combinatorics on Words, pp. 50–64. Springer International Publishing, Cham (2021)
6. Berzish, M., et al.: Towards more efficient methods for solving regular-expression heavy string constraints. Theoretical Computer Science **943**, 50–72 (2023). https://doi.org/10.1016/j.tcs.2022.12.009, https://www.sciencedirect.com/science/article/pii/S030439752200723X
7. Berzish, M., et al.: An SMT solver for regular expressions and linear arithmetic over string length. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification, pp. 289–312. Springer International Publishing, Cham (2021)
8. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-457
9. 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)
10. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
11. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290362
12. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In: Filiot, E., Jungers, R., Potapov, I. (eds.) Reachability Problems, pp. 93–106. Springer International Publishing, Cham (2019)
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4), 543–560 (2003). https://doi.org/10.1016/S1571-0661(05)82542-3, https://www.sciencedirect.com/science/article/pii/S1571066105825423, bMC'2003, First International Workshop on Bounded Model Checking
14. Gao, Y., Moreira, N., Reis, R., Yu, S.: A survey on operational state complexity. CoRR **abs/1509.03254** (2015), http://arxiv.org/abs/1509.03254
15. Hojjat, H., Rümmer, P., Shamakhi, A.: On strings in software model checking. In: Lin, A.W. (ed.) Programming Languages and Systems, pp. 19–30. Springer International Publishing, Cham (2019)

16. Jez, A.: Word Equations in Nondeterministic Linear Space. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 80, pp. 95:1–95:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.95, http://drops.dagstuhl.de/opus/volltexte/2017/7408
17. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: A certified string solver. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 210–224. CPP 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3497775.3503691
18. Karhumäki, J., Mignosi, F., Plandowski, W.: The expressibility of languages and relations by word equations. J. ACM **47**(3), 483–505 (may 2000). https://doi.org/10.1145/337244.337255
19. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: A solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 105–116. ISSTA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1572272.1572286
20. Klieber, W., Kwon, G.: Efficient CNF encoding for selecting 1 from N objects. In: Fourth Workshop on Constraints in Formal Verification (CFV) (2007)
21. Kulczynski, M., Lotz, K., Nowotka, D., Poulsen, D.B.: Solving string theories involving regular membership predicates using SAT. In: Legunsen, O., Rosu, G. (eds.) Model Checking Software, pp. 134–151. Springer International Publishing, Cham (2022)
22. Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: Zaligvinder: A generic test framework for string solvers. J. Softw.: Evolution and Process **n/a**(n/a), e2400. https://doi.org/10.1002/smr.2400, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2400
23. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Math. USSR, Sb. 32, 129–198 (1977). https://doi.org/10.1070/SM1977v032n02ABEH002376
24. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A multi-armed string solver. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) Formal Methods, pp. 389–406. Springer International Publishing, Cham (2021)
25. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
26. Murray, N.V.: Completely non-clausal theorem proving. Artificial Intelligence **18**(1), 67–85 (1982). https://doi.org/10.1016/0004-3702(82)90011-X, https://www.sciencedirect.com/science/article/pii/000437028290011X
27. Nötzli, A., Reynolds, A., Barbosa, H., Barrett, C.W., Tinelli, C.: Even faster conflicts and lazier reductions for string solvers. 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. 205–226. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_11, https://doi.org/10.1007/978-3-031-13188-2_11
28. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation **2**(3), 293–304 (1986). https://doi.org/10.1016/S0747-7171(86)80028-1, https://www.sciencedirect.com/science/article/pii/S0747717186800281

29. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. In: 40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039), pp. 495–500 (1999). https://doi.org/10.1109/SFFCS.1999.814622
30. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 3–18. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_1
31. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: 2010 IEEE Symposium on Security and Privacy, pp. 513–528 (2010). https://doi.org/10.1109/SP.2010.38
32. Wetzler, N., Heule, M., Jr., W.A.H.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8561, pp. 422–429. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31, https://doi.org/10.1007/978-3-319-09284-3_31
33. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods Syst. Design **44**(1), 44–70 (2014). https://doi.org/10.1007/s10703-013-0189-1, https://doi.org/10.1007/s10703-013-0189-1

The Golem Horn Solver

Martin Blicha[1,2](✉), Konstantin Britikov[1], and Natasha Sharygina[1]

[1] Università della Svizzera Italiana, Lugano, Switzerland
{blichm,britik,sharygin}@usi.ch
[2] Charles University, Prague, Czech Republic

Abstract. The logical framework of Constrained Horn Clauses (CHC) models verification tasks from a variety of domains, ranging from verification of safety properties in transition systems to modular verification of programs with procedures. In this work we present Golem, a flexible and efficient solver for satisfiability of CHC over linear real and integer arithmetic. Golem provides flexibility with modular architecture and multiple back-end model-checking algorithms, as well as efficiency with tight integration with the underlying SMT solver. This paper describes the architecture of Golem and its back-end engines, which include our recently introduced model-checking algorithm TPA for deep exploration. The description is complemented by extensive evaluation, demonstrating the competitive nature of the solver.

Keywords: Constrained Horn Clauses · Model Checking

1 Introduction

The framework of *Constrained Horn Clauses* (CHC) has been proposed as a unified, purely logic-based, intermediate format for software verification tasks [33]. CHC provides a powerful way to model various verification problems, such as safety, termination, and loop invariant computation, across different domains like transition systems, functional programs, procedural programs, concurrent systems, and more [33–35,41]. The key advantage of CHC is the separation of modelling from solving, which aligns with the important software design principle—*separation of concerns*. This makes CHCs highly reusable, allowing a specialized CHC solver to be used for different verification tasks across domains and programming languages. The main focus of the front end is then to translate the source code into the language of constraints, while the back end can focus solely on the well-defined formal problem of deciding satisfiability of a CHC system.

CHC-based *verification* is becoming increasingly popular, with several frameworks developed in recent years, including SeaHorn, Korn and TriCera for C [27,28,36], JayHorn for Java [44], RustHorn for Rust [48], HornDroid for Android [18], SolCMC and SmartACE for Solidity [2,57]. A novel CHC-based approach for *testing* also shows promising results [58]. The growing demand from verifiers drives the development of specialized *Horn* solvers. Different solvers implement different techniques based on, e.g., model-checking approaches (such

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 209–223, 2023.
https://doi.org/10.1007/978-3-031-37703-7_10

as predicate abstraction [32], CEGAR [22] and IC3/PDR [16,26]), machine learning, automata, or CHC transformations. Eldarica [40] uses predicate abstraction and CEGAR as the core solving algorithm. It leverages Craig interpolation [23] not only to guide the predicate abstraction but also for acceleration [39]. Additionally, it controls the form of the interpolants with *interpolation abstraction* [46,53]. Spacer [45] is the default algorithm for solving CHCs in Z3 [51]. It extends PDR-style algorithm for nonlinear CHC [38] with under-approximations and leverages *model-based projection* for predecessor computation. Recently it was enriched with *global guidance* [37]. Ultimate TreeAutomizer [25] implements automata-based approaches to CHC solving [43,56]. HoIce [20] implements a machine-learning-based technique adapted from the ICE framework developed for discovering inductive invariants of transition systems [19]. FreqHorn [29,30] combines syntax-guided synthesis [4] with data derived from unrollings of the CHC system.

According to the results of the international competition on CHC solving CHC-COMP [24,31,54], solvers applying model-checking techniques, namely Spacer and Eldarica, are regularly outperforming the competitors. These are the solvers most often used as the back ends in CHC-based verification projects. However, only specific algorithms have been explored in these tools for CHC solving, limiting their application for diverse verification tasks. Experience from software verification and model checking of transition systems shows that in contrast to the state of affairs in CHC solving, it is possible to build a flexible infrastructure with a unified environment for multiple back-end solving algorithms. CPAchecker [6–11], and Pono [47] are examples of such tools.

This work aims to bring this flexibility to the general domain-independent framework of constrained Horn clauses. We present Golem, a new solver for CHC satisfiability, that provides a unique combination of flexibility and efficiency.[1] Golem implements several SMT-based model-checking algorithms: our recent model-checking algorithm based on *Transition Power Abstraction* (TPA) [13,14], and state-of-the-art model-checking algorithms Bounded Model Checking (BMC) [12], k-induction [55], Interpolation-based Model Checking (IMC) [49], Lazy Abstractions with Interpolants (LAWI) [50] and Spacer [45]. Golem achieves efficiency through tight integration with the underlying interpolating SMT solver OpenSMT [17,42] and preprocessing transformations based on *predicate elimination*, *clause merging* and *redundant clause elimination*. The flexible and modular framework of OpenSMT enables customization for different algorithms; its powerful interpolation modules, particularly, offer fine control (in size and strength) with multiple interpolant generation procedures. We report experimentation that confirms the advantage of multiple diverse solving techniques and shows that Golem is competitive with state-of-the-art Horn solvers on large sets of problems.[2] Overall, Golem can serve as an efficient back

[1] Golem is available at https://github.com/usi-verification-and-security/golem.

[2] This is in line with results from CHC-COMP 2021 and 2022 [24,31]. In 2022, Golem beat other solvers except Z3-Spacer in the LRA-TS, LIA-Lin and LIA-Nonlin tracks.

end for domain-specific verification tools and as a research tool for prototyping and evaluating SMT- and interpolation-based verification techniques in a unified setting.

2 Tool Overview

In this section, we describe the main components and features of the tool together with the details of its usage. For completeness, we recall the terminology related to CHCs first.

Constrained Horn Clauses. A constrained Horn clause is formula $\varphi \wedge B_1 \wedge B_2 \wedge \ldots \wedge B_n \implies H$, where φ is the *constraint*, a formula in the background theory, $B_1, \ldots, B_n$ are uninterpreted predicates, and H is an uninterpreted predicate or *false*. The antecedent of the implication is commonly denoted as the *body* and the consequent as the *head*. A clause with more than one predicate in the body is called *nonlinear*. A nonlinear system of CHCs has at least one nonlinear clause; otherwise, the system is linear.

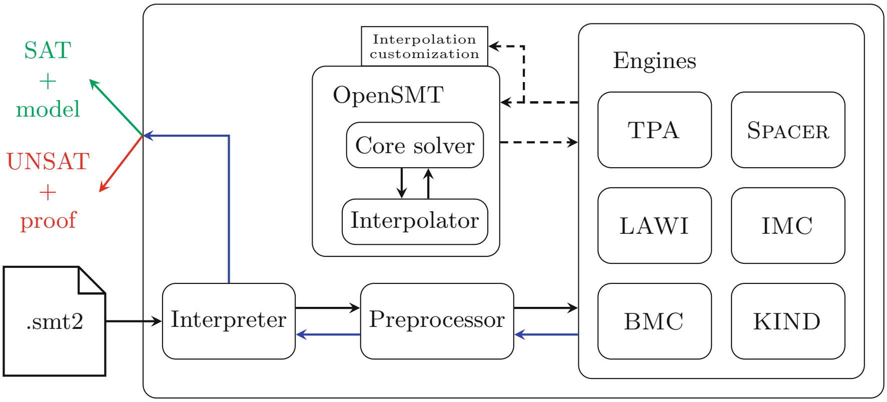


Fig. 1. High-level architecture of Golem

Architecture. The flow of data inside Golem is depicted in Fig. 1. The system of CHCs is read from `.smt2` file, a script in an extension of the language of SMT-LIB.[3] `Interpreter` interprets the SMT-LIB script and builds the internal representation of the system of CHCs. In Golem, CHCs are first *normalized*, then the system is translated into an internal graph representation. Normalization rewrites clauses to ensure that each predicate has only variables as arguments. The graph representation of the system is then passed to the `Preprocessor`, which applies various transformations to simplify the input graph. `Preprocessor` then hands the transformed graph to the chosen back-end engine. Engines in

[3] https://chc-comp.github.io/format.html.

Golem implement various SMT-based model-checking algorithms for solving the CHC satisfiability problem. There are currently six engines in Golem: TPA, BMC, KIND, IMC, LAWI, and Spacer (see details in Sect. 3). User selects the engine to run using a command-line option `--engine`. Golem relies on the interpolating SMT solver OpenSMT [42] not only for answering SMT queries but also for interpolant computation required by most of the engines. Interpolating procedures in OpenSMT can be customized on demand for the specific needs of each engine [1]. Additionally, Golem re-uses the data structures of OpenSMT for representing and manipulating terms.

Models and Proofs. Besides solving the CHC satisfiability problem, a *witness* for the answer is often required by the domain-specific application. Satisfiability witness is a *model*, an interpretation of the CHC predicates that makes all clauses valid. Unsatisfiability witness is a *proof*, a derivation of the empty clause from the input clauses. In software verification these witnesses correspond to program invariants and counterexample paths, respectively. All engines in Golem produce witnesses for their answer. Witnesses from engines are translated back through the applied preprocessing transformations. Only after this *backtranslation*, the witness matches the original input system and is reported to the user. Witnesses must be explicitly requested with the option `--print-witness`.

Models are internally stored as formulas in the background theory, using only the variables of the (normalized) uninterpreted predicates. They are presented to the user in the format defined by SMT-LIB [5]: a sequence of SMT-LIB's `define-fun` commands, one for each uninterpreted predicate.

For the proofs, Golem follows the trace format proposed by Eldarica. Internally, proofs are stored as a sequence of derivation steps. Every derivation step represents a ground instance of some clause from the system. The ground instances of predicates from the body form the *premises* of the step, and the ground instance of the head's predicate forms the *conclusion* of the step. For the derivation to be valid, the premises of each step must have been derived earlier, i.e., each premise must be a conclusion of some derivation step earlier in the sequence. To the user, the proof is presented as a sequence of derivations of ground instances of the predicates, where each step is annotated with the indices of its premises. See Example 1 below for the illustration of the proof trace.

Golem also implements an internal *validator* that checks the correctness of the witnesses. It validates a model by substituting the interpretations for the predicates and checking the validity of all the clauses with OpenSMT. Proofs are validated by checking all conditions listed above for each derivation step. Validation is enabled with an option `--validate` and serves primarily as a debugging tool for the developers of witness production.

Example 1. Consider the following CHC system and the proof of its unsatisfiability.

$$\begin{array}{rcl@{\qquad\qquad}ll}
x > 0 & \implies & L1(x) & 1.\ L_1(1) & \\
x' = x + 1 & \implies & D(x, x') & 2.\ D(1, 2) & \\
L1(x) \wedge D(x, x') & \implies & L2(x') & 3.\ L_2(2) & ;1,2 \\
L2(x) \wedge x \leq 2 & \implies & \mathit{false} & 4.\ \mathit{false} & ;3
\end{array}$$

The derivation of *false* consists of four derivation steps. Step 1 instantiates the first clause for $x := 1$. Step 2 instantiates the second clause for $x := 1$ and $x' := 2$. Step 3 applies resolution to the instance of the third clause for $x := 1$ and $x' := 2$ and facts derived in steps 1 and 2. Finally, step 4 applies resolution to the instance of the fourth clause for $x := 2$ and the fact derived in step 3.

Preprocessing Transformations. Preprocessing can significantly improve performance by transforming the input CHC system into one more suitable for the back-end engine. The most important transformation in Golem is *predicate elimination*. Given a predicate not present in both the body and the head of the same clause, the predicate can be eliminated by exhaustive application of the resolution rule. This transformation is most beneficial when it also decreases the number of clauses. *Clause merging* is a transformation that merges all clauses with the same uninterpreted predicates in the body and the head to a single clause by disjoining their constraints. This effectively pushes work from the level of the model-checking algorithm to the level of the SMT solver. Additionally, Golem detects and deletes *redundant clauses*, i.e., clauses that cannot participate in the proof of unsatisfiability.

An important feature of Golem is that all applied transformations are *reversible* in the sense that any model or proof for the transformed system can be translated back to a model or proof of the original system.

3 Back-end Engines of Golem

The core components of Golem that solve the problem of satisfiability of a CHC system are referred to as *back-end engines*, or just engines. Golem implements several popular state-of-the-art algorithms from model checking and software verification: BMC, k-induction, IMC, LAWI and Spacer. These algorithms treat the problem of solving a CHC system as a *reachability* problem in the graph representation.

The unique feature of Golem is the implementation of the new model-checking algorithm based on the concept of *Transition Power Abstraction* (TPA). It is capable of much deeper analysis than other algorithms when searching for counterexamples [14], and it discovers *transition* invariants [13], as opposed to the usual (state) invariants.

3.1 Transition Power Abstraction

The TPA engine in Golem implements the model-checking algorithm based on the concept of Transition Power Abstraction. It can work in two modes: The first mode implements the basic TPA algorithm, which uses a single TPA sequence [14]. The second mode implements the more advanced version, split-TPA, which relies on two TPA sequences obtained by splitting the single TPA sequence of the basic version [13]. In Golem, both variants use the under-approximating *model-based projection* for propagating truly reachable states, avoiding full quantifier elimination. Moreover, they benefit from incremental solving available in OpenSMT, which speeds up the satisfiability queries.

The TPA algorithms, as described in the publications, operate on transition systems [13,14]. However, the engine in Golem is not limited to a single transition system. It can analyze a connected *chain of transition systems*. In the software domain, this model represents programs with a sequence of consecutive loops. The extension to the chain of transition systems works by maintaining a separate TPA sequence for each node on the chain, where each node has its own transition relation. The reachable states are propagated forwards on the chain, while safe states—from which final error states are unreachable—are propagated backwards. In this scenario, transition systems on the chain are queried for reachability between various initial and error states. Since the transition relations remain the same, the summarized information stored in the TPA sequences can be re-used across multiple reachability queries. The learnt information summarizing multiple steps of the transition relation is not invalidated when the initial or error states change.

Golem's TPA engine discovers counterexample paths in unsafe transition systems, which readily translate to unsatisfiability proofs for the corresponding CHC systems. For safe transition systems, it discovers safe k-inductive transition invariants. If a model for the corresponding CHC system is required, the engine first computes a quantified inductive invariant and then applies quantifier elimination to produce a quantifier-free inductive invariant, which is output as the corresponding model.[4]

The TPA engine's ability to discover deep counterexamples and transition invariants gives Golem a unique edge for systems requiring deep exploration. We provide an example of this capability as part of the evaluation in Sect. 4.

3.2 Engines for State-of-the-Art Model-Checking Algorithms

Besides TPA, Golem implements several popular state-of-the-art model-checking algorithms. Among them are bounded model checking [12], k-induction [55] and McMillan's interpolation-based model checking [49], which operate on transition systems. Golem faithfully follows the description of the algorithms in the respective publications.

[4] The generation of unsatisfiability proofs also works for the extension to chains of transition systems, while the generation of models for this case is still under development.

Additionally, Golem implements *Lazy Abstractions with Interpolants* (LAWI), an algorithm introduced by McMillan for verification of software [50].[5] In the original description, the algorithm operates on programs represented with *abstract reachability graphs*, which map straightforwardly to *linear* CHC systems. This is the input supported by our implementation of the algorithm in Golem.

The last engine in Golem implements the IC3-based algorithm Spacer [45] for solving general, even nonlinear, CHC systems. Nonlinear CHC systems can model programs with summaries, and in this setting, Spacer computes both under-approximating and over-approximating summaries of the procedures to achieve modular analysis of programs. Spacer is currently the only engine in Golem capable of solving nonlinear CHC systems.

All engines in Golem rely on OpenSMT for answering SMT queries, often leveraging the incremental capabilities of OpenSMT to implement the corresponding model-checking algorithm efficiently. Additionally, the engines IMC, LAWI, Spacer and TPA heavily use the flexible and controllable interpolation framework in OpenSMT [1,52], especially multiple interpolation procedures for linear-arithmetic conflicts [3,15].

4 Experiments

In this section, we evaluate the performance of individual Golem's engines on the benchmarks from the latest edition of CHC-COMP. The goal of these experiments is to 1) demonstrate the usefulness of multiple back-end engines and their potential combined use for solving various problems, and 2) compare Golem against state-of-the-art Horn solvers.

The benchmark collections of CHC-COMP represent a rich source of problems from various domains.[6] Version 0.3.2 of Golem was used for these experiments. Z3-Spacer (Z3 4.11.2) and Eldarica 2.0.8 were run (with default options) for comparison as the best Horn solvers available. All experiments were conducted on a machine with an AMD EPYC 7452 32-core processor and 8×32 GiB of memory; the timeout was set to 300 s. No conflicting answers were observed in any of the experiments. The results are in line with the results of the last editions of CHC-COMP where Golem participated [24,31]. Our artifact for reproducing the experiments is available at https://doi.org/10.5281/zenodo.7973428.

4.1 Category LRA-TS

We ran all engines of Golem on all 498 benchmarks from the LRA-TS (transition systems over linear real arithmetic) category of CHC-COMP.

Table 1 shows the number of benchmarks solved per engine, together with a *virtual best* (VB) engine.[7] On unsatisfiable problems, the differences between the

[5] It is also known as Impact, which was the first tool that implemented the algorithm.

[6] https://github.com/orgs/chc-comp/repositories.

[7] Virtual best engine picks the best performance from all engines for each benchmark.

Table 1. Number of solved benchmarks from LRA-TS category.

	BMC	KIND	IMC	LAWI	SPACER	SPLIT-TPA	VB
SAT	0	260	145	279	195	128	360
UNSAT	86	84	70	76	69	72	86

engines' performance are not substantial, but the BMC engine firmly dominates the others. On satisfiable problems, we see significant differences. Figure 2 plots, for each engine, the number of solved *satisfiable* benchmarks (x-axis) within the given time limit (y-axis, log scale).

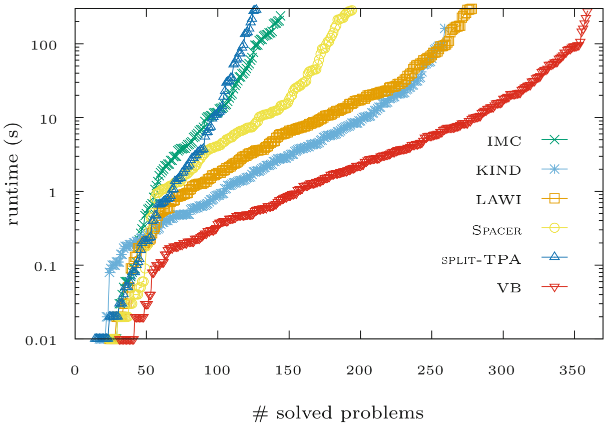


Fig. 2. Performance of GOLEM's engines on SAT problems of LRA-TS category.

The large lead of VB suggests that the solving abilities of the engines are widely complementary. No single engine dominates the others on satisfiable instances. The *portfolio* of techniques available in GOLEM is much stronger than any single one of them.

Moreover, the unified setting enables direct comparison of the algorithms. For example, we can conclude from these experiments that the extra check for k-inductive invariants on top of the BMC-style search for counterexamples, as implemented in the KIND engine, incurs only a small overhead on unsatisfiable problems, but makes the KIND engine very successful in solving satisfiable problems.

4.2 Category LIA-Lin

Next, we considered the LIA-Lin category of CHC-COMP. These are linear systems of CHCs with linear integer arithmetic as the background theory. There are many benchmarks in this category, and for the evaluation at the competition, a subset of benchmarks is selected (see [24,31]). We evaluated the LAWI and Spacer engines of Golem (the engines capable of solving general linear CHC systems) on the benchmarks selected at CHC-COMP 2022 and compared their performance to Z3-Spacer and Eldarica. Notably, we also examined a specific subcategory of LIA-lin, namely `extra-small-lia`[8] with benchmarks that fall into the fragment accepted by Golem's TPA engine.

There are 55 benchmarks in `extra-small-lia` subcategory, all satisfiable, but known to be highly challenging for all tools. The results, given in Table 2, show that split-TPA outperforms not only LAWI and Spacer engines in Golem, but also Z3-Spacer. Only Eldarica solves more benchmars. We ascribe this to split-TPA's capability to perform deep analysis and discover transition invariants.

Table 2. Number of solved benchmarks from `extra-small-lia` subcategory.

Golem				
split-TPA	LAWI	Spacer	Z3-Spacer	Eldarica
22	12	18	18	36

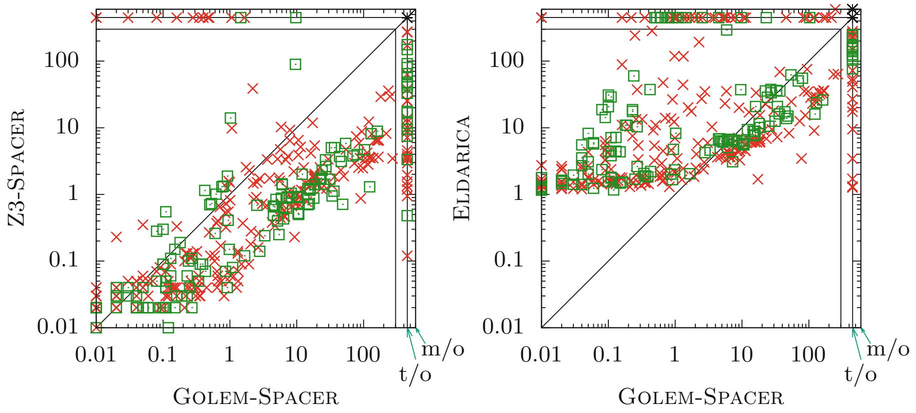
For the whole LIA-Lin category, 499 benchmarks were selected in the 2022 edition of CHC-COMP [24]. The performance of the LAWI and Spacer engines of Golem, Z3-Spacer and Eldarica on this selection is summarized in Table 3. Here, the Spacer engine of Golem significantly outperforms the LAWI engine. Moreover, even though Golem loses to Z3-Spacer, it beats Eldarica. Given that Golem is a prototype, and Z3-Spacer and Eldarica have been developed and optimized for several years, this demonstrates the great potential of Golem.

Table 3. Number of solved benchmarks from LIA-Lin category.

	Golem			
	LAWI	Spacer	Z3-Spacer	Eldarica
SAT	131	184	211	183
UNSAT	77	82	96	60

4.3 Category LIA-Nonlin

Finally, we considered the LIA-Nonlin category of benchmarks of CHC-COMP, which consists of *nonlinear* systems of CHCs with linear integer arithmetic as the background theory. For the experiments, we used the 456 benchmarks selected for the 2022 edition of CHC-COMP. Spacer is the only engine in Golem capable of solving nonlinear CHC systems; thus, we focused on a more detailed comparison of its performance against Z3-Spacer and Eldarica. The results of the experiments are summarized in Fig. 3 and Table 4.



(a) Golem vs Z3-Spacer

(b) Golem vs Eldarica

Fig. 3. Comparison on LIA-Nonlin category (× - SAT, ⊡ - UNSAT). (Color figure online)

Table 4. Number of solved benchmarks from LIA-Nonlin category. The number of *uniquely* solved benchmarks is in parentheses.

	Golem-Spacer	Z3-Spacer	Eldarica
SAT	239 (4)	248 (13)	221 (6)
UNSAT	124 (2)	139 (5)	122 (0)

Overall, Golem solved fewer problems than Z3-Spacer but more than Eldarica; however, *all* tools solved some instances *uniquely*. A detailed comparison is depicted in Fig. 3. For each benchmark, its data point in the plot reflects the runtime of Golem (x-axis) and the runtime of the competitor (y-axis). The plots suggest that the performance of Golem is often orthogonal to Eldarica, but highly correlated with the performance of Z3-Spacer. This is not surprising as the Spacer engine in Golem is built on the same core algorithm. Even though Golem is often slower than Z3-Spacer, there is a non-trivial amount of benchmarks on which Z3-Spacer times out, but which Golem solves fairly quickly. Thus, Golem, while being a newcomer, already complements existing state-of-the-art tools, and more improvements are expected in the near future.

To summarise, the overall experimentation with different engines of Golem demonstrates the advantages of the multi-engine general framework and illustrates the competitiveness of its analysis. It provides a lot of flexibility in addressing various verification problems while being easily customizable with respect to the analysis demands.

[8] https://github.com/chc-comp/extra-small-lia.

5 Conclusion

In this work, we presented GOLEM, a flexible and effective Horn solver with multiple back-end engines, including recently-introduced TPA-based model-checking algorithms. GOLEM is a suitable research tool for prototyping new SMT-based model-checking algorithms and comparing algorithms in a unified framework. Additionally, the effective implementation of the algorithm achieved with tight coupling with the underlying SMT solver makes it an efficient back end for domain-specific verification tools. Future directions for GOLEM include support for VMT input format [21] and analysis of liveness properties, extension of TPA to nonlinear CHC systems, and support for SMT theories of arrays, bit-vectors and algebraic datatypes.

Acknowledgement. This work was partially supported by Swiss National Science Foundation grant 200021_185031 and by Czech Science Foundation Grant 23-06506 S.

References

1. Alt, L.: Controlled and Effective Interpolation. Ph.D. thesis, Università della Svizzera italiana (2016). https://susi.usi.ch/usi/documents/318933
2. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity compiler's model checker. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification, pp. 325–338. Springer International Publishing, Cham (2022)
3. Alt, L., Hyvärinen, A.E.J., Sharygina, N.: LRA interpolants from no man's land. In: Strichman, O., Tzoref-Brill, R. (eds.) Hardware and Software: Verification and Testing, pp. 195–210. Springer International Publishing, Cham (2017)
4. Alur, R., et al.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design, pp. 1–8 (2013)
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017). https://www.SMT-LIB.org
6. Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. Impact. In: 2012 Formal Methods in Computer-Aided Design (FMCAD), pp. 106–113 (Oct 2012)
7. Beyer, D., Dangl, M.: Software verification with PDR: an implementation of the state of the art. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 3–21. Springer International Publishing, Cham (2020)
8. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification, pp. 622–640. Springer International Publishing, Cham (2015)
9. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reason. **60**(3), 299–335 (2018)
10. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, pp. 184–190. Springer, Berlin Heidelberg, Berlin, Heidelberg (2011)

11. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. Tech. Rep. arXiv/CoRR arXiv:2208.05046 (August 2022)
12. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 193–207. Springer, Berlin Heidelberg, Berlin, Heidelberg (1999)
13. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Split transition power abstractions for unbounded safety. In: Griggio, A., Rungta, N. (eds.) Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design - FMCAD 2022. pp. 349–358. TU Wien Academic Press (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_42
14. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Transition power abstractions for deep counterexample detection. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 524–542. Springer International Publishing, Cham (2022)
15. Blicha, M., Hyvärinen, A.E.J., Kofroň, J., Sharygina, N.: Using linear algebra in decomposition of Farkas interpolants. Int. J. Softw. Tools Technol. Transfer **24**(1), 111–125 (2022)
16. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 70–87. Springer, Berlin Heidelberg, Berlin, Heidelberg (2011)
17. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 150–153. Springer, Berlin Heidelberg, Berlin, Heidelberg (2010)
18. Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: Practical and sound static analysis of android applications by SMT solving. In: 2016 IEEE European Symposium on Security and Privacy, pp. 47–62 (2016)
19. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 365–384. Springer International Publishing, Cham (2018)
20. Champion, A., Kobayashi, N., Sato, R.: HoIce: an ICE-based non-linear Horn clause solver. In: Ryu, S. (ed.) Programming Languages and Systems, pp. 146–156. Springer International Publishing, Cham (2018)
21. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools (2021)
22. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification, pp. 154–169. Springer, Berlin Heidelberg, Berlin, Heidelberg (2000)
23. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symbolic Logic **22**(3), 269–285 (1957)
24. De Angelis, E., Vediramana Krishnan, H.G.: CHC-COMP 2022: Competition report. Electron. Proc. Theor. Comput. Sci. **373**, 44–62 (nov 2022)
25. Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A., Podelski, A.: Ultimate TreeAutomizer (CHC-COMP tool description). In: Angelis, E.D., Fedyukovich, G., Tzevelekos, N., Ulbrich, M. (eds.) Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6–7th April 2019. EPTCS, vol. 296, pp. 42–47 (2019)

26. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, pp. 125–134. FMCAD '11, FMCAD Inc, Austin, TX (2011)
27. Ernst, G.: Korn–software verification with Horn clauses (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 559–564. Springer Nature Switzerland, Cham (2023)
28. Esen, Z., Rümmer, P.: TriCera: Verifying C programs using the theory of heaps. In: Griggio, A., Rungta, N. (eds.) Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design - FMCAD 2022, pp. 360–391. TU Wien Academic Press (2022)
29. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Sampling invariants from frequency distributions. In: 2017 Formal Methods in Computer Aided Design (FMCAD), pp. 100–107 (2017)
30. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained Horn clauses using syntax and data. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–9 (2018)
31. Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021. EPTCS, vol. 344, pp. 91–108 (2021)
32. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
33. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 405–416. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012)
34. Gurfinkel, A., Bjørner, N.: The science, art, and magic of constrained Horn clauses. In: 2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 6–10 (2019)
35. Gurfinkel, A.: Program verification with constrained Horn clauses (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification, pp. 19–29. Springer International Publishing, Cham (2022)
36. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification, pp. 343–361. Springer International Publishing, Cham (2015)
37. Hari Govind, V.K., Chen, Y., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification, pp. 101–125. Springer International Publishing, Cham (2020)
38. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
39. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis, pp. 187–202. Springer, Berlin Heidelberg, Berlin, Heidelberg (2012)
40. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: FMCAD, pp. 158–164. IEEE (10 2018)

41. Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. Electronic Proceedings in Theoretical Computer Science 169, 39–52 (dec 2014)
42. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT Solver for Multi-core and Cloud Computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_35
43. Kafle, B., Gallagher, J.P.: Tree automata-based refinement with application to Horn clause verification. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 209–226. Springer, Berlin Heidelberg, Berlin, Heidelberg (2015)
44. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: a framework for verifying Java programs. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification, pp. 352–358. Springer International Publishing, Cham (2016)
45. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (2016)
46. Leroux, J., Rümmer, P., Subotić, P.: Guiding Craig interpolation with domain-specific abstractions. Acta Informatica **53**(4), 387–424 (2016)
47. Mann, M., et al.: Pono: a flexible and extensible SMT-based model checker. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification, pp. 461–474. Springer International Publishing, Cham (2021)
48. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. ACM Trans. Program. Lang. Syst. **43**(4) (oct 2021)
49. McMillan, K.L.: Interpolation and SAT-Based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
50. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, pp. 123–136. Springer, Berlin Heidelberg, Berlin, Heidelberg (2006)
51. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Berlin Heidelberg, Berlin, Heidelberg (2008)
52. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: a framework for producing effective interpolants in SAT-based software verification. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, pp. 683–693. Springer, Berlin Heidelberg, Berlin, Heidelberg (2013)
53. Rümmer, P., Subotić, P.: Exploring interpolants. In: 2013 Formal Methods in Computer-Aided Design, pp. 69–76 (Oct 2013)
54. Rümmer, P.: Competition report: CHC-COMP-20. Electron. Proc. Theor. Comput. Sci. **320**, 197–219 (2020)
55. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) Formal Methods in Computer-Aided Design, pp. 127–144. Springer, Berlin Heidelberg, Berlin, Heidelberg (2000)
56. Wang, W., Jiao, L.: Trace Abstraction Refinement for Solving Horn Clauses. Comput. J.**59**(8), 1236–1251 (08 2016)
57. Wesley, S., Christakis, M., Navas, J.A., Trefler, R., Wüstholz, V., Gurfinkel, A.: Verifying solidity smart contracts via communication abstraction in smartace. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 425–449. Springer International Publishing, Cham (2022)

58. Zlatkin, I., Fedyukovich, G.: Maximizing branch coverage with constrained Horn clauses. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 254–272. Springer International Publishing, Cham (2022)

Model Checking

CoqCryptoLine: A Verified Model Checker with Certified Results

Ming-Hsien Tsai[4](✉), Yu-Fu Fu[2], Jiaxiang Liu[5], Xiaomu Shi[3],
Bow-Yaw Wang[1], and Bo-Yin Yang[1]

[1] Academia Sinica, Taipei, Taiwan
{bywang,byyang}@iis.sinica.edu.tw
[2] Georgia Institute of Technology, Atlanta, USA
yufu@gatech.edu
[3] Institute of Software, Chinese Academy of Sciences, Beijing, China
xshi0811@gmail.com
[4] National Institute of Cyber Security, Taipei, Taiwan
mhtsai208@gmail.com
[5] Shenzhen University, Shenzhen, China
jiaxiang0924@gmail.com

Abstract. We present the verified model checker CoqCryptoLine for cryptographic programs with certified verification results. The CoqCryptoLine verification algorithm consists of two reductions. The algebraic reduction transforms into a root entailment problem; and the bit-vector reduction transforms into an SMT QF_BV problem. We specify and verify both reductions formally using Coq with MathComp. The CoqCryptoLine tool is built on the OCaml programs extracted from verified reductions. CoqCryptoLine moreover employs certified techniques for solving the algebraic and logic problems. We evaluate CoqCryptoLine on cryptographic programs from industrial security libraries.

1 Introduction

CoqCryptoLine [1] is a verified model checker with certified verification results. It is designed for verifying complex non-linear integer computations commonly found in cryptographic programs. The verification algorithms of CoqCryptoLine consist of two reductions. The algebraic reduction transforms polynomial equality checking into a root entailment problem in commutative algebra; the bit-vector reduction reduces range properties to satisfiability of queries in the Quantifier-Free Bit-Vector (QF_BV) logic from Satisfiability Modulo Theories (SMT) [6]. Both verification algorithms are formally specified and verified by the proof assistant Coq with MathComp [7,17]. CoqCryptoLine verification programs are extracted from the formal specification and therefore verified by the proof assistant automatically.

The original version of this chapter was revised: The mistakes in authors affiliation information and typographical errors have been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-031-37703-7_22

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 227–240, 2023.
https://doi.org/10.1007/978-3-031-37703-7_11

To minimize errors from external tools, recent developments in certified verification are employed by CoqCryptoLine. The root entailment problem is solved by the computer algebra system (CAS) Singular [19]. CoqCryptoLine asks the external algebraic tool to provide certificates and validates certificates with the formal polynomial theory in Coq. SMT QF_BV queries on the other hand are answered by the verified SMT QF_BV solver CoqQFBV [33]. Answers to SMT QF_BV queries are therefore all certified as well. With formally verified algorithms and certified answers from external tools, CoqCryptoLine gives verification results with much better guarantees than average automatic verification tools.

Reliable verification tools would not be very useful if they could not check real-world programs effectively. In our experiments, CoqCryptoLine verifies 54 real-world cryptographic programs. 52 of them are from well-known security libraries such as Bitcoin [35] and OpenSSL [30]. They are implementations of field and group operations in elliptic curve cryptography. The remaining two are the Number-Theoretic Transform (NTT) programs from the post-quantum cryptosystem Kyber [10]. All field operations are implemented in a few hundred lines and verified in 6 minutes. The most complicated generic group operation in the elliptic curve Curve25519 consists of about 4000 lines and is verified by CoqCryptoLine in 1.5 h.

Related Work. There are numerous model checkers in the community, e.g. [8, 13, 21–23]. Nevertheless, few of them are formally verified. To our knowledge, the first verification of a model checker was performed in Coq for the modal μ-calculus [34]. The LTL model checker CAVA [15, 27] and the model checker Munta [38, 39] for timed automata were developed and verified using Isabelle/HOL [29], which can be considered as verified counterparts of SPIN [21] and Uppaal [23], respectively. CoqCryptoLine instead checks CryptoLine models [16, 31] that are for the correctness of cryptographic programs. It can be seen as a verified version of CryptoLine. A large body of work studies the correctness of cryptographic programs, e.g. [2–4, 9, 12, 14, 24, 26, 40], cf. [5] for a survey. They either require human intervention or are unverified, while our work is fully automatic and verified. The most relevant work is bvCryptoLine [37], which is the first automated and partly verified model checker for a very limited subset of CryptoLine. We will compare our work with it comprehensively in Sect. 2.3.

2 CoqCryptoLine

CoqCryptoLine is an automatic verification tool that takes a CryptoLine specification as input and returns certified results indicating the validity of the specification. We briefly describe the CryptoLine language [16] followed by the modules, features, and optimizations of CoqCryptoLine in this section.

2.1 CryptoLine Language

A CryptoLine specification contains a CryptoLine program with pre- and post-conditions, where the CryptoLine program usually models some cryptographic program [16,31]. Both the pre- and post-conditions consist of an algebraic part, which is formulated as a conjunction of (modular) equations, and a range part as an SMT QF_BV predicate. A CryptoLine specification is valid if every program execution starting from a program state satisfying the pre-condition ends in a state satisfying the post-condition.

CryptoLine is designed for modeling cryptographic assembly programs. Besides the assignment (mov) and conditional assignment (cmov) statements, CryptoLine provides arithmetic statements such as addition (add), addition with carry (adc), subtraction (sub), subtraction with borrow (sbb), half multiplication (mul) and full multiplication (mull). Most of them have versions that model the carry/borrow flags explicitly (like adds, adcs, subs, sbbs). It also allows bitwise statements, for instance, bitwise AND (and), OR (or) and left-shift (shl). To deal with multi-word arithmetic, CryptoLine further includes multi-word constructs, for example, those that split (split) or join (join) words, as well as multi-word shifts (cshl). CryptoLine is strongly typed, admitting both signed and unsigned interpretations for bit-vector variables and constants. The cast statement converts types explicitly. Finally, CryptoLine also supports special statements (assert and assume) for verification purposes.

2.2 The Architecture of CoqCryptoLine

CoqCryptoLine reduces the verification problem of a CryptoLine specification to instances of root entailment problems and SMT problems over the QF_BV logic. These instances are then solved by respective certified techniques. Moreover, the components in CoqCryptoLine are also specified and verified by the proof assistant Coq with MathComp [7,17]. Figure 1 gives an overview of CoqCryptoLine. In the figure, dashed components represent external tools. Rectangular boxes are verified components and rounded boxes are unverified. Note that all our proof efforts using Coq are transparent to users. No Coq proof is required from users during verification of cryptographic programs with CoqCryptoLine. Details can be found in [36].

Starting from a CryptoLine specification text, the CoqCryptoLine parser translates the text into an abstract syntax tree defined in the Coq module `DSL`. The module gives formal semantics for the typed CryptoLine language [16]. The validity of CryptoLine specifications is also formalized. Similar to most program verification tools, CoqCryptoLine transforms CryptoLine specifications to the static single assignment (SSA) form. The `SSA` module gives our transformation algorithm. It moreover shows that validity of CryptoLine specifications is preserved by the SSA transformation. CoqCryptoLine then reduces the verification problem via two Coq modules.

The `SSA2ZSSA` module contains our algebraic reduction to the root entailment problem. Concretely, a system of (modular) equations is constructed from the given program so that program executions correspond to the roots of the system of (modular) equations. To verify algebraic post-conditions, it suffices to check if the roots for executions are also roots of (modular) equations in the post-condition. However, program executions can deviate from roots of (modular) equations when over- or under-flow occurs. CoqCryptoLine will generate

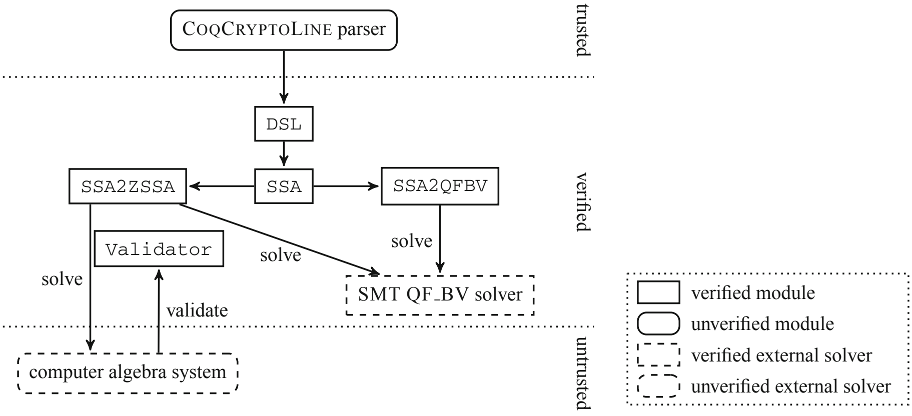


Fig. 1. Overview of CoqCryptoLine

soundness conditions to ensure the executions conform to our (modular) equations. The algebraic verification problem is thus reduced to the root entailment problem provided that soundness conditions hold.

The SSA2QFBV module gives our bit-vector reduction to the SMT QF_BV problem. It constructs an SMT query to check the validity of the given CryptoLine range specification. Concretely, an SMT QF_BV query is built such that all program executions correspond to satisfying assignments to the query and vice versa. To verify the range post-conditions, it suffices to check if satisfying assignments for the query also satisfy the post-conditions. The range verification problem is thus reduced to the SMT QF_BV problem. On the other hand, additional SMT queries are constructed to check soundness conditions for the algebraic reduction. We formally prove the equivalence between soundness conditions and corresponding queries.

With the two formally verified reduction algorithms, it remains to solve the root entailment problems and the SMT QF_BV problems with external solvers. CoqCryptoLine invokes an external computer algebra system (CAS) to solve the root entailment problems, and improves the techniques in [20,37] to validate the (untrusted) returned answers. Currently, the CAS Singular [19] is supported. To solve the SMT QF_BV problems, CoqCryptoLine employs the certified SMT QF_BV solver CoqQFBV [33]. In all cases, instances of the two kinds of problems are solved with certificates. And CoqCryptoLine employs verified certificate checkers to validate the answers to further improve assurance.

Note that the algebraic reduction in SSA2ZSSA is sound but not complete due to the abstraction of bit-accurate semantics into (modular) polynomial equations over integers. Thus a failure in solving the root entailment problem by CAS does not mean that the algebraic post-conditions are violated. On the other hand, the bit-vector reduction in SSA2QFBV is both sound and complete.

The CoqCryptoLine tool is built on OCaml programs extracted from verified algorithms in Coq with MathComp. We moreover integrate the OCaml programs from the certified SMT QF_BV solver CoqQFBV. Our trusted computing base consists of (1) CoqCryptoLine parser, (2) text interface with external SAT solvers (from CoqQFBV), (3) the proof assistant Isabelle [29] (from the SAT solver certificate validator Grat used by CoqQFBV) and (4) the Coq proof assistant. Particularly, sophisticated decision procedures in external CASs and SAT solvers used in CoqQFBV need not be trusted.

2.3 Features and Optimizations

CoqCryptoLine comes with the following features and optimizations implemented in its modules.

Type System. CoqCryptoLine fully supports the type system of the CryptoLine language. The type system is used to model bit-vectors of arbitrary bit-widths with unsigned or signed interpretation. Such a type system allows CoqCryptoLine to model more industrial examples translated from C programs via GCC [16] or LLVM [24] compared to bvCryptoLine [37], which only allows unsigned bit-vectors, all of the same bit-width.

Mixed Theories. With the assert and assume statements supported by CoqCryptoLine, it is possible to make an assertion on the range side (or on the algebraic side) and then make an equivalent assumption on the algebraic side (or resp. on the range side). With this feature, a predicate can be asserted on one side where the predicate is easier to prove, and then assumed on the other side to ease the verification of other predicates. The equivalence between the asserted predicate and the assumed predicate is currently not verified by CoqCryptoLine, though it is achievable. Both assert and assume statements are not available in bvCryptoLine.

Multi-threading. All extracted OCaml code from the verified algorithms in Coq runs sequentially. To speed up, SMT QF_BV problems, as well as root entailment problems, are solved parallelly.

Efficient Root Entailment Problem Solving. CoqCryptoLine can be used as a solver for root entailment problems with certificates validated by a verified validator. A root entailment problem is reduced to an ideal membership problem, which is then solved by computing Gröbner basis [20]. To solve a root entailment problem with a certificate, we need to find a witness of polynomials $c_0, \ldots, c_n$ such that

$$q = \Sigma_{i=0}^{n} c_i p_i \tag{1}$$

where q and p_i's are given polynomials. To compute the witness, bvCryptoLine relies on `gbarith` [32], where new variables are introduced. CoqCryptoLine utilizes the `lift` command in Singular instead without adding fresh variables. We show in the evaluation section that using `lift` is more efficient than using `gbarith`. The witness found is further validated by CoqCryptoLine, which relies on the polynomial normalization procedure `norm_subst` in Coq to check if Eq. 1 holds. bvCryptoLine on the other hand uses the `ring` tactic in Coq, where extra type checking is performed. Elimination of ideal generators through variable substitution is an efficient approach to simplify an ideal membership problem [37]. The elimination procedure implemented in CoqCryptoLine can identify much more variable substitution patterns than those found by bvCryptoLine.

Multi-moduli. Modular equations with multi-moduli are common in post-quantum cryptography. For example, the post-quantum cryptosystem Kyber uses the polynomial ring $\mathbb{Z}_{3329}[X]/\langle X^{256}+1\rangle$ containing two moduli 3329 and $X^{256}+1$. To support multi-moduli in CoqCryptoLine, in the proof of our algebraic reduction, we have to find integers $c_0, \ldots, c_n$ such that $e_1 - e_2 = \Sigma_{i=0}^{n} c_i m_i$ given the proof of $e_1 = e_2 \pmod{m_0, \ldots, m_n}$ where e_1, e_2, and m_i's are integers. Instead of implementing a complicated procedure to find the exact c_i's, we simply invoke the `xchoose` function provided by MathComp to find c_i's based on the proof of $e_1 = e_2 \pmod{m_0, \ldots, m_n}$. Multi-moduli is not supported by bvCryptoLine.

Tight Integration with CoqQFBV*.* CoqCryptoLine verifies every atomic range predicate separately using the certified SMT QF_BV solver CoqQFBV. Constructing a text file as the input to CoqQFBV for every atomic range predicate is not a good idea because the bit-blasting procedure in CoqQFBV is performed several times for the identical program. CoqCryptoLine thus is tightly integrated with CoqQFBV to speed up bit-blasting of the same program using the cache provided by CoqQFBV. bvCryptoLine uses the SMT solver Boolector to prove range predicates without certificates.

Slicing. During the reductions from the verification problem of a CryptoLine specification to instances of root entailment problems and SMT QF_BV problems, a verified static slicing is performed in CoqCryptoLine to produce smaller problems. Unlike the work in [11], which sets all assume statements as additional slicing criteria, the slicing in CoqCryptoLine is capable of pruning unrelated predicates in assume statements. The slicing procedure implemented in CoqCryptoLine is much more complicated than the one in bvCryptoLine due to the presence of assume statements. This feature is provided as command-line option because it makes the verification incomplete. With slicing, the time in verifying industrial examples is reduced dramatically.

3 Walkthrough

We illustrate how CoqCryptoLine is used in this section. The x86_64 assembly subroutine `ecp_nistz256_mul_montx` from OpenSSL [30] shown in Fig. 2 is verified as an example.

An input for CoqCryptoLine contains a CryptoLine specification for the assembly subroutine. The original subroutine is marked between the comments

PROGNAME STARTS and PROGNAME ENDS, which is obtained automatically from the Python script provided by CryptoLine [31].

Prior to the "START" comment are the parameter declaration, pre-condition, and variable initialization. After the "END" comment is the post-condition of the subroutine. After the subroutine ends, the result is moved to the output variables.

The assembly subroutine ecp_nistz256_mul_montx takes two 256-bit unsigned integers a and b and the modulus m as inputs. The 256-bit integer m is the prime $p256 = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ from the NIST curve. The 256-bit integers a and b (less than the prime) are the multiplicands. Each 256-bit input integer $d \in \{a, b, m\}$ is denoted by four 64-bit unsigned integer variables d_i (for $0 \leq i < 4$) in little-endian representation. The expression limbs n [d_0, d_1, ..., d_i] is short for d_0 + d_1*2**n + ... +d_i*2**(i*n)[1]. The inputs and constants are then put in the variables for memory cells with the mov statements. There are two parts to a pre-condition. The first part is for the algebraic reduction; the second part is for the bit-vector reduction:

```
and [ m0=0xffffffffffffffff, m1=0x00000000ffffffff,
      m2=0x0000000000000000, m3=0xffffffff00000001 ]
&&
and [ m0=0xffffffffffffffff@64, m1=0x00000000ffffffff@64,
      m2=0x0000000000000000@64, m3=0xffffffff00000001@64,
      limbs 64 [a0,a1,a2,a3] <u limbs 64 [m0,m1,m2,m3],
      limbs 64 [b0,b1,b2,b3] <u limbs 64 [m0,m1,m2,m3] ]
```

The output 256-bit integer represented by the four variables c_i (for $0 \leq i < 4$) has two requirements. Firstly, the output integer times 2^{256} equals the product of the input integers modulo $p256$. Secondly, the output integer is less than $p256$. Formally, we have this post-condition:

```
eqmod limbs 64 [0, 0, 0, 0, c0, c1, c2, c3]
      limbs 64 [a0, a1, a2, a3] * limbs 64 [b0, b1, b2, b3]
      limbs 64 [m0, m1, m2, m3]
&&
limbs 64 [c0, c1, c2, c3] <u limbs 64 [m0, m1, m2, m3]
```

Here, we employ the algebraic reduction to verify the non-linear modular equality, and the bit-vector reduction to verify the proper range of the output integer.

However, verifying ecp_nistz256_mul_montx takes extra annotations to hint CoqCryptoLine how to verify the post-condition. E.g., in adding two 256-bit integers represented by 64-bit variables, a chain of four 64-bit additions is performed and carries are propagated. The last carry as the chain ends must be zero or the 256-bit sum is incorrect. In ecp_nistz256_mul_montx two interleaved addition chains use the carry and the overflow flags for carries respectively, so we annotate as follows at the end of two interleaving addition chains to tell CoqCryptoLine about the final carries:

[1] ** is the exponentiation operator in CryptoLine.

```
proc main
(uint64 a0, uint64 a1, uint64 a2, uint64 a3,
 uint64 b0, uint64 b1, uint64 b2, uint64 b3,
 uint64 m0, uint64 m1, uint64 m2, uint64 m3) =
{ and [ m0 = 0xffffffffffffffff,
        m1 = 0x00000000ffffffff,
        m2 = 0x0000000000000000,
        m3 = 0xffffffff00000001 ]
&&
  and [ m0 = 0xffffffffffffffff@64,
        m1 = 0x00000000ffffffff@64,
        m2 = 0x0000000000000000@64,
        m3 = 0xffffffff00000001@64,
        limbs 64 [a0, a1, a2, a3] <u
            limbs 64 [m0, m1, m2, m3],
        limbs 64 [b0, b1, b2, b3] <u
            limbs 64 [m0, m1, m2, m3] ] }

mov L0x7fffffffd9b0 a0; mov L0x7fffffffd9b8 a1;
mov L0x7fffffffd9c0 a2; mov L0x7fffffffd9c8 a3;
mov L0x7fffffffd9d0 b0; mov L0x7fffffffd9d8 b1;
mov L0x7fffffffd9e0 b2; mov L0x7fffffffd9e8 b3;

mov L0x55555557c000 0xffffffffffffffff@uint64;
mov L0x55555557c008 0x00000000ffffffff@uint64;
mov L0x55555557c010 0x0000000000000000@uint64;
mov L0x55555557c018 0xffffffff00000001@uint64;

(* ecp_nistz256_mul_montx STARTS *)
mov rdx L0x7fffffffd9d0;
mov r9 L0x7fffffffd9b0;
mov r10 L0x7fffffffd9b8;
mov r11 L0x7fffffffd9c0;
mov r12 L0x7fffffffd9c8;
mull r9 r8 rdx r9;
mull r10 rcx rdx r10;
mov r14 0x20@uint64;
mov r13 0@uint64;

...

mov r8 0@uint64;
clear carry;
clear overflow;
mull rbp rcx rdx L0x7fffffffd9b0;
```

```
adcs carry r9 r9 rcx carry;
adcs overflow r10 r10 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9b8;
adcs carry r10 r10 rcx carry;
adcs overflow r11 r11 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9c0;
adcs carry r11 r11 rcx carry;
adcs overflow r12 r12 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9c8;
mov rdx r9;

adcs carry r12 r12 rcx carry;
split ddc rcx r9 32;
shl rcx rcx 32;
adcs overflow r13 r13 rbp overflow;
split rbp dc r9 32;

assert true && rbp=ddc;
assume rbp=ddc && true;

adcs carry r13 r13 r8 carry;
adcs overflow r8 r8 r8 overflow;

assert true && and [carry=0@1,overflow=0@1];
assume and [carry=0,overflow=0] && true;

...

mov L0x7fffffffda00 r8;
mov L0x7fffffffda08 r9;
(* ecp_nistz256_mul_montx ENDS *)

mov c0 L0x7fffffffd9f0;
mov c1 L0x7fffffffd9f8;
mov c2 L0x7fffffffda00;
mov c3 L0x7fffffffda08;

{ eqmod limbs 64 [0, 0, 0, 0, c0, c1, c2, c3]
        limbs 64 [a0, a1, a2, a3] *
        limbs 64 [b0, b1, b2, b3]
        limbs 64 [m0, m1, m2, m3]
&&
  limbs 64 [c0, c1, c2, c3] <u
      limbs 64 [m0, m1, m2, m3] }
```

Fig. 2. CryptoLine Model for `ecp_nistz256_mul_montx`

```
assert true && and [ carry=0@1, overflow=0@1 ];
assume and [ carry=0, overflow=0 ] && true;
```

The ASSERT statement verifies that both the carry and overflow flags are zeroes through the bit-vector reduction. The ASSUME statement then passes this information to the algebraic reduction. Effectively, CoqCryptoLine checks that both flags are zero for all inputs satisfying the pre-condition, then uses those facts as lemmas to verify the post-condition with the algebraic reduction.

The full specification for `ecp_nistz256_mul_montx` has 230 lines, including 50 lines of manual annotations. 20 are straightforward annotations for variable declaration and initialization. The remaining 30 lines of annotations are hints to CoqCryptoLine, which then verifies the post-condition in 30 s with 24 threads.

The illustration of the typical verification flow shows how a user constructs a CryptoLine specification. The pre-condition for program inputs, the post-condition for outputs, and variable initialization must be specified manually. Additional annotations may be added as hints. Notice that hints only tell CoqCryptoLine *what*, not *why* properties should hold. Proofs of annotated hints and the post-condition are found by CoqCryptoLine automatically. Consequently, manual annotations are minimized and verification efforts are reduced significantly.

4 Evaluation

We evaluate CoqCryptoLine on 52 benchmarks from four industrial security libraries Bitcoin [35], boringSSL [14,18], nss [25], and OpenSSL [30]. The C reference and optimized avx2 implementations of the Number-Theoretic Transform (NTT) from the post-quantum key encapsulation mechanism Kyber [10] are also evaluated. Among the total 54 benchmarks, 43 benchmarks contain features not supported by bvCryptoLine such as signed variables. All experiments are performed on an Ubuntu 22.04.1 machine with a 3.20GHz Intel Xeon Gold 6134M CPU and 1TB RAM.

Benchmarks from security libraries are various field and group operations from elliptic curve cryptography (ECC). In ECC, rational points on curves are represented by elements in large finite fields. In Bitcoin, the finite field is the residue system modulo the prime $p256k1 = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. For other security libraries (boringSSL, nss, and OpenSSL), we verify the operations in Curve25519 using the residue system modulo the prime $p25519 = 2^{255} - 19$ as the underlying field. Rational points on elliptic curves form a group. The group operation in turn is implemented by a number of field operations.

In lattice-based post-quantum cryptosystems, polynomial rings are used. Specifically, the polynomial ring $\mathbb{Z}_{3329}[X]/\langle X^{256} + 1\rangle$ is used in Kyber. To speed up multiplication in the polynomial ring, Kyber requires the multiplication to be implemented by NTT. NTT is a discrete Fast Fourier Transform over finite fields. Instead of complex roots of unity, NTT uses the principal roots of unity in fields. Mathematically, the Kyber NTT computes the following ring isomorphism

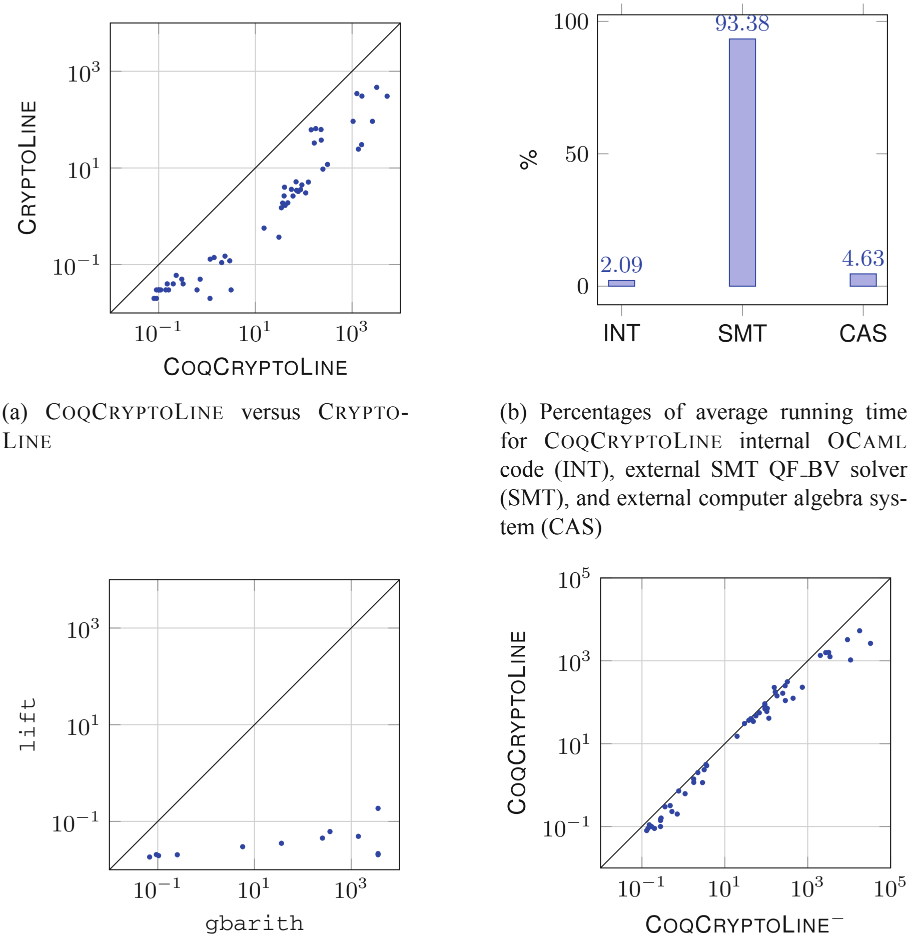
$$\mathbb{Z}_{3329}[X]/\langle X^{256} + 1\rangle \cong \mathbb{Z}_{3329}[X]/\langle X^2 - \zeta_0\rangle \times \cdots \times \mathbb{Z}_{3329}[X]/\langle X^2 - \zeta_{127}\rangle$$

where ζ_i's are the principal roots of unity.

We first compare CoqCryptoLine with all optimizations described in this paper against the unverified model checker CryptoLine [16]. Both tools invoke the computer algebra system Singular [19], but CryptoLine neither lets Singular produce certificates nor certifies answers from Singular. CoqCryptoLine moreover uses the certified SMT QF_BV solver CoqQFBV [33]; CryptoLine uses the uncertified but very efficient Boolector [28].

For the ECC experiments, CoqCryptoLine verifies all field operations in 6 minutes. It takes a few thousand seconds to verify group operations. The most complex implementation (x25519_scalar_mult_generic) from boringSSL (4274 statements) takes about 1.5 hours.[2] For Kyber, CoqCryptoLine verifies in 2642 and 1048 seconds, respectively, that the reference and avx2 NTT implementations indeed compute the isomorphism. The unverified CryptoLine in comparison finishes verification in about 95 seconds. A summary of the comparison between CoqCryptoLine and CryptoLine is shown in Fig. 3a. Though CoqCryptoLine is much slower than CryptoLine, the running time (1.5 hours) for the most complex implementation is still acceptable.

[2] Two (out of three) modular polynomial equations in the post-condition are certified.



(a) CoqCryptoLine versus CryptoLine

(b) Percentages of average running time for CoqCryptoLine internal OCaml code (INT), external SMT QF_BV solver (SMT), and external computer algebra system (CAS)

(c) `gbarith` versus `lift`

(d) CoqCryptoLine$^-$ versus CoqCryptoLine

Fig. 3. Running time (in seconds) comparisons

Figure 3b shows the percentages of average running time for CoqCryptoLine internal OCaml code (INT), external SMT QF_BV solver (SMT), and external computer algebra system (CAS). External solvers take much more time than the internal OCaml program does. Between external solvers, the external computer algebra system takes 4.63% of the time and the external SMT QF_BV solver spends 93.28% of the time.

To show the performance of the `lift` optimization, we run CoqCryptoLine and bvCryptoLine on root entailment problems generated from the benchmarks. Here we only consider 12 root entailment problems that trigger `gbarith` in bvCryptoLine. Figure 3c shows the running time of Singular in solving root entailment problems based on `gbarith` in bvCryptoLine and `lift` in CoqCryptoLine. bvCryptoLine fails to solve 3 root entailment problems in one hour. For the other 9 root entailment problems, `lift` outperforms `gbarith`.

We also compare CoqCryptoLine with and without slicing. The version of CoqCryptoLine without slicing is denoted by CoqCryptoLine$^-$. The running time comparison between CoqCryptoLine and CoqCryptoLine$^-$ in Fig. 3d shows that slicing reduces the running time obviously.

5 Conclusion

CoqCryptoLine is a verified model checker for cryptographic programs with certified results. Its modules are formally verified in Coq with MathComp. CoqCryptoLine moreover employs external tools and validates their answers with certificates. We evaluate CoqCryptoLine on benchmarks from industrial security libraries (Bitcoin, boringSSL, nss and OpenSSL) and a post-quantum cryptography standard candidate (Kyber). In our experiments, CoqCryptoLine verifies most cryptographic programs with certificates in a reasonable time (6 min). Benchmarks with thousands of lines are verified in 1.5 h. To our knowledge, this is the first certified verification on operations of the elliptic curve secp256k1 used in Bitcoin, and the avx2 and reference implementations of Kyber number-theoretic transform.

Acknowledgments. The authors in Academia Sinica are partially funded by National Science and Technology Council grants NSTC110-2221-E-001-008-MY3, NSTC111-2221-E-001-014-MY3, NSTC111-2634-F-002-019, the Sinica Investigator Award AS-IA-109-M01, the Data Safety and Talent Cultivation Project AS-KPQ-109-DSTCP, and the Intel Fast Verified Postquantum Software Project. The authors in Shenzhen University and ISCAS are partially funded by Shenzhen Science and Technology Innovation Commission (JCYJ20210324094202008), the National Natural Science Foundation of China (62002228, 61836005), and the Natural Science Foundation of Guangdong Province (2022A1515011458, 2022A1515010880).

References

1. CoqCryptoLine GitHub repository (2023). https://github.com/fmlab-iis/coq-cryptoline
2. Affeldt, R.: On construction of a library of formally verified low-level arithmetic functions. Innov. Syst. Softw. Eng. **9**(2), 59–77 (2013)
3. Almeida, J.B., et al.: Jasmin: High-assurance and high-speed cryptography. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 1807–1823. ACM (2017)
4. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Programm. Lang. Syst. **37**(2), 7:1–7:31 (2015)
5. Barbosa, M., et al.: Sok: Computer-aided cryptography. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021, pp. 777–795. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00008

6. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
8. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
9. Bond, B., et al.: Vale: Verifying high-performance cryptographic assembly code. In: USENIX Security Symposium, pp. 917–934. USENIX Association (2017)
10. Bos, J., et al.: CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM. In: Smith, M., Piessens, F. (eds.) IEEE European Symposium on Security and Privacy, pp. 353–367. IEEE (2018)
11. Chalupa, M., Strejcek, J.: Evaluation of program slicing in software verification. In: Ahrendt, W., Tarifa, S.L.T. (eds.) Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11918, pp. 101–119. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_6
12. Chen, Y.F., et al.: Verifying Curve25519 software. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM SIGSAC Conference on Computer and Communications Security, pp. 299–309. ACM (2014)
13. Cimatti, A., et al.: NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
14. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: IEEE Symposium on Security and Privacy, pp. 1202–1219. IEEE (2019)
15. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.: A fully verified executable LTL model checker. 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. 463–478. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_31
16. Fu, Y.F., Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Signed cryptographic program verification with typed cryptoline. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM SIGSAC Conference on Computer and Communications Security, pp. 1591–1606. ACM (2019)
17. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. J. Formalized Reason. **3**(2), 95–152 (2010)
18. Google: Boringssl (2021). https://boringssl.googlesource.com/boringssl/
19. Greuel, G.M., Pfister, G.: A Singular Introduction to Commutative Algebra. Springer-Verlag (2002)
20. Harrison, J.: Automating elementary number-theoretic proofs using Gröbner bases. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 51–66. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_5
21. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)

22. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002). http://research.microsoft.com/users/lamport/tla/book.html
23. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1-2), 134–152 (1997). https://doi.org/10.1007/s100090050010
24. Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic in cryptographic C programs. In: Lawall, J., Marinov, D. (eds.) IEEE/ACM International Conference on Automated Software Engineering, pp. 552–564. IEEE (2019)
25. Mozilla: Network security services (2021). https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
26. Myreen, M.O., Curello, G.: Proof Pearl: a verified bignum implementation in x86-64 machine code. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 66–81. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_5
27. Neumann, R.: Using promela in a fully verified executable LTL model checker. In: Giannakopoulou, D., Kroening, D. (eds.) Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8471, pp. 105–114. Springer (2014). https://doi.org/10.1007/978-3-319-12154-3_7
28. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisfiability, Boolean Modeling Comput. **9**(1), 53–58 (2014)
29. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
30. OpenSSL: OpenSSL library. https://github.com/openssl/openssl (2021)
31. Polyakov, A., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic assembly programs in cryptographic primitives. In: Schewe, S., Zhang, L. (eds.) International Conference on Concurrency Theory, pp. 4:1–4:16. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
32. Pottier, L.: Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008). http://ceur-ws.org/Vol-418/paper5.pdf
33. Shi, X., Fu, Y.F., Liu, J., Tsai, M.H., Wang, B.Y., Yang, B.Y.: CoqQFBV: a scalable certified SMT quantifier-free bit-vector solver. In: Leino, R., Silva, A. (eds.) International Conference on Computer Aided Verification. Springer, Lecture Notes in Computer Science (2021)
34. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Steffen, B. (ed.) Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1384, pp. 167–183. Springer (1998). https://doi.org/10.1007/BFb0054171
35. The Bitcoin Developers: Bitcoin source code (2021). https://github.com/bitcoin/bitcoin
36. Tsai, M.H., Fu, Y.F., Shi, X., Liu, J., Wang, B.Y., Yang, B.Y.: Automatic certified verification of cryptographic programs with COQCRYPTOLINE. IACR Cryptol. ePrint Arch. p. 1116 (2022)

37. Tsai, M.H., Wang, B.Y., Yang, B.Y.: Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In: Evans, D., Malkin, T., Xu, D. (eds.) ACM SIGSAC Conference on Computer and Communications Security, pp. 1973–1987. ACM (2017)
38. Wimmer, S.: Munta: A verified model checker for timed automata. In: André, É., Stoelinga, M. (eds.) Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11750, pp. 236–243. Springer (2019). https://doi.org/10.1007/978-3-030-29662-9_14
39. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10805, pp. 61–78. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_4
40. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 1789–1806. ACM (2017)

Incremental Dead State Detection in Logarithmic Time

Caleb Stanford[1](✉) and Margus Veanes[2]

[1] University of California, Davis, USA
cdstanford@ucdavis.edu
[2] Microsoft Research, Redmond, USA
margus@microsoft.com

Abstract. Identifying live and dead states in an abstract transition system is a recurring problem in formal verification; for example, it arises in our recent work on efficiently deciding regex constraints in SMT. However, state-of-the-art graph algorithms for maintaining reachability information *incrementally* (that is, as states are visited and before the entire state space is explored) assume that new edges can be added from any state at any time, whereas in many applications, outgoing edges are added from each state as it is explored. To formalize the latter situation, we propose *guided incremental digraphs* (GIDs), incremental graphs which support labeling *closed* states (states which will not receive further outgoing edges). Our main result is that dead state detection in GIDs is solvable in $O(\log m)$ amortized time per edge for m edges, improving upon $O(\sqrt{m})$ per edge due to Bender, Fineman, Gilbert, and Tarjan (BFGT) for general incremental directed graphs.

We introduce two algorithms for GIDs: one establishing the logarithmic time bound, and a second algorithm to explore a lazy heuristics-based approach. To enable an apples-to-apples experimental comparison, we implemented both algorithms, two simpler baselines, and the state-of-the-art BFGT baseline using a common directed graph interface in Rust. Our evaluation shows 110-530x speedups over BFGT for the largest input graphs over a range of graph classes, random graphs, and graphs arising from regex benchmarks.

Keywords: Dead State Detection · Graph Algorithms · Online Algorithms · SMT

1 Introduction

Classifying states in a transition system as live or dead is a recurring problem in formal verification. For example, given an expression, can it be simplified to the identity? Given an input to a nondeterministic program, can it reach a terminal state, or can it reach an infinitely looping state? Given a state in an automaton, can it reach an accepting state? State classification is relevant to satisfiability modulo theories (SMT) solvers [8,9,24,51], where theory-specific partial decision procedures often work by exploring the state space to find a reachable path that

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 241–264, 2023.
https://doi.org/10.1007/978-3-031-37703-7_12

corresponds to a satisfying string or, more generally, a sequence of constructors. To a first approximation, the core problem in all of these cases amounts to classifying each state u in a directed graph as *live*, meaning that a feasible, accepting, or satisfiable state is reachable from u; or *dead*, meaning that all states reachable from u are infeasible, rejecting, or unsatisfiable.

Motivating Applications. We originally encountered the problem of incremental state classification during our prior work while building Z3's regex solver [61] for the SMT theory of string and regex constraints [4,13,15]. Our solver leveraged *derivatives* (in the sense of Brzozowski [18] and Antimirov [5]) to explore the states of the finite state machine corresponding to the regex incrementally (as the graph is built), to avoid the prohibitive cost of expanding all states initially. This turns out to require solving the live and dead state detection problem in the finite state machine presented as an incremental directed graph.[1] Concretely, consider the regex $(\mathbf{.}^*\alpha\mathbf{.}^{100})^C \cap (\mathbf{.}\alpha)$, where $\mathbf{.}$ matches any character, $\cap$ is regex intersection, ^{C} is regex complement, and α matches any digit (0-9). A traditional solver would expand the left and right operands as state machines, but the left operand $(\mathbf{.}^*\alpha\mathbf{.}^{100})^C$ is astronomically large as a DFA, causing the solver to hang. The derivative-based technique instead constructs the derivative regex: $(\mathbf{.}^*\alpha\mathbf{.}^{100})^C \cap (\mathbf{.}^{100})^C \cap \alpha$. At this stage we have a graph of two states and one edge, where the states are the two regexes just described, and the edge is the derivative relation. After one more derivative operation, the regex is reduced to one that is clearly nonempty as it accepts the empty string.

It is important that a derivative-based solver identify nonempty (live) and empty (dead) regexes *incrementally* because it does not generally construct the entire state space before terminating (see the graph update rule Upd, p. 626 [61]). Moreover, the nonemptiness problem for extended regexes is non-elementary [62] — and still PSPACE-complete for more restricted fragments — which strongly favors a lazy approach over brute-force search.

Regexes are just one possible application; the algorithms we will present here are broadly applicable to any context where the states have a bounded (per-node) out-degree. For example, they could be applied in LTL model checking when lazily exploring the state space of a nondeterministic Büchi automaton (NBA), where the NBA is too expensive to construct up front. The important fact is that each state of the automaton has only finitely many outgoing edges, and when all these are added, we can hope to check for dead states incrementally.

Prior Work. Traditionally, while live state detection can be done incrementally, dead state detection is often done exhaustively (i.e., after the entire state space is explored). For example, bounded and finite-state model checkers based on translations to automata [20,43,58], as well as classical dead-state elimination algorithms [12,16,37], typically work on a fixed state space after it has been fully enumerated. However, we reiterate that exhaustive exploration is prohibitive for large (e.g., exponential or infinite) state spaces which arise in an SMT

[1] The specific setting is regexes with intersection and complement (*extended* [31,44] or *generalized* [26] regexes), which are found natively in security applications [6,61]. Other solvers have also leveraged derivatives [45] and laziness in general [36].

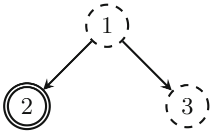


Fig. 1. GID consisting of the sequence of updates E(1, 2), E(1, 3), T(2). Terminal states are drawn as double circles. After the update T(2), states 1 and 2 are known to be live. State 3 is not dead in this GID, as a future update may cause it to be live.

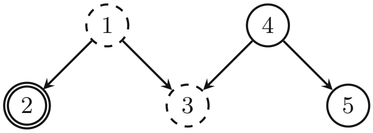


Fig. 2. GID extending Fig. 1 with additional updates E(4, 3), E(4, 5), C(4), C(5). Closed states are drawn as solid circles. After the update C(5) (but not earlier), state 5 is dead. State 4 is not dead because it can still reach state 3.

verification context. We also have good evidence that incremental feedback can improve SMT solver performance: a representative success story is the e-graph data structure [23,67], which maintains an equivalence relation among expressions incrementally; because it applies to general expressions, it is theory-independent and re-usable. Incremental state space exploration could lead to similar benefits if applied to SMT procedures which still rely on exhaustive search.

However, in order to perform incremental dead state detection, we currently lack algorithms which match offline performance. As we discuss in Sect. 2, the best-known existing solutions would require maintaining strong connected components (SCCs) incrementally. For SCC maintenance and the related simpler problem of cycle detection, amortized algorithms are known with $O(m^{3/2})$ total time for m edge additions [10,33], with some recently announced improvements [11,14]. Note that this is in sharp contrast to $O(m)$ for the offline variants of these problems, which can be solved by breadth-first or depth-first search. More generally, research suggests there are computational barriers to solving unconstrained reachability problems in incremental and dynamic graphs [1,29].

This Paper. To improve on prior algorithms, our key observation is that in many applications (including our motivating applications above), edges are not added adversarially, but *from one state at a time* as the states are explored. As a result, we know when a state will have no further outgoing edges. This enables us to (i) identify dead states incrementally, rather than only after the whole state space is explored; and (ii) obtain more efficient algorithms than currently exist for general graph reachability.

We introduce *guided incremental digraphs* (GIDs), a variation on incremental graphs. Like an incremental directed graph, a guided incremental digraph may be updated by adding new edges between states, or a state may be labeled as *closed*, meaning it will receive no further outgoing edges. Some states are designated as *terminal*, and we say that a state is *live* if it can reach a terminal state and *dead* if it will never reach a terminal state in any extension – i.e. if all reachable states from it are closed (see Figs. 1 and 2). To our knowledge, the problem of detecting dead states in such a system has not been studied by existing work in graph algorithms. Our problem can be solved through solving SCC maintenance, but not necessarily the other way around (Sect. 2, Proposition 1). We provide two new algorithms for dead-state detection in GIDs.

First, we show that the dead-state detection problem for GIDs can be solved in time $O(m \cdot \log m)$ for m edge additions, within a logarithmic factor of the $O(m)$ cost for offline search. The worst-case performance of our algorithm thus strictly improves on the $O(m^{3/2})$ upper bound for SCC maintenance in general incremental graphs. Our algorithm is technically sophisticated, and utilizes several data structures and existing results in online algorithms: in particular, Union-Find [63] and Henzinger and King's Euler Tour Trees [35]. The main idea is that, rather than explicitly computing the set of SCCs, for closed states we maintain a single path to a non-closed (open) state. This turns out to reduce the problem to quickly determining whether two states are currently assigned a path to the same open state. On the other hand, Euler Tour Trees can solve *undirected* reachability for graphs that are forests in logarithmic time.[2] The challenge then lies in figuring out how to reduce directed connectivity in the graph of paths to an undirected forest connectivity problem. At the same time, we must maintain this reduction under Union-Find state merges, in order to deal with cycles that are found in the graph along the way.

While as theorists we would like to believe that asymptotic complexity is enough, the truth is that the use of complex data structures (1) can be prohibitively expensive in practice due to constant-factor overheads, and (2) can make algorithms substantially more difficult to implement, leading practitioners to prefer simpler approaches. To address these needs, in addition to the logarithmic-time algorithm, we provide a second *lazy* algorithm which avoids the user of Euler Tour Trees, and only uses union-find. This algorithm is based on an optimization of adding shortcut *jump* edges for long paths in the graph to quickly determine reachability. This approach aims to perform well in practice on typical graphs, and is evaluated in our evaluation along with the logarithmic time algorithm, though we do not prove its asymptotic complexity.

Finally, we implement and empirically evaluate both of our algorithms for GIDs against several baselines in 5.5k lines of code in Rust [47]. Our evaluation focuses on the performance of the GID data structure itself, rather than its end-to-end performance in applications. To ensure an apples-to-apples comparison with existing approaches, we put particular focus on providing a directed graph data structure backend shared by all algorithms, so that the cost of graph search as well as state and edge merges is identical across algorithms. We implement two naïve baselines, as well as an implementation of the state-of-the-art solution

[2] Reachability in dynamic forests can also be solved by Sleator-Tarjan trees [59], Frederickson's Topology Trees [30], or Top Trees [3]. Of these, we found Euler Tour Trees the easiest to work with in our implementation. See also [64].

based on maintaining SCCs, BFGT [10] in our framework. To our knowledge, the latter is the first implementation of BFGT specifically for SCC maintenance. On a collection of generated benchmark GIDs, random GIDs, and GIDs directly pulled from the regex application, we demonstrate a substantial improvement over BFGT for both of our algorithms. For example, for larger GIDs (those with over 100K updates), we observe a 110-530x speedup over BFGT.

Contributions. Our primary contributions are:

- *Guided incremental digraphs* (GIDs), a formalization of incremental live and dead state detection which supports labeling *closed* states. (Section 2)
- Two algorithms for the state classification problem in GIDs: first, an algorithm that works in amortized $O(\log m)$ time per update, improving upon the state-of-the-art amortized $O(\sqrt{m})$ per update for incremental graphs; and second, a simpler algorithm based on lazy heuristics. (Section 3)
- An open-source implementation[3] of GIDs in Rust, and an evaluation which demonstrates up to two orders of magnitude speedup over BFGT. (Section 4)

Following the above, we expand on the application of GIDs to regex solving in SMT (Sect. 5) and survey related work (Sect. 6).

2 Guided Incremental Digraphs

2.1 Problem Statement

An incremental digraph is a sequence of edge updates $\texttt{E}(u, v)$, where the algorithmic challenge in this context is to produce some output after each edge is received (e.g., whether or not a cycle exists). If the graph also contains updates $\texttt{T}(u)$ labeling a state as *terminal*, then we say that a state is *live* if it can reach a terminal state in the current graph. In a *guided* incremental digraph, we also include updates $\texttt{C}(u)$ labeling a state as *closed*, meaning that will not receive any further outgoing edges.

Definition 1. Define a *guided incremental digraph (GID)* to be a sequence of updates, where each update is one of the following:

(i) a new directed *edge* $\texttt{E}(u, v)$;
(ii) a label $\texttt{T}(u)$ which indicates that u is *terminal*; or
(iii) a label $\texttt{C}(u)$ which indicates that u is *closed*, i.e. no further edges will be added going out from u (or labels to u).

The GID is *valid* if the *closed* labels are correct: there are no instances of $\texttt{E}(u, v)$ or $\texttt{T}(u)$ after an update $\texttt{C}(u)$. The *denotation* of G is the directed graph (V, E) where V is the set of all states u which have occurred in any update in the sequence, and E is the set of all (u, v) such that $\texttt{E}(u, v)$ occurs in G. An *extension* of a valid GID G is a valid GID G' such that G is a prefix of G'.

[3] https://github.com/cdstanford/gid.

In a valid GID G, we say that a state u is *live* if there is a path from u to a terminal state in the denotation of G; and a state u is *dead* if it is not live in *any* extension of G. Notice that in a GID without any $\mathtt{C}(u)$ updates, no states are dead as an edge may be added in an extension which makes them live.

We provide an example of a valid GID in Figs. 1 and 2 consisting of the following sequence of updates: $\mathtt{E}(1, 2)$, $\mathtt{E}(1, 3)$, $\mathtt{T}(2)$, $\mathtt{E}(4, 3)$, $\mathtt{E}(4, 5)$, $\mathtt{C}(4)$, $\mathtt{C}(5)$. Terminal states $\mathtt{T}(u)$ are drawn as double circles; closed states, as single circles $\mathtt{C}(u)$; and states that are not closed, as dashed circles.

Definition 2. Given as input a valid GID, the *GID state classification problem* is to output, in an online fashion after each update, the set of new live and new dead states. That is, output $\mathtt{Live}(u)$ or $\mathtt{Dead}(u)$ on the smallest prefix of updates such that u is live or dead on that prefix, respectively.

2.2 Existing Approaches

In many applications, one might choose to classify dead states offline, after the entire state space is enumerated. This leads to a linear-time algorithm via either DFS or BFS, but it does not solve our problem (Definition 2) because it is not incremental. Naïve application of this idea leads to $O(m)$ per update for m updates ($O(m^2)$ total), as we may redo the entire search after each update.

For acyclic graphs, there exists an amortized $O(1)$-time per update algorithm for the problem (Definition 2): maintain the graph as a list of forward- and backward-edges at each state. When a state v is marked terminal, do a DFS along backward-edges to determine all states u that can reach v not already marked as live, and mark them live. When a state v is marked closed, visit all forward-edges from v; if all are dead, mark v as dead and recurse along all backward-edges from v. As each edge is visited only when marking a state live or dead, it is only visited a constant number of times overall (though we may use more than $O(1)$ time on some particular update pass). Additionally, the live state detection part of this procedure still works for graphs containing cycles.

The challenge, therefore, lies primarily in detecting dead states in graphs which may contain cycles. For this, the breakthrough approach from [10] maintains a *condensed* graph which is acyclic, where the vertices in the condensed graph represent strongly connected components (SCCs) of states. The mapping from states to SCCs is maintained using a Union-Find [63] data structure. Maintaining the condensed graph requires $O(\sqrt{m})$ time per update. To avoid confusing closed and non-closed states, we also have to make sure that they are not merged into the same SCC; the easiest solution to this is to withhold all edges from each state u in the graph until u are closed, which ensures that u must be in a SCC on its own. Once we have the condensed graph with these modifications, the same algorithm as in the previous paragraph works to identify live and dead states. Since each edge is only visited when a state is marked closed or live, each edge is visited only once throughout the algorithm, we use only amortized $O(1)$ additional time to calculate live and dead states. While this SCC maintenance algorithm ignores the fact that edges do not occur from closed states $\mathtt{C}(u)$, this still proves the following result:

Live	Some reachable state from u is terminal.
Dead	All reachable states from u (including u) are closed and not terminal.
Unknown	u is closed, but not live or dead.
Open	u is not live and not closed.
Terminal	A state u labeled by $\texttt{T}(u)$.
Closed	A state u labeled by $\texttt{C}(u)$.
Canonical	A state x such that $\texttt{UF.find}(x) = x$.
u, v, w	States (may or may not be canonical).
x, y, z	Canonical states (i.e., states in the condensed graph).
Successor $\texttt{succ}(x)$	For an unknown, canonical state x, a uniquely chosen v such that (x, v) is an edge, and following the path of successors leads to an open state.

Fig. 3. *Top:* Basic classification of GID states into four disjoint categories. *Bottom:* Additional terminology used in this paper.

Proposition 1. *GID state classification reduces to SCC maintenance. That is, suppose we have an algorithm over incremental graphs that maintains the set of SCCs in $O(f(m, n))$ total time given n states and m edge additions.*[4] *Then there exists an algorithm to solve GID state classification in $O(f(m, n))$ total time.*

Despite this reduction one way, there is no obvious reduction the other way – from cycle detection or SCCs to Definition 2. This is because, while the existence of a cycle of non-live states implies bi-reachability between all states in the cycle, it does not necessarily imply that all of the bi-reachable states are dead.

3 Algorithms

This section presents Algorithm 2, which solves the state classification problem in logarithmic time (Theorem 3); and Algorithm 3, an alternative lazy solution. Both algorithms are optimized versions of Algorithm 1, a first-cut algorithm which establishes the structure of our approach. We begin by establishing some basic terminology shared by all of the algorithms (see Fig. 3).

States in a GID can be usefully classified as exactly one of four *statuses*: *live*, *dead*, *unknown*, or *open*, where *unknown* means "closed but not yet live or dead", and *open* means "not closed and not live". Note that a state may be live and neither open nor closed; this terminology keeps the classification disjoint. Pragmatically, for live states it does not matter if they are classified as open or closed, since edges from those states no longer have any effect. However, all dead and unknown states are closed, and no states are both open and closed.

Given this classification, the intuition is that for each unknown state u, we only need *one* path from u to an open state to prove that it is not dead; we want to maintain one such path for all unknown states. To maintain all of these paths

[4] To be precise, "maintains" means that (i) we can check whether two states are in the same SCC in $O(1)$ time; and (ii) we can iterate over all the states, edges from, or edges into a SCC in $O(1)$ time per state or edge.

simultaneously, we maintain an acyclic directed *forest* structure on unknown and open states where the roots are open states, and all non-root states have a single edge to another state, called its *successor*. Edges other than successor edges can be temporarily ignored, except for when marking live states; these are kept as *reserve* edges. Specifically, we add every edge (u, v) as a backward-edge from v (to allow propagating live states), but for edges not in the forest we keep (u, v) in a reserve list from u. We store all edges, including backward-edges, in the original order (u, v). The reserve list edge becomes relevant only when either (i) u is marked as closed, or (ii) u's successor is marked as dead.

In order to deal with cycles, we need to maintain the forest of unknown states not on the original graph, but on a union-find *condensed graph*, similar to [63]. When we find a cycle of unknown states, we *merge* all states in the cycle by calling the union method in the union-find. We refer to a state as *canonical* if it is the canonical representative of its equivalence class in the union find; the condensed graph is a forest on canonical states. We use x, y, z to denote canonical states (states in the condensed graph), and u, v, w to denote the original states (not known to be canonical). Following [63], we maintain edges as linked lists rather than sets, and using the original states instead of canonical states; this is important as it allows combining edge lists in $O(1)$ time when merging states.

3.1 First-Cut Algorithm

Algorithm 1 is a first cut based on these ideas. The procedures OnEdge and OnTerminal contain all the logic to identify live states, using `bck` to look up backward-edges; OnTerminal doubles as a "mark live" function when it is called by OnEdge. The procedure OnClosed tries to assign a successor edge to a newly closed state, to prove that it is not dead. In case we run out of reserve edges, the state is marked dead and we recursively call OnClosed along backward-edges, which will either set a new successor or mark them dead.

The union-find data structure `UF` provides `UF.union`(v_1, v_2), `UF.find`(v), and `UF.iter`(v): `UF.union` merges v_1 and v_2 to refer to the same canonical state, `UF.find` returns the canonical state for v, and `UF.iter` iterates over states equivalent to v. These use amortized $\alpha(n)$ for n updates, where $\alpha(n) \in o(\log n)$ is the inverse Ackermann function. We only merge states if they are bi-reachable from each other, and both unknown; this implies that all states equivalent to a state x have the same status. Each edge (u, v) is always stored in the maps `res` and `bck` using its original states (i.e., edge labels are not updated when states are merged); but we can quickly obtain the corresponding edge on canonical states via (`UF.find`(u),`UF.find`(v)). Once a state is marked `Live` or `Dead`, its edge maps are no longer used.

Invariants. Altogether, we respect the following invariants. *Successor* and *no cycles* describe the forest structure, and, *edge representation* ensures that all edges in the input GID are represented somehow in the current graph.

- *Merge equivalence:* For all states u and v, if `UF.find`(u) = `UF.find`(v), then u and v are bi-reachable and both closed. (This implies that u and v are both live, both dead, or both unknown.)

Algorithm 1. First-cut algorithm.

```
V: a type for states (integers) (variables $u, v, \ldots$)
E: the type of edges, equal to (V, V)
UF: a union-find data structure over V
X: the set of canonical states in UF (variables $x, y, z, \ldots$)
status: a map from X to Live, Dead, Unknown, or Open
succ: a map from X to V
res and bck: maps from X to linked lists of E
procedure OnEdge(E($u, v$))
    $x \leftarrow$ UF.find($u$); $y \leftarrow$ UF.find($v$)
    if status($y$) = Live then
        OnTerminal(T($x$))                          ▷ mark $x$ and its ancestors live
    else if status($x$) $\neq$ Live then            ▷ status($x$) must be Open
        append $(u, v)$ to res($x$)
        append $(u, v)$ to bck($y$)
procedure OnTerminal(T($v$))
    $y \leftarrow$ UF.find($v$)
    for all $x$ in DFS backwards (along bck) from $y$ not already Live do
        status($x$) $\leftarrow$ Live
        output Live($x'$) for all $x'$ in UF.iter($x$)
procedure OnClosed(C($v$))
    $y \leftarrow$ UF.find($v$)
    if status($y$) $\neq$ Open then return          ▷ $y$ is already live or closed
    while res($y$) is nonempty do
        pop $(v, w)$ from res($y$); $z \leftarrow$ UF.find($w$)
        if status($z$) = Dead then continue
        else if CheckCycle($y, z$) then
            for all $z'$ in cycle from $z$ to $y$ do $z \leftarrow$ Merge($z, z'$)
        else
            status($y$) $\leftarrow$ Unknown; succ($y$) $\leftarrow z$;
            return
    status($y$) $\leftarrow$ Dead; output Dead($y'$) for all $y'$ in UF.iter($y$)
    ToRecurse $\leftarrow \varnothing$
    for all $(u, v)$ in bck($y$) do
        $x \leftarrow$ UF.find($u$)
        if status($x$) = Unknown and UF.find(succ($x$)) = $y$ then
            status($x$) $\leftarrow$ Open       ▷ temporary – marked closed on recursive call
            add $x$ to ToRecurse
    for all $x$ in ToRecurse do OnClosed(C($x$))
procedure CheckCycle($y, z$) returning bool
    while status($z$) = Unknown do $z \leftarrow$ UF.find(succ($z$))  ▷ get root state from $z$
    return $y = z$
procedure Merge($x, y$) returning V
    $z \leftarrow$ UF.union($x, y$)
    bck($z$) $\leftarrow$ bck($x$) + bck($y$)                ▷ $O(1)$ linked list append
    res($z$) $\leftarrow$ res($x$) + res($y$)                ▷ $O(1)$ linked list append
    return $z$
```

- *Status correctness:* For all u, $\texttt{status}(\texttt{UF.find}(u))$ equals the status of u.
- *Successor edges:* If x is unknown, then $\texttt{succ}(x)$ is defined and is an unknown or open state. If x is open, then $\texttt{succ}(x)$ is not defined.
- *No cycles:* There are no cycles among the set of edges $(x, \texttt{UF.find}(\texttt{succ}(x)))$, over all unknown and open canonical states x.
- *Edge representation:* For all edges (u, v) in the input GID, at least one of the following holds: (i) $(u, v) \in \texttt{res}(\texttt{UF.find}(v))$; (ii) $v = \texttt{succ}(\texttt{UF.find}(u))$; (iii) $\texttt{UF.find}(u) = \texttt{UF.find}(v)$; (iv) u is live; or (v) v is dead.

Theorem 1. *Algorithm 1 is correct.*

Proof (Summary). The full proof can be found in the arXiv version [60]. The *status correctness* invariant implies correct output at each step, so it suffices to argue that all of the invariants above are preserved. Upon receiving $\texttt{E}(u, v)$ or $\texttt{T}(u)$, some dead, unknown, or open states may become live, but this does not change the status of any other states. The main challenge of the proof is the recursive procedure OnClosedC(u). On recursive calls, some states are *temporarily* marked `Open`, meaning they are roots in the forest structure. During recursive calls, we need a slightly generalized invariant: each forest root corresponds to a pending call to OnClosedC(u) (i.e., an element of `ToRecurse` for some call on the stack) and is a state that is dead iff all of its reserve edges are dead. After we prove this (generalized) invariant, when OnClosedC(u) terminates, we know that there are no more temporary open states, and the forest structure implies that all closed states are correctly marked as unknown.

Complexity. The core inefficiency in Algorithm 1 — what we need to improve — lies in CheckCycle. The procedure repeatedly sets $z \leftarrow \texttt{succ}(z)$ to find the tree root, which in general could be linear time in the number of edges. For example, this inefficiency results in $O(m^2)$ work for a linear graph read in backwards order: `E`(2, 1), `C`(2), `E`(3, 2), `C`(3), ..., `E`(n, n-1), `C`(n).

All other procedures use amortized $\alpha(m)$ time per update for m updates, using array lists to represent the maps `fwd`, `bck`, and `succ` for $O(1)$ lookups. To do the amortized analysis, the cost of each call to OnClosed can be assigned *either* to the target of an edge being marked dead, *or* to an edge being merged as part of a cycle, and both of these events can only happen once per edge added to the GID. And the OnTerminal calls and loop iterations only run once per edge in the graph when the target of that edge is marked live or terminal.

3.2 Logarithmic Algorithm

At its core, CheckCycle requires solving an *undirected* reachability problem on a graph that is restricted to a forest. However, the forest is changed not just by edge additions, but edge additions *and* deletions. While undirected reachability and reachability in directed graphs are both difficult to solve incrementally, reachability in *dynamic forests* can be solved in $O(\log m)$ time per operation. This is the main intuition for our solution, using an Euler Tour Trees data structure `EF` of Henzinger and King [35], shown in Algorithm 2.

Algorithm 2. Logarithmic time algorithm.

```
All data from Algorithm 1; succ: a map from X to E (instead of to V)
EF: Euler Tour Trees data structure providing: EF.add, EF.remove, EF.connected
procedure OnEdge, Merge as in Algorithm 1
procedure OnTerminal(T($v$))
    $y \leftarrow$ UF.find($v$)
    for all $x$ in DFS backwards (along bck) from $y$ not already Live do
        if status($x$) = Unknown then
            ▷ The following line is not strictly necessary, but simplifies the analysis
            $(u, v) \leftarrow$ succ($x$); delete succ($x$); EF.remove($u, v$)
        status($x$) $\leftarrow$ Live; output Live($x'$) for all $x'$ in UF.iter($x$)
procedure OnClosed(C($v$))
    $y \leftarrow$ UF.find($v$)
    if status($y$) $\neq$ Open then return
    while res($y$) is nonempty do
        pop $(v, w)$ from res($y$); $z \leftarrow$ UF.find($w$)
        if status($z$) = Dead then continue
        else if CheckCycle($y, z$) then
            for all $z'$ in cycle from $z$ to $y$ do $z \leftarrow$ Merge($z, z'$)
        else
            status($x$) $\leftarrow$ Unknown; succ($x$) $\leftarrow (v, w)$
            EF.add($v, w$); return  ▷ undirected edge; use original labels (not $(x, y)$)
    status($y$) $\leftarrow$ Dead; ToRec $\leftarrow \varnothing$; output Dead($y'$) for all $y'$ in UF.iter($y$)
    for all $(u, v)$ in bck($y$) do
        $x \leftarrow$ UF.find($u$)
        if status($x$) = Unknown then
            $(u', v') \leftarrow$ succ($x$)
            if UF.find($v'$) = $y$ then
                EF.remove($u', v'$); status($x$) $\leftarrow$ Open; delete succ($x$); add $x$ to ToRec
    for all $x$ in ToRec do OnClosed(C($x$))
procedure CheckCycle($y, z$) returning bool
    return EF.connected($y, z$)
```

Unfortunately, this idea does not work straightforwardly – once again because of the presence of cycles in the original graph. We cannot simply store the forest as a condensed graph with edges on condensed states. As we saw in Algorithm 1, it was important to store successor edges as edges into V, rather than edges into X – this is the only way that we can merge states in $O(1)$, without actually inspecting the edge lists. If we needed to update the forest edges to be in X, this could require $O(m)$ work to merge two $O(m)$-sized edge lists as each edge might need to be relabeled in the EF graph.

To solve this challenge, we instead store the EF data structure on the original states, rather than the condensed graph; but we ensure that *each canonical state is represented by a tree of original states.* When adding edges between canonical states, we need to make sure to remember the original label (u, v), so that we can later remove it using the original labels (this happens when its target becomes dead). When an edge would create a cycle, we instead simply ignore it in the `EF` graph, because a line of connected trees forms a tree.

Summary and Invariants. In summary, the algorithm reuses the data, procedures, and invariants from Algorithm 1, with the following important changes: (1) We maintain the EF data structure `EF`, a forest on `V`. (2) The successor edges are stored as their original edge labels (u, v), rather than just as a target state. (3) The procedure OnClosed is rewritten to maintain the graph `EF`. (4) The *successor edges* and *no cycles* invariants use the new `succ` representation: that is, they are constraints on the edges $(x, \texttt{UF.find}(v))$, where $\texttt{succ}(x) = (u, v)$. (5) We add the following two constraints on edges in `EF`, depending on whether those states are equivalent in the union-find structure.

- *EF inter-edges:* For all *inequivalent* u, v, (u, v) is in the EF if and only if $(u, v) = \texttt{succ}(\texttt{UF.find}(u))$ or $(v, u) = \texttt{succ}(\texttt{UF.find}(v))$.
- *EF intra-edges:* For all unknown canonical states x, the set of edges (u, v) in the EF between states belonging to x forms a tree.

Theorem 2. *Algorithm 2 is correct.*

Proof. Observe that the `EF` inter-edges constraint implies that `EF` only contains edges between unknown and open states, together with isolated trees. In the modified OnTerminal procedure, when marking states as live we remove inter-edges, so we preserve this invariant.

Next we argue that given the invariants about `EF`, for an *open* state y the CheckCycle procedure returns true if and only if (y, z) would create a directed cycle. If there is a cycle of canonical states, then because canonical states are connected trees in `EF`, the cycle can be lifted to a cycle on original states, so y and z must already be connected in this cycle without the edge (y, z). Conversely, if y and z are connected in `EF`, then there is a path from y to z, and this can be projected to a path on canonical states. However, because y is open, it is a root in the successor forest, so any path from y along successor edges travels only on backward-edges; hence z is an ancestor of y in the *directed* graph, and thus (y, z) creates a directed cycle.

This leaves the OnClosed procedure. Other than the `EF` lines, the structure is the same as in Algorithm 1, so the previous invariants are still preserved, and it remains to check the `EF` invariants. When we delete the successor edge and temporarily mark $\texttt{status}(x) = \texttt{Open}$ for recursive calls, we also remove it from `EF`, preserving the inter-edge invariant. Similarly, when we add a successor edge to x, we add it to `EF`, preserving the inter-edge invariant. So it remains to consider when the set of canonical states changes, which is when merging states in a cycle. Here, a line of canonical states is merged into a single state, and a line of connected trees is still a tree, so the intra-edge invariant still holds for the new canonical state, and we are done. □

Theorem 3. *Algorithm 2 uses amortized logarithmic time per edge update.*

Proof. By the analysis of Algorithm 1, each line of the algorithm is executed $O(m)$ times and there are $O(m)$ calls to CheckCycle. Each line of code is

Algorithm 3. Lazy algorithm.

```
All data from Algorithm 1; jumps: a map from X to lists of V
procedure OnEdge, OnTerminal OnClosed as in Algorithm 1
procedure CheckCycle($y, z$) returning bool
    return $y$ = GetRoot($z$)
procedure GetRoot($z$) returning V
    if status($z$) = Open then return $z$
    if jumps($z$) is empty then push succ($z$) to jumps($z$)        ▷ set 0th jump
    repeat pop $w$ from jumps($z$); $z'$ = UF.find($w$)             ▷ remove dead jumps
    until status($z'$) $\neq$ Dead
    push $z'$ to jumps($z$); result $\leftarrow$ GetRoot($z'$)
    $n \leftarrow$ length(jumps($z$)); $n' \leftarrow$ length(jumps($z'$))
    if $n \leq n'$ then push jumps($z'$)[$n - 1$] to jumps($z$)     ▷ set $n$th jump
    return result
procedure Merge($x, y$) returning V
    $z \leftarrow$ UF.union($x, y$)
    bck($z$) $\leftarrow$ bck($x$) + bck($y$); res($z$) $\leftarrow$ res($x$) + res($y$)
    jumps($z$) $\leftarrow$ empty; return $z$
```

either constant-time, $\alpha(m) = o(\log m)$ time for the UF calls, or $O(\log m)$ time for the EF calls, so in total the algorithm takes $O(m \log m)$ time total, or amortized $O(\log m)$ time per edge. □

3.3 Lazy Algorithm

While the asymptotic complexity of $\log m$ could be the end of the story, in practice, we found the cost of the EF calls to be a significant overhead. The technical details of Euler Tour Trees include building an AVL-tree cycle for each tree, where the cycle contains each state of the graph once and each edge in the graph twice. While this is elegant, it turns out that adding *one edge* to EF results in no less than *seven* modifications to the AVL tree: a split at the source, then a split at the target, then an edge addition in both directions (u, v) and (v, u) to the cycle, and finally the four resulting trees need to be glued together (using three merge operations).[5] Each one of these operations comes with a rebalancing operation which could do $\Omega(\log m)$ tree rotations and pointer dereferences to visit the nodes in the AVL tree. Some optimizations may be possible – including, e.g., combining rebalancing operations or considering variants of AVL trees with better cache locality. Nonetheless, these constant-factor overheads constitute a serious practical drawback for Algorithm 2.

To address this, in this section, we investigate a simpler, lazy algorithm which avoids EF and directly optimizes Algorithm 1. For this, one idea in the right direction is to store for each state a direct pointer to the root which results from

[5] Our implementation actually uses nine modifications, as the splits at the source and target also disconnect the source and target states.

repeatedly calling `succ`. But there are two issues with this. First, maintaining this may be difficult (when the root changes, potentially updating a linear number of root pointers). Second, the root may be marked dead, in which case we have to re-compute all pointers to that root.

Instead, we introduce a *jump list* from each state: intuitively, it will contain states after calling successor once, twice, four times, eight times, and so on at powers of two; and it will be updated lazily, at most once for every visit to the state. When a jump becomes obsolete (the target dead), we just pop off the largest jump, so we do not lose all of our work in building the list. We maintain the following additional information: for each unknown canonical state x, a nonempty list of *jumps* $[v_0, v_1, v_2, \dots, v_k]$, such that v_0 is reachable from x, v_1 is reachable from v_0, v_2 is reachable from v_1, and so on, and $v_1 = \texttt{succ}(x)$.

The resulting algorithm is shown in Algorithm 3. The key procedure is GetRoot$z$, which is called when adding a reserve edge (y, z) to the graph. In addition to all invariants from Algorithm 1, we maintain the following invariants for *every* unknown canonical state x, where $\texttt{jumps}(x)$ is a list of states $v_0, v_1, v_2, \dots, v_k$. *First jump:* if the jump list is nonempty, then $v_0 = \texttt{succ}(v)$. *Reachability:* v_{i+1} is reachable from v_i for all i. The jump list also satisfies the following *powers of two* invariant: on the path of canonical states from v_0 to v_i, the total number of states (including all states in each equivalence class) is at least 2^i. While this invariant is not necessary for correctness, it is the key to the algorithm's practical efficiency: it follows from this that *if* the jump list is fully saturated for every state, querying GetRoot$z$ will take only logarithmic time. However, since jump lists are updated lazily, the jump list may not be saturated, so this does not establish a true asymptotic complexity for the algorithm.

Theorem 4. *Algorithm 3 is correct.*

Proof. The *first jump* and *reachability* invariants imply that $v_1, v_2, \dots$ is some sublist of the states along the path from an unknown state to its root, potentially followed by some dead states. We need to argue that the subprocedure GetRoot (i) receives the same verdict as repeatedly calling `succ` to find a cycle in the first-cut algorithm and (ii) preserve both invariants. For *first jump*, if the jump list is empty, then GetRoot ensures that the first jump is set to the successor state. For *reachability*, popping dead states from the jump list clearly preserves the invariant, as does adding on a state along the path to the root, which is done when $k' \geq k$. Merging states preserves both invariants trivially because we throw the jump list away, and marking states live preserves both invariants trivially since the jump list is only maintained and used for unknown states. □

4 Experimental Evaluation

The primary goal of our evaluation has been to experimentally validate the performance of GIDs as a data structure in isolation, rather than their use in a particular application. Our evaluation seeks to answer the following questions:

Q1 How does our approach (Algorithms 2 and 3) compare to the state-of-the-art approach based on maintaining SCCs?

Q2 How does the performance of the studied algorithms vary when the class of input graphs changes (e.g., sparse vs. dense, structured vs. random)?

Q3 Finally, how do the studied algorithms perform on GIDs taken from the example application to regexes described in Sect. 5?

To answer **Q1**, we put substantial implementation effort into a common framework on which a fair comparison could be made between different approaches. To this end, we implemented GIDs as a data structure in Rust which includes a graph data structure on top of which all algorithms are built. In particular, this equalizes performance across algorithms for the following baseline operations: state and edge addition and retrieval, DFS and BFS search, edge iteration, and state merging. We chose Rust for our implementation for its performance, and because there does not appear to be an existing publicly available implementation of BFGT in any other language.[6] The number of lines of code used to implement these various structures is summarized in Fig. 4. We implement Algorithms 2 and 3 and compare them with the following baselines:

BFGT The state-of-the-art approach based on SCC maintenance, using worst-case amortized $O(\sqrt{m})$ time per update [10].

Simple A simpler version of BFGT that uses a forward-DFS to search for cycles. Like Algorithm 1, it can take $\Theta(m^2)$ in the worst case.

Naïve A greedy upper bound for all approaches which re-computes the entire set of dead states using a linear-time DFS after each update.

To answer **Q2**, first, we compiled a range of basic graph classes which are designed to expose edge case behavior in the algorithms, as well as randomly generated graphs. We focus on graphs with no live states, as live states are treated similarly by all algorithms. Most of the generated graphs come in $2 \times 2 = 4$ variants: (i) the states are either read in a forwards- or backwards- order; and (ii) they are either *dead* graphs, where there are no open states at the end and so everything gets marked dead; or *unknown* graphs, where there is a single open state at the end, so most states are unknown. In the unknown case, it is sufficient to have one open state at the end, as many open states can be reduced to the case of a single open state where all edges point to that one. We include GIDs from line graphs and cycle graphs (up to 100K states in multiples of 3); complete and complete acyclic graphs (up to 1K states); and bipartite graphs (up to 1K states). These are important cases, for example, because the reverse-order line and cycle graphs are a potential worst case for Simple and BFGT.

Second, to exhibit more dynamic behavior, we generated random graphs: sparse graphs with a fixed out-degree from each state, chosen from $1, 2, 3$, or 10 (up to 100K states); and dense graphs with a fixed probability of each edge, chosen from .01, .02, or .03 (up to 10K states). Each case uses 10 different random seeds. As with the basic graphs, states are read in some order and marked closed.

[6] That is, BFGT for SCC maintenance. BFGT for cycle detection has been implemented before, for instance, in [28] and formally verified in [32].

Implementation Component	LoC
Common Framework	1197
Naïve Algorithm	78
Simple Algorithm	98
BFGT Algorithm	265
Algorithm 2 (Ours)	253
Algorithm 3 (Ours)	283
Euler Tour Trees	1510
Experimental Scripts	556
Separated Unit Tests	800
Utility	217
Other	69
Total	5326

Category	Benchmark	Source	Qty
Basic	Line		24
	Cycle		24
	Complete		18
	Bipartite		14
	Total		80
Random	Sparse		260
	Dense		130
	Total		390
Regex	RegExLib [15]	2061	37
	Handwritten [61]	70	26
	Additional		11
	Total		74

Fig. 4. *Left:* Lines of code for each algorithm and other implementation components. *Right:* Benchmark GIDs used in our evaluation. Where present, the source column indicates the quantity prior to filtering out trivially small graphs.

To answer **Q3**, we wrote a backend to extract a GID at runtime from Z3's regex solver [61]. While the backend of the solver is precisely a GID — and so could be passed to our GID implementation dynamically — this setup includes many extraneous overheads, including rewriting expressions and computing derivatives when adding nodes to the graph. While some of these overheads may be possible to eliminate, and we are fairly confident that GIDs would be a bottleneck for sufficiently large input examples, this makes it difficult to isolate the performance impact of the GID data structure itself, which is the sole focus of this paper. We therefore instrumented the Z3 solver code to export the (incremental) sequence of graph updates that would be performed during a run of Z3 on existing regex benchmarks. For each benchmark, this instrumented code produces a faithful representation of the sequence of graph updates that actually occur in a run of the SMT solver on this particular benchmark. For each regex benchmark, we thus get a GID benchmark for the present paper. The benchmarks focus on *extended* regexes, rather than plain classical regexes as these are the ones for which dead state detection is relevant (see Sect. 5). We include GIDs for the RegExLib benchmarks [15] and the handcrafted Boolean benchmarks reported in [61]. We add to these 11 additional examples designed to be difficult GID cases. The collection of regex benchmarks we used (just described) is available on GitHub.[7]

From both the Q2 and Q3 benchmarks, we filter out any benchmark which takes under 10 milliseconds for all of the algorithms to solve (including Naïve), and we use a 60 second timeout. The evaluation was run on a 2020 MacBook Air (MacOS Monterey) with an Apple M1 processor and 8GB of memory.

[7] https://github.com/cdstanford/regex-smt-benchmarks.

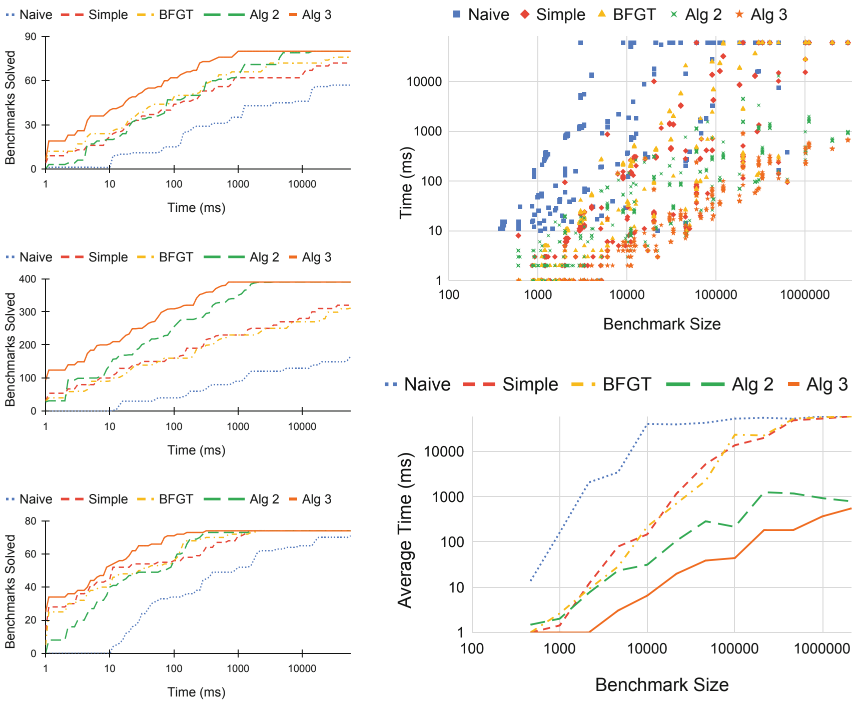


Fig. 5. Evaluation results. *Left:* Cumulative plot showing the number of benchmarks solved in time t or less for basic GID classes (top), randomly generated GIDs (middle), and regex-derived GIDs (bottom). *Top right:* Scatter plot showing the size of each benchmark vs time to solve. *Bottom right:* Average time to solve benchmarks of size closest to s, where values of s are chosen in increments of $1/3$ on a log scale.

Correctness. To ensure that all of our implementations our correct, we invested time into unit testing and checked output correctness on all of our collected benchmarks, including several cases which exposed bugs in previous versions of one or more algorithms. In total, all algorithms are vetted against 25 unit tests from handwritten edge cases that exposed prior bugs, 373 unit tests from benchmarks, and 30 module-level unit tests for specific functions.

Results. Figure 5 shows the results. Algorithm 3 shows significant improvements over the state-of-the-art, solving more benchmarks in a smaller amount of time across basic GIDs, random GIDs, and regex GIDs. Algorithm 2 also shows state-of-the-art performance, similar to BFGT on basic and regex GIDs and significantly better on random GIDs. On the bottom right, since looking at average time is not meaningful for benchmarks of widely varying size, we stratify the size of benchmarks into buckets, and plot time-to-solve as a function of size. Both x-axis and y-axis are on a log scale. The plot shows that Algorithm 3 exhibits up to two orders of magnitude speedup over BFGT for larger GIDs – we see speedups of 110x to 530x for GIDs in the top five size buckets (GIDs of size nearest to 100K, ~200K, ~500K, 1M, and ~2M).

New Implementations of Existing Work. Our implementation contributes, to our knowledge, the first implementation of BFGT specifically for SCC maintenance. In addition, it is one of the first implementations of Euler Tour Trees (see [7] for another), including the AVL tree backing for tours, and likely the first implementation in Rust.

5 Application to Extended Regular Expressions

In this section, we explain how precisely the GID state classification problem arises in the context of derivative-based solvers [45,61]. We first define *extended* regexes [31] (regexes extended with intersection `&` and complement `~`) modulo a symbolic alphabet $\mathcal{A}$ of *predicates* that represent sets of characters. We explain the main idea behind *symbolic derivatives*, as found in [61]; these generalize Brzozowski [18] and Antimirov derivatives [5] (see also [19,42] for other proposals). Symbolic derivatives provide the foundation for incrementally creating a GID. Then we show, through an example, how a solver can incrementally expand derivatives to reduce the satisfiability problem to the GID state classification problem (Definition 2).

Define a *regex* by the following grammar, where $\varphi \in \mathcal{A}$ denotes a predicate:

$$RE ::= \varphi \mid \varepsilon \mid RE_1 \cdot RE_2 \mid RE^* \mid RE_1 \,\texttt{|}\, RE_2 \mid RE_1 \,\texttt{\&}\, RE_2 \mid \texttt{\~{}}RE$$

Let R^k represent the concatenation of R k times. The *symbolic derivative* of a regex R, denoted $\delta(R)$, is a regex which describes the set of *suffixes* of strings in R after the first character is removed. The formal definition can be found in [61] and in the arXiv version of the present paper [60].

To apply Definition 1 to regexes: **states** are regexes; **edges** are transitions from a regex to its derivatives; and **terminal** states are the so-called *nullable* regexes, where a regex is nullable if it matches the empty string. Nullability can be computed inductively over the structure of regexes: for example, ε and R^* are nullable, and $R_1 \,\texttt{\&}\, R_2$ is nullable iff both R_1 and R_2 are nullable. A **live** state here is thus a regex that reaches a nullable regex via 0 or more edges. This implies that there exists a concrete string matching it. Conversely, **dead** states are always empty, i.e. they match no strings, but can reach other dead states, creating strongly connected components of closed states none of which are live. For example, the *false* predicate $\bot$ of $\mathcal{A}$ serves as the regex that matches *nothing* and is trivially a dead state. Thus $\texttt{\~{}}\bot$ is equivalent to $\texttt{.}^*$, where $\texttt{.}$ is the *true* predicate and is trivially a live state.

5.1 Reduction from Incremental Regex Emptiness to GIDs

For simplicity, suppose we want to determine the satisfiability of a single regex constraint $s \in R$, where s is a string variable and R is a concrete regex. (This is not overly restrictive – any number of simultaneous regex constraints for a string s can be combined into single regex constraint by using the Boolean operations of regexes.) For example, let $L = \texttt{\~{}}(\texttt{.}^*\alpha\texttt{.}^{100})$ and $R = L \,\texttt{\&}\, (\texttt{.}\alpha)$, where α is the “is digit” predicate that is true of characters that are digits (often denoted `\d`). The solver manipulates regex membership constraints on strings by unfolding them [61]. The constraint $s \in R$, that essentially tests nonemptiness of R with s as a witness, becomes

$$(s = \epsilon \wedge \mathit{Nullable}(R)) \vee (s \neq \epsilon \wedge s_{1..} \in \delta_{s_0}(R))$$

where, $s \neq \epsilon$ since R is not nullable, $s_{i..}$ is the suffix of s from index i, and

$$\delta(R) = \delta(L) \textcircled{\&} \delta(\mathtt{.}\alpha) = (\alpha \,\mathtt{?}\, L \,\mathtt{\&}\, \mathtt{\sim}(\mathtt{.}^{100}) \,\mathtt{:}\, L) \textcircled{\&} \alpha = (\alpha \,\mathtt{?}\, L \,\mathtt{\&}\, \mathtt{\sim}(\mathtt{.}^{100}) \,\mathtt{\&}\, \alpha \,\mathtt{:}\, L \,\mathtt{\&}\, \alpha)$$

Let $R_1 = L \,\mathtt{\&}\, \mathtt{\sim}(\mathtt{.}^{100}) \,\mathtt{\&}\, \alpha$ and $R_2 = L \,\mathtt{\&}\, \alpha$. So R has two outgoing transitions $R \xrightarrow{\alpha} R_1$ and $R \xrightarrow{\neg\alpha} R_2$ that contribute the edges (R, R_1) and (R, R_2) into the GID. Note that these edges depend only on R and not on s_0.

We continue the search incrementally by checking the two branches of the if-then-else constraint, where R_1 and R_2 are again not nullable (so $s_{1..} \neq \epsilon$):

$$\begin{array}{l}
s_0 \in \alpha \wedge s_{2..} \in \delta_{s_1}(R_1) \quad \vee \quad s_0 \in \neg\alpha \wedge s_{2..} \in \delta_{s_1}(R_2) \\
\delta(R_1) = (\alpha \,\mathtt{?}\, L \,\mathtt{\&}\, \mathtt{\sim}(\mathtt{.}^{100}) \,\mathtt{\&}\, \mathtt{\sim}(\mathtt{.}^{99}) \,\mathtt{:}\, L \,\mathtt{\&}\, \mathtt{\sim}(\mathtt{.}^{99})) \textcircled{\&} (\alpha \,\mathtt{?}\, \varepsilon \,\mathtt{:}\, \bot) = (\alpha \,\mathtt{?}\, \varepsilon \,\mathtt{:}\, \bot) \\
\delta(R_2) = (\alpha \,\mathtt{?}\, L \,\mathtt{\&}\, \mathtt{\sim}(\mathtt{.}^{100}) \,\mathtt{:}\, L) \textcircled{\&} (\alpha \,\mathtt{?}\, \varepsilon \,\mathtt{:}\, \bot) = (\alpha \,\mathtt{?}\, \varepsilon \,\mathtt{:}\, \bot)
\end{array}$$

It follows that $R_1 \xrightarrow{\alpha} \varepsilon$ and $R_2 \xrightarrow{\alpha} \varepsilon$, so the edges (R_1, ε) and (R_2, ε) are added to the GID where ϵ is a trivial terminal state. In fact, after R_1 the search already terminates because we then have the path $(R, R_1)(R_1, \epsilon)$ that implies that R is live. The associated constraints $s_0 \in \alpha$ and $s_1 \in \alpha$ and the final constraint that $s_{2..} = \epsilon$ can be used to extract a concrete witness, e.g., $s =$ ‘‘42".

Soundness of the algorithm follows from that if R is nonempty ($s \in R$ is *satisfiable*), then we eventually arrive at a nullable (terminal) regex, as in the example run above. To achieve *completeness* – and to eliminate dead states as early as possible – we incrementally construct a GID corresponding to the set of regexes seen so far (as above). After all the feasible transitions from R to its derivatives in $\delta(R)$ are added to the GID as edges (WLOG in one batch), the state R becomes closed. *Crucially, due to the* ***symbolic form*** *of* $\delta(R)$*, no derivative is missing.* Therefore R is known to be empty precisely as soon as R is detected as a dead state in the GID. An additional benefit is that the algorithm is independent of the size of the universe of $\mathcal{A}$, that may be very large (e.g. the Unicode character set), or even infinite. We get the following theorem that uses finiteness of the closure of symbolic derivatives [61, Theorem 7.1]:

Theorem 5. *For any regex* R*, (1) If* R *is nonempty, then the decision procedure eventually marks* R *live. (2) If* R *is empty, then the decision procedure marks* R *dead at the earliest stage that it is know to be dead, and terminates.*

6 Related Work

Online Graph Algorithms. Online graph algorithms are typically divided into problems over *incremental* graphs (where edges are added), *decremental* graphs (where edges are deleted), and *dynamic* graphs (where edges are both added and deleted), with core data structures discussed in [27,49]. Important problems include *transitive closure*, *cycle detection*, *topological ordering*, and *strongly connected component (SCC) maintenance*.

For incremental topological ordering, [46] is an early work, and [33] presents two different algorithms, one for *sparse graphs* and one for *dense graphs* – the algorithms are also extended to work with SCCs. The sparse algorithm was subsequently simplified in [10] and is the basis of our implementation named BFGT in Sect. 4. A unified approach of several algorithms based on [10] is presented in [21] that uses a notion of *weak topological order* and a labeling technique that estimates transitive closure size. Further extensions of [10] are studied in [11,14] based on randomization.

For dynamic directed graphs, a topological sorting algorithm that is experimentally preferable for sparse graphs is discussed in [56], and a related article [55] discusses strongly connected components maintenance. Transitive closure for dynamic graphs is studied in [57], improving upon some algorithms presented earlier in [34]. One major application for these algorithms is in pointer analysis [54].

For *undirected* forests, fully dynamic reachability is solvable in amortized logarithmic time per edge via multiple possible approaches [3,30,35,59,64]; our implementation uses Euler Tour Trees [35].

Data Structures for SMT. *UnionFind* [63] is a foundational data structure used in SMT. *E-graphs* [23,67] are used to ensure *functional extensionality*, where two expressions are equivalent if their subexpressions are equivalent [25,52]. In both UnionFind and E-graphs, the maintained relation is an *equivalence* relation. In contrast, maintaining live and dead states involves tracking reachability rather than equivalence. To the best of our knowledge, the specific formulation of incremental reachability we consider here is new.

Dead State Elimination in Automata. A DFA or NFA may be viewed as a GID, so state classification in GIDs solves dead state elimination in DFAs and NFAs, while additionally working in an incremental fashion. Dead state elimination is also known as *trimming* [37] and plays an important role in automata *minimization* [12,38,48]. The literature on minimization is vast, and goes back to the 1950s [16,17,39–41,50,53]; see [65] for a taxonomy, [2] for an experimental comparison, and [22] for the symbolic case. Watson et. al. [66] propose an *incremental* minimization algorithm, in the sense that it can be halted at any point to produce a partially minimized, equivalent DFA; unlike in our setting, the DFA's states and transitions are fixed and read in a predetermined order.

Acknowledgments. We thank the anonymous reviewers of CAV 2021, TACAS 2022, and CAV 2023 for feedback leading to substantial improvements to both our paper and our results. Special thanks to Nikolaj Bjørner, for his collaboration and involvement with Z3, and Yu Chen, for helpful discussions in which he proposed the idea for the first-cut algorithm.

References

1. Abboud, A., Williams, V.V.: Popular conjectures imply strong lower bounds for dynamic problems. In: 2014 IEEE 55th Annual Symposium on Foundations of Computer Science, pp. 434–443. IEEE (2014)
2. Almeida, M., Moreira, N., Reis, R.: On the performance of automata minimization algorithms. Tech. Rep. DCC-2007-03, University of Porto (2007)
3. Alstrup, S., Holm, J., Lichtenberg, K.D., Thorup, M.: Maintaining information in fully dynamic trees with top trees. Acm Trans. Algorithms (talg) **1**(2), 243–264 (2005)
4. Amadini, R.: A survey on string constraint solving. ACM Comput. Surv. (CSUR) **55**(1), 1–38 (2021)
5. Antimirov, V.: Partial derivatives of regular expressions and finite automata constructions. Theoret. Comput. Sci. **155**, 291–319 (1995)
6. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, 30 October - 2 November 2018, pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8602994
7. Bakaric, R.: Euler tour tree representation (GitHub repository) (2019). https://github.com/RobertBakaric/EulerTour
8. Barbosa, H., et al.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
9. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
10. Bender, M.A., Fineman, J.T., Gilbert, S., Tarjan, R.E.: A new approach to incremental cycle detection and related problems. ACM Trans. Algorithms **12**(2), 14:1–14:22 (2015). https://doi.org/10.1145/2756553, https://arxiv.org/abs/1112.0784
11. Bernstein, A., Chechik, S.: Incremental topological sort and cycle detection in o(m sqrt(n)) expected total time. In: Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, pp. 21–34. Society for Industrial and Applied Mathematics (2018)
12. Berstel, J., Boasson, L., Carton, O., Fagnot, I.: Minimization of automata. Handbook of Automata (2011)
13. Berzish, M., et al.: An SMT solver for regular expressions and linear arithmetic over string length. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 289–312. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_14
14. Bhattacharya, S., Kulkarni, J.: An improved algorithm for incremental cycle detection and topological ordering in sparse graphs. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 2509–2521. SIAM (2020)
15. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An SMT-LIB format for sequences and regular expressions. In: SMT workshop, pp. 76–86 (2012), RegExLib benchmarks can be found at https://github.com/cdstanford/regex-smt-benchmarks/, originally downloaded from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/nbjorner-microsoft.automata.smtbenchmarks.zip
16. Blum, N.: An $O(n \log n)$ implementation of the standard method for minimizing n-state finite automata. Inf. Process. Lett. **57**, 65–69 (1996)

17. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Proceedings of the Symposium on Mathematical Theory ofAutomata, New York, pp. 529–561 (1963)
18. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM (JACM) **11**(4), 481–494 (1964)
19. Caron, P., Champarnaud, J.-M., Mignot, L.: Partial derivatives of an extended regular expression. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 179–191. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21254-3_13
20. Clarke, E., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 415–427. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58179-0_72
21. Cohen, E., Fiat, A., Kaplan, H., Roditty, L.: A Labeling Approach to Incremental Cycle Detection. arXiv preprint arXiv:1310.8381 (Oct 2013). https://arxiv.org/abs/1310.8381
22. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: ACM SIGPLAN Notices - POPL 2014, vol. 49(1), pp. 541–553 (2014). https://doi.org/10.1145/2535838.2535849
23. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
24. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011)
25. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM (JACM) **27**(4), 758–771 (1980)
26. Ellul, K., Krawetz, B., Shallit, J., Wang, M.W.: Regular expressions: New results and open problems. J. Autom. Lang. Comb. **10**(4), 407–437 (2005)
27. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. Algorithms Theory Comput. Handbook **1**, 1–9 (1999)
28. Fairbanks, J., Besançon, M., Simon, S., Hoffiman, J., Eubank, N., Karpinski, S.: An optimized graphs package for the Julia programming language (2021). https://github.com/JuliaGraphs/Graphs.jl/, commit 075a01eb6a
29. Fan, W., Hu, C., Tian, C.: Incremental graph computations: Doable and undoable. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 155–169 (2017)
30. Frederickson, G.N.: A data structure for dynamically maintaining rooted trees. J. Algorithms **24**(1), 37–65 (1997). https://arxiv.org/pdf/cs/0310065.pdf
31. Gelade, W., Neven, F.: Succinctness of the complement and intersection of regular expressions. arXiv preprint arXiv:0802.2869 (2008)
32. Guéneau, A., Jourdan, J.H., Charguéraud, A., Pottier, F.: Formal proof and analysis of an incremental cycle detection algorithm. In: Interactive Theorem Proving. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
33. Haeupler, B., Kavitha, T., Mathew, R., Sen, S., Tarjan, R.E.: Incremental cycle detection, topological ordering, and strong component maintenance. ACM Trans. Algorithms **8**(1.3), 1–33 (2012). https://doi.org/10.1145/2071379.2071382
34. Henzinger, M., King, V.: Fully dynamic biconnectivity and transitive closure. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science, pp. 664–672, Milwaukee, WI (1995)
35. Henzinger, M.R., King, V.: Randomized fully dynamic graph algorithms with polylogarithmic time per operation. J. ACM (JACM) **46**(4), 502–516 (1999)

36. Hooimeijer, P., Weimer, W.: Solving string constraints lazily. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 377–386 (2010)
37. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley (1979)
38. Hopcroft, J.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Theory of Machines and Computations: Proceedings of an International Symposium on the Theory of Machines and Computations Held at Technion in Haifa, pp. 189–196. Academic Press, New York (1971)
39. Hopcroft, J.E., Ullman, J.D.: Formal languages and their relation to automata. Addison-Wesley Longman Publishing Co., Inc., Boston (1969)
40. Huffman, D.: The synthesis of sequential switching circuits. J. Franklin Inst. **257**(3–4), 161–190, 275–303 (1954)
41. Kameda, T., Weiner, P.: On the state minimization of nondeterministic finite automata. IEEE Trans. Comput. **C-19**(7), 617–627 (1970)
42. Keil, M., Thiemann, P.: Symbolic solving of extended regular expression inequalities. In: FSTTCS 2014, pp. 175–186. LIPIcs (2014)
43. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Design **19**(3), 291–314 (2001)
44. Kupferman, O., Zuhovitzky, S.: An improved algorithm for the membership problem for extended regular expressions. In: Diks, K., Rytter, W. (eds.) MFCS 2002. LNCS, vol. 2420, pp. 446–458. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45687-2_37
45. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: Lutz, C., Ranise, S. (eds.) FroCoS 2015. LNCS (LNAI), vol. 9322, pp. 135–150. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24246-0_9
46. Marchetti-Spaccamela, A., Nanni, U., Rohnert, H.: Maintaining a topological order under edge insertions. Inf. Process. Lett. **59**(1), 53–58 (1996). https://doi.org/10.1016/0020-0190(96)00075-0
47. Matsakis, N.D., Klock, F.S.: The Rust language. ACM SIGAda Ada Letters **34**(3), 103–104 (2014). https://www.rust-lang.org/
48. Mayr, R., Clemente, L.: Advanced automata minimization. In: POPL 2013, pp. 63–74 (2013)
49. Mehlhorn, K.: Data Structures and Algorithms, Graph Algorithms and NP-Completeness, vol. 2. Springer (1984). https://doi.org/10.1007/978-3-642-69897-2
50. Moore, E.F.: Gedanken-experiments on sequential machines, pp. 129–153. Automata studies, Annals of mathematics studies (1956)
51. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
52. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM (JACM) **27**(2), 356–364 (1980)
53. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
54. Pearce, D.J.: Some directed graph algorithms and their application to pointer analysis. Ph.D. thesis, Imperial College, London (2005)

55. Pearce, D.J., Kelly, P.H.J.: A dynamic algorithm for topologically sorting directed acyclic graphs. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 383–398. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24838-5_29
56. Pearce, D.J., Kelly, P.H.J.: A dynamic topological sort algorithm for directed acyclic graphs. ACM J. Experimental Algorithmics **11**(1.7), 1–24 (2006)
57. Roditty, L., Zwick, U.: Improved dynamic reachability algorithms for directed graphs. SIAM J. Comput. **37**(5), 1455–1471 (2008). https://doi.org/10.1137/060650271
58. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_11
59. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
60. Stanford, C., Veanes, M.: Incremental dead state detection in logarithmic time (extended version for arxiv). arXiv preprint arXiv:2301.05308 (2023)
61. Stanford, C., Veanes, M., Bjørner, N.: Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), pp. 620–635 (2021)
62. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, pp. 1–9 (1973)
63. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. JACM **22**, 215–225 (1975)
64. Tarjan, R.E., Werneck, R.F.: Dynamic trees in practice. J. Exp. Algorithmics (JEA) **14**, 4–5 (2010)
65. Watson, B.W.: A taxonomy of finite automata minimization algorithms. Computing Science Report 93/44, Eindhoven University of Technology (January 1995)
66. Watson, B.W., Daciuk, J.: An efficient incremental DFA minimization algorithm. Nat. Lang. Eng. **9**(1), 49–64 (2003). https://doi.org/10.1017/S1351324903003127
67. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: fast and extensible equality saturation. In: Proceedings of the ACM on Programming Languages 5(POPL), pp. 1–29 (2021)

Model Checking Race-Freedom When "Sequential Consistency for Data-Race-Free Programs" is Guaranteed

Wenhao Wu[1](✉), Jan Hückelheim[2], Paul D. Hovland[2], Ziqing Luo[1], and Stephen F. Siegel[1]

[1] University of Delaware, Newark, DE 19716, USA
`{wuwenhao,ziqing,siegel}@udel.edu`
[2] Argonne National Laboratory, Lemont, IL 60439, USA
`{jhueckelheim,hovland}@anl.gov`

Abstract. Many parallel programming models guarantee that if all sequentially consistent (SC) executions of a program are free of data races, then all executions of the program will appear to be sequentially consistent. This greatly simplifies reasoning about the program, but leaves open the question of how to verify that all SC executions are race-free. In this paper, we show that with a few simple modifications, model checking can be an effective tool for verifying race-freedom. We explore this technique on a suite of C programs parallelized with OpenMP.

Keywords: data race · model checking · OpenMP

1 Introduction

Every multithreaded programming language requires a memory model to specify the values a thread may obtain when reading a variable. The simplest such model is *sequential consistency* [22]. In this model, an execution is an interleaved sequence of the execution steps from each thread. The value read at any point is the last value that was written to the variable in this sequence.

There is no known efficient way to implement a full sequentially consistent model. One reason for this is that many standard compiler optimizations are invalid under this model. Because of this, most multithreaded programming languages (including language extensions) impose a requirement that programs do not have *data races*. A data race occurs when two threads access the same variable without appropriate synchronization, and at least one access is a write. (The notion of appropriate synchronization depends on the specific language.) For data race-free programs, most standard compiler optimizations remain valid. The Pthreads library is a typical example, in that programs with data races

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 265–287, 2023.
https://doi.org/10.1007/978-3-031-37703-7_13

have no defined behavior, but race-free programs are guaranteed to behave in a sequentially consistent manner [25].

Modern languages use more complex "relaxed" memory models. In this model, an execution is not a single sequence, but a set of events together with various relations on those events. These relations—e.g., *sequenced before*, *modification order*, *synchronizes with*, *dependency-ordered before*, *happens before* [21]—must satisfy a set of complex constraints spelled out in the language specification. The complexity of these models is such that only the most sophisticated users can be expected to understand and apply them correctly. Fortunately, these models usually provide an escape, in the form of a substantial and useful language subset which is guaranteed to behave sequentially consistently, as long as the program is race-free. Examples include Java [23], C and C++ since their 2011 versions (see [8] and [21, §5.1.2.4 Note 19]), and OpenMP [26, §1.4.6].

The "guarantee" mentioned above actually consists of two parts: (1) all executions of data race-free programs in the language subset are sequentially consistent, and (2) if a program in the language subset has a data race, then it has a sequentially consistent execution with a data race [8]. Putting these together, we have, for any program P in the language subset:

> (SC4DRF) *If all sequentially consistent executions of P are data race-free, then all executions of P are sequentially consistent.*

The consequence of this is that the programmer need only understand sequentially consistent semantics, both when trying to ensure P is race-free, and when reasoning about other aspects of the correctness of P. This approach provides an effective compromise between usability and efficient implementation.

Still, it is the programmer's responsibility to ensure that all sequentially consistent executions of the program are race-free. Unfortunately, this problem is undecidable [4], so no completely algorithmic solution exists. As a practical matter, detecting and eliminating races is considered one of the most challenging aspects of parallel program development. One source of difficulty is that compilers may "miscompile" racy programs, i.e., translate them in unintuitive, non-semantics-preserving ways [7]. After all, if the source program has a race, the language standard imposes no constraints, so any output from the compiler is technically correct.

Researchers have explored various techniques for race checking. Dynamic analysis tools (e.g., [18]) have experienced the most uptake. These techniques can analyze a single execution precisely, and report whether a race occurred, and sometimes can draw conclusions about closely related executions. But the behavior of many concurrent programs depends on the program input, or on specific thread interleavings, and dynamic techniques cannot explore all possible behaviors. Moreover, dynamic techniques necessarily analyze the behavior of the executable code that results from compilation. As explained above, racy programs may be miscompiled, even possibly removing the race, in which case a dynamic analysis is of limited use.

Approaches based on static analysis, in contrast, have the potential to verify race-freedom. This is extremely challenging, though some promising research prototypes have been developed (e.g., [10]). The most significant limitation is imprecision: a tool may report that race-free code has a possible race— a “false alarm”. Some static approaches are also not sound, i.e., they may fail to detect a race in a racy program; like dynamic tools, these approaches are used more as bug hunters than verifiers.

Finite-state model checking [15] offers an interesting compromise. This approach requires a finite-state model of the program, which is usually achieved by placing small bounds on the number of threads, the size of inputs, or other program parameters. The reachable states of the model can be explored through explicit enumeration or other means. This can be used to implement a sound and precise race analysis of the model. If a race is found, detailed information can be produced, such as a program trace highlighting the two conflicting memory accesses. Of course, if the analysis concludes the model is race-free, it is still possible that a race exists for larger parameter values. In this case, one can increase those values and re-run the analysis until time or computational resources are exhausted. If one accepts the “small scope hypothesis”—the claim that most defects manifest in small configurations of a system—then model checking can at least provide strong evidence for the absence of data races. In any case, the results provide specific information on the scope that is guaranteed to be race-free, which can be used to guide testing or further analysis.

The main limitation of model checking is state explosion, and one of the most effective techniques for limiting state explosion is *partial order reduction* (POR) [17]. A typical POR technique is based on the following observation: from a state s at which a thread t is at a “local” statement—i.e., one which commutes with all statements from other threads—then it is often not necessary to explore all enabled transitions from s; instead, the search can explore only the enabled transitions from t. Usually local statements are those that access only thread-local variables. But if the program is known to be race-free, shared variable accesses can also be considered “local” for POR. This is the essential observation at the heart of recent work on POR in the verification of Pthreads programs [29].

In this paper, we explore a new model checking technique that can be used to verify race-freedom, as well as other correctness properties, for programs in which threads synchronize through locks and barriers. The approach requires two simple modifications to the standard state reachability algorithm. First, each thread maintains a history of the memory locations accessed since its last synchronization operation. These sets are examined for races and emptied at specific synchronization points. Second, a novel POR is used in which only lock (release and acquire) operations are considered non-local. In Sect. 2, we present a precise mathematical formulation of the technique and a theorem that it has the claimed properties, including that it is sound and precise for verification of race-freedom of finite-state models.

Using the CIVL symbolic execution and model checking platform [31], we have implemented a prototype tool, based on the new technique, for verifying race-freedom in C/OpenMP programs. OpenMP is an increasingly popular directive-based language for writing multithreaded programs in C, C++, or Fortran. A large sub-language of OpenMP has the SC4DRF guarantee.[1] While the theoretical model deals with locks and barriers, it can be applied to many OpenMP constructs that can be modeled using those primitives, such as atomic operations and critical sections. This is explained in Sect. 3, along with the results of some experiments applying our tool to a suite of C/OpenMP programs. In Sect. 4, we discuss related work and Sect. 5 concludes.

2 Theory

We begin with a simple mathematical model of a multithreaded program that uses locks and barriers for synchronization.

Definition 1. Let TID be a finite set of positive integers. A *multithreaded program with thread ID set* TID comprises

1. a set Lock of *locks*
2. a set Shared of *shared states*
3. for each $i \in \mathsf{TID}$:
   (a) a set Local_i, the *local states of thread* i, which is the union of five disjoint subsets, $\mathsf{Acquire}_i$, $\mathsf{Release}_i$, $\mathsf{Barrier}_i$, Nsync_i, and Term_i
   (b) a set Stmt_i of *statements*, which includes the *lock statements* $\mathsf{acquire}_i(l)$ and $\mathsf{release}_i(l)$ (for $l \in \mathsf{Lock}$), and the *barrier-exit* statement exit_i; all others statements are known as *nsync (non-synchronization) statements*
   (c) for each $\sigma \in \mathsf{Acquire}_i \cup \mathsf{Release}_i \cup \mathsf{Barrier}_i$, a local state $\mathsf{next}(\sigma) \in \mathsf{Local}_i$
   (d) for each $\sigma \in \mathsf{Acquire}_i \cup \mathsf{Release}_i$, a lock $\mathsf{lock}(\sigma) \in \mathsf{Lock}$
   (e) for each $\sigma \in \mathsf{Nsync}_i$, a nonempty set $\mathsf{stmts}(\sigma) \subseteq \mathsf{Stmt}_i$ of nsync statements and function

$$\mathsf{update}(\sigma) \colon \mathsf{stmts}(\sigma) \times \mathsf{Shared} \to \mathsf{Local}_i \times \mathsf{Shared}.$$

All of the sets Local_i and Stmt_i ($i \in \mathsf{TID}$) are pairwise disjoint. □

Each thread has a unique thread ID number, an element of TID. A local state for thread i encodes the values of all thread-local variables, including the program counter. A shared state encodes the values of all shared variables. (Locks are not considered shared variables.) A thread at an *acquire* state σ is attempting to acquire the lock $\mathsf{lock}(\sigma)$. At a *release* state, the thread is about to release a lock. At a *barrier* state, a thread is waiting inside a barrier. After executing one of the three operations, each thread moves to a unique next local state. A thread that reaches a *terminal* state has terminated. From an *nsync* state, any positive number of statements are enabled, and each of these statements may read and update the local state of the thread and/or the shared state.

[1] Any OpenMP program that does not use non-sequentially consistent atomic directives, `omp_test_lock`, or `omp_test_nest_lock` [26, §1.4.6].

For $i \in \mathsf{TID}$, the *local graph* of thread i is the directed graph with nodes Local_i and an edge $\sigma \to \sigma'$ if either (i) $\sigma \in \mathsf{Acquire}_i \cup \mathsf{Release}_i \cup \mathsf{Barrier}_i$ and $\sigma' = \mathsf{next}(\sigma)$, or (ii) $\sigma \in \mathsf{Nsync}_i$ and there is some $\zeta' \in \mathsf{Shared}$ such that (σ', ζ') is in the image of $\mathsf{update}(\sigma)$.

Fix a multithreaded program P and let

$$\mathsf{LockState} = (\mathsf{Lock} \to \{0\} \cup \mathsf{TID})$$

$$\mathsf{State} = \left(\prod_{i \in \mathsf{TID}} \mathsf{Local}_i \right) \times \mathsf{Shared} \times \mathsf{LockState} \times 2^{\mathsf{TID}}.$$

A *lock state* specifies the owner of each lock. The owner is a thread ID, or 0 if the lock is free. The elements of State are the (global) *states* of P. A state specifies a local state for each thread, a shared state, a lock state, and the set of threads that are currently blocked at a barrier.

Let $i \in \mathsf{TID}$ and $L_i = \mathsf{Local}_i \times \mathsf{Shared} \times \mathsf{LockState} \times 2^{\mathsf{TID}}$. Define

$$\mathsf{enabled}_i \colon L_i \to 2^{\mathsf{Stmt}_i}$$

$$\lambda \mapsto \begin{cases} \{\mathsf{acquire}_i(l)\} & \text{if } \sigma \in \mathsf{Acquire}_i \wedge l = \mathsf{lock}(\sigma) \wedge \theta(l) = 0 \\ \{\mathsf{release}_i(l)\} & \text{if } \mathit{sigma} \in \mathsf{Release}_i \wedge l = \mathsf{lock}(\sigma) \wedge \theta(l) = i \\ \{\mathsf{exit}_i\} & \text{if } \sigma \in \mathsf{Barrier}_i \wedge i \notin w \\ \mathsf{stmts}(\sigma) & \text{if } \sigma \in \mathsf{Nsync}_i \\ \emptyset & \text{otherwise.} \end{cases}$$

where $\lambda = (\sigma, \zeta, \theta, w) \in L_i$. This function returns the set of statements that are enabled in thread i at a given state. This function does not depend on the local states of threads other than i, which is why those are excluded from L_i. An acquire statement is enabled if the lock is free; a release is enabled if the calling thread owns the lock. A barrier exit is enabled if the thread is not currently in the barrier blocked set.

Execution of an enabled statement in thread i updates the state as follows:

$$\mathsf{execute}_i \colon \{(\lambda, t) \in L_i \times \mathsf{Stmt}_i \mid t \in \mathsf{enabled}_i(\lambda)\} \to L_i$$

$$(\lambda, t) \mapsto \begin{cases} (\sigma', \zeta, \theta[l \mapsto i], w') & \text{if } \sigma \in \mathsf{Acquire}_i \wedge t = \mathsf{acquire}_i(l) \wedge \sigma' = \mathsf{next}(\sigma) \\ (\sigma', \zeta, \theta[l \mapsto 0], w') & \text{if } \sigma \in \mathsf{Release}_i \wedge t = \mathsf{release}_i(l) \wedge \sigma' = \mathsf{next}(\sigma) \\ (\sigma', \zeta, \theta, w') & \text{if } \sigma \in \mathsf{Barrier}_i \wedge t = \mathsf{exit}_i \wedge \sigma' = \mathsf{next}(\sigma) \\ (\sigma', \zeta', \theta, w') & \begin{array}{l} \text{if } \sigma \in \mathsf{Nsync}_i \wedge t \in \mathsf{stmts}(\sigma) \wedge \\ \mathsf{update}(\sigma)(t, \zeta) = (\sigma', \zeta') \end{array} \end{cases}$$

where $\lambda = (\sigma, \zeta, \theta, w)$ and in each case above

$$w' = \begin{cases} w \cup \{i\} & \text{if } \sigma' \in \mathsf{Barrier}_i \wedge w \cup \{i\} \neq \mathsf{TID} \\ \emptyset & \text{if } \sigma' \in \mathsf{Barrier}_i \wedge w \cup \{i\} = \mathsf{TID} \\ w & \text{otherwise.} \end{cases}$$

Note a thread arriving at a barrier will have its ID added to the barrier blocked set, unless it is the last thread to arrive, in which case all threads are released from the barrier.

At a given state, the set of enabled statements is the union over all threads of the enabled statements in that thread. Execution of a statement updates the state as above, leaving the local states of other threads untouched:

$$\begin{gathered}\mathsf{enabled}\colon \mathsf{State} \to 2^{\mathsf{Stmt}}\\ s \mapsto \bigcup_{j\in\mathsf{TID}} \mathsf{enabled}_j(\xi_j, \zeta, \theta, w)\\ \mathsf{execute}\colon \{(s,t) \in \mathsf{State} \times \mathsf{Stmt} \mid t \in \mathsf{enabled}(s)\} \to \mathsf{State}\\ (s,t) \mapsto \langle \xi[i \mapsto \sigma], \zeta', \theta', w'\rangle,\end{gathered}$$

where $s = \langle \xi, \zeta, \theta, w\rangle \in \mathsf{State}$, $t \in \mathsf{enabled}(s)$, $i = \mathsf{tid}(t)$, and $\mathsf{execute}_i(\xi_i, \zeta, \theta, w, t) = \langle \sigma, \zeta', \theta', w'\rangle$.

Definition 2. A *transition* is a triple $s \xrightarrow{t} s'$, where $s \in \mathsf{State}$, $t \in \mathsf{enabled}(s)$, and $s' = \mathsf{execute}(s,t)$. An *execution* α of P is a (finite or infinite) chain of transitions $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots$. The *length* of α, denoted $|\alpha|$, is the number of transitions in α. □

Note that an execution is completely determined by its initial state s_0 and its statement sequence $t_1 t_2 \cdots$.

Having specified the semantics of the computational model, we now turn to the concept of the *data race*. The traditional definition requires the notion of "conflicting" accesses: two accesses to the same memory location conflict when at least one is a write. The following abstracts this notion:

Definition 3. A symmetric binary relation $\mathsf{conflict}$ on Stmt is a *conflict relation* for P if the following hold for all $t_1, t_2 \in \mathsf{Stmt}$:

1. if $(t_1, t_2) \in \mathsf{conflict}$ then t_1 and t_2 are nsync statements from different threads
2. if t_1 and t_2 are nsync statements from different threads and $(t_1, t_2) \notin \mathsf{conflict}$, then for all $s \in \mathsf{State}$, if $t_1, t_2 \in \mathsf{enabled}(s)$ then
$$\mathsf{execute}(\mathsf{execute}(s, t_1), t_2) = \mathsf{execute}(\mathsf{execute}(s, t_2), t_1).$$ □

Fix a conflict relation for P for the remainder of this section.

The next ingredient in the definition of *data race* is the *happens-before* relation. This is a relation on the set of *events* generated by an execution. An event is an element of $\mathsf{Event} = \mathsf{Stmt} \times \mathbb{N}$.

Definition 4. Let $\alpha = (s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots)$ be an execution. The *trace of* α is the sequence of events $\mathsf{tr}(\alpha) = \langle t_1, n_1\rangle\langle t_2, n_2\rangle \cdots$, of length $|\alpha|$, where n_i is the number of $j \in [1, i]$ for which $\mathsf{tid}(t_j) = \mathsf{tid}(t_i)$. We write $[\alpha]$ for the set of events occurring in $\mathsf{tr}(\alpha)$. □

A trace labels the statements executed by a thread with consecutive integers starting from 1. Note the cardinality of $[\alpha]$ is $|\alpha|$, as no two events in $\mathsf{tr}(\alpha)$ are equal. Also, $[\alpha]$ is invariant under transposition of two adjacent commuting transitions from different threads.

Given an execution α, the *happens-before relation of* α, denoted $\mathsf{HB}(\alpha)$, is a binary relation on $[\alpha]$. It is the transitive closure of the union of three relations:

1. the intra-thread order relation
$$\{(\langle t_1, n_1\rangle, \langle t_2, n_2\rangle) \in [\alpha] \times [\alpha] \mid \mathsf{tid}(t_1) = \mathsf{tid}(t_2) \wedge n_1 < n_2\}.$$
2. the release-acquire relation. Say $\mathsf{tr}(\alpha) = e_1 e_2 \ldots$ and $e_i = \langle t_i, n_i\rangle$. Then (e_i, e_j) is in the release-acquire relation if there is some $l \in \mathsf{Lock}$ such that all of the following hold: (i) $1 \leq i < j \leq |\alpha|$, (ii) t_i is a release statement on l, (iii) t_j is an acquire statement on l, and (iv) whenever $i < k < j$, t_k is not an acquire statement on l.
3. the barrier relation. For any $e = \langle t, n\rangle \in [\alpha]$, let $i = \mathsf{tid}(t)$ and define
$$\mathsf{epoch}(e) = |\{e' \in [\alpha] \mid e' = \langle \mathsf{exit}_i, j\rangle \text{ for some } j \in [1, n]\}|,$$
the number of barrier exit events in thread i preceding or including e. The barrier relation is
$$\{(e, e') \in [\alpha] \times [\alpha] \mid \mathsf{epoch}(e) < \mathsf{epoch}(e')\}.$$

Two events "race" when they conflict but are not ordered by happens-before:

Definition 5. Let α be an execution and $e, e' \in [\alpha]$. Say $e = \langle t, n\rangle$ and $e' = \langle t', n'\rangle$. We say e and e' *race in* α if $(t, t') \in \mathsf{conflict}$ and neither (e, e') nor (e', e) is in $\mathsf{HB}(\alpha)$. The *data race relation of* α is the symmetric binary relation on $[\alpha]$
$$\mathsf{DR}(\alpha) = \{(e, e') \in [\alpha] \times [\alpha] \mid e \text{ and } e' \text{ race in } \alpha\}. \qquad \square$$

Now we turn to the problem of detecting data races. Our approach is to explore a modified state space. The usual state space is a directed graph with node set State and transitions for edges. We make two modifications. First, we add some "history" to the state. Specifically, each thread records the nsync statements it has executed since its last lock event or barrier exit. This set is checked against those of other threads for conflicts, just before it is emptied after its next lock event or barrier exit. The second change is a reduction: any state that has an enabled statement that is not a lock statement will have outgoing edges from only one thread in the modified graph.

A well-known technical challenge with partial order reduction concerns cycles in the reduced state space. We deal with this challenge by assuming that P comes with some additional information. Specifically, for each i, we are given a set R_i, with $\mathsf{Release}_i \cup \mathsf{Acquire}_i \subseteq R_i \subseteq \mathsf{Local}_i$, satisfying: any cycle in the local graph of thread i has at least one node in R_i. In general, the smaller R_i, the more effective the reduction. In many application domains, there are no cycles in the local graphs, so one can take $R_i = \mathsf{Release}_i \cup \mathsf{Acquire}_i$. For example, standard *for* loops in C, in which the loop variable is incremented by a fixed amount at each iteration, do not introduce cycles, because the loop variable will take on a new value at each iteration. For *while* loops, one may choose one node from the loop body to be in R_i. *Goto* statements may also introduce cycles and could require additions to R_i.

Definition 6. The *race-detecting state graph* for P is the pair $G = (V, E)$, where

$$V = \mathsf{State} \times \Big(\prod_{i \in \mathsf{TID}} 2^{\mathsf{Stmt}_i} \Big)$$

and $E \subseteq V \times \mathsf{Stmt} \times V$ consists of all $(\langle s, \mathbf{a} \rangle, t, \langle s', \mathbf{a}' \rangle)$ such that, letting σ_i be the local state of thread i in s,

1. $s \xrightarrow{t} s'$ is a transition in P
2. $\forall i \in \mathsf{TID}$, $\mathbf{a}'_i = \begin{cases} \mathbf{a}_i \cup \{t\} & \text{if } t \text{ is an nsync statement in thread } i \\ \emptyset & \text{if } t = \mathsf{exit}_0 \text{ or } i = \mathsf{tid}(t) \wedge \sigma_i \in R_i \\ & \text{otherwise} \end{cases}$
3. if there is some $i \in \mathsf{TID}$ such that $\sigma_i \notin R_i$ and thread i has an enabled statement at s, then $\mathsf{tid}(t)$ is the minimal such i. □

The race-detecting state graph may be thought of as a directed graph in which the nodes are V and edges are labeled by statements. Note that at a state where all threads are in the barrier, exit_0 is the only enabled statement in the race-detecting state graph, and its execution results in emptying all the $\mathbf{a}_i$. A lock event in thread i results in emptying $\mathbf{a}_i$ only.

Definition 7. Let P be a multithreaded program and $G = (V, E)$ the race-detecting state graph for P.

1. Let $u = \langle s, \mathbf{a} \rangle \in V$ and $i \in \mathsf{TID}$. We say *thread i detects a race in u* if there exist $j \in \mathsf{TID} \setminus \{i\}$, $t_1 \in \mathbf{a}_i$, and $t_2 \in \mathbf{a}_j$ such that $(t_1, t_2) \in \mathsf{conflict}$.
2. Let $e = v \xrightarrow{t} v' \in E$, $i = \mathsf{tid}(t)$, σ the local state of thread i at v, and σ' the local state of thread i at v'. We say *e detects a race* if either (i) $\sigma \in R_i \setminus \mathsf{Acquire}_i$ and thread i detects a race in v, (ii) $\sigma' \in \mathsf{Acquire}_i$ and thread i detects a race in v', or (ii) $t = \mathsf{exit}_0$ and any thread detects a race in v.
3. We say *G detects a race from u* if E contains an edge that is reachable from u and detects a race, or there is some $v = \langle s, \mathbf{a} \rangle \in V$ that is reachable from u, and $i \in \mathsf{TID}$, such that $\mathsf{enabled}(s) = \emptyset$ and thread i detects a race in v. □

Definition 7 suggests a method for detecting data races in a multithreaded program. The nodes and edges of the race-detecting state graph reachable from an initial node are explored. (The order in which they are explored is irrelevant.) When an edge from a thread at an $R_i \setminus \mathsf{Acquire}_i$ state is executed, the elements of $\mathbf{a}_i$ are compared with those in $\mathbf{a}_j$ for all $j \in \mathsf{TID} \setminus \{i\}$ to see if a conflict exists, and if so, a data race is reported. When an edge in thread i terminates at an $\mathsf{Acquire}_i$ state, a similar race check takes place. When an exit_0 occurs, or a node with no outgoing edges is reached, $\mathbf{a}_i$ and $\mathbf{a}_j$ are compared for all $i, j \in \mathsf{TID}$ with $i \neq j$. This approach is sound and precise in the following sense:

Theorem 1. *Let P be a multithreaded program, and $G = (V, E)$ the race-detecting state graph for P. Let $s_0 \in \mathsf{State}$ and let $u_0 = \langle s_0, \emptyset^{\mathsf{TID}} \rangle \in V$. Assume the set of nodes reachable from u_0 is finite. Then*

1. *P has an execution from s_0 with a data race if, and only if, G detects a race from u_0.*
2. *If there is a data race-free execution of P from s_0 to some state s_f with $\mathsf{enabled}(s_f) = \emptyset$ then there is a path in G from u_0 to a node with state component s_f.*

A proof of Theorem 1 is given in https://arxiv.org/abs/2305.18198.

Example 1. Consider the 2-threaded program represented in pseudocode:

```
$t_1$: acquire($l_1$); x=1; release($l_1$);
$t_2$: acquire($l_2$); x=2; release($l_2$);
```

where l_1 and l_2 are distinct locks. Let $R_i = \mathsf{Release}_i \cup \mathsf{Acquire}_i$ $(i = 1, 2)$. One path in the race-detecting state graph G executes as follows:

```
acquire($l_1$); x=1; release($l_1$); acquire($l_2$); x=2; release($l_2$);.
```

A data race occurs on this path since the two assignments conflict but are not ordered by happens-before. The race is not detected, since at each lock operation, the statement set in the other thread is empty. However, there is another path

```
acquire($l_1$); x=1; acquire($l_2$); x=2; release($l_1$);
```

in G, and on this path the race is detected at the release.

3 Implementation and Evaluation

We implemented a verification tool for C/OpenMP programs using the CIVL symbolic execution and model checking framework. This tool can be used to verify absence of data races within bounds on certain program parameters, such as input sizes and the number of threads. (Bounds are necessary so that the number of states is finite.) The tool accepts a C/OpenMP program and transforms it into CIVL-C, the intermediate verification language of CIVL. The CIVL-C program has a state space similar to the race-detecting state graph described in Sect. 2. The standard CIVL verifier, which uses model checking and symbolic execution techniques, is applied to the transformed code and reports whether the given program has a data race, and, if so, provides precise information on the variable involved in the race and an execution leading to the race.

The approach is based on the theory of Sect. 2, but differs in some implementation details. For example, in the theoretical approach, a thread records the set of non-synchronization statements executed since the thread's last synchronization operation. This data is used only to determine whether a conflict took place between two threads. Any type of data that can answer this question would work equally well. In our implementation, each thread instead records the set of memory locations read, and the set of memory locations modified, since the last synchronization. A conflict occurs if the read or write set of one thread intersects the write set of another read. As CIVL-C provides robust support for tracking memory accesses, this approach is relatively straightforward to implement by a program transformation.

In Sect. 3.1, we summarize the basics of OpenMP. In Sect. 3.2, we provide the necessary background on CIVL-C and the primitives used in the transformation. In Sect. 3.3, we describe the transformation itself. In Sect. 3.4, we report the results of experiments using this tool.

All software and other artifacts necessary to reproduce the experiments, as well as the full results, are included in a VirtualBox virtual machine available at https://doi.org/10.5281/zenodo.7978348.

3.1 Background on OpenMP

OpenMP is a pragma-based language for parallelizing programs written in C, C++ and Fortran [13]. OpenMP was originally designed and is still most commonly used for shared-memory parallelization on CPUs, although the language is evolving and supports an increasing number of parallelization styles and hardware targets. We introduce here the OpenMP features that are currently supported by our implementation in CIVL. An example that uses many of these features is shown in Fig. 1.

The `parallel` construct declares the following structured block as a *parallel region*, which will be executed by all threads concurrently. Within such a parallel region, programmers can use *worksharing* constructs that cause certain parts of the code to be executed only by a subset of threads. Perhaps most importantly, the *loop worksharing construct* can be used inside a parallel region to declare a `omp for` loop whose iterations are mapped to different threads. The mapping of iterations to threads can be controlled through the `schedule` clause, which can take values including `static`, `dynamic`, `guided` along with an integer that defines the *chunk size*. If no schedule is explicitly specified, the OpenMP run time is allowed to use an arbitrary mapping. Furthermore, a structured block within a worksharing loop may be declared as `ordered`, which will cause this block to be executed sequentially in order of the iterations of the worksharing loop. Worksharing for non-iterative workloads is supported through the `sections` construct, which allows the programmer to define a number of different structured blocks of code that will be executed in parallel by different threads.

Programmers may use pragmas and clauses for `barriers`, `atomic` updates, and locks. OpenMP supports named `critical` sections, allowing no more than one thread at a time to enter a critical section with that name, and unnamed critical sections that are associated with the same global mutex. OpenMP also offers `master` and `single` constructs that are executed only by the *master thread* or one arbitrary thread.

```
#pragma omp parallel shared(b) private(i) shared(u,v)
{ // parallel region: all threads will execute this
  #pragma omp sections         // sections worksharing construct
  {
    #pragma omp section        // one thread will do this...
    { b = 0; v = 0; }
    #pragma omp section        // while another thread does this...
    u = rand();
  }
  // loop worksharing construct partitions iterations by schedule.  Each thread has a
  // private copy of b; these are added back to original shared b at end of loop...
  #pragma omp for reduction(+:b) schedule(dynamic,1)
  for (i=0; i<10; i++) {
    b = b + i;
    #pragma omp atomic seq_cst // atomic update to v
    v+=i;
    #pragma omp critical (collatz) // one thread at a time enters critical section
    u = (u%2==0) ? u/2 : 3*u+1;
  }
}
```

Fig. 1. OpenMP Example

Variables are shared by all threads by default. Programmers may change the default, as well as the scope of individual variables, for each parallel region using the following clauses: `private` causes each thread to have its own variable instance, which is uninitialized at the start of the parallel region and separate from the original variable that is visible outside the parallel region. The `firstprivate` scope declares a private variable that is initialized with the value of the original variable, whereas the `lastprivate` scope declares a private variable that is uninitialized, but whose final value is that of the logically last worksharing loop iteration or lexically last section. The `reduction` clause initializes each instance to the neutral element, for example 0 for `reduction(+)`. Instances are combined into the original variable in an implementation-defined order.

CIVL can model OpenMP types and routines to query and control the number of threads (`omp_set_num_threads`, `omp_get_num_threads`), get the current thread ID (`omp_get_thread_num`), interact with locks (`omp_init_lock`, `omp_destroy_lock`, `omp_set_lock`, `omp_unset_lock`, and obtain the current wall clock time (`omp_get_wtime`).

3.2 Background on CIVL-C

The CIVL framework includes a front-end for preprocessing, parsing, and building an AST for a C program. It also provides an API for transforming the AST. We used this API to build a tool which consumes a C/OpenMP program and produces a CIVL-C "model" of the program. The CIVL-C language includes most of sequential C, including functions, recursion, pointers, structs, and dynamically allocated memory. It adds nested function definitions and primitives for concurrency and verification.

In CIVL-C, a thread is created by *spawning* a function: `$spawn f(...);`. There is no special syntax for shared or thread-local variables; any variable that is in scope for two threads is shared. CIVL-C uses an interleaving model of concurrency similar to the formal model of Sect. 2. Simple statements, such as assignments, execute in one atomic step.

Threads can synchronize using *guarded commands*, which have the form `$when (e)`S. The first atomic substatement of S is guaranteed to execute only from a state in which e evaluates to *true*. For example, assume thread IDs are numbered from 0, and a lock value of -1 indicates the lock is free. The *acquire* lock operation may be implemented as `$when (l<0) l=tid;`, where `l` is an integer shared variable and `tid` is the thread ID. A *release* is simply `l=-1;`.

A convenient way to spawn a set of threads is `$parfor (int` i:d)S. This spawns one thread for each element of the 1d-domain d; each thread executes S with i bound to one element of the domain. A 1d-domain is just a set of integers; e.g., if a and b are integer expressions, the domain expression a`..`b represents the set $\{a, a+1, \ldots, b\}$. The thread that invokes the `$parfor` is blocked until all of the spawned threads terminate, at which point the spawned threads are destroyed and the original thread proceeds.

CIVL-C provides primitives to constrain the interleaving semantics of a program. The program state has a single atomic lock, initially free. At any state, if there is a thread t that owns the atomic lock, only t is enabled. When the atomic lock is free, if there is some thread at a `$local_start` statement, and the first statement following `$local_start` is enabled, then among such threads, the thread with lowest ID is the only enabled thread; that thread executes `$local_start` and obtains the lock. When t invokes `$local_end`, t relinquishes the atomic lock. Intuitively, this specifies a block of code to be executed atomically by one thread, and also declares that the block should be treated as a local statement, in the sense that it is not necessary to explore all interleavings from the state where the local is enabled.

Local blocks can also be broken up at specified points using function `$yield`. If t owns the atomic lock and calls `$yield`, then t relinquishes the lock and does not immediately return from the call. When the atomic lock is free, there is no thread at a `$local_start`, a thread t is in a `$yield`, and the first statement following the `$yield` is enabled, then t may return from the `$yield` call and re-obtain the atomic lock. This mechanism can be used to implement the race-detecting state graph: thread i begins with `$local_start`, yields at each R_i node, and ends with `$local_end`.

CIVL's standard library provides a number of additional primitives. For example, the concurrency library provides a barrier implementation through a type `$barrier`, and functions to initialize, destroy, and invoke the barrier.

The *mem* library provides primitives for tracking the sets of memory locations (a variable, an element of an array, field of a struct, etc.) read or modified through a region of code. The type `$mem` is an abstraction representing a set of memory locations, or *mem-set*. The state of a CIVL-C thread includes a stack of mem-sets for writes and a stack for reads. Both stacks are initially empty. The function `$write_set_push` pushes a new empty mem-set onto the write stack. At any point when a memory location is modified, the location is

```
int nthreads = ...;
$mem reads[nthreads], writes[nthreads];
void check_conflict(int i, int j) {
  $assert($mem_disjoint(reads[i], writes[j]) && $mem_disjoint(writes[i], reads[j]) &&
          $mem_disjoint(writes[i], writes[j]));
}
void check_and_clear_all() {
  for (int i=0; i<nthreads; i++)
    for (int j=i+1; j<nthreads; j++) check_conflict(i, j);
  for (int i=0; i<nthreads; i++) reads[i] = writes[i] = $mem_empty();
}
void run(int tid) {
  void pop() { reads[tid]=$read_set_pop(); writes[tid]=$write_set_pop(); }
  void push() { $read_set_push(); $write_set_push(); }
  void check() {
    for (int i=0; i<nthreads; i++) { if (i==tid) continue; check_conflict(tid, i); }
  }
  // local variable declarations
  $local_start(); push(); S pop(); $local_end();
}
for (int i=0; i<nthreads; i++) reads[i] = writes[i] = $mem_empty();
$parfor (int tid:0..nthreads-1) run(tid);
check_and_clear_all();
```

Fig. 2. Translation of `#pragma omp parallel` S

added to the top entry on the write stack. Function `$write_set_pop` pops the write stack, returning the top mem-set. The corresponding functions for the read stack are `$read_set_push` and `$read_set_pop`. The library also provides various operations on mem-sets, such as `$mem_disjoint`, which consumes two mem-sets and returns *true* if the intersection of the two mem-sets is empty.

3.3 Transformation for Data Race Detection

The basic structure for the transformation of a parallel construct is shown in Fig. 2. The user specifies on the command line the default number of threads to use in a parallel region. After this, two shared arrays are allocated, one to record the read set for each thread, and the other the write set. Rather than updating these arrays immediately with each read and write event, a thread updates them only at specific points, in such a way that the shared sets are current whenever a data race check is performed.

The auxiliary function `check_conflict` asserts no read-write or write-write conflict exists between threads i and j. Function `check_and_clear_all` checks that no conflict exists between any two threads and clears the shared mem-sets.

Each thread executes function `run`. A local copy of each private variable is declared (and, for `firstprivate` variables, initialized) here. The body of this function is enclosed in a local region. The thread begins by pushing new entries onto its read and write stacks. As explained in Sect. 3.2, this turns on memory access tracking. The body S is transformed in several ways. First, references to the private variable are replaced by references to the local copy. Other OpenMP constructs are translated as follows.

Lock operations. Several OpenMP operations are modeled using locks. The `omp_set_lock` and `omp_unset_lock` functions are the obvious examples, but we also use locks to model the behavior of atomic and critical section constructs. In any case, a lock acquire operation is translated to

```
pop(); check(); $yield(); acquire(l); push();
```

The thread first pops its stacks, updating its shared mem-sets. At this point, the shared structures are up-to-date, and the thread uses them to check for conflicts with other threads. This conforms with Definition 7(2), that a race check occur upon arrival at an acquire location. It then yields to other threads as it attempts to acquire lock l. Once acquired, it pushes new empty entries onto its stack and resumes tracking. A release statement becomes

```
pop(); $yield(); check(); release(l); push();
```

It is similar to the acquire case, except that the check occurs upon leaving the release location, i.e., after the yield. A similar sequence is inserted in any loop (e.g., a *while* loop or a *for* loop not in standard form) that may create a cycle in the local space, only without the release statement.

Barriers. An explicit or implicit barrier in S becomes

```
pop(); $local_end(); $barrier_call(); if (tid==0) check_and_clear_all();
$barrier_call(); $local_start(); push();.
```

The CIVL-C `$barrier_call` function must be invoked outside of a local region, as it may block. Once all threads are in the barrier, a single thread (0) checks for conflicts and clears all the shared mem-sets. A second barrier call is used to prevent other threads from racing ahead before this check and clear is complete. This protocol mimics the events that take place atomically with an exit_0 transition in Sect. 2.

Atomic and Critical Sections. An OpenMP atomic construct is modeled by introducing a global "atomic lock" which is acquired before executing the atomic statement and then released. The acquire and release actions are then transformed as described above. Similarly, a lock is introduced for each critical section name (and the anonymous critical section); this lock is acquired before entering a critical section with that name and released when departing.

Worksharing Constructs. Upon arriving at a `for` construct, a thread invokes a function that returns the set of iterations for which the thread is responsible. The partitioning of the iteration space among the threads is controlled by the construct clauses and various command line options. If the construct specifies the distribution strategy precisely, then the model uses only that distribution. If the construct does not specify the distribution, then the decisions are based on command line options. One option is to explore all possible distributions. In this case, when the first thread arrives, a series of nondeterministic choices is made to construct an arbitrary distribution. The verifier explores all possible choices, and therefore all possible distributions. This enables a complete analysis of the loop's execution space, but at the expense of a combinatorial explosion with the number of threads or iterations. A different command line option allows the user to specify a particular default distribution strategy, such as *cyclic*. These options give the user some control over the completeness-tractability tradeoff. For `sections`, only cyclic distribution is currently supported, and a `single` construct is executed by the first thread to arrive at the construct.

3.4 Evaluation

We applied our verifier to a suite comprised of benchmarks from DataRaceBench (DRB) version 1.3.2 [35] and some examples written by us that use different concurrency patterns. As a basis for comparison, we applied a state-of-the-art static analyzer for OpenMP race detection, LLOV v.0.3 [10], to the same suite.[2]

LLOV v.0.3 implements two static analyses. The first uses polyhedral analysis to identify data races due to loop-carried dependencies within OpenMP parallel loops [9]. It is unable to identify data races involving critical sections, atomic operations, master or single directives, or barriers. The second is a phase interval analysis to identify statements or basic blocks (and consequently memory accesses within those blocks) that may happen in parallel [10]. Phases are separated by explicit or implicit barriers and the minimum and maximum phase in which a statement or basic block may execute define the phase interval. The phase interval analysis errs in favor of reporting accesses as potentially happening in parallel whenever it cannot prove that they do not; consequently, it may produce false alarms.

The DRB suite exercises a wide array of OpenMP language features. Of the 172 benchmarks, 88 use only the language primitives supported by our CIVL OpenMP transformer (see Sect. 3.1). Some of the main reasons benchmarks were excluded include: use of C++, `simd` and `task` directives, and directives for GPU programming. All 88 programs also use only features supported by LLOV. Of the 88, 47 have data races and 41 are labeled race-free.

We executed CIVL on the 88 programs, with the default number of OpenMP threads for a parallel region bounded by 8 (with a few exceptions, described below). We chose cyclic distribution as the default for OpenMP *for* loops. Many of the programs consume positive integer inputs or have clear hard-coded integer parameters. We manually instrumented 68 of the 88, inserting a few lines of CIVL-C code, protected by a preprocessor macro that is defined only when the program is verified by CIVL. This code allows each parameter to be specified on the CIVL command line, either as a single value or by specifying a range. In a few cases (e.g., DRB055), "magic numbers" such as 500 appear in multiple places,

[2] While there are a number of effective dynamic race detectors, the goal of those tools is to detect races on a particular execution. Our goal is more aligned with that of static analyzers: to cover as many executions as possible, including for different inputs, number of threads, and thread interleavings.

```
// DRB140 (race)
int a, i;
#pragma omp parallel private(i)
{
 #pragma omp master
 a = 0;
 #pragma omp for reduction(+:a)
 for (i=0; i<10; i++)
  a = a + i;
}
```

```
// DRB014 (race)
int n=100, m=100;
double b[n][m];
#pragma omp parallel for \
   private(j)
for (i=1;i<n;i++)
 for (j=0;j<m;j++)
  // out of bound access
  b[i][j]=b[i][j-1];
```

```
// diffusion1 (race)
double *u, *v;
// alloc + init u, v
for (t=0; t<steps; t++) {
 #pragma omp parallel for
 for (i=1; i<n-1; i++) {
  u[i]=v[i]+c*(v[i-1]+v[i]);
 }
 u=v; v=u; // incorrect swap
}
```

Fig. 3. Excerpts from three benchmarks with data races: two from DataRaceBench (left and middle) and erroneous 1d-diffusion (right).

which we replaced with an input parameter controlled by CIVL. These modifications are consistent with the "small scope" approach to verification, which requires some manual effort to properly parameterize the program so that the "scope" can be controlled.

We used the range 1..10 for inputs, again with a few exceptions. In three cases, verification did not complete within 3 min and we lowered these bounds as follows: for DRB043, thread bound 8 and input bound 4; for the Jacobi iteration kernel DRB058, thread bound 4 and bound of 5 on both the matrix size and number of iterations; for DRB062, thread bound 4 and input bound 5.

CIVL correctly identified 40 of the 41 data-race-free programs, failing only on DRB139 due to nested parallel regions. It correctly reported a data race for 45 of the 47 programs with data races, missing only DRB014 (Fig. 3, middle) and DRB015. In both cases, CIVL reports a bound issue for an access to `b[i][j-1]` when $\texttt{i} > 0$ and $\texttt{j} = 0$, but fails to report a data race, even when bound checking is disabled.

LLOV correctly identified 46 of the 47 programs with data races, failing to report a data race for DRB140 (Fig. 3, left). The semantics for reduction specify that the loop behaves as if each thread creates a private copy, initially 0, of the shared variable `a`, and updates this private copy in the loop body. At the end of the loop, the thread adds its local copy onto the original shared variable. These final additions are guaranteed to not race with each other. In CIVL, this is modeled using a lock. However, there is no guarantee that these updates do not race with other code. In this example, thread 0 could be executing the assignment `a=0` while another thread is adding its local result to `a`—a data race. This race issue can be resolved by isolating the reduction loop with barriers.

LLOV correctly identified 38 out of 41 data-race-free programs. It reported false alarms for DRB052 (no support for indirect addressing), DRB054 (failure to propagate array dimensions and loop bounds from a variable assignment), and DRB069 (failure to properly model OpenMP lock behavior).

The DRB suite contains few examples with interesting interleaving dependencies or pointer alias issues. To complement the suite, we wrote 10 additional C/OpenMP programs based on widely-used concurrency patterns (cf. [1]):

– 3 implementations of a synchronization signal sent from one thread to another, using locks or busy-wait loops with critical sections or atomics;

```
// atomic3 (no race)
int x=0, s=0;
#pragma omp parallel sections \
   shared(x,s) num_threads(2)
{
 #pragma omp section
 {
  x=1;
  #pragma omp atomic write seq_cst
  s=1;
 }
 #pragma omp section
 {
  int done = 0;
  while (!done) {
   #pragma omp atomic read seq_cst
   done = s;
  }
  x=2;
 }
}
```

```
// bar2 (no race)
// ...create/initialize locks l0, l1;
#pragma omp parallel num_threads(2)
{
  int tid = omp_get_thread_num();
  if (tid == 0) omp_set_lock(&l0);
  else if (tid == 1) omp_set_lock(&l1);
  #pragma omp barrier
  if (tid == 0) x=0;
  if (tid == 0) {
    omp_unset_lock(&l0);
    omp_set_lock(&l1);
  } else if (tid == 1) {
    omp_set_lock(&l0);
    omp_unset_lock(&l1);
  }
  if (tid == 1) x=1;
  #pragma omp barrier
  if (tid == 0) omp_unset_lock(&l1);
  else if (tid == 1) omp_unset_lock(&l0);
}
```

Fig. 4. Code for synchronization using an atomic variable (left) and a 2-thread barrier using locks (right).

- 3 implementations of a 2-thread barrier, using busy-wait loops or locks;
- 2 implementations of a 1d-diffusion simulation, one in which two copies of the main array are created by two separate malloc calls; one in which they are inside a single malloced object; and
- an instance of a single-producer, single-consumer pattern; and a multiple-producer, multiple-consumer version, both using critical sections.

For each program, we created an erroneous version with a data race, for a total of 20 tests. These codes are included in the experimental archive, and two are excerpted in Fig. 4.

CIVL obtains the expected result in all 20. While we wrote these additional examples to verify that CIVL can reason correctly about programs with complex interleaving semantics or alias issues, for completeness we also evaluated them with LLOV. It should be noted, however, that the authors of LLOV warn that it "...does not provide support for the OpenMP constructs for synchronization..." and "...can produce False Positives for programs with explicit synchronizations with barriers and locks." [9] It is therefore unsurprising that the results were somewhat mixed: LLOV produced no output for 6 of our examples (the racy and race-free versions of diffusion2 and the two producer-consumer codes) and produced the correct answer on 7 of the remaning 14. On these problems, LLOV reported a race for both the racy and race-free version, with the exception of diffusion1 (Fig. 3, right), where a failure to detect the alias between u and v leads it to report both versions as race-free.

CIVL's verification time is significantly longer than LLOV's. On the DRB benchmarks, total CIVL time for the 88 tests was 27 min. Individual times ranged from 1 to 150 seconds: 66 took less than 5s, 80 took less than 30s, and 82 took less than 1 min. (All CIVL runs used an M1 MacBook Pro with 16GB memory.)

Total CIVL runtime on the 20 extra tests was 210s. LLOV analyzes all 88 DRB problems in less than 15 s (on a standard Linux machine).

4 Related Work

By Theorem 1, if barriers are the only form of synchronization used in a program, only a single interleaving will be explored, and this suffices to verify race-freedom or to find all states at the end of each barrier epoch. This is well known in other contexts, such as GPU kernel verification (cf. [5]).

Prior work involving model checking and data races for unstructured concurrency includes Schemmel et al. [29]. This work describes a technique, using symbolic execution and POR, to detect defects in Pthreads programs. The approach involves intricate algorithms for enumerating configurations of prime event structures, each representing a set of executions. The completeness results deal with the detection of defects under the assumption that the program is race-free. While the implementation does check for data races, it is not clear that the theoretical results guarantee a race will be found if one exists.

Earlier work of Elmas et al. describes a sound and precise technique for verifying race-freedom in finite-state lock-based programs [16]. It uses a bespoke POR-based model checking algorithm that associates significant and complex information with the state, including, for each shared memory location, a set of locks a thread should hold when accessing that location, and a reference to the node in the depth first search stack from which the last access to that location was performed.

Both of these model checking approaches are considerably more complex than the approach of this paper. We have defined a simple state-transition system and shown that a program has a data race if and only if a state or edge satisfying a certain condition is reachable in that system. Our approach is agnostic to the choice of algorithm used to check reachability. The earlier approaches are also path-precise for race detection, i.e., for each execution path, a race is detected if and only if one exists on that path. As we saw in the example following Theorem 1, our approach is not path-precise, nor does it have to be: to verify race-freedom, it is only necessary to find one race in one execution, if one exists. This partly explains the relative simplicity of our approach.

A common approach for verifying race-freedom is to establish *consistent correlation*: for each shared memory location, there is some lock that is held whenever that location is accessed. LOCKSMITH [27] is a static analysis tool for multithreaded C programs that takes this approach. The approach should never report that a racy program is race-free, but can generate false alarms, since there are race-free programs that are not consistently correlated. False alarms can also arise from imprecise approximations of the set of shared variables, alias analysis, and so on. Nevertheless, the technique appears very effective in practice.

Static analysis-based race-detection tools for OpenMP include OMPRacer [33]. OMPRacer constructs a static graph representation of the happens-before relation of a program and analyzes this graph, together with a novel whole-program pointer analysis and a lockset analysis, to detect races. It may miss violations as a consequence of unsound decisions that aim to improve performance on real applications. The tool is not open source. The authors subsequently released OpenRace [34], designed to be extensible to other parallelism dialects; similar to OMPRacer, OpenRace may miss violations. Prior papers by the authors present details of static methods for race detection, without a tool that implements these methods [32].

PolyOMP [12] is a static tool that uses a polyhedral model adapted for a subset of OpenMP. Like most polyhedral approaches, it works best for affine loops and is precise in such cases. The tool additionally supports may-write access relations for non-affine loops, but may report false alarms in that case. DRACO [36] also uses a polyhedral model and has similar drawbacks.

Hybrid static and dynamic tools include Dynamatic [14], which is based on LLVM. It combines a static tool that finds candidate races, which are subsequently confirmed with a dynamic tool. Dynamatic may report false alarms and miss violations.

ARCHER [2] is a tool that statically determines many sequential or provably non-racy code sections and excludes them from dynamic analysis, then uses TSan [30] for dynamic race detection. To avoid false alarms, ARCHER also encodes information about OpenMP barriers that are otherwise not understood by TSan. A follow-up paper discusses the use of the OMPT interface to aid dynamic race detection tools in correctly identifying issues in OpenMP programs [28], as well as SWORD [3], a dynamic tool that can stay within user-defined memory bounds when tracking races, by capturing a summary on disk for later analysis.

ROMP [18] is a dynamic/static tool that instruments executables using the DynInst library to add checks for each memory access and uses the OMPT interface at runtime. It claims to support all of OpenMP except `target` and `simd` constructs, and models "logical" races even if they are not triggered because the conflicting accesses happen to be scheduled on the same thread. Other approaches for dynamic race detection and tricks for memory and run-time efficient race bookkeeping during execution are described in [11,19,20,24].

Deductive verification approaches have also been applied to OpenMP programs. An example is [6], which introduces an intermediate parallel language and a specification language based on permission-based separation logic. C programs that use a subset of OpenMP are manually annotated with "iteration contracts" and then automatically translated into the intermediate form and verified using VerCors and Viper. Successfully verified programs are guaranteed to be race-free. While these approaches require more work from the user, they do not require bounding the number of threads or other parameters.

5 Conclusion

In this paper, we introduced a simple model-checking technique to verify that a program is free from data races. The essential ideas are (1) each thread "remembers" the accesses it performed since its last synchronization operation, (2) a partial order reduction scheme is used that treats all memory accesses as local, and (3) checks for conflicting accesses are performed around synchronizations. We proved our technique is sound and precise for finite-state models, using a simple mathematical model for multithreaded programs with locks and barriers. We implemented our technique in a prototype tool based on the CIVL symbolic execution and model checking platform and applied it to a suite of C/OpenMP programs from DataRaceBench. Although based on completely different techniques, our tool achieved performance comparable to that of the state-of-the-art static analysis tool, LLOV v.0.3.

Limitations of our tool include incomplete coverage of the OpenMP specification (e.g., `target`, `simd`, and `task` directives are not supported); the need for some manual instrumentation; the potential for state explosion necessitating small scopes; and a combinatorial explosion in the mappings of threads to loop iterations, OpenMP sections, or single constructs. In the last case, we have compromised soundness by selecting one mapping, but in future work we will explore ways to efficiently cover this space. On the other hand, in contrast to LLOV and because of the reliance on model checking and symbolic execution, we were able to verify the presence or absence of data races even for programs using unstructured synchronization with locks, critical sections, and atomics, including barrier algorithms and producer-consumer code.

Acknowledgements. This material is based upon work by the RAPIDS Institute, supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, under contract DE-AC02-06CH11357 and award DE-SC0021162. Support was also provided by U.S. National Science Foundation awards CCF-1955852 and CCF-2019309.

References

1. Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (2000). https://www.pearson.ch/HigherEducation/Pearson/EAN/9780201357523/Foundations-of-Multithreaded-Parallel-and-Distributed-Programming
2. Atzeni, S., et al.: ARCHER: Effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 53–62 (2016). https://doi.org/10.1109/IPDPS.2016.68
3. Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Laguna, I., Lee, G.L., Ahn, D.H.: SWORD: A bounded memory-overhead detector of OpenMP data races in production runs. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 845–854 (2018). https://doi.org/10.1109/IPDPS.2018.00094
4. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Trans. Electronic Comput. EC **15**(5), 757–763 (1966). https://doi.org/10.1109/PGEC.1966.264565
5. Betts, A., et al.: The design and implementation of a verification technique for GPU kernels. ACM Trans. Program. Lang. Syst. **37**(3) (2015). https://doi.org/10.1145/2743017

6. Blom, S., Darabi, S., Huisman, M., Safari, M.: Correct program parallelisations. Int. J. Softw. Tools Technol. Trans. **23**(5), 741–763 (2021). https://doi.org/10.1007/s10009-020-00601-z
7. Boehm, H.J.: How to miscompile programs with "benign" data races. In: Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism, HotPar 2011, pp. 1–6. USENIX Association, Berkeley, CA, USA (2011). http://dl.acm.org/citation.cfm?id=2001252.2001255
8. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 68–78. PLDI '08, Association for Computing Machinery, New York (2008). https://doi.org/10.1145/1375581.1375591
9. Bora, U., Das, S., Kukreja, P., Joshi, S., Upadrasta, R., Rajopadhye, S.: LLOV: A fast static data-race checker for OpenMP programs. ACM Trans. Archit. Code Optimiz. (TACO) **17**(4), 1–26 (2020). https://doi.org/10.1145/3418597
10. Bora, U., Vaishay, S., Joshi, S., Upadrasta, R.: OpenMP aware MHP analysis for improved static data-race detection. In: 2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). pp. 1–11 (2021). https://doi.org/10.1109/LLVMHPC54804.2021.00006
11. Boushehrinejadmoradi, N., Yoga, A., Nagarakatte, S.: On-the-fly data race detection with the enhanced openmp series-parallel graph. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) IWOMP 2020. LNCS, vol. 12295, pp. 149–164. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_10
12. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for spmd programs and its use in static data race detection. In: Ding, C., Criswell, J., Wu, P. (eds.) LCPC 2016. LNCS, vol. 10136, pp. 106–120. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52709-3_10
13. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998). https://doi.org/10.1109/99.660313
14. Davis, M.J.: Dynamatic: An OpenMP Race Detection Tool Combining Static and Dynamic Analysis. Undergraduate research scholars thesis, Texas A&M University (2021). https://oaktrust.library.tamu.edu/handle/1969.1/194411
15. Edmund M. Clarke, J., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking, 2 edn. MIT press, Cambridge, MA, USA (2018). https://mitpress.mit.edu/books/model-checking-second-edition
16. Elmas, T., Qadeer, S., Tasiran, S.: Precise race detection and efficient model checking using locksets. Tech. Rep. MSR-TR-2005-118, Microsoft Research (2006). https://www.microsoft.com/en-us/research/publication/precise-race-detection-and-efficient-model-checking-using-locksets/
17. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60761-7
18. Gu, Y., Mellor-Crummey, J.: Dynamic data race detection for OpenMP programs. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis (2018). https://doi.org/10.1109/SC.2018.00064
19. Ha, O.-K., Jun, Y.-K.: Efficient thread labeling for on-the-fly race detection of programs with nested parallelism. In: Kim, T., Adeli, H., Kim, H., Kang, H., Kim, K.J., Kiumi, A., Kang, B.-H. (eds.) ASEA 2011. CCIS, vol. 257, pp. 424–436. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-27207-3_47

20. Ha, O.K., Kuh, I.B., Tchamgoue, G.M., Jun, Y.K.: On-the-fly detection of data races in OpenMP programs. In: Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, pp. 1–10. PADTAD 2012, Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2338967.2336808
21. International Organization for Standardization: ISO/IEC 9899:2018. Information technology – Programming languages – C (2018). https://www.iso.org/standard/74528.html
22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **C-28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
23. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 378–391. POPL '05, Association for Computing Machinery, New York (2005). https://doi.org/10.1145/1040305.1040336
24. Mellor-Crummey, J.: On-the-fly detection of data races for programs with nested fork-join parallelism. In: Supercomputing 1991: Proceedings of the 1991 ACM/IEEE Conference On Supercomputing, pp. 24–33. IEEE (1991). https://doi.org/10.1145/125826.125861
25. Open Group: IEEE Std 1003.1: Standard for information technology–Portable Operating System Interface (POSIX(R)) base specifications, issue 7: General concepts: Memory synchronization (2018). https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_12
26. OpenMP Architecture Review Board: OpenMP Application Programming Interface (Nov 2021). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, version 5.2
27. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Practical static race detection for C. ACM Trans. Program. Lang. Syst. **33**, 3:1–3:55 (2011). https://doi.org/10.1145/1889997.1890000
28. Protze, J., Hahnfeld, J., Ahn, D.H., Schulz, M., Müller, M.S.: OpenMP tools interface: synchronization information for data race detection. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 249–265. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_17
29. Schemmel, D., Büning, J., Rodríguez, C., Laprell, D., Wehrle, K.: Symbolic partial-order execution for testing multi-threaded programs. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 376–400. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_18
30. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, pp. 62–71. WBIA 2009. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1791194.1791203
31. Siegel, S.F., et al.: CIVL: The concurrency intermediate verification language. In: SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, New York (Nov 2015). https://doi.org/10.1145/2807591.2807635, article no. 61, pages 1-12
32. Swain, B., Huang, J.: Towards incremental static race detection in OpenMP programs. In: 2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness), pp. 33–41. IEEE (2018). https://doi.org/10.1109/Correctness.2018.00009

33. Swain, B., Li, Y., Liu, P., Laguna, I., Georgakoudis, G., Huang, J.: OMPRacer: A scalable and precise static race detector for OpenMP programs. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14. IEEE (2020). https://doi.org/10.1109/SC41405.2020.00058
34. Swain, B., Liu, B., Liu, P., Li, Y., Crump, A., Khera, R., Huang, J.: OpenRace: An open source framework for statically detecting data races. In: 2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness), pp. 25–32. IEEE (2021). https://doi.org/10.1109/Correctness54621.2021.00009
35. Verma, G., Shi, Y., Liao, C., Chapman, B., Yan, Y.: Enhancing DataRaceBench for evaluating data race detection tools. In: 2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness), pp. 20–30 (2020). https://doi.org/10.1109/Correctness51934.2020.00008
36. Ye, F., Schordan, M., Liao, C., Lin, P.H., Karlin, I., Sarkar, V.: Using polyhedral analysis to verify OpenMP applications are data race free. In: 2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness), pp. 42–50. IEEE (2018). https://doi.org/10.1109/Correctness.2018.00010

Searching for i-Good Lemmas to Accelerate Safety Model Checking

Yechuan Xia[1], Anna Becchi[2], Alessandro Cimatti[2], Alberto Griggio[2],
Jianwen Li[1](✉), and Geguang Pu[1,3](✉)

[1] East China Normal University, Shanghai, China
{jwli,ggpu}@sei.ecnu.edu.cn
[2] Fondazione Bruno Kessler, Trento, Italy
{abecchi,cimatti,griggio}@fbk.eu
[3] Shanghai Trusted Industrial Control Platform Co., Ltd., Shanghai, China

Abstract. IC3/PDR and its variants have been the prominent approaches to safety model checking in recent years. Compared to the previous model-checking algorithms like BMC (Bounded Model Checking) and IMC (Interpolation Model Checking), IC3/PDR is attractive due to its completeness (vs. BMC) and scalability (vs. IMC). IC3/PDR maintains an over-approximate state sequence for proving the correctness. Although the sequence refinement methodology is known to be crucial for performance, the literature lacks a systematic analysis of the problem. We propose an approach based on the definition of *i- good lemmas*, and the introduction of two kinds of heuristics, i.e., branching and refer-skipping, to steer the search towards the construction of *i*-good lemmas. The approach is applicable to IC3 and its variant CAR (Complementary Approximate Reachability), and it is very easy to integrate within existing systems. We implemented the heuristics into two open-source model checkers, IC3Ref and SimpleCAR, as well as into the mature nuXmv platform, and carried out an extensive experimental evaluation on HWMCC benchmarks. The results show that the proposed heuristics can effectively compute more *i*-good lemmas, and thus improve the performance of all the above checkers.

1 Introduction

Safety model checking is a fundamental problem in verification. The goal is to prove that all the reachable states of the transition system $\langle I, T\rangle$ satisfy a property P. The field has been dominated by SAT-based techniques since the introduction of Bounded Model Checking (BMC) [9]. The first wave of SAT-based model-checking algorithms, including BMC, k-induction [31] and Interpolation-based Model Checking [25] have been superseded by the research deriving from the seminal work of Bradley [11]. The IC3 algorithm maintains an over-approximate state sequence for proving the correctness; it avoids unrolling the transition relation by localizing reasoning to *frames*, used to incrementally build an inductive invariant by discovering inductive clauses.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 288–308, 2023.
https://doi.org/10.1007/978-3-031-37703-7_14

IC3 (also known as PDR [17]) has spawned several variants, including those that attempt to combine forward and backward search [29]. Particularly relevant in this paper is CAR (Complementary Approximate Reachability), which combines the forward overapproximation with a backward underapproximation [23].

It has been noted that different ways to refine the over-approximating sequence can impact the performance of the algorithm. For example, [21] attempts to discover *good* lemmas, that can be "pushed to the top" since they are inductive. In this paper, we propose an alternative way to drive the refinement of the over-approximating sequence. We identify *i- good lemmas*, i.e. lemmas that are inductive with respect to the i-th overapproximating level. The intuition is that such i-good lemmas are useful in the search since they are fundamental to reach a fix point in the safe case. In order to guide the search towards the discovery of i-good lemmas, we propose a heuristic approach based on two key insights, i.e., branching and refer-skipping. First, with branching we try to control the way the SAT solver extracts unsatisfiable cores by privileging variables occurring in i-good lemmas. Second, we control lemma generalization by avoiding dropping literals occurring in a subsuming lemma in the previous layer (refer-skipping).

The proposed approach is applicable both to IC3/PDR and CAR, and it is very simple to implement. Yet, it appears to be quite effective in practice. We implemented the i-good lemma heuristics in two open-source implementations of IC3 and CAR, and also in the mature, state-of-the-art IC3 implementation available inside the nuXmv model checker [12], and we carried out an extensive experimental evaluation on Hardware Model Checking Competition (HWMCC) benchmarks. Analysis of the results suggests that increasing the ratio of i-good lemmas leads to an increase in performance, and the heuristics appear to be quite effective in driving the search towards i-good lemmas. In terms of performance, this results in significant improvements for all the tools when equipped with the proposed approach.

This paper is structured as follows. In Sect. 2 we present the problem and the IC3/PDR and CAR algorithms. In Sect. 3 we present the intuition underlying i-good lemmas and the algorithms to find them. In Sect. 4 we overview the related work. In Sect. 5 we present the experimental evaluation. In Sect. 6 we draw some conclusions and present directions for future work.

2 Preliminaries

2.1 Boolean Transition System

A Boolean transition system Sys is a tuple $\langle X, Y, I, T\rangle$, where X and X' denote the set of state variables in the present state and the next state, respectively, and Y denotes the set of input variables. The state space of Sys is the set of possible assignments to X. $I(X)$ is a Boolean formula corresponding to the set of initial states, and $T(X, Y, X')$ is a Boolean formula representing the transition relation. State s_2 is a successor of state s_1 with input y iff $s_1 \wedge y \wedge s_2' \models T$, which is also denoted by $(s_1, y, s_2) \in T$. In the following, we will also write $(s_1, s_2) \in T$ meaning that $(s_1, y, s_2) \in T$ for some assignment y to the input variables. A *path* of length k is a finite state sequence $s_1, s_2, \ldots, s_k$, where $(s_i, s_{i+1}) \in T$ holds for $(1 \leq i \leq k-1)$. A state t is reachable from s in k steps if there is a path of length k from s to t. Let S be a set of states in *Sys*. We overload T and denote the set of successors of states in S as $T(S) = \{t \mid (s,t) \in T, s \in S\}$. Conversely, we define the set of predecessors of states in S as $T^{-1}(S) = \{s \mid (s,t) \in T, t \in S\}$. Recursively, we define $T^0(S) = S$ and $T^{i+1}(S) = T(T^i(S))$ where $i \geq 0$; the notation $T^{-i}(S)$ is defined analogously. In short, $T^i(S)$ denotes the states that are reachable from S in i steps, and $T^{-i}(S)$ denotes the states that can reach S in i steps.

2.2 Safety Checking and Reachability Analysis

Given a transition system $Sys = \langle X, Y, I, T\rangle$ and a safety property P, which is a Boolean formula over X, a model checker either proves that P holds for any state reachable from an initial state in I, or disproves P by producing a *counterexample*. In the former case, we say that the system is *safe*, while in the latter case, it is *unsafe*. A *counterexample* is a finite path from an initial state s to a state t violating P, i.e., $t \in \neg P$, and such a state is called a *bad* state. In symbolic model checking, safety checking is reduced to symbolic reachability analysis. Reachability analysis can be performed in a forward or backward search. Forward search starts from initial states I and searches for bad states by computing $T^i(I)$ with increasing values of i, while backward search begins with states in $\neg P$ and searches for initial states by computing $T^{-i}(\neg P)$ with increasing values of i. Table 1 gives the corresponding formal definitions.

Table 1. Exact reachability analysis.

	Forward	Backward
Base	$F_0 = I$	$B_0 = \neg P$
Induction	$F_{i+1} = T(F_i)$	$B_{i+1} = T^{-1}(B_i)$
Safe Check	$F_{i+1} \subseteq \bigcup_{0\leq j\leq i} F_j$	$B_{i+1} \subseteq \bigcup_{0\leq j\leq i} B_j$
Unsafe Check	$F_i \cap \neg P \neq \emptyset$	$B_i \cap I \neq \emptyset$

For forward search, F_i denotes the set of states that are reachable from I within i steps, which is computed by iteratively applying T. At each iteration, we first compute a new F_i, and then perform safe checking and unsafe checking. If the safe/unsafe checking hits, the search terminates. Intuitively, unsafe checking $F_i \cap \neg P \neq \emptyset$ indicates some bad states are within F_i and safe checking $F_{i+1} \subseteq \bigcup_{0\leq j\leq i} F_j$ indicates that all reachable states from I have been checked and none of them violate P. For backward search, B_i is the set of states that can reach $\neg P$ in i steps, and the search procedure is analogous to the forward one.

Notations. A *literal* is an atomic variable or its negation. If l is a literal, we denote its corresponding variable with $var(l)$. A *cube* (resp. *clause*) is a conjunction (resp. disjunction) of literals. The negation of a clause is a cube and vice versa. A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses. For simplicity, we also treat a CNF formula ϕ as a set of clauses and make no difference between the formula and its set representation. Similarly, a cube or a clause c can be treated as a set of literals or a Boolean formula, depending on the context.

We say a CNF formula ϕ is satisfiable if there exists an assignment of its Boolean variables, called a *model*, that makes ϕ true; otherwise, ϕ is unsatisfiable. A SAT solver is a tool that can decide the satisfiability of a CNF formula ϕ. In addition to providing a yes/no answer, modern SAT solvers can also produce *models* for satisfiable formulas, and *unsatisfiable cores* (UC), i.e. a reason for unsatisfiability, for unsatisfiable ones. More precisely, in the following we shall assume to have a SAT solver that supports the following API (which is standard in state-of-the-art SAT solvers based on the CDCL algorithm [24]):

- **is_SAT**$(\phi, \mathcal{A})$ checks the satisfiability of ϕ under the given assumptions $\mathcal{A}$, which is a list of literals. This is logically equivalent to checking the satisfiability of $\phi \wedge \bigwedge \mathcal{A}$, but is typically more efficient;
- **get_UC**() retrieves an UC of the assumption literals of the previous SAT call when the formula $\phi \wedge \bigwedge \mathcal{A}$ is unsatisfiable. That is, the result is a set $uc \subseteq \mathcal{A}$ such that $\phi \wedge \bigwedge uc$ is unsatisfiable;
- **get_model**() retrieves the model of the formula $\phi \wedge \bigwedge \mathcal{A}$ of the previous SAT call, if the formula is satisfiable.

2.3 Overview of IC3 and CAR

IC3 is a SAT-based and complete safety model checking algorithm proposed in [11], which only needs to unroll the system at most once. PDR [17] is a re-implementation of IC3 which optimizes the original version in different aspects. To prove the correctness of a given system $Sys = \langle X, Y, I, T \rangle$ w.r.t. the safety property P, IC3/PDR maintains a monotone over-approximate state sequence O such that (1) $O_0 = I$ and (2) $O_{i+1} \supseteq O_i \cup T(O_i)$ for $i \geq 0$. From the perspective of reachability analysis, IC3 performs as shown in the left part of Table 2. Since O is monotone, the states search can converge as soon as $O_{i+1} = O_i$ holds for some $i \geq 0$. Otherwise, a state path (counterexample) starting from I to some state in $\neg P$ can be detected ($T^{-i}(\neg P) \cap I \neq \emptyset$).

Table 2. A high-level description of IC3 (left) and (Forward) CAR (right).

	Over-approximate	Under-approximate
Base	$O_0 = I$	-
Induction	$O_{i+1} \supseteq O_i \cup T(O_i)$	-
Safe Check	$\exists i \cdot O_{i+1} = O_i$	-
Unsafe Check	-	$\exists i \cdot T^{-i}(\neg P) \cap I \neq \emptyset$

	Over-approximate	Under-approximate
Base	$O_0 = I$	$U_0 = \neg P$
Induction	$O_{i+1} \supseteq T(O_i)$	$U_{i+1} \subseteq T^{-1}(U_i)$
Safe Check	$\exists i \cdot O_{i+1} \subseteq \bigcup_{0 \leq j \leq i} O_j$	-
Unsafe Check	-	$\exists i \cdot U_i \cap I \neq \emptyset$

CAR [23] is a recently proposed algorithm, which can be considered as a general version of IC3. The main points CAR differs from IC3 are as follows:

Algorithm 1. Overview of IC3

```
 1: procedure IC3($I$, $T$, $P$)
 2:     if is_SAT($I \wedge \neg P$) then                       // unsafe check of initial state
 3:         return unsafe
 4:     $O_0 := I$, $k := 1$, $O_k := \top$
 5:     while true do
 6:         while is_SAT($O_k \wedge \neg P$) do
 7:             $s :=$ get_model()                               // $s \models O_k \wedge \neg P$
 8:             if UnsafeCheck($s$, $k-1$) then
 9:                 return unsafe                                // counterexample found
10:         $k := k+1$, $O_k := \top$
11:         if SafeCheck($k$) then
12:             return safe                                      // property proved
13:
14: function UnsafeCheck($s$, $i$)
15:     while is_SAT($O_i \wedge \neg s \wedge T$, $s'$) do
16:         if $i = 0$ then
17:             return true
18:         $t :=$get_predecessor($s$, $i$)                      // $(t, s) \in T$, see Algorithm 4
19:         if UnsafeCheck($t$, $i-1$) then
20:             return true
21:     $c :=$ generalize($\{l \mid l' \in$ get_UC()$\}$, $i$)    // $c \subseteq s$, see Algorithm 3
22:     $O_j := O_j \cap \neg c, 1 \leq j \leq i+1$
23:     return false
24:
25: function SafeCheck($k$)
26:     propagation($k$)                                         // see Algorithm 4
27:     $i := 0$
28:     while $i < k$ do
29:         if $O_i = O_{i+1}$ then
30:             return true
31:     return false
```

- The over-approximate state sequence O in CAR is not necessarily monotone. Therefore, CAR has to apply the standard invariant-checking approach, i.e., finding a position $i \geq 0$ such that $O_{i+1} \subseteq \bigcup_{0 \leq j \leq i} O_j$ holds, as shown in the right part of Table 2.
- Besides the O sequence, CAR also maintains an under-approximate state sequence U that stores reachable (real) states from $\neg P$, see Table 2. The motivation to introduce the U sequence is to re-use the intermediate states that are computed during proving. Although it is straightforward for IC3 to introduce such a sequence, the effect on the performance remains unknown.
- CAR can be performed in both forward, i.e., proving from I while searching states back from $\neg P$, and backward, i.e., proving back from $\neg P$ while searching states from I. Although Backward CAR is not good at proving, it is advantageous in finding bugs, i.e., checking unsafety [16,22]. Relevant work on reverse IC3/PDR [28], which corresponds to Backward CAR, has been studied but the results did not clearly support its advantage on bug-finding.

An overview of IC3 and (forward) CAR is shown in Algorithm 1 and Algorithm 2 respectively. At a high level, both algorithms have a similar structure, consisting of an alternation of two phases: unsafe check and safe check. The unsafe check (line 14 of Algorithm 1, line 14 of Algorithm 2) tries to find a state sequence that is a path between I and $\neg P$; if such a sequence can be found, then it is a counterexample witnessing the violation of P; otherwise, the O_i are

Algorithm 2. Overview of CAR

```
 1: procedure CAR_FORWARD($I$, $T$, $P$)
 2:     if is_SAT($I \wedge \neg P$) then                    // unsafe check of initial state
 3:         return unsafe
 4:     $O_0 := I$, $U := \{\neg P\}$, $k := 0$
 5:     while true do
 6:         while is_SAT($U$) do
 7:             $s :=$ get_model()                           // $s \in U$
 8:             if UnsafeCheck($s$, $k$) then
 9:                 return unsafe                            // counterexample found
10:         if SafeCheck($k$) then
11:             return safe                                  // property proved
12:         $k := k + 1$, $O_k := \top$
13:
14: function UnsafeCheck($s$, $i$)
15:     while is_SAT($O_i \wedge T$, $s'$) do
16:         if $i = 0$ then
17:             return true
18:         $t :=$ get_predecessor($s$, $i$)                // $(t, s) \in T$, see Algorithm 5
19:         $U := U \cup \{t\}$
20:         if UnsafeCheck($t$, $i - 1$) then
21:             return true
22:     $c :=$ generalize($\{l \mid l' \in$ get_UC()$\}$, $i$)    // $c \subseteq s$, see Algorithm 3
23:     $O_{i+1} := O_{i+1} \cap \neg c$
24:     return false
25:
26: function SafeCheck($k$)
27:     propagation($k$)                                     // see Algorithm 5
28:     $i := 0$
29:     while $i < k$ do
30:         if not is_SAT($O_{i+1} \wedge \neg(\bigvee_{0 \leq j \leq i} O_j)$) then
31:             return true
32:     return false
```

strengthened with additional clauses until O_k is strong enough to imply P.[1] The safe check (line 25 of Algorithm 1, line 26 of Algorithm 2) tries to propagate the clauses in O_i to O_{i+1} and check if a fixpoint is reached. If so then the algorithm terminates. Both algorithms make use of similar additional procedures, which will be detailed in the following section, when we introduce our novel heuristics.

3 Finding *i*-Good Lemmas

In this section, we introduce the concept of *i-good lemmas*, define the heuristics to steer the search towards i-good lemmas and describe the IC3 and CAR algorithms enhanced with i-good lemmas. For the sake of convenient description, we fix the input system $Sys = \langle X, Y, I, T \rangle$ and the property P to be verified. In describing the implementation of our heuristics, we shall necessarily assume that the reader has some familiarity with the low-level details of IC3 and CAR, for which we refer to [11,17,23]. Specifically, we shall use pseudo-code descriptions of the main components of the algorithms (Algorithm 3, 4, and 5), in which the modifications required to implement our heuristics are highlighted in blue.

[1] Note that in the unsafe check, the meaning of the SAT query **is_SAT**($O_i \wedge T$, s') is different between CAR and IC3 (line 15 Algorithm 2) so that when it is unsatisfiable the obtained clauses have different semantics.

3.1 What Are i-good Lemmas

The over-approximate state sequence O in IC3 (resp. CAR) is a finite sequence, in which every element O_i $(0 \leq i < |O|)$, namely *frame* i, is an over-approximation of the states of the system that are reachable in up to (resp. exactly) i steps from I, and which is strong enough to imply P. Such sequence O has the form of $P \wedge C$, where C is a CNF, and each clause in C is called a *lemma*. For both algorithms, the goal is that of transforming the sequence O to construct an over-approximation of all the reachable states of the system (over an unbounded horizon) that still implies P. When this happens, such over-approximation is an inductive invariant that proves P. The key idea, common to both IC3 and to CAR, is to construct the invariant *incrementally* and by reasoning in a *localized manner*, by (i) considering increasingly-long sequences of overapproximations, and by (ii) trying to propagate forward individual lemmas from a frame O_i to its successor O_{i+1}, until a fixpoint is reached[2]. The forward propagation procedure is crucial for ensuring the convergence of the algorithm in practice: for IC3 (resp. CAR), it checks whether a lemma c at frame i represents also an overapproximation of all the states reachable in up to (resp. exactly) $i+1$ steps, and therefore can be added to frame $i+1$. It is immediate to see that the successful propagation of *all* lemmas from i to $i+1$, for some i, is a sufficient condition for the termination of both IC3 and CAR with a safe result. In fact, for IC3, this is also a necessary condition.

We now introduce the notion of *i-good lemma.*

Definition 1 (i-Good Lemma). *Let c be a lemma that was added at frame i by* IC3/CAR *(at some previous step in the execution of the algorithm), i.e. $O_i \models c$. We say that c is i-good if c now holds also at frame $i+1$, i.e. $O_{i+1} \models c$.*

The following theorems are then consequences of the definition.

Theorem 1. IC3 *terminates with safe at frame i $(i > 0)$, if and only if every lemma at frame i is i-good.*

Theorem 2. CAR *terminates with safe at frame i $(i > 0)$, if every lemma at frame i is i-good.*

Such theorems provide the theoretical foundation on which we base our main conjecture: the computation of i-good lemmas can be helpful for both IC3 and CAR to accelerate the convergence in proving properties. Intuitively, an i-good lemma shows the promise of being independent of the reachability layer, and hence holds in general.

[2] The algorithms differ in the way they check reaching the fixpoint, but this difference will be ignored unless otherwise stated.

3.2 Searching for i-good Lemmas

Our conjecture is that there exists, on average, a positive correlation between the ratio of i-good lemmas vs the total amount of lemmas computed by IC3/CAR during generalization and the efficiency of the algorithm.

Ensuring that only i-good lemmas are produced is as hard as solving the verification problem itself, since this is essentially equivalent to synthesizing an inductive invariant which implies P. However, there are two situations in which it is easy to *identify* i-good lemmas, for both IC3 and CAR:

1. In the *propagation* procedure, if a lemma c can be successfully pushed from frame i to frame $i+1$, then c is i-good;
2. In the *generalize* procedure, if the current lemma c at frame i is generalized to a lemma $p \subseteq c$ such that $p \in O_{i-1}$, then p is $(i-1)$-good; additionally, if we can *guide the generalization* of c so that it produces p, then p becomes $(i-1)$-good.

Therefore, we do not attempt to compute only i-good lemmas, but rather, our main idea is to use some (cheap) heuristics to increase the probability of producing i-good lemmas during the normal execution of IC3 and CAR.

We exploit the above observations to design two heuristics that try to bias the search for lemmas towards those that are more likely to be i-good, which we call respectively branching and refer-skipping.

Branching. The branching strategy [26] is an important feature of modern CDCL (Conflict-Driven Clause Learning) SAT solvers [7]. Traditional scoring schemes for branching such as VSIDS and EVSIDS have been extensively evaluated in [10]. In CDCL SAT solvers, decision variables are selected according to their priority. Whenever a conflict occurs, the priority of each variable in the clause is increased. To this end, variables that have recently been involved in conflicts are more likely to be selected as decision variables.

We adopt a similar idea in our branching heuristic for IC3/CAR to bias the unsatisfiable cores produced by the SAT solver, by ordering the assumptions in SAT queries according to their score. This is based on the fact that modern SAT solvers based on CDCL apply the assumption literals in the order given by the user, and (as a consequence of how CDCL works) the unsatisfiable core produced when the formula is unsatisfiable depends on such order, with literals occurring earlier in the assumption list being more likely to be included in the core. For example, assume the SAT query is **is_SAT**$(\neg 1 \wedge (2 \vee \neg 3), 1 \wedge \neg 2 \wedge 3)$, which is unsatisfiable, then the returned UC from the SAT solver, e.g., Minisat [5,18], will be $\{1\}$. If the order of assumptions is changed to $3 \wedge \neg 2 \wedge 1$, then the UC will be $\{3, \neg 2\}$.

Since UCs are the source for lemmas in both IC3 and CAR, the first idea of our branching heuristic is that of sorting the assumption literals in SAT queries according to *how often they occur in recent i- good lemmas*. Concretely, this is implemented as follows:

- We introduce a mapping $S_{[v]} : v \rightarrow score_v, v \in X$ from each variable to its score (priority). Initially, all variables have the same score of 0.

- Before each SAT query in which a (negated) lemma c (or its next-state version c') is part of the assumptions, c is sorted in descending order of $S_{[var(l)]}$, where $l \in c$, to give higher priority to assumption literals with higher scores. This corresponds to the calls to the function SORT(c) in the pseudo-code description of the main components of IC3 and CAR: at the beginning of UNSAFECHECK (Algorithm 1 and 2), in GET_PREDECESSOR (line 6 of Algorithm 4, line 6 of Algorithm 5), and in GENERALIZATION (line 25 of Algorithm 4, line 23 of Algorithm 5).
- Whenever IC3 or CAR discovers an i-good lemma c, all the variables in c are *rewarded* by increasing their score. A lemma c is determined to be i-good either when it is propagated forward from frame i to frame $i+1$ (function PROPAGATION of Algorithm 4 and 5) or when c is the result of a generalization from $d \supseteq c$ at frame $i+1$ such that c is already in frame i (function GENERALIZE, Algorithm 3). In the pseudo-code, the reward steps correspond to the calls to the function REWARD(c) at line 12 of Algorithm 3, line 42 of Algorithm 4, and line 37 of Algorithm 5. The REWARD function first decays the scores of all the variables in $S_{[v]}$ by a small amount (we multiply by 0.99 in our implementation), and then increments the score of all the variables in c (by 1 in our implementation).

  In order to determine whether GENERALIZE produced an i-good lemma, we also use the function GET_PARENTNODE(c) (line 3 of Algorithm 3), which returns a cube p in frame $i-1$ such that $p \subseteq c$ when c belongs to frame i. (If multiple such p exist, the one with the highest score is returned).
- When performing inductive generalization of a lemma c at frame i (Algorithm 3), in which c is strengthened by trying to drop literals from it as long as the result is still a valid lemma for frame i, the literals of c are sorted in increasing order of $S_{[var(l)]}$, with $l \in c$. This corresponds to the call to the function REVERSE_SORT(c) at line 2 of Algorithm 3 in the pseudo-code.

Algorithm 3. Lemma Generalization of IC3/CAR

```
1:  function GENERALIZE($c$, $i$, $rec\_lvl = 1$)
2:      REVERSED_SORT($c$)          // sort literals in c in increasing order of priority
3:      $\neg p$ :=GET_PARENTNODE($\neg c$)          // $\neg p \in F_{i-1}(O_{i-1})$ and $p \subseteq c$
4:      $req := p$          // skip literals in $p$
5:      for each $l \in c$ and $l \notin req$ do
6:          $cm := c \setminus \{l\}$
7:          if DOWN($cm$, $i$, $rec\_lvl$, $req$) then          // CTG-based dropping, see Algorithm 4 and 5
8:              $c := cm$
9:          else
10:             $req := req \cup \{l\}$          // failed to drop $l$
11:     if $c \setminus p = \varnothing$ then          // whether $c$ is a good lemma
12:         REWARD(c)          // raise priority of variables in $c$
13:     return $c$
```

Skipping Literals by Reference. Lemma generalization is a crucial process in IC3/CAR that affects performance significantly. Given the original lemma c

Algorithm 4. Auxiliary functions for IC3

```
function GET_PREDECESSOR($s$, $i$)                          // generalization of predecessors
    ASSERT(is_SAT($O_i \wedge \neg s \wedge T$, $s'$))          // precondition: $\exists t$ that $(t, s) \in T$
    $\mu$ := get_model()
    $in := \{l \in \mu | var(l) \in Y\}$
    $t := \{l \in \mu | var(l) \in X\}$
    SORT($t$)                              // sort literals in s in descending order of priority
    while not is_SAT($O_i \wedge in \wedge \neg s'$, $t$) do
        if $t$ =get_UC() then
            break
        $t$ :=get_UC()
    return $t$

function DOWN($c$, $i$, $rec\_lvl$, $req$)                     // CTG-based dropping literals
    $cex\_num := 0$
    while true do
        if is_SAT($I \wedge c$) then
            return false
        if not is_SAT($O_i \wedge \neg c \wedge T$, $c'$) then
            $c := \{l | l' \in$ get_UC()$\}$
            return true
        else if $rec\_lvl >$ MAX_REC_LVL then                        // MAX_REC_LVL = 3
            return false
        else
            $cex$ := GET_PREDECESSOR($c$, $i$)          // cex as a counter-example of generalization
            SORT($cex$)                        // sort literals in s in descending order of priority
            if $cex\_num <$ MAX_CEX_NUM and $i > 0$ and not is_SAT($O_{i-1} \wedge \neg cex \wedge T$, $cex'$)
               and not is_SAT($I \wedge cex$) then                     // MAX_CEX_NUM = 3
                $c_{cex}$ :=GENERALIZE($\{l | l' \in$ get_UC()$\}$, $i-1, rec\_lvl + 1$)
                $O_k := O_k \cap \neg c_{cex}, 1 \leq k \leq i-1$
                $cex\_num + +$
            else
                $cex\_num := 0$
                if $(c \setminus cex) \cap req \neq \varnothing$ then
                    return false
                $c := c \cap cex$

function PROPAGATION($k$)
    $i := 1$
    for $i < k$ do
        for $\neg c \in O_i$ do
            if not SAT($O_i \wedge \neg c \wedge T, c'$) then
                $O_{i+1} := O_{i+1} \cap \neg c$
                REWARD(c)                         // raise priority of variables in c
```

to be added into frame i ($i > 0$), the GENERALIZE procedure tries to compute a new lemma g such that $g \subseteq c$ and g is also valid to be added to frame i (O_i). The main idea of generalization is to try to drop literals in the original lemma one by one, to see whether the left part can still be a valid lemma.

There are several generalization algorithms with different trade-offs between efficiency (in terms of the number of SAT queries) and effectiveness (in terms of the potential reduction in the size of the generalized lemma), e.g. [11,17,20]. More in general, there might be multiple different ways in which a lemma c can be generalized, with results of uncomparable strength (i.e. there might be both $g_1 \subseteq c$ and $g_2 \subseteq c$ such that $g_1 \not\subseteq g_2$ and $g_2 \not\subseteq g_1$).

The main idea of the refer-skipping heuristic is to bias the generalization to increase the likelihood that the result g is a $(i-1)$-good lemma. Consider the generalization of lemma $c = \neg 1 \vee 2 \vee \neg 3$ at frame i ($i > 1$). If there is already a

Algorithm 5. Auxiliary functions for CAR

```
function GET_PREDECESSOR($s$, $i$)                         // generalization of predecessors
    ASSERT(is_SAT($O_i \wedge T$, $s'$))                   // precondition: $\exists t$ that $(t, s) \in T$
    $\mu$ := get_model()
    $in := \{l \in \mu | var(l) \in Y\}$
    $t := \{l \in \mu | var(l) \in X\}$
    SORT($t$)                          // sort literals in s in descending order of priority
    while not is_SAT($O_i \wedge in \wedge \neg s'$, $t$) do
        if $t$ =get_UC() then
            break
        $t$ :=get_UC()
    return $t$

function DOWN($c$, $i$, $rec\_lvl$)                        // CTG-based dropping literals
    $cex\_num := 0$
    while true do
        if not is_SAT($O_i \wedge T$, $c'$) then
            $c$ :=$\{l | l' \in$ get_UC()$\}$
            return true
        else if $rec\_lvl >$ MAX_REC_LVL then              // MAX_REC_LVL = 3
            return false
        else
            $cex$ := GET_PREDECESSOR($c$, $i$)
            SORT($cex$)                    // sort literals in s in descending order of priority
            if $cex\_num <$ MAX_CEX_NUM and $i > 0$
               and not is_SAT($O_{i-1} \wedge T$, $cex$) then      // MAX_CEX_NUM = 3
                $c_{cex}$ :=GENERALIZE($\{l | l' \in$ get_UC()$\}$, $i-1, rec\_lvl+1$)
                $O_{i-1} := O_{i-1} \cap \neg c_{cex}$
                $cex\_num + +$
            else
                return false

function PROPAGATION($k$)
    $i := 1$
    for $i < k$ do
        for $\neg c \in O_i$ do
            if not SAT($O_i \wedge T, c'$) then
                $O_{i+1} := O_{i+1} \cap \neg c$
                REWARD(c)                              // raise priority of variables in c
```

lemma $g = \neg 1 \vee \neg 3$ at frame $i-1$, we say that g is a *candidate* $(i-1)$*-good lemma* for the generalization of c. In order to drive the generalization of c towards g, we *blacklist* the literals of g, so that GENERALIZE will never attempt to drop them from c. As such, we call g a reference for skipping generalization. In general, there might be multiple references for a given lemma. Currently, our strategy in refer-skipping is to just pick the one first found.

The implementation of refer-skipping is based on existing generalization algorithms and only needs to add less than 10 lines in the pseudo-code (see line 4-10 of Algorithm 3). As shown in the algorithm, a variable set *req* is maintained to store variables that fail to be dropped so that they are not tried to be removed again later. In order to use refer-skipping, we simply initialize *req* with the variables occurring in the candidate $(i-1)$-good lemma that is returned by the GET_PARENTNODE procedure (line 3 of Algorithm 3).

Finally, note that although in our pseudo-code (and in our implementation) we use the CTG algorithm of [20], the idea discussed here can be applied also to the other variants of generalization just as easily.

4 Related Work

In the field of safety model checking, after the introduction of IC3 [11], several variants have been presented: [20] presents the counterexample-guided generalization (CTG) of a lemma by blocking states that interfere with it, which significantly improves the performance of IC3; AVY [33] introduces the ideas of IC3 into IMC (Interpolant Model Checking) [25] to induce a better model checking algorithm; its upgrade version kAVY [32] uses k-induction to guide the interpolation and IC3/PDR generalization inside; [28] proposes to combine IC3/PDR with reverse IC3/PDR; the subsequent work [29] interleaves a forward and a backward execution of IC3 and strengthens one frame sequence by leveraging the proof-obligations from the other; IC3-INN [15] enables IC3 to leverage the internal signal information of the system to induce a variant of IC3 that can perform better on certain industrial benchmarks; [30] introduces under-approximation in PDR to improve the performance of bug-finding.

The importance of discovering inductive lemmas for improving convergence is first noted in [17]. In PDR terminology, inductive lemmas are the ones belonging to frame O_∞, as they represent an over-approximation of all the reachable states.

The most relevant related work is [21], where a variant of IC3 named QUIP is proposed for implementing the pushing of the discovered lemmas to O_∞. At its essence, QUIP adds the negation of a discovered lemma c as a *may*-proof-obligation, hence trying to push c to the next frame. Counterexamples of may-proof-obligations represent an under-approximation of the reachable states and are stored to disprove the inductiveness of other lemmas. In QUIP terminology, such lemmas are classified as *bad lemmas*, as they have no chance of being part of the inductive invariant. Since the pushing is not limited to the current number of frames, inductive lemmas are discovered when all the clauses of a frame can be pushed ($O_k \setminus O_{k+1} = \emptyset$ for a level k), and then added in O_∞. In QUIP terminology, lemmas belonging to O_∞ are classified as *good lemmas*, and are always kept during the algorithm. Observe that the concept of *good* lemma in [21] is a stronger version of Definition 1, which instead is *local* to a frame i and characterizes lemmas that can be propagated one frame ahead.

Both QUIP and our heuristic try to accomplish a similar task, which is prioritizing the use of already discovered lemmas during the generalization. There are however several differences: QUIP proceeds by adding additional proof-obligations to the queue and by progressively proving the inductiveness of a lemma relative to any frame. Our approach, on the other hand, is based on a cheap heuristic strategy that *locally* guides the generalization prioritizing the locally good lemmas. Some i-good lemmas computed may not be part of the final invariant and can not be pushed later; in QUIP, such lemmas would not be considered good. In our view, pushing them is not necessarily a waste of effort, because they still strengthen the frames and their presence might be necessary to deduce the final invariant. Finally, it is worth mentioning that our heuristic is much simpler to implement and integrate into different PDR-based engines.

The idea of ordering literals when performing inductive generalization is already proposed in [11] and adopted, as a default strategy, in several implementations of IC3 [3,17,19], yielding modest improvements on HWMCC benchmarks, however without clear trends identified (see [17,19]). Compared to such works, our approach has two main differences. First, these heuristics favor literals occurring more frequently in all previous frames, whereas our approach is driven by the role of lemmas and prefers the variables occurring only in those are i-good. Second, our use of ordering heuristics is more pervasive: unlike in previous works, where variable ordering heuristics are only used during the lemma generalization, we use ordering everywhere the SAT results affect search direction, which makes it more effective to bias the search.

5 Evaluation

5.1 Experimental Setup

We integrated the branching and refer-skipping heuristics into three systems: the IC3Ref [3] and SimpleCAR [6] (open-source) model checkers, which implement the IC3 and (Forward and Backward[3]) CAR algorithms respectively, and the mature, state-of-the-art implementation of IC3 available inside the nuXmv model checker [12]. We make our implementations and data for reproducing the experiments available at https://github.com/youyusama/i-Good_Lemmas_MC.

Since our approach is related to QUIP [21], we include the evaluation of QUIP, and IC3 (mainly as the baseline for QUIP), as implemented[4] in IIMC [4]. We also consider the PDR implementation in the ABC model checker [1], which is state-of-the-art in hardware model checking.

Table 3. Tools and algorithms evaluated in the experiments.

Tools	Algorithms	Available Flags
IC3Ref [3]	IC3 (ic3)	-br | -rs | -sh
SimpleCAR [6]	Forward CAR (fcar)	-br | -rs | -sh
nuXmv [12]	IC3 (nuXmv)	-br | -rs | -sh
IIMC [4]	QUIP (iimc-quip)	–
IIMC [4]	IC3 (iimc-ic3)	–
ABC [1]	PDR (abc-pdr)	–

Table 3 summarizes the tested tools, algorithms, and their flags. We use the flag "-br" to enable the branching heuristic and "-rs" to enable refer-skipping. Furthermore, we evaluate also another configuration (denoted as "-sh"), in which the calls to SORT() functions in Algorithms 4 and 5 are replaced by random

[3] Although there is an implementation of Backward CAR in SimpleCAR, this methodology corresponds to reverse IC3. As a result, we did not include Backward CAR in this paper and left the evaluation in future work.

[4] As far as we know, this is the only publicly available QUIP implementation.

shuffles, thus simulating a strategy that orders variables randomly. When no flag is active, IC3Ref runs the instances with its own strategy of sorting variables, present in the original implementation.

We evaluate all the tools on 749 benchmarks, in *aiger* format, of the SINGLE safety property track of the 2015 and 2017 editions of HWMCC [8][5]. We ran the experiments on a cluster, which consists of 2304 2.5GHz CPUs in 240 nodes running RedHat 4.8.5 with a total of 96GB RAM. For each test, we set the memory limit to 8GB and the time limit to 5 h. During the experiments, each model-checking run has exclusive access to a dedicated node.

To increase our confidence in the correctness of the results, we compare the results of the solvers to make sure they are all consistent (modulo timeouts). For the cases with unsafe results, we also check the provided counterexample with the *aigsim* tool from the Aiger package [2]. We have no discrepancies in the results, and all unsafe cases successfully pass the *aigsim* check.

5.2 Experimental Results

Overview. The results of the experimental evaluation are discussed below. We first consider the aggregated results, as reported in Table 4. For each tool, we group the results obtained with the various configurations; we report the total number of benchmarks solved, distinguishing between safe and unsafe benchmarks; we also report the benchmarks gained and lost by the configurations with branching and/or refer-skipping active, relative to the baseline where branching and refer-skipping are not active. We can draw the following conclusions.

- The proposed heuristics are in general effective in improving performance. Each of the model checkers, with at least one of branching and refer-skipping active, consistently outperforms the respective baseline in terms of the number of benchmarks solved.
- The same holds within the safe instances, with the exception of refer-skipping in nuXmv that solves two safe benchmarks less than the baseline.
- The heuristics also yield a uniform improvement over the baseline in the unsafe instances.
- The combination of branching and refer-skipping gives further improvements over a single technique, with the exception of nuXmv with branching, which cumulatively solves 5 more benchmarks than nuXmv with branching and refer-skipping.
- The gain is not uniform across the instances. For example, nuXmv with branching gains 52 benchmarks (44 safe and 8 unsafe) that are not solved by nuXmv baseline, while losing 13 (safe) benchmarks. This level of variability can be expected, given a heuristic approach, but further investigation is needed to assess the underlying phenomena.

[5] From HWMCC 2019, the official format used in the competition is switched from Aiger to Btor2 [27], a format for word-level model checking. As a result, we did not include those instances in our experiments.

- The performance of using the heuristics guided by random variable ordering does not differ significantly from the baseline in terms of aggregate results. There are some differences (as expected) at the level of individual instances, especially for CAR, but no clear trend emerges overall.
- The comparison also shows that the considered systems compare well against the state-of-the-art system ABC, and QUIP; QUIP turns out to be quite inefficient and is disregarded in the following. Note that the original implementation of QUIP is not available; the fact that the available version of QUIP implemented on top of IIMC does not seem to achieve the same improvements reported in the original paper [21] (the code for which is unfortunately not available) suggests that the QUIP is far from trivial to implement. As the reference, QUIP performs even worse than the IC3 implementation in IIMC, whose performance is similar to the IC3Ref baseline, see Table 4.

Table 4. Summary of overall results among different configurations.

Configuration	#Solved	#Safe	#Unsafe	Gained(safe/unsafe)	Lost(safe/unsafe)
ic3 -br -rs	*439*	*313*	*126*	25(18/7)	6(4/2)
ic3 -br	428	302	*126*	22(15/7)	14(12/2)
ic3 -rs	430	308	122	21(17/4)	11(8/3)
ic3 -sh	420	299	121	–	–
ic3	417	297	120	9(7/2)	12(9/3)
fcar -br -rs	*444*	*319*	*125*	54(43/11)	1(0/1)
fcar -br	429	308	121	43(33/10)	5(1/4)
fcar -rs	410	295	115	23(22/1)	4(3/1)
fcar -sh	394	277	117	31(22/9)	28(21/7)
fcar	391	276	115	–	–
nuXmv -br -rs	497	353	*144*	49(39/10)	15(15/0)
nuXmv -br	*502*	*360*	142	52(44/8)	13(13/0)
nuXmv -rs	473	333	140	26(19/7)	16(15/1)
nuXmv -sh	464	327	137	7(4/3)	6(6/0)
nuXmv	463	329	134	–	–
abc-pdr	430	315	115	–	–
iimc-ic3	418	307	111	–	–
iimc-quip	377	281	96	–	–

Similar insights can be obtained from Fig. 1, which clearly shows the positive effect of improvements in performance.

Detailed Statistics. As shown in Table 4 and Fig. 1, nuXmv is highly optimized and has a much better performance than other open-source IC3 implementations, but enabling both heuristics is still useful to improve its overall performance by solving 34 more instances. For IC3Ref and SimpleCAR, the increased numbers of solved cases are 19 and 53, respectively. Moreover, from Table 4, nuXmv/IC3Ref/SimpleCAR is able to solve 24/14/43 more safe and 10/5/10 more unsafe instances with both heuristics.

A comparison of the performance of the tools with and without the heuristics is shown in Fig. 2. All three solvers are able to reduce their time cost when equipping with branching and refer-skipping (see the last row of the figure). Explicitly, 67.8% of the instances cost less or equal to check by ‘nuXmv -br -rs’, and the corresponding portions for ‘ic3 -br -rs’ and ‘fcar -br -rs’ are 77.9% and 87.0%. The variability occurs when considering only a single heuristic, which needs to be explored in the future. For example, ‘fcar -br’ and ‘nuXmv -rs’ generally cost slightly more time than ‘fcar’ and ‘nuXmv’, respectively.

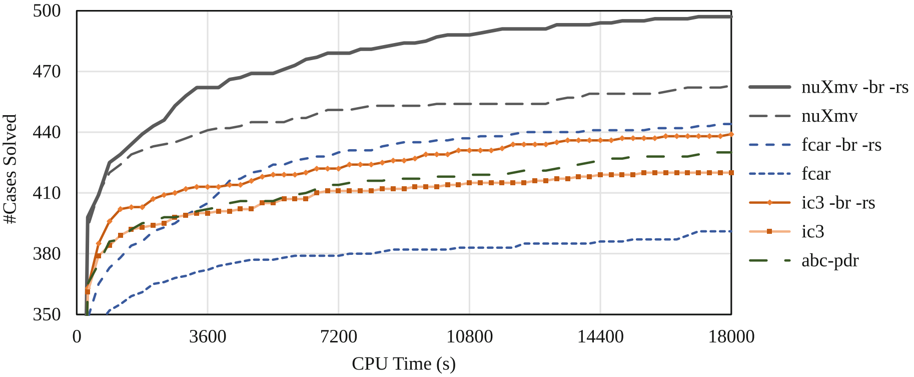


Fig. 1. Comparisons among the implementations of IC3, PDR and CAR under different configurations. (To make the figure more readable, we skip the results with a single heuristic, which are still shown in Table 4.)

According to Table 4, either branching or refer-skipping is effective for improving nuXmv, IC3Ref, and SimpleCAR. For nuXmv and SimpleCAR, branching is more useful, considering that ‘nuXmv -br’ (resp. ‘fcar -br’) solves 39 (resp. 38) more instances than ‘nuXmv’ (resp. ‘fcar’), with 31 (resp. 32) safe and 8 (resp. 6) unsafe. For IC3Ref, the improvement with either heuristic seems relatively modest, i.e., ‘ic3 -br’ solves 8 more instances than ‘ic3’, with 3 safe and 5 unsafe, while ‘ic3 -rs’ solves 10 more instances than ‘ic3’, with 9 safe and 1 unsafe.

As listed above, ‘ic3 -br -rs’ loses only 6 instances that are solved by ‘ic3’, while ‘fcar -br -rs’ even loses only 1 instance that is solved by ‘fcar’, which indicates the performance domination of ‘fcar -br -rs’ over ‘fcar’. For ‘nuXmv -br -rs’, the number of lost cases is 15, which is still modest when compared to the gain of 49. So enabling branching and refer-skipping together makes the checkers pay a limited cost. The same applies to the situations when equipping with only one single heuristic for the checkers, see Table 4.

5.3 Why Do branching and refer-skipping Work?

To measure why branching and refer-skipping work, we introduce sr, i.e. the **s**uccess **r**ate in computing i-good lemmas. Formally, $sr = N_g/N$ where N_g is

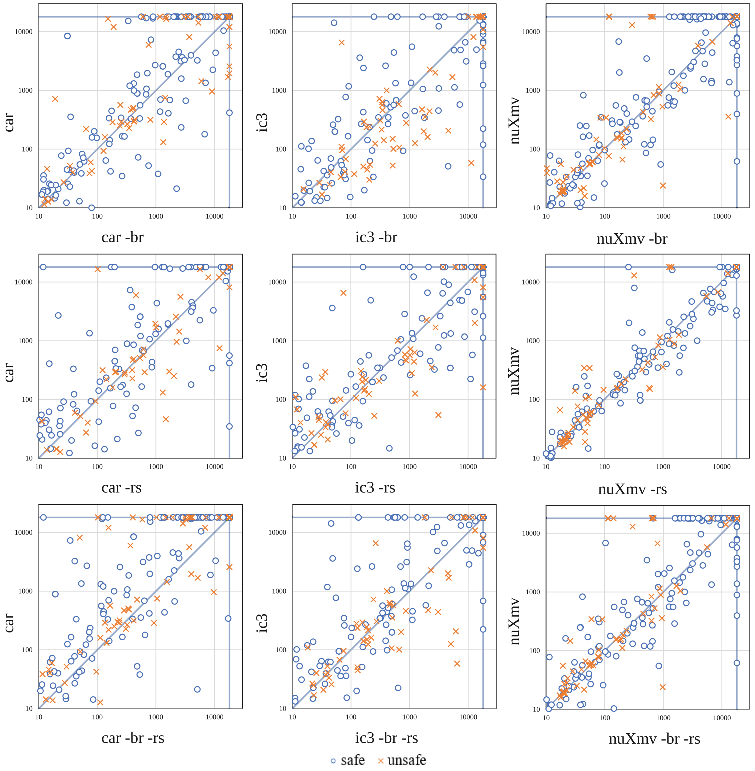


Fig. 2. Time comparison between IC3/CAR with and without two heuristics on safe-unsafe cases. The baseline is always on the y-axis. Points above the diagonal indicate better performance with the heuristics active. Points on the borders indicate timeouts (18000 s).

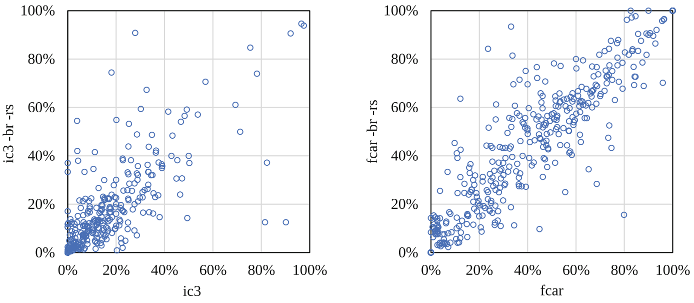


Fig. 3. Comparison on the success rate (sr) to compute i-good lemmas between IC3/CAR with and without branching and refer-skipping.

the number of generalizations that successfully return i-good lemmas, while N is the total number of generalization calls. We instrumented the two open-source checkers IC3Ref and SimpleCAR in order to compute sr for each terminating run (including each run with/without a returned result at timeout).

- Consider the results presented in Fig. 3. The figure shows the comparison of the success rate in computing i-good lemmas between IC3/CAR with and without the heuristics. 'ic3 -br -rs' computes more i-good lemmas than 'ic3' on 54% tested instances, while 'fcar -br -rs' computes more i-good lemmas than 'fcar' on 67% tested instances, the portion of which is even higher. This supports the conjecture that enabling branching and refer-skipping makes IC3/CAR compute more i-good lemmas.
- Now consider Fig. 4. The figure shows the comparison between the deviation of success rate to compute i-good lemmas (Y axis) and the deviation of checking (CPU) time (X axis) for IC3/CAR with and without the heuristics. The meaning of each point in the plot is explained in the title of the figure. In general, the more points located in the first quadrant, the better our claim can be supported.

  Clearly, the plot for both IC3 and CAR in Fig. 4 supports the conjecture that searching more i-good lemmas can help achieve better model-checking performance (time cost).

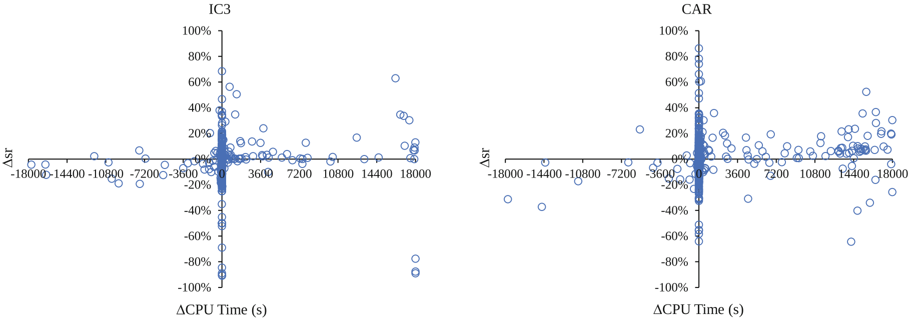


Fig. 4. Comparison between the deviation of the success rate (sr) to compute i-good lemmas (Y axis) and the deviation of checking (CPU) time (X axis) for IC3/CAR with and without the heuristics. For each instance, let the checking time of 'ic3'/'fcar' be t and that of 'ic3 -br -rs'/'fcar -br -rs' be t'. Each point has $t - t'$ as the x value and $sr' - sr$ as the y value.

Finally, we argue that computing as many i-good lemmas as possible is the direction to take to improve the performance of IC3 and its variants. branching and refer-skipping are two heuristics that can enable IC3/CAR to compute more i-good lemmas. However, there can be more efficient ways to compute i-good lemmas, which is left for our future work.

6 Conclusions and Future Work

In this paper, we proposed a heuristic-based approach to improve the performance of IC3-based safety model checking. The idea is to steer the search of the over-approximation sequence towards i-good lemmas, i.e. lemmas that can be pushed from frame i to frame $i + 1$. On the one side, we attempt to control the way the SAT solver extracts the unsat cores, by privileging variables occurring in i-good lemmas (branching); on the other, we control lemma generalization by avoiding dropping literals that occur in a subsuming lemma in the previous layer (refer-skipping). The approach is very simple to implement and has been integrated into two open-source model checkers and an industrial-strength, closed-source model checker. The experimental evaluation, carried out on a wide set of benchmarks, shows that the approach yields computational benefits on all the implementations. Further analysis shows a correlation between i-good lemmas and performance improvements and suggests that the proposed heuristics are effective in finding more i-good lemmas.

In the future, we plan to investigate the reasons for performance improvement/degradation at the level of the single benchmarks. We will also attempt to integrate the proposed ideas with the ideas in QUIP, explore different kinds of heuristics, and lift this approach to the safety checking of infinite-state systems [13,14].

Acknowledgment. We thank anonymous reviewers for their helpful comments. This work is supported by National Natural Science Foundation of China (Grant #U21B2015 and #62002118) and Shanghai Collaborative Innovation Center of Trusted Industry Internet Software. This work has been partly supported by the project "AI@TN" funded by the Autonomous Province of Trento and by the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU.

References

1. ABC. https://github.com/berkeley-abc/abc
2. AIGER Tools. http://fmv.jku.at/aiger/aiger-1.9.9.tar.gz
3. IC3Ref. https://github.com/arbrad/IC3ref
4. IIMC-QUIP. https://github.com/ryanberryhill/iimc
5. Minisat 2.2.0. https://github.com/niklasso/minisat
6. SimpleCAR. https://github.com/lijwen2748/simplecar/releases/tag/v0.1
7. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M.: Proceedings of sat competition 2022: Solver and benchmark descriptions. Department of Computer Science Series of Publications B, vol. B-2022-1. http://hdl.handle.net/10138/347211
8. Biere, A.: AIGER Format. http://fmv.jku.at/aiger/FORMAT
9. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
10. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 405–422. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_29

11. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
12. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
13. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 46–61. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_4
14. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools. CoRR abs/ arXiv: 2109.12821 (2021)
15. Dureja, R., Gurfinkel, A., Ivrii, A., Vizel, Y.: Ic3 with internal signals. In: 2021 Formal Methods in Computer Aided Design (FMCAD), pp. 63–71 (2021)
16. Dureja, R., Li, J., Pu, G., Vardi, M.Y., Rozier, K.Y.: Intersection and rotation of assumption literals boosts bug-finding. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 180–192. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_12
17. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD 2011, pp. 125–134. FMCAD Inc., Austin, Texas (2011)
18. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
19. Griggio, A., Roveri, M.: Comparing different variants of the ic3 algorithm for hardware model checking. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(6), 1026–1039 (2015)
20. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in ic3. In: 2013 Formal Methods in Computer-Aided Design, pp. 157–164. IEEE (2013)
21. Ivrii, A., Gurfinkel, A.: Pushing to the top. In: Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design, FMCAD 2015, pp. 65–72. FMCAD Inc., Austin, Texas (2015)
22. Li, J., Dureja, R., Pu, G., Rozier, K.Y., Vardi, M.Y.: SimpleCAR: an efficient bug-finding tool based on approximate reachability. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 37–44. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_5
23. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: Proceedings of the 36th International Conference on Computer-Aided Design, ICCAD 2017, pp. 95–100. IEEE Press (2017)
24. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. In: Handbook of satisfiability, vol. 185 (2009)
25. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
26. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th annual Design Automation Conference, pp. 530–535 (2001)
27. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 587–595. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_32

28. Seufert, T., Scholl, C.: Combining pdr and reverse pdr for hardware model checking. In: 2018 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 49–54 (2018)
29. Seufert, T., Scholl, C.: fbpdr: In-depth combination of forward and backward analysis in property directed reachability. In: Teich, J., Fummi, F. (eds.) Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, 25–29 March 2019, pp. 456–461. IEEE (2019)
30. Seufert, T., Scholl, C., Chandrasekharan, A., Reimer, S., Welp, T.: Making progress in property directed reachability. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 355–377. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_18
31. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
32. Vediramana Krishnan, H.G., Vizel, Y., Ganesh, V., Gurfinkel, A.: Interpolating strong induction. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 367–385. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_21
33. Vizel, Y., Gurfinkel, A.: Interpolating property directed reachability. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 260–276. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_17

Second-Order Hyperproperties

Raven Beutner, Bernd Finkbeiner, Hadar Frenkel(✉),
and Niklas Metzger

CISPA Helmholtz Center for Information Security,
Saarbrücken, Germany
`{raven.beutner,finkbeiner,hadar.frenkel,`
`niklas.metzger}@cispa.de`

Abstract. We introduce Hyper^{2}LTL, a temporal logic for the specification of hyperproperties that allows for second-order quantification over sets of traces. Unlike first-order temporal logics for hyperproperties, such as HyperLTL, Hyper^{2}LTL can express complex epistemic properties like common knowledge, Mazurkiewicz trace theory, and asynchronous hyperproperties. The model checking problem of Hyper^{2}LTL is, in general, undecidable. For the expressive fragment where second-order quantification is restricted to smallest and largest sets, we present an approximate model-checking algorithm that computes increasingly precise under- and overapproximations of the quantified sets, based on fixpoint iteration and automata learning. We report on encouraging experimental results with our model-checking algorithm, which we implemented in the tool `HySO`.

1 Introduction

About a decade ago, Clarkson and Schneider coined the term *hyperproperties* [21] for the rich class of system requirements that relate multiple computations. In their definition, hyperproperties generalize trace properties, which are sets of traces, to *sets of* sets of traces. This covers a wide range of requirements, from information-flow security policies to epistemic properties describing the knowledge of agents in a distributed system. Missing from Clarkson and Schneider's original theory was, however, a concrete specification language that could express customized hyperproperties for specific applications and serve as the common semantic foundation for different verification methods.

A first milestone towards such a language was the introduction of the temporal logic HyperLTL [20]. HyperLTL extends linear-time temporal logic (LTL) with quantification over traces. Suppose, for example, that an agent i in a distributed system observes only a subset of the system variables. The agent *knows* that some LTL formula φ is true on some trace π iff φ holds on *all* traces π' that agent i cannot distinguish from π. If we denote the indistinguishability of π and π' by $\pi \sim_i \pi'$, then the property that *there exists a trace* π *where agent* i *knows* φ can be expressed as the HyperLTL formula

$$\exists \pi. \forall \pi'. \, \pi \sim_i \pi' \rightarrow \varphi(\pi'),$$

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 309–332, 2023.
https://doi.org/10.1007/978-3-031-37703-7_15

where we write $\varphi(\pi')$ to denote that the trace property φ holds on trace π'.

While HyperLTL and its variations have found many applications [28,32,44], the expressiveness of these logics is limited, leaving many widely used hyperproperties out of reach. A prominent example is *common knowledge*, which is used in distributed applications to ensure simultaneous action [30,40]. Common knowledge in a group of agents means that the agents not only know *individually* that some condition φ is true, but that this knowledge is "common" to the group in the sense that each agent *knows* that every agent *knows* that φ is true; on top of that, each agent in the group *knows* that every agent *knows* that every agent *knows* that φ is true; and so on, forming an infinite chain of knowledge.

The fundamental limitation of HyperLTL that makes it impossible to express properties like common knowledge is that the logic is restricted to *first-order quantification*. HyperLTL, then, cannot reason about sets of traces directly, but must always do so by referring to individual traces that are chosen existentially or universally from the full set of traces. For the specification of an agent's individual knowledge, where we are only interested in the (non-)existence of a single trace that is indistinguishable and that violates φ, this is sufficient; however, expressing an infinite chain, as needed for common knowledge, is impossible.

In this paper, we introduce Hyper^{2}LTL, a temporal logic for hyperproperties with *second-order quantification* over traces. In Hyper^{2}LTL, the existence of a trace π where the condition φ is common knowledge can be expressed as the following formula (using slightly simplified syntax):

$$\exists \pi. \exists X.\ \pi \in X \land \Big(\forall \pi' \in X. \forall \pi''.\ \big(\bigvee_{i=1}^{n} \pi' \sim_i \pi''\big) \rightarrow \pi'' \in X\Big) \land \forall \pi' \in X. \varphi(\pi').$$

The second-order quantifier $\exists X$ postulates the existence of a set X of traces that (1) contains π; that (2) is closed under the observations of each agent, i.e., for every trace π' already in X, all other traces π'' that some agent i cannot distinguish from π' are also in X; and that (3) only contains traces that satisfy φ. The existence of X is a necessary and sufficient condition for φ being common knowledge on π. In the paper, we show that Hyper^{2}LTL is an elegant specification language for many hyperproperties of interest that cannot be expressed in HyperLTL, including, in addition to epistemic properties like common knowledge, also Mazurkiewicz trace theory and asynchronous hyperproperties.

The model checking problem for Hyper^{2}LTL is much more difficult than for HyperLTL. A HyperLTL formula can be checked by translating the LTL subformula into an automaton and then applying a series of automata transformations, such as self-composition to generate multiple traces, projection for existential quantification, and complementation for negation [8,32]. For Hyper^{2}LTL, the model checking problem is, in general, undecidable. We introduce a method that nevertheless obtains sound results by over- and underapproximating the quantified sets of traces. For this purpose, we study Hyper^{2}LTL$_{\text{fp}}$, a fragment of Hyper^{2}LTL, in which we restrict second-order quantification to the smallest or largest set satisfying some property. For example, to check common knowledge, it suffices to consider the *smallest* set X that is closed under the observations of all agents. This smallest set X is defined by the (monotone) fixpoint operation that adds, in each step, all traces that are indistinguishable to some trace already in X.

We develop an approximate model checking algorithm for Hyper^{2}LTL$_{\mathrm{fp}}$ that uses bidirectional inference to deduce lower and upper bounds on second-order variables, interposed with first-order model checking in the style of HyperLTL. Our procedure is parametric in an oracle that provides (increasingly precise) lower and upper bounds. In the paper, we realize the oracles with *fixpoint iteration* for underapproximations of the sets of traces assigned to the second-order variables, and *automata learning* for overapproximations. We report on encouraging experimental results with our model-checking algorithm, which has been implemented in a tool called `HySO`.

2 Preliminaries

For $n \in \mathbb{N}$ we define $[n] := \{1, \ldots, n\}$. We assume that AP is a finite set of atomic propositions and define $\Sigma := 2^{\mathrm{AP}}$. For $t \in \Sigma^\omega$ and $i \in \mathbb{N}$ define $t(i) \in \Sigma$ as the ith element in t (starting with the 0th); and $t[i, \infty]$ for the infinite suffix starting at position i. For traces $t_1, \ldots, t_n \in \Sigma^\omega$ we write $\mathit{zip}(t_1, \ldots, t_n) \in (\Sigma^n)^\omega$ for the pointwise zipping of the traces, i.e., $\mathit{zip}(t_1, \ldots, t_n)(i) := (t_1(i), \ldots, t_n(i))$.

Transition Systems. 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. A path in $\mathcal{T}$ is an infinite state sequence $s_0 s_1 s_2 \cdots \in S^\omega$, s.t., $s_0 \in S_0$, and $(s_i, s_{i+1}) \in \kappa$ for all i. The associated trace is given by $L(s_0)L(s_1)L(s_2) \cdots \in \Sigma^\omega$ and $\mathit{Traces}(\mathcal{T}) \subseteq \Sigma^\omega$ denotes all traces of $\mathcal{T}$.

Automata. A *non-deterministic Büchi automaton* (NBA) [18] is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ where Σ is a finite alphabet, Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $F \subseteq Q$ is a set of accepting states, and $\delta : Q \times \Sigma \to 2^Q$ is the transition function. A run on a word $u \in \Sigma^\omega$ is an infinite sequence of states $q_0 q_1 q_2 \cdots \in Q^\omega$ such that $q_0 \in Q_0$ and for every $i \in \mathbb{N}$, $q_{i+1} \in \delta(q_i, u(i))$. The run is accepting if it visits states in F infinitely many times, and we define the language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$, as all infinite words on which $\mathcal{A}$ has an accepting run.

HyperLTL. HyperLTL [20] is one of the most studied temporal logics for the specification of hyperproperties. We assume that $\mathcal{V}$ is a fixed set of trace variables. For the most part, we use variations of π (e.g., $\pi, \pi', \pi_1, \ldots$) to denote trace variables. HyperLTL formulas are then generated by the grammar

$$\begin{aligned}\varphi &:= \mathbb{Q}\pi.\, \varphi \mid \psi \\ \psi &:= a_\pi \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi\end{aligned}$$

where $a \in \mathrm{AP}$ is an atomic proposition, $\pi \in \mathcal{V}$ is a trace variable, $\mathbb{Q} \in \{\forall, \exists\}$ is a quantifier, and $\bigcirc$ and $\mathcal{U}$ are the temporal operators *next* and *until*.

The semantics of HyperLTL is given with respect to a *trace assignment* Π, which is a partial mapping $\Pi : \mathcal{V} \rightharpoonup \Sigma^\omega$ that maps trace variables to traces. Given $\pi \in \mathcal{V}$ and $t \in \Sigma^\omega$ we define $\Pi[\pi \mapsto t]$ as the updated assignment that maps π to t. For $i \in \mathbb{N}$ we define $\Pi[i, \infty]$ as the trace assignment defined by $\Pi[i, \infty](\pi) := \Pi(\pi)[i, \infty]$, i.e., we (synchronously) progress all traces by i steps. For quantifier-free formulas ψ we follow the LTL semantics and define

$$\begin{array}{lll}
\Pi \models a_\pi & \text{iff} & a \in \Pi(\pi)(0) \\
\Pi \models \neg\psi & \text{iff} & \Pi \not\models \psi \\
\Pi \models \psi_1 \wedge \psi_2 & \text{iff} & \Pi \models \psi_1 \text{ and } \Pi \models \psi_2 \\
\Pi \models \bigcirc \psi & \text{iff} & \Pi[1, \infty] \models \psi \\
\Pi \models \psi_1 \mathcal{U} \psi_2 & \text{iff} & \exists i \in \mathbb{N}.\, \Pi[i, \infty] \models \psi_2 \text{ and } \forall j < i.\, \Pi[j, \infty] \models \psi_1 \,.
\end{array}$$

The indexed atomic propositions refer to a specific path in Π, i.e., a_π holds iff a holds on the trace bound to π. Quantifiers range over system traces:

$$\Pi \models_{\mathcal{T}} \psi \text{ iff } \Pi \models \psi \qquad \text{and} \qquad \Pi \models_{\mathcal{T}} \mathbb{Q}\pi.\, \varphi \text{ iff } \mathbb{Q}t \in \mathit{Traces}(\mathcal{T}).\, \Pi[\pi \mapsto t] \models \varphi \,.$$

We write $\mathcal{T} \models \varphi$ if $\emptyset \models_{\mathcal{T}} \varphi$ where $\emptyset$ denotes the empty trace assignment.

HyperQPTL. HyperQPTL [45] adds – on top of the trace quantification of HyperLTL – also propositional quantification (analogous to the propositional quantification that QPTL [46] adds on top of LTL). For example, HyperQPTL can express a promptness property which states that there must exist a bound (which is common among all traces), up to which an event must have happened. We can express this as $\exists q. \forall \pi. \diamondsuit q \wedge (\neg q) \mathcal{U} a_\pi$ which states that there exists an evaluation of proposition q such that (1) q holds at least once, and (2) for all traces π, a holds on π before the first occurrence of q. See [8] for details.

3 Second-Order HyperLTL

The (first-order) trace quantification in HyperLTL ranges over the set of all system traces; we thus cannot reason about arbitrary sets of traces as required for, e.g., common knowledge. We introduce a second-order extension of HyperLTL by introducing second-order variables (ranging over sets of traces) and allowing quantification over traces from any such set. We present two variants of our logic that differ in the way quantification is resolved. In Hyper^{2}LTL, we quantify over arbitrary sets of traces. While this yields a powerful and intuitive logic, second-order quantification is inherently non-constructive. During model checking, there thus does not exist an efficient way to even approximate possible witnesses for the sets of traces. To solve this quandary, we restrict Hyper^{2}LTL to Hyper^{2}LTL$_{\text{fp}}$, where we instead quantify over sets of traces that satisfy some minimality or maximality constraint. This allows for large fragments of Hyper^{2}LTL$_{\text{fp}}$ that admit algorithmic approximations to its model checking (by, e.g., using known techniques from fixpoint computations [47,48]).

3.1 Hyper^{2}LTL

Alongside the set $\mathcal{V}$ of trace variables, we use a set $\mathfrak{V}$ of second-order variables (which we, for the most part, denote with capital letters $X, Y, ...$). We assume that there is a special variable $\mathfrak{S} \in \mathfrak{V}$ that refers to the set of traces of the given system at hand, and a variable $\mathfrak{A} \in \mathfrak{V}$ that refers to the set of all traces. We define the Hyper^{2}LTL syntax by the following grammar:

$$\varphi := \mathbb{Q}\pi \in X.\, \varphi \mid \mathbb{Q}X.\, \varphi \mid \psi$$
$$\psi := a_\pi \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc\psi \mid \psi\,\mathcal{U}\,\psi$$

where $a \in \text{AP}$ is an atomic proposition, $\pi \in \mathcal{V}$ is a trace variable, $X \in \mathfrak{V}$ is a second-order variable, and $\mathbb{Q} \in \{\forall, \exists\}$ is a quantifier. We also consider the usual derived Boolean constants (*true*, *false*) and connectives ($\vee$, $\rightarrow$, $\leftrightarrow$) as well as the temporal operators *eventually* ($\Diamond\psi := \mathit{true}\,\mathcal{U}\,\psi$) and *globally* ($\Box\psi := \neg\Diamond\neg\psi$). Given a set of atomic propositions $P \subseteq \text{AP}$ and two trace variables π, π', we abbreviate $\pi =_P \pi' := \bigwedge_{a \in P}(a_\pi \leftrightarrow a_{\pi'})$.

Semantics. Apart from a trace assignment Π (as in the semantics of HyperLTL), we maintain a second-order assignment $\Delta : \mathfrak{V} \rightharpoonup 2^{\Sigma^\omega}$ mapping second-order variables to *sets of traces*. Given $X \in \mathfrak{V}$ and $A \subseteq \Sigma^\omega$ we define the updated assignment $\Delta[X \mapsto A]$ as expected. Quantifier-free formulas ψ are then evaluated in a fixed trace assignment as for HyperLTL (cf. Sect. 2). For the quantifier prefix we define:

$$\begin{aligned} &\Pi, \Delta \models \psi && \text{iff} \quad \Pi \models \psi \\ &\Pi, \Delta \models \mathbb{Q}\pi \in X.\, \varphi && \text{iff} \quad \mathbb{Q}t \in \Delta(X).\, \Pi[\pi \mapsto t], \Delta \models \varphi \\ &\Pi, \Delta \models \mathbb{Q}X.\, \varphi && \text{iff} \quad \mathbb{Q}A \subseteq \Sigma^\omega.\, \Pi, \Delta[X \mapsto A] \models \varphi \end{aligned}$$

Second-order quantification updates Δ with a set of traces, and first-order quantification updates Π by quantifying over traces within the set defined by Δ.

Initially, we evaluate a formula in the empty trace assignment and fix the valuation of the special second-order variable $\mathfrak{S}$ to be the set of all system traces and $\mathfrak{A}$ to be the set of all traces. That is, given a system $\mathcal{T}$ and Hyper^{2}LTL formula φ, we say that $\mathcal{T}$ satisfies φ, written $\mathcal{T} \models \varphi$, if $\emptyset, [\mathfrak{S} \mapsto \mathit{Traces}(\mathcal{T}), \mathfrak{A} \mapsto \Sigma^\omega] \models \varphi$, where we write $\emptyset$ for the empty trace assignment. The model-checking problem for Hyper^{2}LTL is checking whether $\mathcal{T} \models \varphi$ holds.

Hyper^{2}LTL naturally generalizes HyperLTL by adding second-order quantification. As sets range over *arbitrary* traces, Hyper^{2}LTL also subsumes the more powerful logic HyperQPTL. The proof of Lemma 1 is given in the full version of this paper [11].

Lemma 1. Hyper^{2}LTL *subsumes* HyperQPTL *(and thus also* HyperLTL*).*

Syntactic Sugar. In Hyper^{2}LTL, we can quantify over traces within a second-order variable, but we cannot state, within the body of the formula, that some path is a member of some second-order variable. For that, we define $\pi \triangleright X$ (as an atom within the body) as syntactic sugar for $\exists \pi' \in X. \square(\pi' =_{\mathrm{AP}} \pi)$, i.e., π is in X if there exists some trace in X that agrees with π on all propositions. Note that we can only use $\pi \triangleright X$ *outside* of the scope of any temporal operators; this ensures that we can bring the resulting formula into a form that conforms to the Hyper^{2}LTL syntax.

3.2 Hyper^{2}LTL$_{fp}$

The semantics of Hyper^{2}LTL quantifies over arbitrary sets of traces, making even approximations to its semantics challenging. We propose Hyper^{2}LTL$_{\mathrm{fp}}$ as a restriction that only quantifies over sets that are subject to an additional minimality or maximality constraint. For large classes of formulas, we show that this admits effective model-checking approximations. We define Hyper^{2}LTL$_{\mathrm{fp}}$ by the following grammar:

$$\varphi := \mathbb{Q}\, \pi \in X. \varphi \mid \mathbb{Q}\,(X, \bowtie, \varphi). \varphi \mid \psi$$
$$\psi := a_\pi \mid \neg \psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi$$

where $a \in \mathrm{AP}$, $\pi \in \mathcal{V}$, $X \in \mathfrak{V}$, $\mathbb{Q} \in \{\forall, \exists\}$, and $\bowtie \in \{\curlywedge, \curlyvee\}$ determines if we consider smallest ($\curlyvee$) or largest ($\curlywedge$) sets. For example, the formula $\exists\,(X, \curlyvee, \varphi_1). \varphi_2$ holds if there exists some set of traces X, that satisfies both φ_1 and φ_2, and is *a* smallest set that satisfies φ_1. Such minimality and maximality constraints with respect to a (hyper)property arise naturally in many properties. Examples include common knowledge (cf. Sect. 3.3), asynchronous hyperproperties (cf. Sect. 4.2), and causality in reactive systems [22,23].

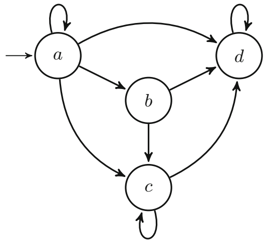
Semantics. For path formulas, the semantics of Hyper^{2}LTL$_{\mathrm{fp}}$ is defined analogously to that of Hyper^{2}LTL and HyperLTL. For the quantifier prefix we define:

$$\begin{array}{lll} \Pi, \Delta \vDash \psi & \text{iff} & \Pi \vDash \psi \\ \Pi, \Delta \vDash \mathbb{Q}\pi \in X. \varphi & \text{iff} & \mathbb{Q}t \in \Delta(X). \Pi[\pi \mapsto t], \Delta \vDash \varphi \\ \Pi, \Delta \vDash \mathbb{Q}(X, \bowtie, \varphi_1). \varphi_2 & \text{iff} & \mathbb{Q}A \in \mathit{sol}(\Pi, \Delta, (X, \bowtie, \varphi_1)). \Pi, \Delta[X \mapsto A] \vDash \varphi_2 \end{array}$$

where $\mathit{sol}(\Pi, \Delta, (X, \bowtie, \varphi_1))$ denotes all solutions to the minimality/maximality condition given by φ_1, which we define by mutual recursion as follows:

$$\mathit{sol}(\Pi, \Delta, (X, \curlyvee, \varphi)) := \{A \subseteq \Sigma^\omega \mid \Pi, \Delta[X \mapsto A] \vDash \varphi \wedge \forall A' \subsetneq A. \Pi, \Delta[X \mapsto A'] \nvDash \varphi\}$$
$$\mathit{sol}(\Pi, \Delta, (X, \curlywedge, \varphi)) := \{A \subseteq \Sigma^\omega \mid \Pi, \Delta[X \mapsto A] \vDash \varphi \wedge \forall A' \supsetneq A. \Pi, \Delta[X \mapsto A'] \nvDash \varphi\}$$

A set A satisfies the minimality/maximality constraint if it satisfies φ and is a least (in case $\bowtie = \curlyvee$) or greatest (in case $\bowtie = \curlywedge$) set that satisfies φ.



$$\begin{aligned}
\pi &= a^n d^\omega \\
K_2(\pi) &= a^{n-1} b d^\omega \\
K_1 K_2(\pi) &= a^{n-1} c d^\omega \\
K_2 K_1 K_2(\pi) &= a^{n-2} b c d^\omega \\
&\dots \\
K_1 K_2 \dots K_2(\pi) &= a c^{n-1} d^\omega
\end{aligned}$$

Fig. 1. Left: An example for a multi-agent system with two agents, where agent 1 observes a and d, and agent 2 observes c and d. Right: The iterative construction of the traces to be considered for common knowledge starting with $a^n d^\omega$.

Note that $sol(\Pi, \Delta, (X,)\!(, \varphi))$ can contain multiple sets or no set at all, i.e., there may not exists a unique least or greatest set that satisfies φ. In Hyper^{2}LTL$_{\text{fp}}$, we therefore add an additional quantification over the set of all solutions to the minimality/maximality constraint. When discussing our model checking approximation algorithm, we present a (syntactic) restriction on φ which guarantees that $sol(\Pi, \Delta, (X,)\!(, \varphi))$ contains a unique element (i.e., is a singleton set). Moreover, our restriction allows us to employ fixpoint techniques to find approximations to this unique solution. In case the solution for $(X,)\!(, \varphi)$ is unique, we often omit the leading quantifier and simply write $(X,)\!(, \varphi)$ instead of $\mathbb{Q}(X,)\!(, \varphi)$.

As we can encode the minimality/maximality constraints of Hyper^{2}LTL$_{\text{fp}}$ in Hyper^{2}LTL (see full version [11]), we have the following:

Proposition 1. *Any* Hyper^{2}LTL$_{\text{fp}}$ *formula φ can be effectively translated into an* Hyper^{2}LTL *formula φ' such that for all transition systems $\mathcal{T}$ we have $\mathcal{T} \models \varphi$ iff $\mathcal{T} \models \varphi'$.*

3.3 Common Knowledge in Multi-agent Systems

To explain common knowledge, we use a variation of an example from [43], and encode it in Hyper^{2}LTL$_{\text{fp}}$. Fig. 1(left) shows a transition system of a distributed system with two agents, agent 1 and agent 2. Agent 1 observes variables a and d, whereas agent 2 observes c and d. The property of interest is *starting from the trace $\pi = a^n d^\omega$ for some fixed $n > 1$, is it common knowledge for the two agents that a holds in the second step.* It is trivial to see that $\bigcirc a$ holds on π. However, for common knowledge, we consider the (possibly) infinite chain of observationally equivalent traces. For example, agent 2 cannot distinguish the traces $a^n d^\omega$ and $a^{n-1} b d^\omega$. Therefore, agent 2 only knows that $\bigcirc a$ holds on π if it also holds on $\pi' = a^{n-1} b d^\omega$. For common knowledge, agent 1 also has to know that agent 2 knows $\bigcirc a$, which means that for all traces that are indistinguishable from π or π' for agent 1, $\bigcirc a$ has to hold. This adds $\pi'' = a^{n-1} c d^\omega$ to the set of traces to verify $\bigcirc a$ against. This chain of reasoning continues as shown in Fig. 1(right). In the last step we add $ac^{n-1}d^\omega$ to the set of indistinguishable traces, concluding that $\bigcirc a$ is not common knowledge.

The following Hyper^{2}LTL$_\text{fp}$ formula specifies the property stated above. The abbreviation $\mathit{obs}(\pi_1, \pi_2) := \square(\pi_1 =_{\{a,d\}} \pi_2) \lor \square(\pi_1 =_{\{c,d\}} \pi_2)$ denotes that π_1 and π_2 are observationally equivalent for either agent 1 or agent 2.

$$\forall \pi \in \mathfrak{S}.\big(\bigwedge_{i=0}^{n-1} \bigcirc^i a_\pi \land \bigcirc^n \square d_\pi\big) \rightarrow$$
$$\Big(X, \curlyvee, \pi \triangleright X \land \big(\forall \pi_1 \in X. \forall \pi_2 \in \mathfrak{S}.\, \mathit{obs}(\pi_1, \pi_2) \rightarrow \pi_2 \triangleright X\big)\Big). \forall \pi' \in X. \bigcirc a_{\pi'}$$

For a trace π of the form $\pi = a^n d^\omega$, the set X represents the *common knowledge set* on π. This set X is the smallest set that (1) contains π (expressed using our syntactic sugar $\triangleright$); and (2) is closed under observations by either agent, i.e., if we find some $\pi_1 \in X$ and some system trace π_2 that are observationally equivalent, π_2 should also be in X. Note that this set is unique (due to the minimality restriction), so we do not quantify it explicitly. Lastly, we require that all traces in X satisfy the property $\bigcirc a$. All sets that satisfy this formula would also include the trace $ac^{n-1}d^\omega$, and therefore no such X exists; thus, we can conclude that starting from trace $a^n d^\omega$, it is *not* common knowledge that $\bigcirc a$ holds. On the other hand, it *is* common knowledge that a holds in the *first* step (cf. Sect. 6).

3.4 Hyper^{2}LTL Model Checking

As Hyper^{2}LTL and Hyper^{2}LTL$_\text{fp}$ allow quantification over arbitrary sets of traces, we can encode the satisfiability of HyperQPTL (i.e., the question of whether some set of traces satisfies a formula) within their model-checking problem; rendering the model-checking problem highly undecidable [34], even for very simple formulas [4].

Proposition 2. *For any* HyperQPTL *formula φ there exists a* Hyper^{2}LTL *formula φ' such that φ is satisfiable iff φ' holds on some arbitrary transition system. The model-checking problem of* Hyper^{2}LTL *is thus highly undecidable (Σ_1^1-hard).*

Proof. Let φ' be the Hyper^{2}LTL formula obtained from φ by replacing each HyperQPTL trace quantifier $\mathbb{Q}\pi$ with the Hyper^{2}LTL quantifier $\mathbb{Q}\pi \in X$, and each propositional quantifier $\mathbb{Q}q$ with $\mathbb{Q}\pi_q \in \mathfrak{A}$ for some fresh trace variable π_q. In the body, we replace each propositional variable q with a_{π_q} for some fixed proposition $a \in \text{AP}$. Then, φ is satisfiable iff the Hyper^{2}LTL formula $\exists X.\varphi'$ holds in some arbitrary system. □

Hyper^{2}LTL$_\text{fp}$ cannot express HyperQPTL satisfiability directly. If there exists a model of a HyperQPTL formula, there may not exist a least one. However, model checking of Hyper^{2}LTL$_\text{fp}$ is also highly undecidable.

Proposition 3. *The model-checking problem of* Hyper^{2}LTL$_\text{fp}$ *is Σ_1^1-hard.*

Proof (Sketch). We can encode the existence of a *recurrent* computation of a Turing machine, which is known to be Σ_1^1-hard [1]. □

Conversely, the *existential* fragment of Hyper^{2}LTL can be encoded back into HyperQPTL satisfiability:

Proposition 4. *Let φ be a* Hyper^{2}LTL *formula that uses only existential second-order quantification and $\mathcal{T}$ be any system. We can effectively construct a formula φ' in* HyperQPTL *such that $\mathcal{T} \vDash \varphi$ iff φ' is satisfiable.*

Lastly, we present some easy fragments of Hyper^{2}LTL for which the model-checking problem is decidable. Here we write $\exists^* X$ (resp. $\forall^* X$) for some sequence of existentially (resp. universally) quantified *second-order* variables and $\exists^* \pi$ (resp. $\forall^* \pi$) for some sequence of existentially (resp. universally) quantified *first-order* variables. For example, $\exists^* X \forall^* \pi$ captures all formulas of the form $\exists X_1, \dots X_n . \forall \pi_1, \dots, \pi_m . \psi$ where ψ is quantifier-free.

Proposition 5. *The model-checking problem of* Hyper^{2}LTL *is decidable for the fragments: $\exists^* X \forall^* \pi$, $\forall^* X \forall^* \pi$, $\exists^* X \exists^* \pi$, $\forall^* X \exists^* \pi$, $\exists X . \exists^* \pi \in X \forall^* \pi' \in X$.*

We refer the reader to the full version [11] for detailed proofs.

4 Expressiveness of Hyper^{2}LTL

In this section, we point to existing logics that can naturally be encoded within our second-order hyperlogics Hyper^{2}LTL and Hyper^{2}LTL$_{\mathrm{fp}}$.

4.1 Hyper^{2}LTL and LTL$_{\mathsf{K},\mathsf{C}}$

LTL$_{\mathsf{K}}$ extends LTL with the knowledge operator K. For some subset of agents A, the formula $\mathsf{K}_A \psi$ holds in timestep i, if ψ holds on all traces equivalent to some agent in A up to timestep i. See full version [11] for detailed semantics. LTL$_{\mathsf{K}}$ and HyperCTL* have incomparable expressiveness [16] but the knowledge operator K can be encoded by either adding a linear past operator [16] or by adding propositional quantification (as in HyperQPTL) [45].

Using Hyper^{2}LTL$_{\mathrm{fp}}$ we can encode LTL$_{\mathsf{K},\mathsf{C}}$, featuring the knowledge operator K *and* the common knowledge operator C (which requires that ψ holds on the closure set of equivalent traces, up to the current timepoint) [41]. Note that LTL$_{\mathsf{K},\mathsf{C}}$ is not encodable by only adding propositional quantification or the linear past operator.

Proposition 6. *For every* LTL$_{\mathsf{K},\mathsf{C}}$ *formula φ there exists an* Hyper^{2}LTL$_{\mathrm{fp}}$ *formula φ' such that for any system $\mathcal{T}$ we have $\mathcal{T} \vDash_{LTL_{\mathsf{K},\mathsf{C}}} \varphi$ iff $\mathcal{T} \vDash \varphi'$.*

Proof (Sketch). We follow the intuition discussed in Sect. 3.3. For each occurrence of a knowledge operator in $\{\mathsf{K}, \mathsf{C}\}$, we use a fresh trace variable to keep track on the points in time with respect to which we need to compare traces. We then use this trace variable to introduce a second-order set that collects all equivalent traces (by the observations of one agent, or the closure of all agents' observations). We then inductively construct a Hyper^{2}LTL$_{\text{fp}}$ formula that captures all the knowledge and common-knowledge sets, over which we check the properties at hand. See full version for more details [11]. □

4.2 Hyper^{2}LTL and Asynchronous Hyperproperties

Most existing hyperlogics (including Hyper^{2}LTL) traverse the traces of a system *synchronously*. However, in many cases such a synchronous traversal is too restricting and we need to compare traces asynchronously. As an example, consider *observational determinism* (OD), which we can express in HyperLTL as $\varphi_{OD} := \forall \pi_1 . \forall \pi_2 . \square (o_{\pi_1} \leftrightarrow o_{\pi_2})$. The formula states that the output of a system is identical across all traces and so (trivially) no information about high-security inputs is leaked. In most systems encountered in practice, this synchronous formula is violated, as the exact timing between updates to o might differ by a few steps (we provide some examples in the full version [11]). However, assuming that an attacker only has access to the memory footprint and not a timing channel, we would only like to check that all traces are *stutter* equivalent (with respect to o).

A range of extensions to existing hyperlogics has been proposed to reason about such asynchronous hyperproperties [3,5,9,17,39]. We consider AHLTL [3]. An AHLTL formula has the form $\mathbb{Q}_1 \pi_1, \ldots, \mathbb{Q}_n \pi_m . \mathbf{E} . \psi$ where ψ is a qunatifier-free HyperLTL formula. The initial trace quantifier prefix is handled as in HyperLTL. However, different from HyperLTL, a trace assignment $[\pi_1 \mapsto t_1, \ldots, \pi_n \mapsto t_n]$ satisfies $\mathbf{E} . \psi$ if there exist stuttered traces $t'_1, \ldots, t'_n$ of $t_1, \ldots, t_n$ such that $[\pi_1 \mapsto t'_1, \ldots, \pi_n \mapsto t'_n] \models \psi$. We write $\mathcal{T} \models_{AHLTL} \varphi$ if a system $\mathcal{T}$ satisfies the AHLTL formula φ. Using this quantification over stutterings we can, for example, express an asynchronous version of observational determinism as $\forall \pi_1 . \forall \pi_2 . \mathbf{E} . \square (o_{\pi_1} \leftrightarrow o_{\pi_2})$ stating that every two traces can be aligned such that they (globally) agree on o. Despite the fact that Hyper^{2}LTL$_{\text{fp}}$ is itself synchronous, we can use second-order quantification to encode asynchronous hyperproperties, as we state in the following proposition.

Proposition 7. *For any AHLTL formula φ there exists a* Hyper^{2}LTL$_{\text{fp}}$ *formula φ' such that for any system $\mathcal{T}$ we have $\mathcal{T} \models_{AHLTL} \varphi$ iff $\mathcal{T} \models \varphi'$.*

Proof. Assume that $\varphi = \mathbb{Q}_1 \pi_1, \ldots, \mathbb{Q}_n \pi_n . \mathbf{E} . \psi$ is the given AHLTL formula. For each $i \in [n]$ we define a formula φ_i as follows

$$\forall \pi_1 \in X_i . \forall \pi_2 \in \mathfrak{A} .$$
$$\Big((\pi_1 =_{\text{AP}} \pi_2) \, \mathcal{U} \, \big((\pi_1 =_{\text{AP}} \pi_2) \land \square \bigwedge_{a \in \text{AP}} a_{\pi_1} \leftrightarrow \bigcirc a_{\pi_2} \big) \Big) \to \pi_2 \triangleright X_i$$

The formula asserts that the set of traces bound to X_i is closed under stuttering, i.e., if we start from any trace in X_i and stutter it once (at some arbitrary position) we again end up in X_i. Using the formulas φ_i, we then construct a Hyper^{2}LTL$_{\mathrm{fp}}$ formula that is equivalent to φ as follows

$$\begin{aligned}\varphi' := \mathbb{Q}_1\pi_1 \in \mathfrak{S}, \dots, \mathbb{Q}_n\pi_n \in \mathfrak{S}.(X_1, \curlyvee, \pi_1 \triangleright X_1 \wedge \varphi_1) \cdots (X_n, \curlyvee, \pi_n \triangleright X_n \wedge \varphi_n)\\ \exists \pi'_1 \in X_1, \dots, \exists \pi'_n \in X_n.\psi[\pi'_1/\pi_1, \dots, \pi'_n/\pi_n]\end{aligned}$$

We first mimic the quantification in φ and, for each trace π_i, construct a least set X_i that contains π_i and is closed under stuttering (thus describing exactly the set of all stuttering of π_i). Finally, we assert that there are traces $\pi'_1, \dots, \pi'_n$ with $\pi'_i \in X_i$ (so π'_i is a stuttering of π_i) such that $\pi'_1, \dots, \pi'_n$ satisfy ψ. It is easy to see that $\mathcal{T} \models_{AHLTL} \varphi$ iff $\mathcal{T} \models \varphi'$ holds for all systems. □

Hyper^{2}LTL$_{\mathrm{fp}}$ captures all properties expressible in AHLTL. In particular, our approximate model-checking algorithm for Hyper^{2}LTL$_{\mathrm{fp}}$ (cf. Sect. 5) is applicable to AHLTL; even for instances where no approximate solutions were previously known. In Sect. 6, we show that our prototype model checker for Hyper^{2}LTL$_{\mathrm{fp}}$ can verify asynchronous properties in practice.

5 Model-Checking Hyper^{2}LTL$_{fp}$

In general, finite-state model checking of Hyper^{2}LTL$_{\mathrm{fp}}$ is highly undecidable (cf. Proposition 2). In this section, we outline a partial algorithm that computes approximations on the concrete values of second-order variables for a fragment of Hyper^{2}LTL$_{\mathrm{fp}}$. At a very high-level, our algorithm (Algorithm 1) iteratively computes under- and overapproximations for second-order variables. It then turns to resolve first-order quantification, using techniques from HyperLTL model checking [8,32], and resolves existential and universal trace quantification on the under- and overapproximation of the second-order variables, respectively. If the verification fails, it goes back to refine second-order approximations.

In this section, we focus on the setting where we are interested in the least sets (using $\curlyvee$), and use techniques to approximate the *least* fixpoint. A similar (dual) treatment is possible for Hyper^{2}LTL$_{\mathrm{fp}}$ formulas that use the largest set. Every Hyper^{2}LTL$_{\mathrm{fp}}$ which uses only minimal sets has the following form:

$$\varphi = \gamma_1.(Y_1, \curlyvee, \varphi_1^{con}).\gamma_2 \dots .(Y_k, \curlyvee, \varphi_k^{con}).\, \gamma_{k+1}.\, \psi \qquad (1)$$

We quantify second-order variables $Y_1, \dots, Y_k$, where, for each $j \in [k]$, Y_j is the least set that satisfies φ_j^{con}. Finally, for each $j \in [k+1]$,

$$\gamma_j = \mathbb{Q}_{l_j+1}\pi_{l_j+1} \in X_{l_j+1} \dots \mathbb{Q}_{l_{j+1}}\pi_{l_{j+1}} \in X_{l_{j+1}}$$

is the block of first-order quantifiers that sits between the quantification of Y_{j-1} and Y_j. Here $X_{l_j+1}, \dots, X_{l_{j+1}} \in \{\mathfrak{S}, \mathfrak{A}, Y_1, \dots, Y_{j-1}\}$ are second-order variables that are quantified before γ_j. In particular, $\pi_1, \dots, \pi_{l_j}$ are the first-order variables quantified before Y_j.

5.1 Fixpoints in Hyper^{2}LTL$_{fp}$

We consider a fragment of Hyper^{2}LTL$_{\mathrm{fp}}$ which we call the *least fixpoint fragment*. Within this fragment, we restrict the formulas $\varphi_1^{con}, \ldots, \varphi_k^{con}$ such that $Y_1, \ldots, Y_k$ can be approximated as (least) fixpoints. Concretely, we say that φ is in the *least fixpoint fragment* of Hyper^{2}LTL$_{\mathrm{fp}}$ if for all $j \in [k]$, φ_j^{con} is a conjunction of formulas of the form

$$\forall \dot{\pi}_1 \in X_1 \ldots \forall \dot{\pi}_n \in X_n.\, \psi_{step} \rightarrow \dot{\pi}_M \triangleright Y_j \tag{2}$$

where each $X_i \in \{\mathfrak{S}, \mathfrak{A}, Y_1, \ldots, Y_j\}$, ψ_{step} is quantifier-free formula over trace variables $\dot{\pi}_1, \ldots, \dot{\pi}_n, \pi_1, \ldots, \pi_{l_j}$, and $M \in [n]$. Intuitively, Eq. (2) states a requirement on traces that should be included in Y_j. If we find traces $\dot{t}_1 \in X_1, \ldots, \dot{t}_n \in X_n$ that, together with the traces $t_1, \ldots, t_{l_j}$ quantified before Y_j, satisfy ψ_{step}, then $\dot{t}_M$ should be included in Y_j.

Together with the minimality constraint on Y_j (stemming from the semantics of Hyper^{2}LTL$_{\mathrm{fp}}$), this effectively defines a (monotone) least fixpoint computation, as ψ_{step} defines exactly the traces to be added to the set. This will allow us to use results from fixpoint theory to compute approximations for the sets Y_j.

Our least fixpoint fragment captures most properties of interest, in particular, common knowledge (Sect. 3.3) and asynchronous hyperproperties (Sect. 4.2). We observe that formulas of the above form ensure that the solution Y_j is unique, i.e., for any trace assignment Π to $\pi_1, \ldots, \pi_{l_j}$ and second-order assignment Δ to $\mathfrak{S}, \mathfrak{A}, Y_1, \ldots, Y_{j-1}$, there is only one element in $sol(\Pi, \Delta, (Y_j, \curlyvee, \varphi_j^{con}))$.

5.2 Functions as Automata

In our (approximate) model-checking algorithm, we represent a concrete assignment to the second-order variables $Y_1, \ldots, Y_k$ using automata $\mathcal{B}_{Y_1}, \ldots, \mathcal{B}_{Y_k}$. The concrete assignment of Y_j can depend on traces assigned to $\pi_1, \ldots, \pi_{l_j}$, i.e., the first-order variables quantified before Y_j. To capture these dependencies, we view each Y_j not as a set of traces but as a function mapping traces of all preceding first-order variables to a set of traces. We represent such a function $f : (\Sigma^\omega)^{l_j} \rightarrow 2^{(\Sigma^\omega)}$ mapping the l_j traces to a set of traces as an automaton $\mathcal{A}$ over Σ^{l_j+1}. For traces $t_1, \ldots, t_{l_j}$, the set $f(t_1, \ldots, t_{l_j})$ is represented in the automaton by the set $\{t \in \Sigma^\omega \mid zip(t_1, \ldots, t_{l_j}, t) \in \mathcal{L}(\mathcal{A})\}$. For example, the function $f(t_1) := \{t_1\}$ can be defined by the automaton that accepts the zipping of a pair of traces exactly if both traces agree on all propositions. This representation of functions as automata allows us to maintain an assignment to Y_j that is parametric in $\pi_1, \ldots, \pi_{l_j}$ and still allows first-order model checking on $Y_1, \ldots, Y_k$.

5.3 Model Checking for First-Order Quantification

First, we focus on first-order quantification, and assume that we are given a concrete assignment for each second-order variable as fixed automata $\mathcal{B}_{Y_1}, \ldots, \mathcal{B}_{Y_k}$

(where $\mathcal{B}_{Y_j}$ is an automaton over Σ^{l_j+1}). Our construction for resolving first-order quantification is based on HyperLTL model checking [32], but needs to work on sets of traces that, themselves, are based on traces quantified before (cf. Sect. 5.2). Recall that the first-order quantifier prefix is $\gamma_1 \cdots \gamma_{k+1} = \mathbb{Q}_1 \pi_1 \in X_1 \cdots \mathbb{Q}_{l_{k+1}} \pi_{l_{k+1}} \in X_{l_{k+1}}$. For each $1 \leq i \leq l_{k+1}$ we inductively construct an automaton $\mathcal{A}_i$ over Σ^{i-1} that summarizes all trace assignments to $\pi_1, \ldots, \pi_{i-1}$ that satisfy the subformula starting with the quantification of π_i. That is, for all traces $t_1, \ldots, t_{i-1}$ we have

$$[\pi_1 \mapsto t_1, \ldots, \pi_{i-1} \mapsto t_{i-1}] \vDash \mathbb{Q}_i \pi_i \in X_i \cdots \mathbb{Q}_{l_{k+1}} \pi_{l_{k+1}} \in X_{l_{k+1}}. \psi$$

(under the fixed second-order assignment for $Y_1, \ldots, Y_k$ given by $\mathcal{B}_{Y_1}, \ldots, \mathcal{B}_{Y_k}$) if and only if $zip(t_1, \ldots, t_{i-1}) \in \mathcal{L}(\mathcal{A}_i)$. In the context of HyperLTL model checking we say $\mathcal{A}_i$ is *equivalent* to $\mathbb{Q}_i \pi_i \in X_i \cdots \mathbb{Q}_{l_{k+1}} \pi_{l_{k+1}} \in X_{l_{k+1}}. \psi$ [8,32]. In particular, $\mathcal{A}_1$ is an automaton over singleton alphabet Σ^0.

We construct $\mathcal{A}_1, \ldots, \mathcal{A}_{l_{k+1}+1}$ inductively, starting with $\mathcal{A}_{l_{k+1}+1}$. Initially, we construct $\mathcal{A}_{l_{k+1}+1}$ (over $\Sigma^{l_{k+1}}$) using a standard LTL-to-NBA construction on the (quantifier-free) body ψ (see [32] for details). Now assume that we are given an (inductively constructed) automaton $\mathcal{A}_{i+1}$ over Σ^i and want to construct $\mathcal{A}_i$. We first consider the case where $\mathbb{Q}_i = \exists$, i.e., the ith trace quantification is existential. Now X_i (the set where π_i is resolved on) either equals $\mathfrak{S}$, $\mathfrak{A}$ or Y_j for some $j \in [k]$. In either case, we represent the current assignment to X_i as an automaton $\mathcal{C}$ over Σ^{T+1} for some $T < i$ that defines the model of X_i based on traces $\pi_1, \ldots, \pi_T$: In case $X_i = \mathfrak{S}$, we set $\mathcal{C}$ to be the automaton over Σ^{0+1} that accepts exactly the traces in the given system $\mathcal{T}$; in case $X_i = \mathfrak{A}$, we set $\mathcal{C}$ to be the automaton over Σ^{0+1} that accepts all traces; If $X_i = Y_j$ for some $j \in [k]$ we set $\mathcal{C}$ to be $\mathcal{B}_{Y_j}$ (which is an automaton over Σ^{l_j+1}).[1] Given $\mathcal{C}$, we can now modify the construction from [32], to resolve first-order quantification: The desired automaton $\mathcal{A}_i$ should accept the zipping of traces $t_1, \ldots, t_{i-1}$ if there exists a trace t such that (1) $zip(t_1, \ldots, t_{i-1}, t) \in \mathcal{L}(\mathcal{A}_{i+1})$, *and* (2) the trace t is contained in the set of traces assigned to X_i as given by $\mathcal{C}$, i.e., $zip(t_1, \ldots, t_T, t) \in \mathcal{L}(\mathcal{C})$. The construction of this automaton is straightforward by taking a product of $\mathcal{A}_{i+1}$ and $\mathcal{C}$. We denote this automaton with `eProduct(`$\mathcal{A}_{i+1}$`,`$\mathcal{C}$`)`. In case $\mathbb{Q}_i = \forall$ we exploit the duality that $\forall \pi. \psi = \neg \exists \pi. \neg \psi$, combining the above construction with automata complementation. We denote this universal product of $\mathcal{A}_{i+1}$ and $\mathcal{C}$ with `uProduct(`$\mathcal{A}_{i+1}$`,`$\mathcal{C}$`)`.

The final automaton $\mathcal{A}_1$ is an automaton over singleton alphabet Σ^0 that is equivalent to $\gamma_1 \cdots \gamma_{k+1}. \psi$, i.e., the entire first-order quantifier prefix. Automaton $\mathcal{A}_1$ thus satisfies $\mathcal{L}(\mathcal{A}_1) \neq \emptyset$ (which we can decide) iff the empty trace assignment satisfies the first-order formula $\gamma_1 \cdots \gamma_{k+1}. \psi$, iff φ (of Eq. (1)) holds within the fixed model for $Y_1, \ldots, Y_k$. For a given fixed second-order assignment (given as automata $\mathcal{B}_{Y_1}, \ldots, \mathcal{B}_{Y_k}$), we can thus decide if the system satisfies the first-order part.

[1] Note that in this case $l_j < i$: if trace π_i is resolved on Y_j (i.e., $X_i = Y_j$), then Y_j must be quantified *before* π_i so there are at most $i - 1$ traces quantified before Y_j.

Algorithm 1

```
verify($\varphi$, $\mathcal{T}$) =
  let $\varphi$ = $\left[\gamma_j \; (Y_j, \Upsilon, \varphi_j^{con})\right]_{j=1}^{k} \gamma_{k+1}. \psi$ where $\gamma_i$ = $\left[\mathbb{Q}_m \pi_m \in X_m\right]_{m=l_i+1}^{l_{i+1}}$
  let N = 0
  let $\mathcal{A}_\mathcal{T}$ = systemToNBA($\mathcal{T}$)
  repeat
    // Start outside-in traversal on second-order variables
    let $\flat$ = $\left[\mathfrak{S} \mapsto (\mathcal{A}_\mathcal{T}, \mathcal{A}_\mathcal{T}), \mathfrak{A} \mapsto (\mathcal{A}_\top, \mathcal{A}_\top)\right]$
    for $j$ from 1 to $k$ do
      $\mathcal{B}_j^l$ := underApprox(($Y_j, \Upsilon, \varphi_j^{con}$),$\flat$,$N$)
      $\mathcal{B}_j^u$ := overApprox(($Y_j, \Upsilon, \varphi_j^{con}$),$\flat$,$N$)
      $\flat(Y_j)$ := $(\mathcal{B}_j^l, \mathcal{B}_j^u)$
    // Start inside-out traversal on first-order variables
    let $\mathcal{A}_{l_{k+1}+1}$ = LTLtoNBA($\psi$)
    for $i$ from $l_{k+1}$ to 1 do
      let ($\mathcal{C}^l$,$\mathcal{C}^u$) = $\flat(X_i)$
      if $\mathbb{Q}_i = \exists$ then
        $\mathcal{A}_i$ := eProduct($\mathcal{A}_{i+1}$, $\mathcal{C}^l$)
      else
        $\mathcal{A}_i$ := uProduct($\mathcal{A}_{i+1}$, $\mathcal{C}^u$)
    if $\mathcal{L}(\mathcal{A}_1) \neq \emptyset$ then
      return SAT
   else
      $N$ = $N$ + 1
```

During the first-order model-checking phase, each quantifier alternations in the formula require complex automata complementation. For the first-order phase, we could also use cheaper approximate methods by, e.g., instantiating the existential trace using a strategy [6,7,25].

5.4 Bidirectional Model Checking

So far, we have discussed the verification of the first-order quantifiers assuming we have a fixed model for all second-order variables $Y_1, \ldots, Y_k$. In our actual model-checking algorithm, we instead maintain under- and overapproximations on each of the $Y_1, \ldots, Y_k$.

In each iteration, we first traverse the second-order quantifiers in an *outside-in* direction and compute lower- and upper-bounds on each Y_j. Given the bounds, we then traverse the first-order prefix in an *inside-out* direction using the current approximations to $Y_1, \ldots, Y_k$. If the current approximations are not precise enough to witness the satisfaction (or violation) of a property, we repeat and try to compute better bounds on $Y_1, \ldots, Y_k$. Due to the different directions of traversal, we refer to our model-checking approach as *bidirectional*. Algorithm 1 provides an overview. Initially, we convert the system $\mathcal{T}$ to an NBA $\mathcal{A}_\mathcal{T}$ accepting exactly the traces of the system. In each round, we compute under- and overapproximations for each Y_j in a mapping $\flat$. We initialize $\flat$ by mapping $\mathfrak{S}$ to $(\mathcal{A}_\mathcal{T}, \mathcal{A}_\mathcal{T})$ (i.e., the value assigned to the system variable is precisely $\mathcal{A}_\mathcal{T}$ for both under- and overapproximation), and $\mathfrak{A}$ to $(\mathcal{A}_\top, \mathcal{A}_\top)$ where $\mathcal{A}_\top$ is an automaton over Σ^1 accepting all traces. We then traverse the second-order quantifiers outside-in (from Y_1 to Y_k) and for each Y_j compute a pair $(\mathcal{B}^l_j, \mathcal{B}^u_j)$ of automata over Σ^{l_j+1} that under- and overapproximate the actual (unique) model of Y_j. We compute these approximations using functions `underApprox` and `overApprox`, which can be instantiated with any procedure that computes sound lower and upper bounds (see Sect. 5.5). During verification, we further maintain a precision bound N (initially set to 0) that tracks the current precision of the second-order approximations.

When $\flat$ contains an under- and overapproximation for each second-order variable, we traverse the first-order variables in an inside-out direction (from $\pi_{l_{k+1}}$ to π_1) and, following the construction outlined in Sect. 5.3, construct automata $\mathcal{A}_{l_k+1}, \dots, \mathcal{A}_1$. Different from the simplified setting in Sect. 5.3 (where we assume a fixed automaton $\mathcal{B}_{Y_j}$ providing a model for each Y_j), the mapping $\flat$ contains only approximations of the concrete solution. We choose which approximation to use according to the corresponding set quantification: In case we construct $\mathcal{A}_i$ and $\mathbb{Q}_i = \exists$, we use the *underapproximation* (thus making sure that any witness trace we pick is indeed contained in the actual model of the second-order variable); and if $\mathbb{Q}_i = \forall$, we use the *overapproximation* (making sure that we consider at least those traces that are in the actual solution). If $\mathcal{L}(\mathcal{A}_1)$ is nonempty, i.e., accepts the empty trace assignment, the formula holds (assuming the approximations returned by `underApprox` and `overApprox` are sound). If not, we increase the precision bound N and repeat.

In Algorithm 1, we only check for the satisfaction of a formula (to keep the notation succinct). Using the second-order approximations in $\flat$ we can also check the negation of a formula (by considering the negated body and dualizing all trace quantifiers). Our tool (Sect. 6) makes use of this and thus simultaneously tries to show satisfaction and violation of a formula.

5.5 Computing Under- and Overapproximations

In this section we provide concrete instantiations for `underApprox` and `overApprox`.

Computing Underapproximations. As we consider the fixpoint fragment, each formula φ^{con}_j (defining Y_j) is a conjunction of formulas of the form in Eq. (2), thus defining Y_j via a least fixpoint computation. For simplicity, we assume that Y_j is defined by the single conjunct, given by Eq. (2) (our construction generalizes easily to a conjunction of such formulas). Assuming fixed models for $\mathfrak{S}$, $\mathfrak{A}$ and $Y_1, \dots, Y_{j-1}$, the fixpoint operation defining Y_j is monotone, i.e., the larger the current model for Y_j is, the more traces we need to add according to Eq. (2). Monotonicity allows us to apply the Knaster-Tarski theorem [47] and compute underapproximations to the fixpoint by iteration.

In our construction of an approximation for Y_j, we are given a mapping $\flat$ that fixes a pair of automata for $\mathfrak{S}$, $\mathfrak{A}$, and $Y_1, \dots, Y_{j-1}$ (due to the outside-in traversal in Algorithm 1). As we are computing an underapproximation, we use the underapproximation for each of the second-order variables in $\flat$. So $\flat(\mathfrak{S})$ and $\flat(\mathfrak{A})$ are automata over Σ^1 and for each $j' \in [j-1]$, $\flat(Y_{j'})$ is an automaton over $\Sigma^{l_{j'}+1}$. Given this fixed mapping $\flat$, we iteratively construct automata $\hat{\mathcal{C}}_0, \hat{\mathcal{C}}_1, \dots$ over Σ^{l_j+1} that capture (increasingly precise) underapproximations on the solution for Y_j. We set $\hat{\mathcal{C}}_0$ to be the automaton with the empty language. We then recursively define $\hat{\mathcal{C}}_{N+1}$ based on $\hat{\mathcal{C}}_N$ as follows: For each second-order variable X_i for $i \in [n]$ used in Eq. (2) we can assume a concrete assignment in the form of an automaton $\mathcal{D}_i$ over Σ^{T_i+1} for some $T_i \leq l_j$: In case $X_i \neq Y_j$ (so $X_i \in \{\mathfrak{S}, \mathfrak{A}, Y_1, \dots, Y_{j-1}\}$), we set $\mathcal{D}_i := \flat(X_i)$. In case $X_i = Y_j$, we set $\mathcal{D}_i := \hat{\mathcal{C}}_N$, i.e., we use the current approximation of Y_j in iteration N. After we have set $\mathcal{D}_1, \dots, \mathcal{D}_n$, we compute an automaton $\dot{\mathcal{C}}$ over Σ^{l_j+1} that accepts $\mathit{zip}(t_1, \dots, t_{l_j}, t)$ iff there exists traces $\dot{t}_1, \dots, \dot{t}_n$ such that (1) $\mathit{zip}(t_1, \dots, t_{T_i}, \dot{t}_i) \in \mathcal{L}(\mathcal{D}_i)$ for all $i \in [n]$, (2) $[\pi_1 \mapsto t_1, \dots, \pi_{l_j} \mapsto t_{l_j}, \dot{\pi}_1 \mapsto \dot{t}_1, \dots, \dot{\pi}_n \mapsto \dot{t}_n] \vDash \psi_{step}$, and (3) trace t equals $\dot{t}_M$ (of Eq. (2)). The intuition is that $\dot{\mathcal{C}}$ captures all traces that should be added to Y_j: Given $t_1, \dots, t_{l_j}$ we check if there are traces $\dot{t}_1, \dots, \dot{t}_n$ for trace variables $\dot{\pi}_1, \dots, \dot{\pi}_n$ in Eq. (2) where (1) each $\dot{t}_i$ is in the assignment for X_i, which is captured by the automaton $\mathcal{D}_i$ over Σ^{T_i+1}, and (2) the traces $\dot{t}_1, \dots, \dot{t}_n$ satisfy φ_{step}. If this is the case, we want to add $\dot{t}_M$ (as stated in Eq. (2)). We then define $\hat{\mathcal{C}}_{N+1}$ as the union of $\hat{\mathcal{C}}_N$ and $\dot{\mathcal{C}}$, i.e. extend the previous model with all (potentially new) traces that need to be added.

Computing Overapproximations. As we noted above, conditions of the form of Eq. (2) always define fixpoint constraints. To compute upper bounds on such fixpoint constructions we make use of Park's theorem, [48] stating that if we find some set (or automaton) $\mathcal{B}$ that is inductive (i.e., when computing all traces that we would need to add assuming the current model of Y_j is $\mathcal{B}$, we end up with traces that are already in $\mathcal{B}$), then $\mathcal{B}$ overapproximates the unique solution (aka. least fixpoint) of Y_j. To derive such an inductive invariant, we employ techniques developed in the context of regular model checking [15] (see Sect. 7). Concretely, we employ the approach from [19] that uses automata learning [2] to find suitable invariants. While the approach from [19] is limited to finite words, we extend it to an ω-setting by interpreting an automaton accepting finite words as one that accepts an ω-word u iff every prefix of u is accepted.[2] As soon as the learner provides a candidate for an equivalence check, we check that it is inductive and, if not, provide some finite counterexample (see [19] for details). If the automaton is inductive, we return it as a potential overapproximation.

[2] This effectively poses the assumption that the step formula specifies a safety property, which seems to be the case for almost all examples. As an example, common knowledge infers a safety property: In each step, we add all traces for which there exists some trace that agrees on all propositions observed by that agent.

Should this approximation not be precise enough, the first-order model checking (Sect. 5.3) returns some concrete counterexample, i.e., some trace contained in the invariant but violating the property, which we use to provide more counterexamples to the learner.

6 Implementation and Experiments

We have implemented our model-checking algorithm in a prototype tool we call `HySO` (**Hy**perproperties with **S**econd **O**rder).[3] Our tool uses `spot` [29] for basic automata operations (such as LTL-to-NBA translations and complementations). To compute under- and overapproximations, we use the techniques described in Sect. 5.5. We evaluate the algorithm on the following benchmarks.

Muddy Children. The muddy children puzzle [30] is one of the classic examples in common knowledge literature. The puzzle consists of n children standing such that each child can see all other children's faces. From the n children, an unknown number $k \geq 1$ have a muddy forehead, and in incremental rounds, the children should step forward if they know if their face is muddy or not. Consider the scenario of $n = 2$ and $k = 1$, so child a sees that child b has a muddy forehead and child b sees that a is clean. In this case, b immediately steps forward, as it knows that its forehead is muddy since $k \geq 1$. In the next step, a knows that its face is clean since b stepped forward in round 1. In general, one can prove that all children step forward in round k, deriving common knowledge.

For each n we construct a transition system $\mathcal{T}_n$ that encodes the muddy children scenario with n children. For every m we design a Hyper^{2}LTL$_{\text{fp}}$ formula φ_m that adds to the common knowledge set X all traces that appear indistinguishable in the first m steps for some child. We then specify that all traces in X should agree on all inputs, asserting that all inputs are common knowledge.[4] We used `HySO` to *fully automatically* check $\mathcal{T}_n$ against φ_m for varying values of n and m, i.e., we checked if, after the first m steps, the inputs of all children are common knowledge. As expected, the above property holds only if $m \geq n$ (in the worst case, where all children are dirty ($k = n$), the inputs of all children only become common knowledge after n steps). We depict the results in Table 1a.

Asynchronous Hyperproperties. As we have shown in Sect. 4.2, we can encode arbitrary AHLTL properties into Hyper^{2}LTL$_{\text{fp}}$. We verified synchronous and asynchronous version of observational determinism (cf. Sect. 4.2) on programs taken from [3,5,9]. We depict the verification results in Table 1b. Recall that Hyper^{2}LTL$_{\text{fp}}$ properties without any second-order variables correspond to

[3] Our tool is publicly available at https://doi.org/10.5281/zenodo.7877144.

[4] This property is not expressible in non-hyper logics such as LTL$_{\mathsf{K,C}}$, where we can only check *trace properties* on the common knowledge set X. In contrast, Hyper^{2}LTL$_{\text{fp}}$ allows us to check *hyperproperties* on X. That way, we can express that some value is common knowledge (i.e., equal across all traces in the set) and not only that a property is common knowledge (i.e., holds on all traces in the set).

Table 1. In Table 1a, we check common knowledge in the muddy children puzzle for n children and m rounds. We give the result (✓ if common knowledge holds and ✗ if it does not), and the running time. In Table 1a, we check synchronous and asynchronous versions of observational determinism. We depict the number of iterations needed and running time. Times are given in seconds.

		m			
		1	2	3	4
n	2	✗ 0.64	✓ 0.59		
	3	✗ 0.79	✗ 0.75	✓ 0.54	
	4	✗ 2.72	✗ 2.21	✗ 1.67	✓ 1.19

(a)

Instance	**Method**	**Res**	$\boldsymbol{t}$
$\mathcal{T}_{syn}$, φ_{OD}	-	✓	0.26
$\mathcal{T}_{asyn}$, φ_{OD}	-	✗	0.31
$\mathcal{T}_{syn}$, φ_{OD}^{asyn}	Iter (0)	✓	0.50
$\mathcal{T}_{asyn}$,φ_{OD}^{asyn}	Iter (1)	✓	0.78
Q1, φ_{OD}	-	✗	0.34
Q1, φ_{OD}^{asyn}	Iter (1)	✓	0.86

(b)

HyperQPTL formulas. HySO can check such properties precisely, i.e., it constitutes a sound-and-complete model checker for HyperQPTL properties with an arbitrary quantifier prefix. The synchronous version of observational determinism is a HyperLTL property and thus needs no second-order approximation (we set the method column to "-" in these cases).

Common Knowledge in Multi-agent Systems. We used HySO for an automatic analysis of the system in Fig. 1. Here, we verify that on initial trace $\{a\}^n\{d\}^\omega$ it is CK that a holds in the first step. We use a similar formula as the one of Sect. 3.3, with the change that we are interested in whether a is CK (whereas we used $\bigcirc a$ in Sect. 3.3). As expected, HySO requires $2n - 1$ iterations to converge. We depict the results in Table 2a.

Mazurkiewicz Traces. Mazurkiewicz traces are an important concept in the theory of distributed computing [27]. Let $I \subseteq \Sigma \times \Sigma$ be an independence relation that determines when two consecutive letters can be switched (think of two actions in disjoint processes in a distributed system). Any $t \in \Sigma^\omega$ then defines the set of all traces that are equivalent to t by flipping consecutive independent actions an arbitrary number of times (the equivalence class of all these traces is called the Mazurkiewicz Trace). See [27] for details. The verification problem for Mazurkiewicz traces now asks if, given some $t \in \Sigma^\omega$, all traces in the Mazurkiewicz trace of t satisfy some property ψ. Using Hyper^{2}LTL$_{\text{fp}}$ we can directly reason about the Mazurkiewicz Trace of any given trace, by requiring that all traces that are equal up to one swap of independent letters are also in a given set (which is easily expressed in Hyper^{2}LTL$_{\text{fp}}$).

Table 2. In Table 1a, we check common knowledge in the example from Fig. 1 when starting with $a^n d^\omega$ for varying values of n. We depict the number of refinement iterations, the result, and the running time. In Table 2b, we verify various properties on Mazurkiewicz traces. We depict whether the property could be verified or refuted by iteration or automata learning, the result, and the time. Times are given in seconds.

n	Method	Res	t
1	Iter (1)	✓	0.51
2	Iter (3)	✓	0.83
3	Iter (5)	✓	1.20
10	Iter (19)	✓	3.81
100	Iter (199)	✓	102.8

(a)

Instance	Method	Res	t
SwapA	Learn	✓	1.07
SwapATwice	Learn	✓	2.13
SwapA$_5$	Iter (5)	✓	1.15
SwapA$_{15}$	Iter (15)	✓	3.04
SwapAViolation$_5$	Iter (5)	✗	2.35
SwapAViolation$_{15}$	Iter (15)	✗	4.21

(b)

Using HySO we verify a selection of such trace properties that often require non-trivial reasoning by coming up with a suitable invariant. We depict the results in Table 2b. In our preliminary experiments, we model a situation where we start with $\{a\}^1\{\}^\omega$ and can swap letters $\{a\}$ and $\{\}$. We then, e.g., ask if on any trace in the resulting Mazurkiewicz trace, a holds at most once, which requires inductive invariants and cannot be established by iteration.

7 Related Work

In recent years, many logics for the formal specification of hyperproperties have been developed, extending temporal logics with explicit path quantification (examples include HyperLTL, HyperCTL* [20], HyperQPTL [10,45], HyperPDL [38], and HyperATL* [5,9]); or extending first and second-order logics with an equal level predicate [25,33]. Others study (ω)-regular [14,37] and context-free hyperproperties [35]; or discuss hyperproperties over data and modulo theories [24,31]. Hyper^{2}LTL is the first temporal logic that reasons about second-order hyperproperties which allows is to capture many existing (epistemic, asynchronous, etc.) hyperlogics while at the same time taking advantage of model-checking solutions that have been proven successful in first-order settings.

Asynchronous Hyperproperties. For asynchronous hyperproperties, Gutfeld et al. [39] present an asynchronous extension of the polyadic μ-calculus. Bozelli et al. [17] extend HyperLTL with temporal operators that are only evaluated if the truth value of some temporal formula changes. Baumeister et al. present AHLTL [3], that extends HyperLTL with a explicit quantification over trajectories and can be directly encoded within Hyper^{2}LTL$_{\mathrm{fp}}$.

Regular Model Checking. Regular model checking [15] is a general verification method for (possibly infinite state) systems, in which each state of the system is interpreted as a finite word. The transitions of the system are given as a finite-state (regular) transducer, and the model checking problem asks if, from some initial set of states (given as a regular language), some bad state is eventually reachable. Many methods for automated regular model checking have been developed [12,13,19,26]. Hyper^{2}LTL can be seen as a logical foundation for ω-regular model checking: Assume the set of initial states is given as a QPTL formula φ_{init}, the set of bad states is given as a QPTL formula φ_{bad}, and the transition relation is given as a QPTL formula φ_{step} over trace variables π and π'. The set of bad states is reachable from a trace (state) in φ_{init} iff the following Hyper^{2}LTL$_{\mathrm{fp}}$ formula holds on the system that generates all traces:

$$\big(X, \curlyvee, \forall \pi \in \mathfrak{S}.\, \varphi_{init}(\pi) \rightarrow \pi \triangleright X \land \\ \forall \pi \in X. \forall \pi' \in \mathfrak{S}.\, \varphi_{step}(\pi, \pi') \rightarrow \pi' \triangleright X\big). \forall \pi \in X. \neg \varphi_{bad}(\pi)$$

Conversely, Hyper^{2}LTL$_{\mathrm{fp}}$ can express more complex properties, beyond the reachability checks possible in the framework of (ω-)regular model checking.

Model Checking Knowledge. Model checking of knowledge properties in multi-agent systems was developed in the tools `MCK` [36] and `MCMAS` [42], which can exactly express LTL$_{\mathsf{K}}$. Bozzelli et al. [16] have shown that HyperCTL* and LTL$_{\mathsf{K}}$ have incomparable expressiveness, and present HyperCTL$^*_{lp}$ – an extension of HyperCTL* that can reason about past – to unify HyperCTL* and LTL$_{\mathsf{K}}$. While HyperCTL$^*_{lp}$ can express the knowledge operator, it cannot capture common knowledge. LTL$_{\mathsf{K,C}}$ [41] captures both knowledge and common knowledge, but the suggested model-checking algorithm only handles a decidable fragment that is reducible to LTL model checking.

8 Conclusion

Hyperproperties play an increasingly important role in many areas of computer science. There is a strong need for specification languages and verification methods that reason about hyperproperties in a uniform and general manner, similar to what is standard for more traditional notions of safety and reliability. In this paper, we have ventured forward from the first-order reasoning of logics like HyperLTL into the realm of second-order hyperproperties, i.e., properties that not only compare individual traces but reason comprehensively about *sets* of such traces. With Hyper^{2}LTL, we have introduced a natural specification language and a general model-checking approach for second-order hyperproperties. Hyper^{2}LTL provides a general framework for a wide range of relevant hyperproperties, including common knowledge and asynchronous hyperproperties, which could previously only be studied with specialized logics and algorithms. Hyper^{2}LTL also provides a starting point for future work on second-order hyperproperties in areas such as cyber-physical [44] and probabilistic systems [28].

Acknowledgements. We thank Jana Hofmann for the fruitful discussions. This work was supported by the European Research Council (ERC) Grant HYPER (No. 101055412), by DFG grant 389792660 as part of TRR 248 – CPEC, and by the German Israeli Foundation (GIF) Grant No. I-1513-407.2019.

References

1. Alur, R., Henzinger, T.A.: A really temporal logic. J. ACM **41**(1) (1994). https://doi.org/10.1145/174644.174651
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2) (1987). https://doi.org/10.1016/0890-5401(87)90052-6
3. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 694–717. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_33
4. 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, LICS 2022. ACM (2022). https://doi.org/10.1145/3531130.3533369
5. 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
6. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: IEEE Computer Security Foundations Symposium, CSF 2022. IEEE (2022). https://doi.org/10.1109/CSF54842.2022.9919658
7. 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
8. Beutner, R., Finkbeiner, B.: AutoHyper: Explicit-state model checking for HyperLTL. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023, vol. 13993. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_8
9. Beutner, R., Finkbeiner, B.: HyperATL*: A logic for hyperproperties in multi-agent systems. Log. Methods Comput, Sci (2023)
10. Beutner, R., Finkbeiner, B.: Model checking omega-regular hyperproperties with AutoHyperQ. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2023. EPiC Series in Computing, EasyChair (2023)
11. Beutner, R., Finkbeiner, B., Frenkel, H., Metzger, N.: Second-order hyperproperties. CoRR abs/2305.17935 (2023). https://doi.org/10.48550/arXiv.2305.17935, https://doi.org/10.48550/arXiv.2305.17935
12. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_24
13. Boigelot, B., Legay, A., Wolper, P.: Omega-regular model checking. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 561–575. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_41
14. Bonakdarpour, B., Sheinvald, S.: Finite-word hyperlanguages. In: Leporati, A., Martín-Vide, C., Shapira, D., Zandron, C. (eds.) LATA 2021. LNCS, vol. 12638, pp. 173–186. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-68195-1_17

15. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_31
16. Bozzelli, L., Maubert, B., Pinchinat, S.: Unifying hyper and epistemic temporal logics. In: Pitts, A. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 167–182. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46678-0_11
17. Bozzelli, L., Peron, A., Sánchez, C.: Asynchronous extensions of HyperLTL. In: Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470583
18. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Studies in Logic and the Foundations of Mathematics, vol. 44. Elsevier (1966)
19. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: Formal Methods in Computer Aided Design, FMCAD 2017. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102244
20. 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.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
21. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6) (2010). https://doi.org/10.3233/JCS-2009-0393
22. Coenen, N., et al.: Explaining hyperproperty violations. In: International Conference on Computer Aided Verification, CAV 2022. LNCS, vol. 13371. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_20
23. 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
24. Coenen, N., Finkbeiner, B., Hofmann, J., Tillman, J.: Smart contract synthesis modulo hyperproperties. To appear at the 36th IEEE Computer Security Foundations Symposium (CSF 2023) (2023)
25. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 121–139. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_7
26. Dams, D., Lakhnech, Y., Steffen, M.: Iterating transducers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 286–297. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_27
27. Diekert, V., Rozenberg, G. (eds.): The Book of Traces. World Scientific (1995). https://doi.org/10.1142/2563
28. Dimitrova, R., Finkbeiner, B., Torfah, H.: Probabilistic hyperproperties of markov decision processes. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 484–500. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_27
29. Duret-Lutz, A., et al.: 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
30. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (1995). https://doi.org/10.7551/mitpress/5803.001.0001
31. Finkbeiner, B., Frenkel, H., Hofmann, J., Lohse, J.: Automata-based software model checking of hyperproperties. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods, 15th International Symposium, NFM 2023, Houston, TX, USA,

16–18 May 2023, Proceedings. LNCS, vol. 13903. Springer (2023). https://doi.org/10.1007/978-3-031-33170-1_22

32. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
33. Finkbeiner, B., Zimmermann, M.: The first-order logic of hyperproperties. In: Symposium on Theoretical Aspects of Computer Science, STACS 2017. LIPIcs, vol. 66. Schloss Dagstuhl (2017). https://doi.org/10.4230/LIPIcs.STACS.2017.30
34. Fortin, M., Kuijer, L.B., Totzke, P., Zimmermann, M.: HyperLTL satisfiability is Σ_1^1-complete, HyperCTL* satisfiability is Σ_1^2-complete. In: International Symposium on Mathematical Foundations of Computer Science, MFCS 2021. LIPIcs, vol. 202. Schloss Dagstuhl (2021). https://doi.org/10.4230/LIPIcs.MFCS.2021.47
35. Frenkel, H., Sheinvald, S.: Realizable and context-free hyperlanguages. In: Ganty, P., Monica, D.D. (eds.) Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, 21–23 September 2022. EPTCS, vol. 370, pp. 114–130 (2022). https://doi.org/10.4204/EPTCS.370.8, https://doi.org/10.4204/EPTCS.370.8
36. Gammie, P., van der Meyden, R.: MCK: model checking the logic of knowledge. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 479–483. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_41
37. Goudsmid, O., Grumberg, O., Sheinvald, S.: Compositional model checking for multi-properties. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 55–80. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_4
38. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Propositional dynamic logic for hyperproperties. In: International Conference on Concurrency Theory, CONCUR 2020. LIPIcs, vol. 171. Schloss Dagstuhl (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.50
39. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Automata and fixpoints for asynchronous hyperproperties. Proc. ACM Program. Lang. 5(POPL) (2021). https://doi.org/10.1145/3434319
40. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM **37**(3), 549–587 (1990)
41. van der Hoek, W., Wooldridge, M.: Model checking knowledge and time. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 95–111. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46017-9_9
42. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transfer **19**(1), 9–30 (2015). https://doi.org/10.1007/s10009-015-0378-x
43. van der Meyden, R.: Common knowledge and update in finite environments. Inf. Comput. **140**(2) (1998). https://doi.org/10.1006/inco.1997.2679
44. Nguyen, L.V., Kapinski, J., Jin, X., Deshmukh, J.V., Johnson, T.T.: Hyperproperties of real-valued signals. In: ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017. ACM (2017). https://doi.org/10.1145/3127041.3127058
45. Rabe, M.N.: A temporal logic approach to information-flow control. Ph.D. thesis, Saarland University (2016)
46. Sistla, A.P.: Theoretical issues in the design and verification of distributed systems. Ph.D. thesis, Harvard University (1983)

47. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications (1955)
48. Winskel, G.: The formal semantics of programming languages - an introduction. MIT Press, Foundation of computing series (1993)

Neural Networks and Machine Learning

Certifying the Fairness of KNN in the Presence of Dataset Bias

Yannan Li(✉), Jingbo Wang, and Chao Wang

University of Southern California, Los Angeles, CA 90089, USA
{yannanli,jingbow,wang626}@usc.edu

Abstract. We propose a method for certifying the fairness of the classification result of a widely used supervised learning algorithm, the k-nearest neighbors (KNN), under the assumption that the training data may have historical bias caused by systematic mislabeling of samples from a protected minority group. To the best of our knowledge, this is the first certification method for KNN based on three variants of the fairness definition: individual fairness, ϵ-fairness, and label-flipping fairness. We first define the fairness certification problem for KNN and then propose sound approximations of the complex arithmetic computations used in the state-of-the-art KNN algorithm. This is meant to lift the computation results from the concrete domain to an abstract domain, to reduce the computational cost. We show effectiveness of this *abstract interpretation* based technique through experimental evaluation on six datasets widely used in the fairness research literature. We also show that the method is accurate enough to obtain fairness certifications for a large number of test inputs, despite the presence of historical bias in the datasets.

1 Introduction

Certifying the fairness of the classification output of a machine learning model has become an important problem. This is in part due to a growing interest in using machine learning techniques to make socially sensitive decisions in areas such as education, healthcare, finance, and criminal justice systems. One reason why the classification output may be biased against an individual from a protected minority group is because the dataset used to train the model may have historical bias; that is, there is systematic mislabeling of samples from the protected minority group. Thus, we must be extremely careful while considering the possibility of using the classification output of a machine learning model, to avoid perpetuating or even amplifying historical bias.

One solution to this problem is to have the ability to certify, with certainty, that the classification output $y = M(x)$ for an individual input x is fair, despite that the model M is learned from a dataset T with historical bias. This is a

This work was partially funded by the U.S. National Science Foundation grants CNS-1702824, CNS-1813117 and CCF-2220345.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 335–357, 2023.
https://doi.org/10.1007/978-3-031-37703-7_16

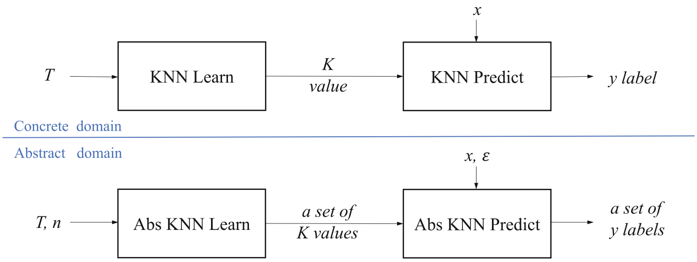


Fig. 1. FairKNN: our method for certifying fairness of KNNs with label bias.

form of *individual fairness* that has been studied in the fairness literature [14]; it requires that the classification output remains the same for input x even if historical bias were not in the training dataset T. However, this is a challenging problem and, to the best of our knowledge, techniques for solving it efficiently are still severely lacking. Our work aims to fill the gap.

Specifically, we are concerned with three variants of the fairness definition. Let the input $x = \langle x_1, \ldots, x_D \rangle$ be a D-dimensional input vector, and $\mathcal{P}$ be the subset of vector indices corresponding to the *protected* attributes (e.g., race, gender, etc.). The first variant of the fairness definition is *individual fairness*, which requires that similar individuals are treated similarly by the machine learning model. For example, if two individual inputs x and x' differ only in some protected attribute x_i, where $i \in \mathcal{P}$, but agree on all the other attributes, the classification output must be the same. The second variant is ϵ-*fairness*, which extends the notion of individual fairness to include inputs whose un-protected attributes differ and yet the difference is bounded by a small constant (ϵ). In other words, if two individual inputs are almost the same in all unprotected attributes, they should also have the same classification output. The third variant is *label-flipping fairness*, which requires the aforementioned fairness requirements to be satisfied even if a biased dataset T has been used to train the model in the first place. That is, as long as the number of mislabeled elements in T is bounded by n, the classification output must be the same.

We want to certify the fairness of the classification output for a popular supervised learning technique called the k-nearest neighbors (KNN) algorithm. Our interest in KNN comes from the fact that, unlike many other machine learning techniques, KNN is a *model-less* technique and thus does not have the high cost associated with training the model. Because of this reason, KNN has been widely adopted in real-world applications [1,4,16,18,23,29,36,45,46]. However, obtaining a fairness certification for KNN is still challenging and, in practice, the most straightforward approach of *enumerating all possible scenarios* and then *checking if the classification outputs obtained in these scenarios agree* would have been prohibitively expensive.

To overcome the challenge, we propose an efficient method based on the idea of *abstract interpretation* [10]. Our method relies on sound approximations to analyze the arithmetic computations used by the state-of-the-art KNN algorithm both accurately and efficiently. Figure 1 shows an overview of our method in the lower half of this figure, which conducts the analysis in an abstract domain, and the default KNN algorithm in the upper half, which operates in the concrete domain. The main difference is that, by staying in the abstract domain, our method is able to analyze a large set of possible training datasets (derived from T due to n label-flips) and a potentially-infinite set of inputs (derived from x due to ϵ perturbation) symbolically, as opposed to analyze a single training dataset and a single input concretely.

To the best of our knowledge, this is the first method for KNN fairness certification in the presence of dataset bias. While Meyer et al. [26,27] and Drews et al. [12] have investigated robustness certification techniques, their methods target decision trees and linear regression, which are different types of machine learning models from KNN. Our method also differs from the KNN data-poisoning robustness verification techniques developed by Jia et al. [20] and Li et al. [24], which do not focus on *fairness* at all; for example, they do not distinguish *protected* attributes from *unprotected* attributes. Furthermore, Jia et al. [20] consider the prediction step only while ignoring the learning step, and Li et al. [24] do not consider label flipping. Our method, in contrast, considers all of these cases.

We have implemented our method and demonstrated the effectiveness through experimental evaluation. We used all of the six popular datasets in the fairness research literature as benchmarks. Our evaluation results show that the proposed method is efficient in analyzing complex arithmetic computations used in the state-of-the-art KNN algorithm, and is accurate enough to obtain fairness certifications for a large number of test inputs. To better understand the impact of historical bias, we also compared the fairness certification success rates across different demographic groups.

To summarize, this paper makes the following contributions:

- We propose an abstract interpretation based method for efficiently certifying the fairness of KNN classification results in the presence of dataset bias. The method relies on sound approximations to speed up the analysis of both the learning and the prediction steps of the state-of-the-art KNN algorithm, and is able to handle three variants of the fairness definition.
- We implement the method and evaluate it on six datasets that are widely used in the fairness literature, to demonstrate the efficiency of our approximation techniques as well as the effectiveness of our method in obtaining sound fairness certifications for a large number of test inputs.

The remainder of this paper is organized as follows. We first present the technical background in Sect. 2 and then give an overview of our method in Sect. 3. Next, we present our detailed algorithms for certifying the KNN prediction step in Sect. 4 and certifying the KNN learning step in Sect. 5. This is followed by our experimental results in Sect. 6. We review the related work in Sect. 7 and, finally, give our conclusion in Sect. 8.

2 Background

Let L be a supervised learning algorithm that takes the training dataset T as input and returns a learned model $M = L(T)$ as output. The training set $T = \{(x, y)\}$ is a set of labeled samples, where each $x \in \mathcal{X} \subseteq \mathbb{R}^D$ has D real-valued attributes, and the $y \in \mathcal{Y} \subseteq \mathbb{N}$ is a class label. The learned model $M : \mathcal{X} \rightarrow \mathcal{Y}$ is a function that returns the classification output $y' \in \mathcal{Y}$ for any input $x' \in \mathcal{X}$.

2.1 Fairness of the Learned Model

We are concerned with *fairness* of the classification output $M(x)$ for an individual input x. Let $\mathcal{P}$ be the set of vector indices corresponding to the protected attributes in $x \in \mathcal{X}$. We say that x_i is a protected attribute (e.g., race, gender, etc.) if and only if $i \in \mathcal{P}$.

Definition 1 (Individual Fairness). *For an input x, the classification output $M(x)$ is fair if, for any input x' such that (1) $x_j \neq x'_j$ for some $j \in \mathcal{P}$ and (2) $x_i = x'_i$ for all $i \notin \mathcal{P}$, we have $M(x) = M(x')$.*

It means two individuals (x and x') differing only in some protected attribute (e.g., gender) but agreeing on all other attributes must be treated equally. While being intuitive and useful, this notion of fairness may be too narrow. For example, if two individuals differ in some unprotected attributes and yet the difference is considered *immaterial*, they must still be treated equally. This can be captured by $\epsilon-$fairness.

Definition 2 (ϵ-Fairness). *For an input x, the classification output $M(x)$ is fair if, for any input x' such that (1) $x_j \neq x'_j$ for some $j \in \mathcal{P}$ and (2) $|x_i - x'_i| \leq \epsilon$ for all $i \notin \mathcal{P}$, we have $M(x) = M(x')$.*

In this case, such inputs x' form a set. Let $\Delta^\epsilon(x)$ be the set of all inputs x' considered in the $\epsilon-$fairness definition. That is, $\Delta^\epsilon(x) := \{x' \mid x_j \neq x'_j$ for some $j \in \mathcal{P}, |x_i - x'_i| \leq \epsilon$ for all $i \notin \mathcal{P}\}$. By requiring $M(x) = M(x')$ for all $x' \in \Delta^\epsilon(x)$, ϵ-fairness guarantees that a larger set of individuals similar to x are treated equally.

Individual fairness can be viewed as a special case of ϵ-fairness, where $\epsilon = 0$. In contrast, when $\epsilon > 0$, the number of elements in $\Delta^\epsilon(x)$ is often large and sometimes infinite. Therefore, the most straightforward approach of certifying fairness by enumerating all possible elements in $\Delta^\epsilon(x)$ would not work. Instead, any practical solution would have to rely on abstraction.

2.2 Fairness in the Presence of Dataset Bias

Due to historical bias, the training dataset T may have contained samples whose output are unfairly labeled. Let the number of such samples be bounded by n.

We assume that there are no additional clues available to help identify the mislabeled samples. Without knowing which these samples are, fairness certification must consider all of the possible scenarios. Each scenario corresponds to a de-biased dataset, T', constructed by flipping back the incorrect labels in T. Let $\mathsf{dBias}_n(T) = \{T'\}$ be the set of these possible de-biased (clean) datasets. Ideally, we want all of them to lead to the same classification output.

Definition 3 (Label-flipping Fairness). *For an input x, the classification output $M(x)$ is fair against label-flipping bias of at most n elements in the dataset T if, for all $T' \in \mathsf{dBias}_n(T)$, we have $M'(x) = M(x)$ where $M' = L(T')$.*

Label-flipping fairness differs from and yet complements individual and ϵ-fairness in the following sense. While individual and ϵ-fairness guarantee equal output for similar inputs, label-flipping fairness guarantees equal output for similar datasets. Both aspects of fairness are practically important. By combining them, we are able to define the entire problem of certifying fairness in the presence of historical bias.

To understand the complexity of the fairness certification problem, we need to look at the size of the set $\mathsf{dBias}_n(T)$, similar to how we have analyzed the size of $\Delta^{\epsilon}(x)$. While the size of $\mathsf{dBias}_n(T)$ is always finite, it can be astronomically large in practice. Let q is the number of unique class labels and m be the actual number of flipped elements in T. Assuming that each flipped label may take any of the other $q-1$ possible labels, the total number of possible *clean sets* is $\binom{|T|}{m} \cdot (q-1)^m$ for each m. Since $m \leq n$, $|\mathsf{dBias}_n(T)| = \sum_{m=1}^{n} \binom{|T|}{m} \cdot (q-1)^m$. Again, the number of elements in $\mathsf{dBias}_n(T)$ is too large to enumerate, which means any practical solution would have to rely on abstraction.

3 Overview of Our Method

Given the tuple $\langle T, \mathcal{P}, n, \epsilon, x \rangle$, where T is the training set, $\mathcal{P}$ represents the protected attributes, n bounds the number of biased elements in T, and ϵ bounds the perturbation of x, our method checks if the KNN classification output for x is fair.

3.1 The KNN Algorithm

Since our method relies on an *abstract interpretation* of the KNN algorithm, we first explain how the KNN algorithm operates in the concrete domain (this subsection), and then lift it to the abstract domain in the next subsection.

As shown in Fig. 2, KNN has a prediction step where `KNN_predict` computes the output label for an input x using T and a given parameter K, and a learning step where `KNN_learn` computes the K value from the training set T.

Unlike many other machine learning techniques, KNN does not have an explicit model M; instead, M can be regarded as the combination of T and K.

```
1  func KNN_predict($T, K, x$) {
2     Let $T_x^K$ = the $K$ nearest neighbors of $x$ in $T$;
3     Let $Freq(T_x^K)$ = the most frequent label in $T_x^K$;
4     return $Freq(T_x^K)$;
5  }
6
7  func KNN_learn($T$) {
8     for (each candidate $k$ value) { // conducting p-fold cross validation
9        Let $\{G_i\}$ = a partition of $T$ into $p$ groups of roughly equal size;
10       Let $err_i^k$ = $\{(x, y) \in G_i \mid y \neq$ KNN_predict($T \setminus G_i, k, x$)$\}$ for each $G_i$;
11    }
12    Let $K$ = $\arg\min_k \frac{1}{p} \sum_{i=1}^{p} \frac{|err_i^k|}{|G_i|}$;
13    return $K$;
14 }
```

Fig. 2. The KNN algorithm, consisting of the prediction and learning steps.

Inside `KNN_predict`, the set T_x^K represents the K-nearest neighbors of x in the dataset T, where distance is measured by Euclidean (or Manhattan) distance in the input vector space. $Freq(T_x^K)$ is the most frequent label in T_x^K.

Inside `KNN_learn`, a technique called *p-fold cross validation* is used to select the optimal value for K, e.g., from a set of candidate k values in the range $[1, |T| \times (p-1)/p]$ by minimizing classification error, as shown in Line 12. This is accomplished by first partitioning T into p groups of roughly equal size (Line 9), and then computing err_i^k (a set of misclassified samples from G_i) by treating G_i as the evaluation set, and $T \setminus G_i$ as the training set. Here, an input $(x, y) \in G_i$ is "misclassified" if the expected output label, y, differs from the output of `KNN_predict` using the candidate k value.

3.2 Certifying the KNN Algorithm

Algorithm 1 shows the top-level procedure of our fairness certification method, which first executes the KNN algorithm in the concrete domain (Lines 1–2), to obtain the default K and y, and then starts our analysis in the abstract domain.

Algorithm 1: Our method for certifying fairness of KNN for input x.

```
1 $K$ = KNN_learn($T$);
2 $y$ = KNN_predict($T, K, x$);
3 $KSet$ = abs_KNN_learn($T, n$);
4 for each $K \in KSet$ do
5     if abs_KNN_predict_same($T, n, K, x, y$) $= False$ then
6         return unknown;
7     end if
8 end for
9 return certified;
```

In the *abstract* learning step (Line 3), instead of considering T, our method considers the set of all clean datasets in $\mathsf{dBias}_n(T)$ symbolically, to compute the set of possible optimal K values, denoted $KSet$.

In the *abstract* prediction step (Lines 4–8), for each K, instead of considering input x, our method considers all perturbed inputs in $\Delta^{\epsilon}(x)$ and all clean datasets in $\mathsf{dBias}_n(T)$ symbolically, to check if the classification output always stays the same. Our method returns "certified" only when the classification output always stays the same (Line 9); otherwise, it returns "unknown" (Line 6).

We only perturb numerical attributes in the input x since perturbing categorical or binary attributes often does not make sense in practice.

In the next two sections, we present our detailed algorithms for abstracting the prediction step and the learning step, respectively.

4 Abstracting the KNN Prediction Step

We start with abstract KNN prediction, which is captured by the subroutine *abs_KNN_predict_same* used in Line 5 of Algorithm 1. It consists of two parts. The first part (to be presented in Sect. 4.1) computes a superset of T_x^K, denoted $overNN$, while considering the impact of ϵ perturbation of the input x. The second part (to be presented in Sect. 4.2) leverages $overNN$ to decide if the classification output always stays the same, while considering the impact of label-flipping bias in the dataset T.

4.1 Finding the *K*-Nearest Neighbors

To compute $overNN$, which is a set of samples in T that *may be* the K nearest neighbors of the test input x, we must be able to compute the distance between x and each sample in T.

This is not a problem at all in the concrete domain, since the K nearest neighbors of x in T, denoted T_x^K, is fixed and is determined solely by the Euclidean distance between x and each sample in T in the attribute space. However, when ϵ perturbation is applied to x, the distance changes and, as a result, the K nearest neighbors of x may also change.

Fortunately, the distance in the attribute space is not affected by label-flipping bias in the dataset T, since label-flipping only impacts sample labels, not sample attributes. Thus, in this subsection, we only need to consider the impact of ϵ perturbation of the input x.

The Challenge. Due to ϵ perturbation, a single test input x becomes a potentially-infinite set of inputs $\Delta^{\epsilon}(x)$. Since our goal is to over-approximate the K nearest neighbors of $\Delta^{\epsilon}(x)$, the expectation is that, as long as there exists some $x' \in \Delta^{\epsilon}(x)$ such that a sample input t in T is one of the K nearest neighbors of x', denoted $t \in T_{x'}^K$, we must include t in the set $overNN$. That is,

$$\bigcup_{x' \in \Delta^{\epsilon}(x)} T_{x'}^K \subseteq overNN \subseteq T.$$

However, finding an efficient way of computing $overNN$ is a challenging task. As explained before, the naive approach of enumerating $x' \in \Delta^{\epsilon}(x)$, computing the K nearest neighbors, $T^{K}_{x'}$, and unionizing all of them would not work. Instead, we need abstraction that is both efficient and accurate enough in practice.

Our solution is that, for each sample t in T, we first analyze the distances between t and all inputs in $\Delta^{\epsilon}(x)$ symbolically, to compute a lower bound and an upper bound of the distances. Then, we leverage these lower and upper bounds to compute the set $overNN$, which is a superset of samples in T that may become the K nearest neighbors of $\Delta^{\epsilon}(x)$.

Bounding Distance Between $\Delta^{\epsilon}(x)$ and t. Assume that $x = (x_1, x_2, ..., x_D)$ and $t = (t_1, t_2, ..., t_D)$ are two real-valued vectors in the D-dimensional attribute space. Let $\epsilon = (\epsilon_1, \epsilon_2, ..., \epsilon_D)$, where $\epsilon_i \geq 0$, be the small perturbation. Thus, the perturbed input is $x' = (x'_1, x'_2, ..., x'_D) = (x_1 + \delta_1, x_2 + \delta_2, ..., x_D + \delta_D)$, where $\delta_i \in [-\epsilon_i, \epsilon_i]$ for all $i = 1, ..., D$.

The distance between x and t is a fixed value $d(x,t) = \sqrt{\sum_{i=1}^{D}(x_i - t_i)^2}$, since both x and the samples t in T are fixed, but the distance between $x' \in \Delta^{\epsilon}(x)$ and t is a function of $\delta_i \in [-\epsilon_i, \epsilon_i]$, since $\sqrt{\sum_{i=1}^{D}(x'_i - t_i)^2} = \sqrt{\sum_{i=1}^{D}(x_i - t_i + \delta_i)^2}$. For ease of presentation, we define the distance as $d^{\epsilon} = \sqrt{\sum_{i=1}^{D} d^{\epsilon}_i}$, where $d^{\epsilon}_i = (x_i - t_i + \delta_i)^2$ is the (squared) distance function in the i-th dimension. Then, our goal becomes computing the lower bound, $LB(d^{\epsilon})$, and the upper bound, $UB(d^{\epsilon})$, in the domain $\delta_i \in [-\epsilon_i, \epsilon_i]$ for all $i = 1, ..., D$.

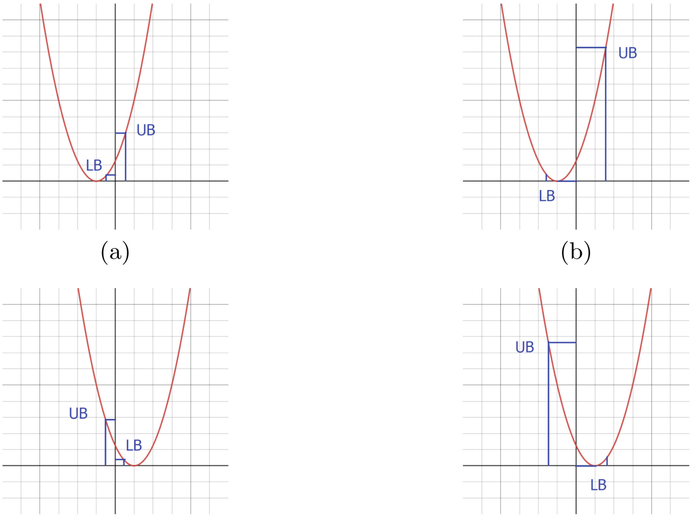
Distance Bounds Are Compositional. Our first observation is that bounds on the distance d^{ϵ} as a whole can be computed using bounds in the individual dimensions. To see why this is the case, consider the (square) distance in the i-th dimension, $d^{\epsilon}_i = (x_i - t_i + \delta_i)^2$, where $\delta_i \in [-\epsilon_i, \epsilon_i]$, and the (square) distance in the j-th dimension, $d^{\epsilon}_j = (x_j - t_j + \delta_j)^2$, where $\delta_j \in [-\epsilon_j, \epsilon_j]$. By definition, d^{ϵ}_i is completely independent of d^{ϵ}_j when $i \neq j$.

Thus, the lower bound of d^{ϵ}, denoted $LB(d^{\epsilon})$, can be calculated by finding the lower bound of each d^{ϵ}_i in the i-th dimension. Similarly, the upper bound of d^{ϵ}, denoted $UB(d^{\epsilon})$, can also be calculated by finding the upper bound of each d^{ϵ}_i in the i-the dimension. That is,

$$LB(d^{\epsilon}) = \sqrt{\sum_{i=1}^{D} LB(d^{\epsilon}_i)} \text{ and } UB(d^{\epsilon}) = \sqrt{\sum_{i=1}^{D} UB(d^{\epsilon}_i)}.$$

Four Cases in Each Dimension. Our second observation is that, by utilizing the mathematical nature of the (square) distance function, we can calculate the minimum and maximum values of d^{ϵ}_i, which can then be used as the lower bound $LB(d^{\epsilon}_i)$ and upper bound $UB(d^{\epsilon}_i)$, respectively.

Specifically, in the i-th dimension, the (square) distance function $d^{\epsilon}_i = ((x_i - t_i) + \delta_i)^2$ may be rewritten to $(\delta_i + A)^2$, where $A = (x_i - t_i)$ is a constant and $\delta_i \in [-\epsilon, +\epsilon]$ is a variable. The function can be plotted in two dimensional space, using δ_i as x-axis and the output of the function as y-axis; thus, it is a quadratic function $Y = (X + A)^2$.



(a) (b) (c) (d)

Fig. 3. Four cases for computing the upper and lower bounds of the distance function $d_i^\epsilon(\delta_i) = (\delta_i + A)^2$ for $\delta_i \in [-\epsilon_i, \epsilon_i]$. In these figures, δ_i is the x-axis, and d_i^ϵ is the y-axis, LB denotes $LB(d_i^\epsilon)$, and UB denotes $UB(d_i^\epsilon)$.

Figure 3 shows the plot, which reminds us of where the minimum and maximum values of a quadratic function is. There are two versions of the quadratic function, depending on whether $A > 0$ (corresponding to the two subfigures at the top) or $A < 0$ (corresponding to the two subfigures at the bottom). Each version also has two cases, depending on whether the perturbation interval $[-\epsilon_i, \epsilon_i]$ falls inside the constant interval $[-|A|, |A|]$ (corresponding to the two subfigures on the left) or falls outside (corresponding to the two subfigures on the right). Thus, there are four cases in total.

In each case, the maximal and minimal values of the quadratic function are different, as shown by the LB and UB marks in Fig. 3.

Case (a). This is when $(x_i - t_i) > 0$ and $-\epsilon_i > -(x_i - t_i)$, which is the same as saying $A > 0$ and $-\epsilon_i > -A$. In this case, function $d_i(\epsilon_i) = (\delta_i + A)^2$ is monotonically increasing w.r.t. variable $\delta_i \in [-\epsilon_i, +\epsilon_i]$.

Thus, $LB(d_i^\epsilon) = (-\epsilon_i + (x_i - t_i))^2$ and $UB(d_i^\epsilon) = (+\epsilon_i + (x_i - t_i))^2$.

Case (b). This is when $(x_i - t_i) > 0$ and $-\epsilon_i < -(x_i - t_i)$, which is the same as saying $A > 0$ and $-\epsilon_i < -A$. In this case, the function is not monotonic. The minimal value is 0, obtained when $\delta_i = -A$. The maximal value is obtained when $\delta_i = +\epsilon_i$.

Thus, $LB(d_i^\epsilon) = 0$ and $UB(d_i^\epsilon) = (+\epsilon_i + (x_i - t_i))^2$.

Case (c). This is when $(x_i - t_i) < 0$ and $\epsilon_i < -(x_i - t_i)$, which is the same as saying $A < 0$ and $\epsilon_i < -A$. In this case, the function is monotonically decreasing w.r.t. variable $\delta_i \in [-\epsilon_i, \epsilon_i]$.

Thus, $LB(d_i^\epsilon) = (\epsilon_i + (x_i - t_i))^2$ and $UB(d_i^\epsilon) = (-\epsilon_i + (x_i - t_i))^2$.

Case (d). This is when $(x_i - t_i) < 0$ and $\epsilon_i > -(x_i - t_i)$, which is the same as saying $A < 0$ and $\epsilon_i > -A$. In this case, the function is not monotonic. The minimal value is 0, obtained when $\delta_i = -A$. The maximal value is obtained when $\delta_i = -\epsilon_i$.

Thus, $LB(d_i^\epsilon) = 0$ and $UB(d_i^\epsilon) = (-\epsilon_i + (x_i - t_i))^2$.

Summary. By combining the above four cases, we compute the bounds of the entire distance function d^ϵ as follows:

$$\left[\ \sqrt{\sum_{i=1}^{D} \max(|x_i - t_i| - \epsilon_i, 0)^2}, \quad \sqrt{\sum_{i=1}^{D} (|x_i - t_i| + \epsilon_i)^2}\ \right]$$

Here, the take-away message is that, since x_i, t_i and ϵ_i are all fixed values, the upper and lower bounds can be computed in constant time, despite that there is a potentially-infinite number of inputs in $\Delta^\epsilon(x)$.

Computing *overNN* **Using Bounds.** With the upper and lower bounds of the distance between $\Delta^\epsilon(x)$ and sample t in the dataset T, denoted $[LB(d^\epsilon(x,t)), UB(d^\epsilon(x,t))]$, we are ready to compute *overNN* such that every $t \in overNN$ *may be* among the K nearest neighbors of $\Delta^\epsilon(x)$.

Let UB_{Kmin} denote the K-th minimum value of $UB(d^\epsilon(x,t))$ for all $t \in T$. Then, we define *overNN* as the set of samples in T whose $LB(d^\epsilon(x,t))$ is *not greater than* UB_{Kmin}. In other words,

$$overNN = \{t \in T \mid LB(d^\epsilon(x,t)) \leq UB_{Kmin}\}.$$

Example. Given a dataset $T = \{t^1, t^2, t^3, t^4, t^5\}$, a test input x, perturbation ϵ, and $K = 3$. Assume that the lower and upper bounds of the distance between $\Delta^\epsilon(x)$ and samples in T are $[25.4, 29.4]$, $[30.1, 34.1]$, $[35.3, 39.3]$, $[37.2, 41.2]$, $[85.5, 90.5]$. Since $K = 3$, we find the 3rd minimum upper bound, $UB_{3min} = 39.3$. By comparing UB_{3min} with the lower bounds, we compute $overNN_3 = \{t^1, t^2, t^3, t^4\}$, since t^5 is the only sample in T whose lower bound is greater than 39.3. All the other four samples *may be* among the 3 nearest neighbors of $\Delta^\epsilon(x)$.

Due to ϵ perturbation, the set $overNN_3$ for $K = 3$ is expected to contain 3 or more samples. That is, since different inputs in $\Delta^\epsilon(x)$ may have different samples as their 3-nearest neighbors, to be conservative, we have to take the union of all possible sets of 3-nearest neighbors.

Algorithm 2: Subroutine `abs_same_label`($overNN, K, y$).

```
1 Let $S$ be a subset of $overNN$ obtained by removing all $y$-labeled elements;
2 Let $y' = Freq(S)$, and $\#y'$ be the count of $y'$-labeled elements in $S$;
3 if $\#y' < K - |S| - 2 * n$ then
4     return $True$;
5 end if
6 return $False$;
```

Soundness Proof. Here we prove that any $t' \notin overNN_K$ cannot be among the K nearest neighbors of any $x' \in \Delta^{\epsilon}(x)$. Since UB_{Kmin} is the K-th minimum $UB(d^{\epsilon}(x,t))$ for all $t \in T$, there must be samples $t^1, t^2, ...t^K$ such that $UB(d^{\epsilon}(x,t^i)) \leq UB_{Kmin}$ for all $i = 1, 2, ...K$. For any $t' \notin overNN$, we have $LB(d^{\epsilon}(x,t')) > UB_{Kmin}$.

Combining the above conditions, we have $LB(d^{\epsilon}(x,t')) > UB(d^{\epsilon}(x,t^i))$ for $i = 1, 2, ...K$. It means at least K other samples are closer to x than t'. Thus, t' cannot be among the K-nearest neighbors of x'.

4.2 Checking the Classification Result

Next, we try to certify that, regardless of which of the K elements are selected from $overNN$, the prediction result obtained using them is always the same.

The prediction label is affected by both ϵ perturbation of the input x and label-flipping bias in the dataset T. Since ϵ perturbation affects which points are identified as the K nearest neighbors, and its impact has been accounted for by $overNN$, from now on, we focus only on label-flipping bias in T.

Our method is shown in Algorithm 2, which takes the set $overNN$, the parameter K, and the expected label y as input, and checks if it is possible to find a subset of $overNN$ with size K, whose most frequent label differs from y. If such a "bad" subset cannot be found, we say that KNN prediction always returns the same label.

To try to find such a "bad" subset of $overNN$, we first remove all elements labeled with y from $overNN$, to obtain the set S (Line 1). After that, there are two cases to consider.

1. If the size of S is equal to or greater than K, then any subset of S with size K must have a different label because it will not contain any element labeled with y. Thus, the condition in Line 3 of Algorithm 2 is not satisfied ($\#y'$ is a positive number, and right-hand side is a negative number), and the procedure returns $False$.
2. If the size of S, denoted $|S|$, is smaller than K, the most likely "bad" subset will be $S_K = S \cup \{$ any $(K - |S|)$ y-labeled elements from $overNN\}$. In this case, we need to check if the most frequent label in S_K is y or not.

In S_K, the most frequent label must be either y (whose count is $K - |S|$) or y' (which is the most frequent label in S, with the count $\#y'$). Moreover, since we can flip up to n labels, we can flip n elements from label y to label y'.

Algorithm 3: Subroutine $abs\_KNN\_learn(T, n)$

```
1 for each candidate $k$ value do
2     Let $\{G_i\}$ = a partition of $T$ into $p$ groups of roughly equal size;
3     $errUB_i^k = \{(x, y) \in G_i \mid \texttt{abs\_may\_err}\ (T \setminus G_i, n, k, x, y) = true\}$ for each $G_i$;
4     $errLB_i^k = \{(x, y) \in G_i \mid \texttt{abs\_must\_err}(T \setminus G_i, n, k, x, y) = true\}$ for each $G_i$;
5     $UB_k = \frac{1}{p} \sum_{i=1}^{p} |errUB_i^k| / |G_i|$;
6     $LB_k = \frac{1}{p} \sum_{i=1}^{p} |errLB_i^k| / |G_i|$;
7 end for
8 Let $minUB = \min(\{UB_1, ..., UB_p\})$;
9 return $KSet = \{k \mid LB_k \leq minUB\}$;
```

Therefore, to check if our method should return $True$, meaning the prediction result is guaranteed to be the same as label y, we only need to compare $K - |S|$ with $\#y' + 2 * n$. This is checked using the condition in Line 3 of Algorithm 2.

5 Abstracting the KNN Learning Step

In this section, we present our method for abstracting the learning step, which computes the optimal K value based on T and the impact of flipping at most n labels. The output is a super set of possible optimal K values, denoted $KSet$.

Algorithm 3 shows our method, which takes the training set T and parameter n as input, and returns $KSet$ as output. To be sound, we require the $KSet$ to include any candidate k value that may become the optimal K for some clean set $T' \in \mathsf{dBias}_n(T)$.

In Algorithm 3, our method first computes the lower and upper bounds of the classification error for each k value, denoted LB_k and UB_k, as shown in Lines 5–6. Next, it computes $minUB$, which is the minimal upper bound for all candidate k values (Line 8). Finally, by comparing $minUB$ with LB_k for each candidate k value, our method decides whether this candidate k value should be put into $KSet$ (Line 9).

We will explain the steps needed to compute LB_k and UB_k in the remainder of this section. For now, assuming that they are available, we explain how they are used to compute $KSet$.

Example. Given the candidate k values, k_1, k_2, k_3, k_4, and their error bounds $[0.1, 0.2]$, $[0.1, 0.3]$, $[0.3, 0.4]$, $[0.3, 0.5]$. The smallest upper bound is $minUB = 0.2$. By comparing $minUB$ with the lower bounds, we compute $KSet = \{k_1, k_2\}$, since only LB_{k_1} and LB_{k_2} are lower than or equal to $minUB$.

Soundness Proof. Here we prove that any $k' \notin KSet$ cannot result in the smallest classification error. Assume that k_s is the candidate k value that has the minimal upper bound ($minUB$), and err_{k_s} is the actual classification error. By definition, we have $err_{k_s} \leq minUB$. Meanwhile, for any $k' \notin KSet$, we have $LB_{k'} >$

Algorithm 4: Subroutine `abs_may_err`(T, n, K, x, y).

```
1 Let $y'$ be, among the non-$y$ labels, the label with the highest count in $T_x^K$;
2 Let $\#y$ be the number of elements in $T_x^K$ with the $y$ label;
3 Let $n'$ be $\min(n, \#y \in T_x^K)$;
4 Changing $n'$ elements in $T_x^K$ from $y$ label to $y'$ label;
5 return $Freq(T_x^K) \neq y$;
```

$minUB$. Combining the two cases, we have $err_{k'} > minUB \geq err_{k_s}$. Here, $err_{k'} > err_{k_s}$ means that k' cannot result in the smallest classification error.

5.1 Overapproximating the Classification Error

To compute the upper bound $errUB_i^k$ defined in Line 3 of Algorithm 3, we use the subroutine `abs_may_err` to check if $(x, y) \in G_i$ may be misclassified when using $T \setminus G_i$ as the training set.

Algorithm 4 shows the implementation of the subroutine, which checks, for a sample (x, y), whether it is possible to obtain a set S by flipping at most n labels in T_x^K such that the most frequent label in S is not y. If it is possible to obtain such a set S, we conclude that the prediction label for x may be an error.

The condition $Freq(T_x^K) \neq y$, computed on T_x^K after the y label of n' elements is changed to y' label, is a sufficient condition under which the prediction label for x may be an error. The rationale is as follows.

In order to make the most frequent label in the set T_x^K different from y, we need to focus on the label most likely to become the new most frequent label. It is the label $y'(\neq y)$ with the highest count in the current T_x^K.

Therefore, Algorithm 4 checks whether y' can become the most frequent label by changing at most n elements in T_x^K from y label to y' label (Lines 3–5).

5.2 Underapproximating the Classification Error

To compute the lower bound $errLB_i^k$ defined in Line 4 of Algorithm 3, we use the subroutine `abs_must_err` to check if $(x, y) \in G_i$ must be misclassified when using $T \setminus G_i$ as the training set.

Algorithm 5 shows the implementation of the subroutine, which checks, for a sample (x, y), whether it is impossible to obtain a set S by flipping at most n labels in T_x^K such that the most frequent label in S is y. In other words, is it impossible to avoid the classification error? If it is impossible to avoid the classification error, we conclude that the prediction label must be an error, and thus the procedure returns $True$

In this sense, all samples in $errLB_i^k$ (computed in Line 4 of Algorithm 3 are guaranteed to be misclassified.

Algorithm 5: Subroutine `abs_must_err`(T, n, K, x, y).

```
1 if $\exists S$ obtained from $T_x^K$ by flipping up to $n$ labels such that $Freq(S) = y$ then
2     return $False$;
3 end if
4 return $True$;
```

The challenge in Algorithm 5 is to check if such a set S can be constructed from T_x^K. The intuition is that, to make y the most frequent label, we should flip the labels of non-y elements to label y. Let us consider two examples first.

Example 1. Given the label counts of T_x^K, denoted $\{l_1 * 4,\ l_4 * 4,\ l_3 * 2\}$, meaning that 4 elements are labeled l_1, 4 elements are labeled l_4, and 2 elements are labeled l_3. Assume that $n = 2$ and $y = l_3$. Since we can flip at most 2 elements, we choose to flip one $l_1 \rightarrow l_3$ and one $l_4 \rightarrow l_3$, to get a set $S = \{l_1 * 3,\ l_4 * 3,\ l_3 * 4\}$.

Example 2. Given the label counts of T_x^K, denoted $\{l_1 * 5,\ l_4 * 3,\ l_3 * 2\}$, $n = 2$, and $y = l_3$. We can flip two $l_1 \rightarrow l_3$ to get a set $S = \{l_1 * 3,\ l_4 * 3,\ l_3 * 4\}$.

The LP Problem. The question is how to decide whether the set S (defined in Line 1 of Algorithm 5) exists. We can formulate it as a linear programming (LP) problem. The LP problem has two constraints. The first one is defined as follows: Let y be the expected label, $l_i \neq y$ be another label, where $i = 1, ..., q$ and q is the total number of class labels (e.g., in the above two examples, the number $q = 3$). Let $\#y$ be the number of elements in T_x^K that have the y label. Similarly, let $\#l_i$ be the number of elements with l_i label. Assume that a set S as defined in Algorithm 5 exists, then all of the labels $l_i \neq y$ must satisfy

$$\#l_i - \#flip_i < \#y + \sum_{i=1}^{q} \#flip_i \quad , \tag{1}$$

where $\#flip_i$ is a variable representing the number of l_i–to–y flips. Thus, in the above formula, the left-hand side is the count of l_i after flipping, the right-hand side is the count of y after flipping. Since y is the most frequent label in S, y should have a higher count than any other label.

The second constraint is

$$\sum_{i=1}^{q} \#flip_i \leq n \quad , \tag{2}$$

which says that the total number of label flips is bounded by the parameter n.

Since the number of class labels (q) is often small (from 2 to 10), this LP problem can be solved quickly. However, the LP problem must be solved $|T|$ times, where $|T|$ may be as large as 50,000. To avoid invoking the LP solver unnecessarily, we propose two easy-to-check conditions. They are *necessary* condition in that, if either of them is violated, the set S does not exist. Thus, we invoke the LP solver only if both conditions are satisfied.

Necessary Conditions. The first condition is derived from Formula (1a), by adding up the two sides of the inequality constraint for all labels $l_i \neq y$. The resulting condition is

$$\left(\sum_{l_i \neq y} \#l_i - \sum_{i=1}^{q} \#flip_i \right) < \left((q-1)\#y + (q-1) \sum_{i=1}^{q} \#flip_i \right).$$

The second condition requires that, in S, label y has a higher count (after flipping) than any other label, including the label $l_p \neq y$ with the highest count in the current T_x^K. The resulting condition is

$$(\#l_p - \#y)/2 < n,$$

since only when this condition is satisfied, it is possible to allow y to have a higher count than l_p, by flipping at most n of the label l_p to y.

These are necessary conditions (but may not be sufficient conditions) because, whenever the first condition does not hold, Eq. (1) does not hold either. Similarly, whenever the second condition does not hold, Eq. (1) does not hold either. In this sense, these two conditions are *easy-to-check* over-approximations of Eq. (1).

6 Experiments

We have implemented our method as a software tool written in Python using the **scikit-learn** machine learning library. We evaluated our tool on six datasets that are widely used in the fairness research literature.

Datasets. Table 1 shows the statistics of each dataset, including the name, a short description, the size ($|T|$), the number of attributes, the protected attributes, and the parameters ϵ and n. The value of ϵ is set to 1% of the attribute range. The bias parameter n is set to 1 for small datasets, 10 for medium datasets, and 50 for large datasets. The protected attributes include *Gender* for all six datasets, and *Race* for two datasets, *Compas* and *Adult*, which are consistent with known biases in these datasets.

In preparation for the experimental evaluation, we have employed state-of-the-art techniques in the machine learning literature to preprocess and balance the datasets for KNN, including encoding, standard scaling, k-bins-discretizer, downsampling and upweighting.

Table 1. Statistics of all of the datasets used during our experimental evaluation.

Dataset	Description	Size $\lvert T\rvert$	# Attr.	Protected Attr.	Parameters ϵ and n
Salary	salary level [42]	52	4	Gender	$\epsilon = 1\%$ attribute range, $n = 1$
Student	academic performance [9]	649	30	Gender	$\epsilon = 1\%$ attribute range, $n = 1$
German	credit risk [13]	1,000	20	Gender	$\epsilon = 1\%$ attribute range, $n = 10$
Compas	recidivism risk [11]	10,500	16	Race+Gender	$\epsilon = 1\%$ attribute range, $n = 10$
Default	loan default risk [47]	30,000	36	Gender	$\epsilon = 1\%$ attribute range, $n = 50$
Adult	earning power [13]	48,842	14	Race+Gender	$\epsilon = 1\%$ attribute range, $n = 50$

Table 2. Results for certifying *label-flipping* and *individual fairness* (gender) on small datasets, for which ground truth can still be obtained by naive enumeration, and compared with our method.

	Certifying label-flipping fairness						Certifying label-flipping + individual fairness					
Name	Ground truth	Time	Our method	Time	Accuracy	Speedup	Ground truth	Time	Our method	Time	Accuracy	Speedup
Salary	50.0%	1.7s	33.3%	0.2s	**66.7%**	8.5X	33.3%	1.5s	33.3%	0.2s	**100%**	7.5X
Student	70.8%	23.0s	60.0%	0.2s	**84.7%**	115X	58.5%	25.2s	44.6%	0.2s	**76.2%**	116X

Methods. For comparison purposes, we implemented six variants of our method, by enabling or disabling the ability to certify label-flipping fairness, the ability to certify individual fairness, and the ability to certify ϵ-fairness.

Except for ϵ-fairness, we also implemented the naive approach of enumerating all $T' \in \mathsf{dBias}_n(T)$. Since the naive approach does not rely on approximation, its result can be regarded as the ground truth (i.e., whether the classification output for an input x is truly fair). Our goal is to obtain the ground truth on small datasets, and use it to evaluate the accuracy of our abstract interpretation based method. However, as explained before, enumeration does not work for ϵ-fairness, since the number of inputs in $\Delta^{\epsilon}(x)$ is infinite.

Our experiments were conducted on a computer with 2 GHz Quad-Core Intel Core i5 CPU and 16 GB of memory. The experiments were designed to answer two questions. First, is our method efficient and accurate enough in handling popular datasets in the fairness literature? Second, does our method help us gain insights? For example, it would be interesting to know whether decision made on an individuals from a protected minority group is more (or less) likely to be certified as fair.

Results on Efficiency and Accuracy. We first evaluate the efficiency and accuracy of our method. For the two small datasets, *Salary* and *Student*, we are able to obtain the ground truth using the naive enumeration approach, and then compare it with the result of our abstract interpretation based method. We want to know how much our results deviate from the ground truth.

Table 2 shows the results obtained by treating *Gender* as the protected attribute. Column 1 shows the name of the dataset. Columns 2–7 compare the naive approach (ground truth) and our method in certifying label-flipping fairness. Columns 8–13 compare the naive approach (ground truth) and our method in certifying label-flipping plus individual fairness.

Table 3. Results for certifying *label-flipping*, *individual*, and *ϵ-fairness* by our method.

Name	Label-flipping fairness	Time	+ Individual fairness	Time	+ ϵ-fairness	Time
Salary (gender)	33.3%	0.2s	33.3%	0.2s	33.3%	0.2s
Student (gender)	60.0%	0.2s	44.6%	0.2s	32.3%	0.2s
German (gender)	48.0%	0.2s	44.0%	0.3s	43.0%	0.2s
Compas (race)	95.0%	0.3s	62.4%	1.4s	56.4%	1.1s
Compas (gender)	95.0%	0.3s	65.3%	1.3s	59.4%	1.0s
Default (gender)	83.2%	2.3s	73.3%	4.4s	64.4%	3.5s
Adult (race)	76.2%	2.2s	65.3%	4.5s	53.5%	5.3s
Adult (gender)	76.2%	2.2s	52.5%	3.5s	43.6%	3.3s

Based on the results in Table 2, we conclude that the accuracy of our method is high (81.9% on average) despite its aggressive use of abstraction to reduce the computational cost. Our method is also 7.5X to 126X faster than the naive approach. Furthermore, the larger the dataset, the higher the speedup.

For medium and large datasets, it is infeasible for the naive enumeration approach to compute and show the ground truth in Table 2. However, the fairness scores of our method shown in Table 3 provide "lower bounds" for the ground truth since our method is sound for certification. For example, when our method reports 95% for *Compas (race)* in Table 3, it means the ground truth must be $\geq$95% (and thus the gap must be $\leq$5%). However, there does not seem to be obvious relationship between the gap and the dataset size – the gap may be due to some unique characterristics of each dataset.

Results on the Certification Rates. We now present the success rates of our certification method for the three variants of fairness. Table 3 shows the results for label-flipping fairness in Columns 2–3, label-flipping plus individual fairness (denoted *+ Individual fairness*) in Columns 4–5, and label-flipping plus ϵ-fairness (denoted *+ ϵ-fairness*) in Columns 6–7. For each variant of fairness, we show the percentage of test inputs that are certified to be fair, together with the average certification time (per test input). In all six datasets, *Gender* was treated as the protected attribute. In addition, *Race* was treated as the protected attribute for *Compas* and *Adult.*

From the results in Table 3, we see that as more stringent fairness standard is used, the certified percentage either stays the same (as in *Salary*) or decreases (as in *Student*). This is consistent with what we expect, since the classification output is required to stay the same for an increasingly larger number of scenarios. For *Compas (race)*, in particular, adding ϵ-fairness on top of label-flipping fairness causes the certified percentage to drop from 62.4% to 56.4%.

Nevertheless, our method still maintains a high certification percentage. Recall that, for *Salary*, the 33.3% certification rate (for *+Individual fairness*) is actually 100% accurate according to comparison with the ground truth in Table 2, while the 44.6% certification rate (for *+Individual fairness*) is actually

76.2% accurate. Furthermore, the efficiency of our method is high: for *Adult*, which has 50,000 samples in the training set, the average certification time of our method remains within a few seconds.

Table 4. Results for certifying *label-flipping* + ϵ*-fairness* with both *Race* and *Gender* as protected attributes.

(a) Compas

	White	Other	Wt. Avg
Male	61.9%	52.2%	52.8%
Female	100%	60.0%	63.7%
Wt. Avg	63.7%	53.7%	54.4%

(b) Adult

	White	Other	Wt. Avg
Male	35.3%	33.3%	35.1%
Female	33.3%	66.7%	37.0%
Wt. Avg	34.7%	44.4%	35.6%

Results on Demographic Groups. Table 4 shows the certified percentage of each demographic group, when both *label-flipping* and ϵ*-fairness* are considered, and both *Race* and *Gender* are treated as protected attributes. The four demographic groups are (1) *White Male*, (2) *White Female*, (3) *Other Male*, and (4) *Other Female*. For each group, we show the certified percentage obtained by our method. In addition, we show the weighted averages for *White* and *Other*, as well as the weighted averages for *Male* and *Female*.

For *Compas*, *White Female* has the highest certified percentage (100%) while *Other Female* has the lowest certified percentage (52.2%); here, the classification output represents the recidivism risk.

For *Adult*, *Other Female* has the highest certified percentage (66.7%) while the other three groups have certified percentages in the range of 33.3%-35.3%.

The differences may be attributed to two sources, one of which is technical and the other is social. The social reason is related to historical bias, which is well documented for these datasets. If the actual percentages (ground truth) is different, the percentages reported by our method will also be different. The technical reason is related to the nature of the KNN algorithm itself, which we explain as follows.

In these datasets, some demographic groups have significantly more samples than others. In KNN, the lowest occurring group may have a limited number of close neighbors. Thus, for each test input x from this group, its K nearest neighbors tend to have a larger radius in the input vector space. As a result, the impact of ϵ perturbation on x will be smaller, resulting in fewer changes to its K nearest neighbors. That may be one of the reasons why, in Table 4, the lowest occurring groups, *White Female* in *Compas* and *Other Female* in *Adult*, have significantly higher certified percentage than other groups.

Results in Table 4 show that, even if a machine learning technique discriminates against certain demographic groups, for an individual, the prediction result produced by the machine learning technique may still be fair. This is closely related to differences (and sometimes conflicts) between *group fairness* and *individual fairness*: while group fairness focuses on statistical parity, individual fairness focuses on similar outcomes for similar individuals. Both are useful notions and in many cases they are complementary.

Caveat. Our work should not be construed as an endorsement nor criticism of the use of machine learning techniques in socially sensitive applications. Instead, it should be viewed as an effort on developing new methods and tools to help improve our understanding of these techniques.

7 Related Work

For fairness certification, as explained earlier in this paper, our method is the first method for certifying KNN in the presence of historical (dataset) bias. While there are other KNN certification and falsification techniques, including Jia et al. [20] and Li et al. [24,25], they focus solely on robustness against data poisoning attacks as opposed to individual and ϵ-fairness against historical bias. Meyer et al. [26,27] and Drews et al. [12] propose certification techniques that handle dataset bias, but target different machine learning techniques (decision tree or linear regression); furthermore, they do not handle ϵ-fairness.

Throughout this paper, we have assumed that the KNN learning (parameter-tuning) step is not tampered with or subjected to fairness violation. However, since the only impact of tampering with the KNN learning step will be changing the optimal value of the parameter K, the biased KNN learning step can be modeled using a properly over-approximated $KSet$. With this new $KSet$, our method for certifying fairness of the prediction result (as presented in Sect. 4) will work AS IS.

Our method aims to certify fairness with certainty. In contrast, there are statistical techniques that can be used to prove that a system is fair or robust with a high probability. Such techniques have been applied to various machine learning models, for example, in *VeriFair* [6] and *FairSquare* [2]. However, they are typically applied to the prediction step while ignoring the learning step, although the learning step may be affected by dataset bias.

There are also techniques for mitigating bias in machine learning systems. Some focus on improving the learning algorithms using random smoothing [33], better embedding [7] or fair representation [34], while others rely on formal methods such as iterative constraint solving [38]. There are also techniques for repairing models to improve fairness [3]. Except for Ruoss et al. [34], most of them focus on group fairness such as demographic parity and equal opportunity; they are significantly different from our focus on certifying individual and ϵ-fairness of the classification results in the presence of dataset bias.

At a high level, our method that leverages a sound over-approximate analysis to certify fairness can be viewed as an instance of the abstract interpretation paradigm [10]. Abstract interpretation based techniques have been successfully used in many other settings, including verification of deep neural networks [17,30], concurrent software [21,22,37], and cryptographic software [43,44].

Since fairness is a type of non-functional property, the verification/certification techniques are often significantly different from techniques used to verify/certify functional correctness. Instead, they are more closely related to techniques for verifying/certifying robustness [8], noninterference [5], and side-channel security [19,39,40,48], where a program is executed multiple times, each time for a different input drawn from a large (and sometimes infinite) set, to see if they all agree on the output. At a high level, this is closely related to differential verification [28,31,32], synthesis of relational invariants [41] and verification of hyper-properties [15,35].

8 Conclusions

We have presented a method for certifying the individual and ϵ-fairness of the classification output of the KNN algorithm, under the assumption that the training dataset may have historical bias. Our method relies on abstract interpretation to soundly approximate the arithmetic computations in the learning and prediction steps. Our experimental evaluation shows that the method is efficient in handling popular datasets from the fairness research literature and accurate enough in obtaining certifications for a large amount of test data. While this paper focuses on KNN only, as a future work, we plan to extend our method to other machine learning models.

References

1. Adeniyi, D.A., Wei, Z., Yongquan, Y.: Automated web usage data mining and recommendation system using k-nearest neighbor (KNN) classification method. Appl. Comput. Inf. **12**(1), 90–108 (2016)
2. Albarghouthi, A., D'Antoni, L., Drews, S., Nori, A.V.: FairSquare: probabilistic verification of program fairness. Proc. ACM Programm. Lang. **1**(OOPSLA), 1–30 (2017)
3. Albarghouthi, A., D'Antoni, L., Drews, S.: Repairing decision-making programs under uncertainty. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 181–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_9
4. Andersson, M., Tran, L.: Predicting movie ratings using KNN (2020)
5. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: IEEE Computer Security Foundations Workshop, pp. 100–114 (2004)
6. Bastani, O., Zhang, X., Solar-Lezama, A.: Probabilistic verification of fairness properties via concentration. Proc. ACM Programm. Lang. **1**(OOPSLA), 1–27 (2019)
7. Bolukbasi, T., Chang, K.W., Zou, J.Y., Saligrama, V., Kalai, A.T.: Man is to computer programmer as woman is to homemaker? Debiasing word embeddings. In: Annual Conference on Neural Information Processing Systems, vol. 29 (2016)
8. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM **55**(8), 107–115 (2012)
9. Cortez, P., Silva, A.M.G.: Using data mining to predict secondary school student performance. EUROSIS-ETI (2008)
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Symposium on Principles of Programming Languages, pp. 238–252 (1977)
11. Dieterich, W., Mendoza, C., Brennan, T.: COMPAS risk scales: demonstrating accuracy equity and predictive parity. Northpointe Inc (2016)

12. Drews, S., Albarghouthi, A., D'Antoni, L.: Proving data-poisoning robustness in decision trees. In: ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 1083–1097 (2020)
13. Dua, D., Graff, C.: UCI machine learning repository (2017). http://archive.ics.uci.edu/ml
14. Dwork, C., Hardt, M., Pitassi, T., Reingold, O., Zemel, R.S.: Fairness through awareness. In: Innovations in Theoretical Computer Science, pp. 214–226 (2012)
15. Finkbeiner, B., Haas, L., Torfah, H.: Canonical representations of k-safety hyperproperties. In: IEEE Computer Security Foundations Symposium, pp. 17–31 (2019)
16. Firdausi, I., Erwin, A., Nugroho, A.S., et al.: Analysis of machine learning techniques used in behavior-based malware detection. In: 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies, pp. 201–203. IEEE (2010)
17. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: IEEE Symposium on Security and Privacy, pp. 3–18 (2018)
18. Guo, G., Wang, H., Bell, D., Bi, Y., Greer, K.: KNN model-based approach in classification. In: Meersman, R., Tari, Z., Schmidt, D.C. (eds.) OTM 2003. LNCS, vol. 2888, pp. 986–996. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39964-3_62
19. Guo, S., Wu, M., Wang, C.: Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 377–388 (2018)
20. Jia, J., Liu, Y., Cao, X., Gong, N.Z.: Certified robustness of nearest neighbors against data poisoning and backdoor attacks. In: The AAAI Conference on Artificial Intelligence (2022)
21. Kusano, M., Wang, C.: Flow-sensitive composition of thread-modular abstract interpretation. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 799–809 (2016)
22. Kusano, M., Wang, C.: Thread-modular static analysis for relaxed memory models. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering, pp. 337–348 (2017)
23. Li, Y., Fang, B., Guo, L., Chen, Y.: Network anomaly detection based on TCM-KNN algorithm. In: ACM Symposium on Information, Computer and Communications Security, pp. 13–19 (2007)
24. Li, Y., Wang, J., Wang, C.: Proving robustness of KNN against adversarial data poisoning. In: International Conference on Formal Methods in Computer-Aided Design, pp. 7–16 (2022)
25. Li, Y., Wang, J., Wang, C.: Systematic testing of the data-poisoning robustness of KNN. In: ACM SIGSOFT International Symposium on Software Testing and Analysis (2023)
26. Meyer, A.P., Albarghouthi, A., D'Antoni, L.: Certifying robustness to programmable data bias in decision trees. In: Annual Conference on Neural Information Processing Systems, pp. 26276–26288 (2021)
27. Meyer, A.P., Albarghouthi, A., D'Antoni, L.: Certifying data-bias robustness in linear regression. CoRR abs/2206.03575 (2022)
28. Mohammadinejad, S., Paulsen, B., Deshmukh, J.V., Wang, C.: DiffRNN: differential verification of recurrent neural networks. In: International Conference on Formal Modeling and Analysis of Timed Systems, pp. 117–134 (2021)

29. Narudin, F.A., Feizollah, A., Anuar, N.B., Gani, A.: Evaluation of machine learning classifiers for mobile malware detection. Soft. Comput. **20**(1), 343–357 (2016)
30. Paulsen, B., Wang, C.: Example guided synthesis of linear approximations for neural network verification. In: International Conference on Computer Aided Verification, pp. 149–170 (2022)
31. Paulsen, B., Wang, J., Wang, C.: ReluDiff: differential verification of deep neural networks. In: International Conference on Software Engineering, pp. 714–726 (2020)
32. Paulsen, B., Wang, J., Wang, J., Wang, C.: NEURODIFF: scalable differential verification of neural networks using fine-grained approximation. In: International Conference on Automated Software Engineering, pp. 784–796 (2020)
33. Rosenfeld, E., Winston, E., Ravikumar, P., Kolter, J.Z.: Certified robustness to label-flipping attacks via randomized smoothing. In: International Conference on Machine Learning, vol. 119, pp. 8230–8241 (2020)
34. Ruoss, A., Balunovic, M., Fischer, M., Vechev, M.T.: Learning certified individually fair representations. In: Annual Conference on Neural Information Processing Systems (2020)
35. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 57–69 (2016)
36. Su, M.Y.: Real-time anomaly detection systems for denial-of-service attacks by weighted k-nearest-neighbor classifiers. Expert Syst. Appl. **38**(4), 3492–3498 (2011)
37. Sung, C., Kusano, M., Wang, C.: Modular verification of interrupt-driven software. In: International Conference on Automated Software Engineering, pp. 206–216 (2017)
38. Wang, J., Li, Y., Wang, C.: Synthesizing fair decision trees via iterative constraint solving. In: Shoham, S., Vizel, Y. (eds.) International Conference on Computer Aided Verification, pp. 364–385. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_18
39. Wang, J., Sung, C., Raghothaman, M., Wang, C.: Data-driven synthesis of provably sound side channel analyses. In: International Conference on Software Engineering, pp. 810–822 (2021)
40. Wang, J., Sung, C., Wang, C.: Mitigating power side channels during compilation. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 590–601 (2019)
41. Wang, J., Wang, C.: Learning to synthesize relational invariants. In: International Conference on Automated Software Engineering, pp. 65:1–65:12 (2022)
42. Weisberg, S.: Applied Linear Regression, p. 194. Wiley (1985)
43. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 15–26 (2018)
44. Wu, M., Wang, C.: Abstract interpretation under speculative execution. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 802–815 (2019)
45. Wu, W., Zhang, W., Yang, Y., Wang, Q.: DREX: developer recommendation with k-nearest-neighbor search and expertise ranking. In: Asia-Pacific Software Engineering Conference, pp. 389–396 (2011)
46. Xie, M., Hu, J., Han, S., Chen, H.H.: Scalable hypergrid K-NN-based online anomaly detection in wireless sensor networks. IEEE Trans. Parallel Distrib. Syst. **24**(8), 1661–1670 (2012)

47. Yeh, I.C., Lien, C.h.: The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. Exp. Syst. Appl. **36**(2), 2473–2480 (2009)
48. Zhang, J., Gao, P., Song, F., Wang, C.: SCInfer: refinement-based verification of software countermeasures against side-channel attacks. In: International Conference on Computer Aided Verification, pp. 157–177 (2018)

Monitoring Algorithmic Fairness

Thomas A. Henzinger, Mahyar Karimi, Konstantin Kueffner, and Kaushik Mallik(✉)

Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
{tah,mahyar.karimi,konstantin.kueffner,kaushik.mallik}@ist.ac.at

Abstract. Machine-learned systems are in widespread use for making decisions about humans, and it is important that they are *fair*, i.e., not biased against individuals based on sensitive attributes. We present runtime verification of algorithmic fairness for systems whose models are unknown, but are assumed to have a Markov chain structure. We introduce a specification language that can model many common algorithmic fairness properties, such as demographic parity, equal opportunity, and social burden. We build monitors that observe a long sequence of events as generated by a given system, and output, after each observation, a quantitative estimate of how fair or biased the system was on that run until that point in time. The estimate is proven to be correct modulo a variable error bound and a given confidence level, where the error bound gets tighter as the observed sequence gets longer. Our monitors are of two types, and use, respectively, frequentist and Bayesian statistical inference techniques. While the frequentist monitors compute estimates that are objectively correct with respect to the ground truth, the Bayesian monitors compute estimates that are correct subject to a given prior belief about the system's model. Using a prototype implementation, we show how we can monitor if a bank is fair in giving loans to applicants from different social backgrounds, and if a college is fair in admitting students while maintaining a reasonable financial burden on the society. Although they exhibit different theoretical complexities in certain cases, in our experiments, both frequentist and Bayesian monitors took less than a millisecond to update their verdicts after each observation.

1 Introduction

Runtime verification complements traditional static verification techniques, by offering lightweight solutions for checking properties based on a single, possibly long execution trace of a given system [8]. We present new runtime verification techniques for the problem of bias detection in decision-making software. The use of software for making critical decisions about humans is a growing trend; example areas include judiciary [13,20], policing [23,49], banking [48], etc. It is important that these software systems are unbiased towards the protected attributes

This work is supported by the European Research Council under Grant No.: ERC-2020-AdG101020093.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 358–382, 2023.
https://doi.org/10.1007/978-3-031-37703-7_17

of humans, like gender, ethnicity, etc. However, they have often shown biases in their decisions in the past [20,47,55,57,58]. While there are many approaches for mitigating biases before deployment [20,47,55,57,58], recent runtime verification approaches [3,34] offer a new complementary tool to oversee *algorithmic fairness* in AI and machine-learned decision makers during deployment.

To verify algorithmic fairness at runtime, the given decision-maker is treated as a *generator* of events with an unknown model. The goal is to algorithmically design lightweight but rigorous *runtime monitors* against quantitative formal specifications. The monitors observe a long stream of events and, after each observation, output a quantitative, statistically sound estimate of how fair or biased the generator was until that point in time. While the existing approaches [3,34] considered only sequential decision making models and built monitors from the frequentist viewpoint in statistics, we allow the richer class of Markov chain models and present monitors from both the frequentist and the Bayesian statistical viewpoints.

Monitoring algorithmic fairness involves on-the-fly statistical estimations, a feature that has not been well-explored in the traditional runtime verification literature. As far as the algorithmic fairness literature is concerned, the existing works are mostly *model-based*, and either minimize decision biases of machine-learned systems at *design-time* (i.e., pre-processing) [11,41,65,66], or verify their absence at *inspection-time* (i.e., post-processing) [32]. In contrast, we verify algorithmic fairness at *runtime*, and do not require an explicit model of the generator. On one hand, the model-independence makes the monitors trustworthy, and on the other hand, it complements the existing model-based static analyses and design techniques, which are often insufficient due to partially unknown or imprecise models of systems in real-world environments.

We assume that the sequences of events generated by the generator can be modeled as sequences of states visited by a finite unknown Markov chain. This implies that the generator is well-behaved and the events follow each other according to some fixed probability distributions. Not only is this assumption satisfied by many machine-learned systems (see Sect. 1.1 for examples), it also provides just enough structure to lay the bare-bones foundations for runtime verification of algorithmic fairness properties. We emphasize that we do not require knowledge of the transition probabilities of the underlying Markov chain.

We propose a new specification language, called the Probabilistic Specification Expressions (PSEs), which can formalize a majority of the existing algorithmic fairness properties in the literature, including demographic parity [21], equal opportunity [32], disparate impact [25], etc. Let Q be the set of events. Syntactically, a PSE is a restricted arithmetic expression over the (unknown) transition probabilities of a Markov chain with the state space Q. Semantically, a PSE φ over Q is a function that maps every Markov chain M with the state space Q to a real number, and the value $\varphi(M)$ represents the degree of fairness or bias (with respect to φ) in the generator M. Our monitors observe a long sequence of events from Q, and after each observation, compute a statistically rigorous estimate of $\varphi(M)$ with a PAC-style error bound for a given confidence level. As the observed sequence gets longer, the error bound gets tighter.

Algorithmic fairness properties that are expressible using PSEs are quantitative refinements of the traditional qualitative fairness properties studied in formal methods. For example, a qualitative fairness property may require that if a certain event A occurs infinitely often, then another event B should follow infinitely often. In particular, a coin is qualitatively fair if infinitely many coin tosses contain both infinitely many heads and infinitely many tails. In contrast, the coin will be algorithmically fair (i.e., unbiased) if approximately half of the tosses come up heads. Technically, while qualitative weak and strong fairness properties are ω-regular, the algorithmic fairness properties are statistical and require counting. Moreover, for a qualitative fairness property, the satisfaction or violation cannot be established based on a finite prefix of the observed sequence. In contrast, for any given finite prefix of observations, the value of an algorithmic fairness property can be estimated using statistical techniques, assuming the future behaves statistically like the past (the Markov assumption).

As our main contribution, we present two different monitoring algorithms, using tools from frequentist and Bayesian statistics, respectively. The central idea of the *frequentist monitor* is that the probability of every transition of the monitored Markov chain M can be estimated using the fraction of times the transition is taken per visit to its source vertex. Building on this, we present a practical implementation of the frequentist monitor that can estimate the value of a given PSE from an observed finite sequence of states. For the coin example, after every new toss, the frequentist monitor will update its estimate of probability of seeing heads by computing the fraction of times the coin came up heads so far, and then by using concentration bounds to find a tight error bound for a given confidence level. On the other hand, the central idea of the *Bayesian monitor* is that we begin with a prior belief about the transition probabilities of M, and having seen a finite sequence of observations, we can obtain an updated posterior belief about M. For a given confidence level, the output of the monitor is computed by applying concentration inequalities to find a tight error bound around the mean of the posterior belief. For the coin example, the Bayesian monitor will begin with a prior belief about the degree of fairness, and, after observing the outcome of each new toss, will compute a new posterior belief. If the prior belief agrees with the true model with a high probability, then the Bayesian monitor's output converges to the true value of the PSE more quickly than the frequentist monitor. In general, both monitors can efficiently estimate more complicated PSEs, such as the ratio and the squared difference of the probabilities of heads of two different coins. The choice of the monitor for a particular application depends on whether an objective or a subjective evaluation, with respect to a given prior, is desired.

Both frequentist and Bayesian monitors use registers (and counters as a restricted class of registers) to keep counts of the relevant events and store the intermediate results. If the size of the given PSE is n, then, in theory, the frequentist monitor uses $\mathcal{O}(n^4 2^n)$ registers and computes its output in $\mathcal{O}(n^4 2^n)$ time after each new observation, whereas the Bayesian monitor uses $\mathcal{O}(n^2 2^n)$ registers and computes its output in $\mathcal{O}(n^2 2^n)$ time after each new observation.

The computation time and the required number of registers get drastically reduced to $O(n^2)$ for the frequentist monitor with PSEs that contain up to one division operator, and for the Bayesian monitor with polynomial PSEs (possibly having negative exponents in the monomials). This shows that under given circumstances, one or the other type of the monitor can be favorable computation-wise. These special, efficient cases cover many algorithmic fairness properties of interest, such as demographic parity and equal opportunity.

Our experiments confirm that our monitors are fast in practice. Using a prototype implementation in Rust, we monitored a couple of decision-making systems adapted from the literature. In particular, we monitor if a bank is fair in lending money to applicants from different demographic groups [48], and if a college is fair in admitting students without creating an unreasonable financial burden on the society [54]. In our experiments, both monitors took, on an average, less than a millisecond to update their verdicts after each observation, and only used tens of internal registers to operate, thereby demonstrating their practical usability at runtime.

In short, we advocate that runtime verification introduces a new set of tools in the area of algorithmic fairness, using which we can monitor biases of deployed AI and machine-learned systems in real-time. While existing monitoring approaches only support sequential decision making problems and use only the frequentist statistical viewpoint, we present monitors for the more general class of Markov chain system models using both frequentist and Bayesian statistical viewpoints.

All proofs can be found in the longer version of the paper [33].

1.1 Motivating Examples

We first present two real-world examples from the algorithmic fairness literature to motivate the problem; these examples will later be used to illustrate the technical developments.

The Lending Problem [48]**:** Suppose a bank lends money to individuals based on certain attributes, like credit score, age group, etc. The bank wants to maximize profit by lending money to only those who will repay the loan in time—called the "true individuals." There is a sensitive attribute (e.g., ethnicity) classifying the population into two groups g and $\overline{g}$. The bank will be considered fair (in lending money) if its lending policy is independent of an individual's membership in g or $\overline{g}$. Several *group fairness* metrics from the literature are relevant in this context. *Disparate impact* [25] quantifies the *ratio* of the probability of an individual from g getting the loan to the probability of an individual from $\overline{g}$ getting the loan, which should be close to 1 for the bank to be considered fair. *Demographic parity* [21] quantifies the *difference* between the probability of an individual from g getting the loan and the probability of an individual from $\overline{g}$ getting the loan, which should be close to 0 for the bank to be considered fair. *Equal opportunity* [32] quantifies the *difference* between the probability of a *true* individual from g getting the loan and the probability of a *true* individual from $\overline{g}$ getting the loan, which should be close to 0 for the bank to be considered fair.

A discussion on the relative merit of various different algorithmic fairness notions is out of scope of this paper, but can be found in the literature [15,22,43,62]. We show how we can monitor whether a given group fairness criteria is fulfilled by the bank, by observing a sequence of lending decisions.

The College Admission Problem [54]**:** Consider a college that announces a cutoff of grades for admitting students through an entrance examination. Based on the merit, every truly qualified student belongs to group g, and the rest to group $\overline{g}$. Knowing the cutoff, every student can choose to invest a sum of money—proportional to the gap between the cutoff and their true merit—to be able to reach the cutoff, e.g., by taking private tuition classes. On the other hand, the college's utility is in minimizing admission of students from $\overline{g}$, which can be accomplished by raising the cutoff to a level that is too expensive to be achieved by the students from $\overline{g}$ and yet easy to be achieved by the students from g. The *social burden* associated to the college's cutoff choice is the expected expense of every student from g, which should be close to 0 for the college to be considered fair (towards the society). We show how we can monitor the social burden, by observing a sequence of investment decisions made by the students from g.

1.2 Related Work

There has been a plethora of work on algorithmic fairness from the machine learning standpoint [10,12,21,32,38,42,45,46,52,59,63,66]. In general, these works improve algorithmic fairness through de-biasing the training dataset (pre-processing), or through incentivizing the learning algorithm to make fair decisions (in-processing), or through eliminating biases from the output of the machine-learned model (post-processing). All of these are interventions in the design of the system, whereas our monitors treat the system as already deployed.

Recently, formal methods-inspired techniques have been used to guarantee algorithmic fairness through the verification of a learned model [2,9,29,53,61], and enforcement of robustness [6,30,39]. All of these works verify or enforce algorithmic fairness *statically* on all runs of the system with high probability. This requires certain knowledge about the system model, which may not be always available. Our runtime monitor dynamically verifies whether the current run of an opaque system is fair.

Our frequentist monitor is closely related to the novel work of Albarghouthi et al. [3], where the authors build a programming framework that allows runtime monitoring of algorithmic fairness properties on programs. Their monitor evaluates the algorithmic fairness of repeated "single-shot" decisions made by machine-learned functions on a sequence of samples drawn from an underlying unknown but fixed distribution, which is a special case of our more general Markov chain model of the generator. They do not consider the Bayesian point of view. Moreover, we argue and empirically show in Sect. 4 that our frequentist approach produces significantly tighter statistical estimates than their approach on most PSEs. On the flip side, their specification language is more expressive, in that they allow atomic variables for expected values of events, which is useful for specifying individual fairness criteria [21]. We only consider group fairness, and leave individual fairness as part of future research. Also, they allow logical operators (like boolean connectives) in their specification language. However, we obtain tighter statistical estimates for the core arithmetic part of algorithmic fairness properties (through PSEs), and point out that we can deal with logical operators just like they do in a straightforward manner.

Shortly after the first manuscript of this paper was written, we published a separate work for monitoring long-run fairness in sequential decision making problems, where the feature distribution of the population may dynamically change due to the actions of the individuals [34]. Although this other work generalizes our current paper in some aspects (support for dynamic changes in the model), it only allows sequential decision making models (instead of Markov chains) and does not consider the Bayesian monitoring perspective.

There is a large body of research on monitoring, though the considered properties are mainly temporal [5,7,19,24,40,50,60]. Unfortunately, these techniques do not directly extend to monitoring algorithmic fairness, since checking algorithmic fairness requires statistical methods, which is beyond the limit of finite automata-based monitors used by the classical techniques. Although there are works on quantitative monitoring that use richer types of monitors (with counters/registers like us) [28,35,36,56], the considered specifications do not easily extend to statistical properties like algorithmic fairness. One exception is the work by Ferrère et al. [26], which monitors certain statistical properties, like mode and median of a given sequence of events. Firstly, they do not consider algorithmic fairness properties. Secondly, their monitors' outputs are correct only as the length of the observed sequence approaches infinity (asymptotic guarantee), whereas our monitors' outputs are *always* correct with high confidence (finite-sample guarantee), and the precision gets better for longer sequences.

Although our work uses similar tools as used in statistical verification [1,4,14,17,64], the goals are different. In traditional statistical verification, the system's runs are chosen probabilistically, and it is verified if any run of the system satisfies a boolean property with a certain probability. For us, the run is given as input to the monitor, and it is this run that is verified against a quantitative algorithmic fairness property with statistical error bounds. To the best of our knowledge, existing works on statistical verification do not consider algorithmic fairness properties.

2 Preliminaries

For any alphabet Σ, the notation Σ^* represents the set of all finite words over Σ. We write $\mathbb{R}$, $\mathbb{N}$, and $\mathbb{N}^+$ to denote the sets of real numbers, natural numbers (including zero), and positive integers, respectively. For a pair of real (natural) numbers a, b with $a < b$, we write $[a, b]$ ($[a \mathinner{..} b]$) to denote the set of all real (natural) numbers between and including a and b. For a given $c, r \in \mathbb{R}$, we write $[c \pm r]$ to denote the set $[c - r, c + r]$. For simpler notation, we will use $|\cdot|$ to denote both the cardinality of a set and the absolute value of a real number, whenever the intended use is clear.

For a given vector $v \in \mathbb{R}^n$ and a given $m \times n$ real matrix M, for some m, n, we write v_i to denote the i-th element of v and write M_{ij} to denote the element at the i-th row and the j-th column of M. For a given $n \in \mathbb{N}^+$, a *simplex* is the set of vectors $\Delta(n) := \{x \in [0,1]^{n+1} \mid \sum_{i=1}^{n+1} x_i = 1\}$. Notice that the dimension of $\Delta(n)$ is $n+1$ (and not n), a convention that is standard due to the interpretation of $\Delta(n)$ as the $n+1$ vertices of an n-dimensional polytope. A *stochastic matrix* of dimension $m \times m$ is a matrix whose every row is in $\Delta(m-1)$, i.e. $M \in \Delta(m-1)^m$. Random variables will be denoted using uppercase symbols from the Latin alphabet (e.g. X), while the associated outcomes will be denoted using lowercase font of the same symbol (x is an outcome of X). We will interchangeably use the expected value $\mathbb{E}(X)$ and the mean μ_X of X. For a given set S, define $\mathcal{D}(S)$ as the set of every random variable—called a *probability distribution*[1]—with set of outcomes being 2^S. A Bernoulli random variable that produces "1" (the alternative is "0") with probability p is written as *Bernoulli*(p).

2.1 Markov Chains as Randomized Generators of Events

We use finite Markov chains as sequential randomized generators of events. A (finite) Markov chain $\mathcal{M}$ is a triple (Q, M, π), where $Q = [1..N]$ is a set of states for a finite N, $M \in \Delta(N-1)^N$ is a stochastic matrix called the transition probability matrix, and $\pi \in \mathcal{D}(Q)$ is the distribution over initial states. We often refer to a pair of states $(i,j) \in Q \times Q$ as an *edge*. The Markov chain $\mathcal{M}$ generates an infinite sequence of random variables $X_0 = \pi, X_1, \ldots$, with $X_i \in \mathcal{D}(Q)$ for every i, such that the Markov property is satisfied: $\mathbb{P}(X_{n+1} = i_{n+1} \mid X_0 = i_0, \ldots, X_n = i_n) = \mathbb{P}(X_{n+1} = i_{n+1} \mid X_n = i_n)$, which is $M_{i_n i_{n+1}}$ in our case. A finite *path* $\vec{x} = x_0, \ldots, x_n$ of $\mathcal{M}$ is a finite word over Q such that for every $t \in [0;n]$, $\mathbb{P}(X_t = x_t) > 0$. Let *Paths*$(\mathcal{M})$ be the set of every finite path of $\mathcal{M}$.

We use Markov chains to model the probabilistic interaction between a machine-learned decision maker with its environment. Intuitively, the Markov assumption on the model puts the restriction that the decision maker does not change over time, e.g., due to retraining.

In Fig. 1 we show the Markov chains for the lending and the college admission examples from Sect. 1.1. The Markov chain for the lending example captures the sequence of loan-related probabilistic events, namely, that a loan applicant is randomly sampled and the group information (g or $\overline{g}$) is revealed, a probabilistic decision is made by the decision-maker and either the loan was granted (gy or $\overline{g}y$, depending on the group) or refused ($\overline{y}$), and if the loan is granted then with some probabilities it either gets repaid (z) or defaulted ($\overline{z}$). The Markov chain for the college admission example captures the sequence of admission events, namely, that a candidate is randomly sampled and the group is revealed ($g, \overline{g}$), and when the candidate is from group g (truly qualified) then the amount of money invested for admission is also revealed.

[1] An alternate commonly used definition of probability distribution is directly in terms of the probability measure induced over S, instead of through the random variable.

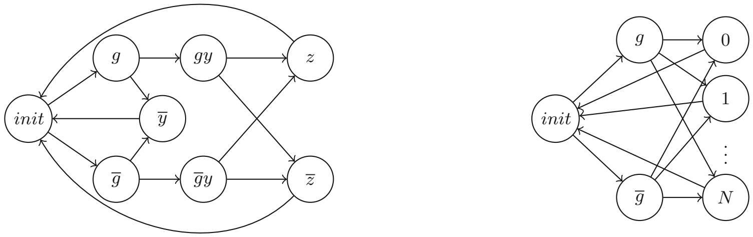


Fig. 1. Markov chains for the lending and the college-admission examples. (**left**) The lending example: The state *init* denotes the initiation of the sampling, and the rest represent the selected individual, namely, g and $\overline{g}$ denote the two groups, (gy) and $(\overline{g}y)$ denote that the individual is respectively from group g and group $\overline{g}$ and the loan was granted, $\overline{y}$ denotes that the loan was refused, and z and $\overline{z}$ denote whether the loan was repaid or not. (**right**) The college admission example: The state *init* denotes the initiation of the sampling, the states $g, \overline{g}$ represent the group identity of the selected candidate, and the states $\{0, \ldots, N\}$ represent the amount of money invested by a truly eligible candidate.

2.2 Randomized Register Monitors

Randomized register monitors, or simply monitors, are adapted from the (deterministic) polynomial monitors of Ferrère et al. [27]. Let R be a finite set of integer variables called registers. A function $v \colon R \to \mathbb{N}$ assigning concrete value to every register in R is called a valuation of R. Let $\mathbb{N}^R$ denote the set of all valuations of R. Registers can be read and written according to relations in the signature $S = \langle 0, 1, +, -, \times, \div, \leq \rangle$. We consider two basic operations on registers:

- A *test* is a conjunction of atomic formulas over S and their negation;
- An *update* is a mapping from variables to terms over S.

We use $\Phi(R)$ and $\Gamma(R)$ to respectively denote the set of tests and updates over R. *Counters* are special registers with a restricted signature $S = \langle 0, 1, +, -, \leq \rangle$.

Definition 1 (Randomized register monitor). *A randomized register monitor is a tuple $(\Sigma, \Lambda, R, \lambda, T)$ where Σ is a finite input alphabet, Λ is an output alphabet, R is a finite set of registers, $\lambda \colon \mathbb{N}^R \to \Lambda$ is an output function, and $T \colon \Sigma \times \Phi(R) \to \mathcal{D}(\Gamma(R))$ is the randomized transition function such that for every $\sigma \in \Sigma$ and for every valuation $v \in \mathbb{N}^R$, there exists a unique $\phi \in \Phi(R)$ with $v \models \phi$ and $T(\sigma, \phi) \in \mathcal{D}(\Gamma(R))$. A deterministic register monitor is a randomized register monitor for which $T(\sigma, \phi)$ is a Dirac delta distribution, if it is defined.*

A *state* of a monitor $\mathcal{A}$ is a valuation of its registers $v \in \mathbb{N}^R$. The monitor $\mathcal{A}$ *transitions* from state v to a *distribution* over states given by the random variable $Y = T(\sigma, \phi)$ on input $\sigma \in \Sigma$ if there exists ϕ such that $v \models \phi$. Let γ be an outcome of Y with $\mathbb{P}(Y = \gamma) > 0$, in which case the registers are updated as $v'(x) = v(\gamma(x))$ for every $x \in R$, and the respective concrete transition is written as $v \xrightarrow{\sigma} v'$. A *run* of $\mathcal{A}$ on a word $w_0 \dots w_n \in \Sigma^*$ is a sequence of concrete transitions $v_0 \xrightarrow{w_0} v_1 \xrightarrow{w_1} \dots \xrightarrow{w_n} v_{n+1}$. The probabilistic transitions of $\mathcal{A}$ induce a probability distribution over the sample space of finite runs of the monitor, denoted $\widehat{\mathbb{P}}(\cdot)$. For a given finite word $w \in \Sigma^*$, the *semantics* of the monitor $\mathcal{A}$ is given by a random variable $[\![\mathcal{A}]\!](w) := \lambda(Y)$ inducing the probability measure $\mathbb{P}_{\mathcal{A}}$, where Y is the random variable representing the distribution over the final state in a run of $\mathcal{A}$ on the word w, i.e., $\mathbb{P}_{\mathcal{A}}(Y = v) := \widehat{\mathbb{P}}(\{r = r_0 \dots r_m \in \Sigma^* \mid r \text{ is a run of } \mathcal{A} \text{ on } w \text{ and } r_m = v\})$.

Example: A Monitor for Detecting the (Unknown) Bias of a Coin. We present a simple deterministic monitor that computes a PAC estimate of the bias of an unknown coin from a sequence of toss outcomes, where the outcomes are denoted as "h" for heads and "t" for tails. The input alphabet is the set of toss outcomes, i.e., $\Sigma = \{h, t\}$, the output alphabet is the set of every bias intervals, i.e., $\Gamma = \{[a, b] \mid 0 \leq a < b \leq 1\}$, the set of registers is $R = \{r_n, r_h\}$, where r_n and r_h are counters counting the total number of tosses and the number of heads, respectively, and the output function λ maps every valuation of r_n, r_h to an interval estimate of the bias that has the form $\lambda \equiv {}^{v(r_h)}/_{v(r_n)} \pm \varepsilon(r_n, \delta)$, where $\delta \in [0, 1]$ is a given upper bound on the probability of an incorrect estimate and $\varepsilon(r_n, \delta)$ is the estimation error computed using PAC analysis. For instance, after observing a sequence of 67 tosses with 36 heads, the values of the registers will be $v(r_n) = 67$ and $v(r_h) = 36$, and the output of the monitor will be $\lambda(67, 36) = {}^{36}/_{67} \pm \varepsilon(n, \delta)$ for some appropriate $\varepsilon(\cdot)$. Now, suppose the next input to the monitor is h, in which case the monitor's transition is given as $T(h, \cdot) = (r_n + 1, r_h + 1)$, which updates the registers to the new values $v'(r_n) = 67 + 1 = 68$ and $v'(r_h) = 36 + 1 = 37$. For this example, the tests $\Phi(R)$ over the registers are redundant, but they can be used to construct monitors for more complex properties.

3 Algorithmic Fairness Specifications and Problem Formulation

3.1 Probabilistic Specification Expressions

To formalize algorithmic fairness properties, like the ones in Sect. 1.1, we introduce *probabilistic specification expressions* (PSE). A PSE φ over a given finite set Q is an algebraic expression with some restricted set of operations that uses variables labeled v_{ij} with $i, j \in Q$ and whose domains are the real interval $[0, 1]$. The syntax of φ is:

$$\xi ::= v \in \{v_{ij}\}_{i,j \in Q} \mid \xi \cdot \xi \mid 1 \div \xi, \tag{1a}$$

$$\varphi ::= \kappa \in \mathbb{R} \mid \xi \mid \varphi + \varphi \mid \varphi - \varphi \mid \varphi \cdot \varphi \mid (\varphi), \tag{1b}$$

where $\{v_{ij}\}_{i,j \in Q}$ are the variables with domain $[0, 1]$ and κ is a constant. The expression ξ in (1a) is called a *monomial* and is simply a product of powers of variables with integer exponents. A *polynomial* is a weighted sum of monomials with constant weights.[2] Syntactically, polynomials form a strict subclass of the expressions definable using (1b), because the product of two polynomials is not a polynomial, but is a valid expression according to (1b). A PSE φ is *division-free* if there is no division operator involved in φ. The *size* of an expression φ is the total number of arithmatic operators (i.e. $+, -, \cdot, \div$) in φ. We use V_φ to denote the set of variables appearing in the expression φ, and for every $V \subseteq V_\varphi$ we define $Dom(V) \coloneqq \{i \in Q \mid \exists v_{ij} \in V \vee \exists v_{ki} \in V\}$ as the set containing any state of the Markov chain that is involved in some variable in V.

The semantics of a PSE φ is interpreted *statically* on the unknown Markov chain M: we write $\varphi(M)$ to denote the evaluation or the value of φ by substituting every variable v_{ij} in φ with M_{ij}. E.g., for a Markov chain with state space $\{1, 2\}$ and transition probabilities $M_{11} = 0.2$, $M_{12} = 0.8$, $M_{21} = 0.4$, and $M_{22} = 0.6$, the expression $\varphi = v_{11} - v_{21}$ has the evaluation $\varphi(M) = 0.2 - 0.4 = -0.2$. We will assume that for every expression $(1 \div \xi)$, $\xi(M) \neq 0$.

Example: Group Fairness. Using PSEs, we can express the group fairness properties for the lending example described in Sect. 1.1, with the help of the Markov chain in the left subfigure of Fig. 1:

Disparate impact [25]: $v_{gy} \div v_{\overline{g}y}$

Demographic parity [21]: $v_{gy} - v_{\overline{g}y}$

The equal opportunity criterion requires the following probability to be close to zero: $p = \mathbb{P}(y \mid g, z) - \mathbb{P}(y \mid \overline{g}, z)$, which is tricky to monitor as p contains the counter-factual probabilities representing "the probability that an individual from a group would repay had the loan been granted." We apply Bayes' rule, and turn p into the following equivalent form: $p' = \frac{\mathbb{P}(z|g,y)\cdot\mathbb{P}(y|g)}{\mathbb{P}(z|g)} - \frac{\mathbb{P}(z|\overline{g},y)\cdot\mathbb{P}(y|\overline{g})}{\mathbb{P}(z|\overline{g})}$. Assuming $\mathbb{P}(z \mid g) = c_1$ and $\mathbb{P}(z \mid \overline{g}) = c_2$, where c_1 and c_2 are known constants, the property p' can be encoded as a PSE as below:

Equal opportunity [32]: $(v_{(gy)z} \cdot v_{gy}) \div c_1 - (v_{(\overline{g}y)z} \cdot v_{\overline{g}y}) \div c_2.$

Example: Social Burden. Using PSEs, we can express the social burden of the college admission example described in Sect. 1.1, with the help of the Markov chain depicted in the right subfigure of Fig. 1:

Social burden [54]: $1 \cdot v_{g1} + \ldots + N \cdot v_{gN}.$

3.2 The Monitoring Problem

Informally, our goal is to build monitors that observe a single long path of a Markov chain and, after each observation, output a new estimate for the value of the PSE. Since the monitor's estimate is based on statistics collected from

[2] Although monomials and polynomials usually only have positive exponents, we take the liberty to use the terminologies even when negative exponents are present.

a finite path, the output may be incorrect with some probability, where the source of this probability is different between the frequentist and the Bayesian approaches. In the frequentist approach, the underlying Markov chain is fixed (but unknown), and the randomness stems from the sampling of the observed path. In the Bayesian approach, the observed path is fixed, and the randomness stems from the uncertainty about a prior specifying the Markov chain's parameters. The commonality is that, in both cases, we want our monitors to estimate the value of the PSE up to an error with a fixed probabilistic confidence.

We formalize the monitoring problem separately for the two approaches. A *problem instance* is a triple (Q, φ, δ), where $Q = [1\,..\,N]$ is a set of states, φ is a PSE over Q, and $\delta \in [0, 1]$ is a constant. In the frequentist approach, we use $\mathbb{P}_s$ to denote the probability measure induced by *sampling* of paths, and in the Bayesian approach we use $\mathbb{P}_\theta$ to denote the probability measure induced by the *prior* probability density function $p_\theta \colon \Delta(n-1)^n \to \mathbb{R} \cup \{\infty\}$ over the transition matrix of the Markov chain. In both cases, the output alphabets of the monitors contain every real interval.

Problem 1 (Frequentist monitor). *Suppose (Q, φ, δ) is a problem instance given as input. Design a monitor $\mathcal{A}$ such that for every Markov chain $\mathcal{M}$ with transition probability matrix M and for every finite path $\vec{x} \in Paths(\mathcal{M})$:*

$$\mathbb{P}_{s,\mathcal{A}}\left(\varphi(M) \in [\![\mathcal{A}]\!](\vec{x})\right) \geq 1 - \delta, \tag{2}$$

where $\mathbb{P}_{s,\mathcal{A}}$ is the joint probability measure of $\mathbb{P}_s$ and $\mathbb{P}_\mathcal{A}$.

Problem 2 (Bayesian monitor). *Suppose (Q, φ, δ) is a problem instance and p_θ is a prior density function, both given as inputs. Design a monitor $\mathcal{A}$ such that for every Markov chain $\mathcal{M}$ with transition probability matrix M and for every finite path $\vec{x} \in Paths(\mathcal{M})$:*

$$\mathbb{P}_{\theta,\mathcal{A}}\left(\varphi(M) \in [\![\mathcal{A}]\!](\vec{x}) \mid \vec{x}\right) \geq 1 - \delta, \tag{3}$$

where $\mathbb{P}_{\theta,\mathcal{A}}$ is the joint probability measure of $\mathbb{P}_\theta$ and $\mathbb{P}_\mathcal{A}$.

Notice that the state space of the Markov chain and the input alphabet of the monitor are the same, and so, many times, we refer to observed states as (input) symbols, and vice versa. The estimate $[l, u] = [\![\mathcal{A}]\!](\vec{x})$ is called the $(1-\delta)\cdot 100\%$ *confidence interval* for $\varphi(M)$.[3] The radius, given by $\varepsilon = 0.5 \cdot (u - l)$, is called the *estimation error*, and the quantity $1 - \delta$ is called the *confidence*. The estimate gets more precise as the error gets smaller and the confidence gets higher.

In many situations, we are interested in a *qualitative* question of the form "is $\varphi(M) \leq c$?" for some constant c. We point out that, once the quantitative problem is solved, the qualitative questions can be answered using standard procedures by setting up a hypothesis test [44, p. 380].

[3] While in the Bayesian setting *credible intervals* would be more appropriate, we use confidence intervals due to uniformity and the relative ease of computation. To relate the two, our confidence intervals are over-approximations of credible intervals (non-unique) that are centered around the posterior mean.

4 Frequentist Monitoring

Suppose the given PSE is only a single variable $\varphi = v_{ij}$, i.e., we are monitoring the probability of going from state i to another state j. The frequentist monitor $\mathcal{A}$ for φ can be constructed in two steps: (1) empirically compute the average number of times the edge (i, j) was taken per visit to the state i on the observed path of the Markov chain, and (2) compute the $(1-\delta) \cdot 100\%$ confidence interval using statistical concentration inequalities.

Now consider a slightly more complex PSE $\varphi' = v_{ij} + v_{ik}$. One approach to monitor φ', proposed by Albarghouthi et al. [3], would be to first compute the $(1-\delta) \cdot 100\%$ confidence intervals $[l_1, u_1]$ and $[l_2, u_2]$ separately for the two constituent variables v_{ij} and v_{ik}, respectively. Then, the $(1-2\delta) \cdot 100\%$ confidence interval for φ' would be given by the sum of the two intervals $[l_1, u_1]$ and $[l_2, u_2]$, i.e., $[l_1+l_2, u_1+u_2]$; notice the drop in overall confidence due to the union bound. The drop in the confidence level and the additional error introduced by the interval arithmetic accumulate quickly for larger PSEs, making the estimate unusable. Furthermore, we lose all the advantages of having any dependence between the terms in the PSE. For instance, by observing that v_{ij} and v_{ik} correspond to the mutually exclusive transitions i to j and i to k, we know that $\varphi'(M)$ is always less than 1, a feature that will be lost if we use plain merging of individual confidence intervals for v_{ij} and v_{ik}. We overcome these issues by estimating the value of the PSE as a whole as much as possible. In Fig. 2, we demonstrate how the ratio between the estimation errors from the two approaches vary as the number of summands (i.e., n) in the PSE $\varphi = \sum_{i=1}^{n} v_{1n}$ changes; in both cases we fixed the overall δ to 0.05 (95% confidence). The ratio remains the same for different observation lengths. Our approach is always at least as accurate as their approach [3], and is significantly better for larger PSEs.

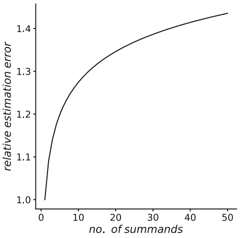


Fig. 2. Variation of ratio of the est. error using the existing approach [3] to est. error using our approach, w.r.t. the size of the chosen PSE.

4.1 The Main Principle

We first explain the idea for division-free PSEs, i.e., PSEs that do not involve any division operator; later we extend our approach to the general case.

Divison-Free PSEs: In our algorithm, for every variable $v_{ij} \in V_\varphi$, we introduce a *Bernoulli*(M_{ij}) random variable Y^{ij} with the mean M_{ij} unknown to us. We make an observation y_p^{ij} for every p-th visit to the state i on a run, and if j follows immediately afterwards then record $y_p^{ij} = 1$ else record $y_p^{ij} = 0$. This gives us a sequence of observations $\overrightarrow{y}^{ij} = y_1^{ij}, y_2^{ij}, \ldots$ corresponding to the sequence of i.i.d. random variables $\overrightarrow{Y}^{ij} = Y_1^{ij}, Y_2^{ij}, \ldots$. For instance, for the run 121123 we obtain $\overrightarrow{y}^{12} = 1, 0, 1$ for the variable v_{12}.

The heart of our algorithm is an aggregation procedure of every sequence of random variable $\{\overrightarrow{Y}^{ij}\}_{v_{ij} \in V_\varphi}$ to a single i.i.d. sequence $\overrightarrow{W}$ of an auxiliary random variable W, such that the mean of W is $\mu_W = \mathbb{E}(W) = \varphi(M)$. We can then use known concentration inequalities on the sequence $\overrightarrow{W}$ to estimate μ_W. Since μ_W exactly equals $\varphi(M)$ by design, we obtain a tight concentration bound on $\varphi(M)$. We informally explain the main idea of constructing $\overrightarrow{W}$ using simple examples; the details can be found in Algorithm 2.

Sum and Difference: Let $\varphi = v_{ij} + v_{kl}$. We simply combine $\overrightarrow{Y}^{ij}$ and $\overrightarrow{Y}^{kl}$ as $W_p = Y_p^{ij} + Y_p^{kl}$, so that $w_p = y_p^{ij} + y_p^{kl}$ is the corresponding observation of W_p. Then $\mu_{W_p} = \varphi(M)$ holds, because $\mu_{W_p} = \mathbb{E}(W_p) = \mathbb{E}(Y_p^{ij} + Y_p^{kl}) = \mathbb{E}(Y_p^{ij}) + \mathbb{E}(Y_p^{kl}) = M_{ij} + M_{kl}$. Similar approach works for $\varphi = v_{ij} - v_{kl}$.

Multiplication: For multiplications, the same linearity principle will not always work, since for random variables A and B, $\mathbb{E}(A \cdot B) = \mathbb{E}(A) \cdot \mathbb{E}(B)$ *only if* A and B are statistically independent, which will not be true for specifications of the form $\varphi = v_{ij} \cdot v_{ik}$. In this case, the respective Bernoulli random variables Y_p^{ij} and Y_p^{ik} are dependent: $\mathbb{P}(Y_p^{ij} = 1) \cdot \mathbb{P}(Y_p^{ik} = 1) = M_{ij} \cdot M_{ik}$, but $\mathbb{P}(Y_p^{ij} = 1 \wedge Y_p^{ik} = 1)$ is always 0 (since *both* j and k cannot be visited following the p-th visit to i).

To benefit from independence once again, we temporally shift one of the random variables by defining $W_p = Y_{2p}^{ij} \cdot Y_{2p+1}^{ik}$, with $w_p = y_{2p}^{ij} \cdot y_{2p+1}^{ik}$. Since the random variables Y_{2p}^{ij} and Y_{2p+1}^{ik} are independent, as they use separate visits of state i, hence we obtain $\mu_{W_p} = M_{ij} \cdot M_{ik}$. For independent multiplications of the form $\varphi = v_{ij} \cdot v_{kl}$ with $i \neq k$, we can simply use $W_p = Y_p^{ij} \cdot Y_p^{ik}$.

In general, we use the ideas of aggregation and temporal shift on the syntax tree of the PSE φ, inductively. With an aggregated sequence of observations for the auxiliary variable W for φ, we can find an estimate for $\varphi(M)$ using the Hoeffding's inequality. We present the detailed algorithm of this monitor, namely `FreqMonitorDivFree`, in Algorithm 1.

The General Case (PSEs With Division Operators): We observe that every arbitrary PSE φ of size n can be transformed into a semantically equivalent PSE of the form $\varphi_a + \frac{\varphi_b}{\varphi_c}$ of size $\mathcal{O}(n^2 2^n)$, where φ_a, φ_b, and φ_c are all division-free. Once in this form, we can employ three different `FreqMonitorDivFree` monitors from Algorithm 1 to obtain separate interval estimates for φ_a, φ_b, and φ_c, which are then combined using standard interval arithmetic and the resulting confidence of the estimate is obtained through the union bound. The steps for constructing the (general-case) `FrequentistMonitor` are shown in Algorithm 2, and the detailed analysis can be found in the proof of Theorem 1.

Bounding Memory: Consider a PSE $\varphi = v_{ij} + v_{kl}$. The outcome w_p for φ can only be computed when both the Bernoulli outcomes y_p^{ij} and y_p^{kl} are available. If at any point only one of the two is available, then we need to store the available one so that it can be used later when the other one gets available. It can be shown that the storage of "unmatched" outcomes may need unbounded memory.

To bound the memory, we use the insight that a *random reshuffling* of the i.i.d. sequence y_p^{ij} would still be i.i.d. with the same distribution, so that we do not need to store the exact order in which the outcomes appeared. Instead, for every $v_{ij} \in V_\varphi$, we only store the number of times we have seen the state i and the edge (i, j) in counters c_i and c_{ij}, respectively. Observe that $c_i \geq \sum_{v_{ik} \in V_\varphi} c_{ik}$, where the possible difference accounts for the visits to irrelevant states, denoted as a dummy state $\top$. Given $\{c_{ik}\}_k$, whenever needed, we generate in x_i a *random reshuffling* of the sequence of states, together with $\top$, seen after the past visits to i. From the sequence stored in x_i, for every $v_{ik} \in V_\varphi$, we can consistently determine the value of y_p^{ik} (consistency dictates $y_p^{ik} = 1 \Rightarrow y_p^{ij} = 0$). Moreover, we reuse space by resetting x_i whenever the sequence stored in x_i is no longer needed. It can be shown that the size of every x_i can be at most the size of the expression [33, Proof of Thm. 2]. This random reshuffling of the observation sequences is the cause of the probabilistic transitions of the frequenitst monitor.

4.2 Implementation of the Frequentist Monitor

Fix a problem instance (Q, φ, δ), with size of φ being n. Let φ be transformed into φ^l by relabeling duplicate occurrences of v_{ij} using distinct labels $v_{ij}^1, v_{ij}^2, \ldots$. The set of labeled variables in φ^l is V_φ^l, and $|V_\varphi^l| = \mathcal{O}(n)$. Let $\mathit{SubExpr}(\varphi)$ denote the set of every subexpression in the expression φ, and use $[l_\varphi, u_\varphi]$ to denote the range of values the expression φ can take for every valuation of every variable as per the domain $[0, 1]$. Let $\mathit{Dep}(\varphi) = \{i \mid \exists v_{ij} \in V_\varphi\}$, and every subexpression $\varphi_1 \cdot \varphi_2$ with $\mathit{Dep}(\varphi_1) \cap \mathit{Dep}(\varphi_2) \neq \emptyset$ is called a *dependent multiplication*.

Implementation of `FreqMonitorDivFree` in Algorithm 1 has two main functions. *Init* initializes the registers. *Next* implements the transition function of the monitor, which attempts to compute a new observation w for $\overrightarrow{W}$ (Line 4) after observing a new input σ', and if successful it updates the output of the monitor by invoking the *UpdateEst* function. In addition to the registers in *Init* and *Next* labeled in the pseudocode, following registers are used internally:

- x_i, $i \in \mathit{Dom}(V_\varphi)$: reshuffled sequence of states that followed i.
- t_{ij}^l: the index of x_i that was used to obtain the latest outcome of v_{ij}^l.

Now, we summarize the main results for the frequentist monitor.

Theorem 1 (Correctness). *Let (Q, φ, δ) be a problem instance. Algorithm 2 implements a monitor for (Q, φ, δ) that solves Problem 1.*

Theorem 2 (Computational resources). *Let (Q, φ, δ) be a problem instance and $\mathcal{A}$ be the monitor implemented using the* `FrequentistMonitor` *routine of Algorithm 2. Suppose the size of φ is n. The monitor $\mathcal{A}$ requires $\mathcal{O}(n^4 2^{2n})$ registers, and takes $\mathcal{O}(n^4 2^{2n})$ time to update its output after receiving a new input*

Algorithm 1. `FreqMonitorDivFree`

Parameters: Q, φ, δ
Output: Λ

```
function $Init(\sigma)$
    $\varphi^l \xleftarrow{\text{unique labeling}} \varphi$
    for all $v_{ij} \in V_\varphi$ do
        $c_{ij} \leftarrow 0$                ▷# of $(i,j)$
        $c_i \leftarrow 0$                   ▷# of $i$
    $n \leftarrow 0$                         ▷length of $\vec{w}$
    $\sigma \leftarrow \sigma$               ▷prev. symbol
    $\mu_\Lambda \leftarrow \bot$            ▷est. mean
    $\varepsilon_\Lambda \leftarrow \bot$    ▷est. error
    $ResetX()$                               ▷reset $x_i$-s
    Compute $l_\varphi, u_\varphi$           ▷int. arith.
```

```
function $Next(\sigma')$
    $c_\sigma \leftarrow c_\sigma + 1$               ▷update counters
    $c_{\sigma\sigma'} \leftarrow c_{\sigma\sigma'} + 1$
    $w \leftarrow Eval(\varphi^l)$
    if $w \neq \bot$ then
        $n \leftarrow n + 1$
        $\Lambda \leftarrow UpdateEst(w, n)$
        $ResetX()$
    $\sigma \leftarrow \sigma'$
    return $\Lambda$
```

```
function $Eval(\varphi^l)$
    if $r_{\varphi^l} = \bot$ then
        if $\varphi^l \equiv \varphi_1^l + \varphi_2^l$ then
            $r_{\varphi^l} \leftarrow Eval(\varphi_1^l) + Eval(\varphi_2^l)$
        else if $\varphi^l \equiv \varphi_1^l - \varphi_2^l$ then
            $r_{\varphi^l} \leftarrow Eval(\varphi_1^l) - Eval(\varphi_2^l)$
        else if $\varphi^l \equiv \varphi_1^l \cdot \varphi_2^l$ then
            if $Dep(V_{\varphi_1}^l) \cap Dep(V_{\varphi_2}^l) = \emptyset$ then
                $r_{\varphi^l} \leftarrow Eval(\varphi_1^l) \cdot Eval(\varphi_2^l)$
            else                                         ▷dep. mult.
                for $v_{ij}^l \in V_{\varphi_2}^l \cap Dep(V_{\varphi_1}^l)$ do
                    $t_{ij}^l \leftarrow \max(\{t_{ik}^m \mid v_{ik}^m \in V_{\varphi_1}^l\})$
                    $t_{ij}^l \leftarrow t_{ij}^l + 1$          ▷make indep.
                $r_{\varphi^l} \leftarrow Eval(\varphi_1^l) \cdot Eval(\varphi_2^l)$
        else if $\varphi^l \equiv v_{ij}^l$ then
            if $x_i[t_{ij}^l + 1] = \bot$ then
                $ExtractOutcome(x_i, t_{ij}^l + 1)$
            if $x_i[t_{ij}^l + 1] = j \neq \bot$ then
                $r_{\varphi^l} \leftarrow 1$
            else
                $r_{\varphi^l} \leftarrow 0$
        else if $\varphi^l \equiv c$ then
            $r_{\varphi^l} \leftarrow c$
    return $r_{\varphi^l}$
```

```
function $UpdateEst(w, n)$
    $\mu_\Lambda \leftarrow \frac{\mu_\Lambda \cdot (n-1) + w}{n}$
    $\varepsilon_\Lambda \leftarrow \sqrt{-\frac{(u_\varphi - l_\varphi)^2}{2n} \cdot \ln\left(\frac{\delta}{2}\right)}$
    return $[\mu_\Lambda \pm \varepsilon_\Lambda]$
```

```
function $ExtractOutcome(x_i, t)$
    ▷generate a shuffled sequence of symbols seen after $i$ so that $|x_i| = t$
    Let $U \leftarrow \{j \in Q \mid v_{ij} \in V_\varphi\}$
    for $p = |x_i| + 1, \ldots, t$ do
        $q \leftarrow \forall u \in U$ .
                pick $u$ w/ prob. $\frac{c_{iu}}{c_i}$,
                pick $\top$ w/ prob. $\frac{(c_i - \sum_j c_{ij})}{c_i}$
        $c_i \leftarrow c_i - 1$
        if $q \neq \top$ then
            $c_{iq} \leftarrow c_{iq} - 1$
        $x_i[|x_i| + 1] \leftarrow q$
```

```
function $ResetX()$
    for all $i \in Dom(V_\varphi)$ do
        $x_i \leftarrow \emptyset$
    for all $v_{ij}^l \in V_\varphi^l$ do
        $t_{ij}^l \leftarrow 0$
```

Algorithm 2. `FrequentistMonitor`

Parameters: Q, φ, δ
Output: Λ

```
function $Init(\sigma)$
    $\varphi_a + \frac{\varphi_b}{\varphi_c} \xleftarrow{\text{change form}} \varphi^l \xleftarrow{\text{labeling}} \varphi$
    $\mathcal{A}_a \leftarrow$ FreqMonitorDivFree$(Q, \varphi_a, \delta/3)$
    $\mathcal{A}_b \leftarrow$ FreqMonitorDivFree$(Q, \varphi_b, \delta/3)$
    $\mathcal{A}_c \leftarrow$ FreqMonitorDivFree$(Q, \varphi_c, \delta/3)$
    $\mathcal{A}_a.Init(\sigma)$
    $\mathcal{A}_b.Init(\sigma)$
    $\mathcal{A}_c.Init(\sigma)$
```

```
function $Next(\sigma')$
    $[\mu_a \pm \varepsilon_a] \leftarrow \mathcal{A}_a.Next(\sigma')$
    $[\mu_b \pm \varepsilon_b] \leftarrow \mathcal{A}_b.Next(\sigma')$
    $[\mu_c \pm \varepsilon_c] \leftarrow \mathcal{A}_c.Next(\sigma')$
    if $\mu_a \neq \bot \wedge \mu_b \neq \bot \wedge \mu_c \neq \bot$ then
        $[\mu_\Lambda \pm \varepsilon_\Lambda] \leftarrow [\mu_a \pm \varepsilon_a] + \frac{[\mu_b \pm \varepsilon_b]}{[\mu_c \pm \varepsilon_c]}$
    return $[\mu_\Lambda \pm \varepsilon_\Lambda]$
```

symbol. For the special case of φ containing at most one division operator (division by constant does not count), $\mathcal{A}$ requires only $\mathcal{O}(n^2)$ registers, and takes only $\mathcal{O}(n^2)$ time to update its output after receiving a new input symbol.

There is a tradeoff between the estimation error, the confidence, and the length of the observed sequence of input symbols. For instance, for a fixed confidence, the longer the observed sequence is, the smaller is the estimation error. The following theorem establishes a lower bound on the length of the sequence for a given upper bound on the estimation error and a fixed confidence.

Theorem 3 (Convergence speed). *Let (Q, φ, δ) be a problem instance where φ does not contain any division operator, and let $\mathcal{A}$ be the monitor computed using Algorithm 2. Suppose the size of φ is n. For a given upper bound on estimation error $\overline{\varepsilon} \in \mathbb{R}$, the minimum number of visits to every state in $Dom(V_\varphi)$ for obtaining an output with error at most $\overline{\varepsilon}$ and confidence at least $1 - \delta$ on any path is given by:*

$$-\frac{(u_\varphi - l_\varphi)^2 \ln\left(\frac{\delta}{2}\right) n}{2\overline{\varepsilon}^2}, \tag{4}$$

where $[l_\varphi, u_\varphi]$ is the set of possible values of φ for every valuation of every variable (having domain $[0,1]$) in φ.

The bound follows from the Hoeffding's inequality, together with the fact that every dependent multiplication increments the required number of samples by 1. A similar bound for the general case with division is left open.

5 Bayesian Monitoring

Fix a problem instance $(Q = [1\,..\,N], \varphi, \delta)$. Let $\mathbb{M} = \Delta(N-1)^N$ be the shorthand notation for the set of transition probability matrices of the Markov chains with state space Q. Let $p_\theta \colon \mathbb{M} \to [0,1]$ be the prior probability density function over $\mathbb{M}$, which is assumed to be specified using the matrix beta distribution (the definition can be found in standard textbooks on Bayesian statistics [37, pp. 280]). Let $\mathbb{1}$ be a matrix, with its size dependent on the context, whose every element is 1. We make the following common assumption [31,37, p. 50]:

Assumption 1 (Prior). *We are given a parameter matrix $\theta \geq \mathbb{1}$, and p_θ is specified using the matrix beta distribution with parameter θ. Moreover, the initial state of the Markov chain is fixed.*

When $\theta = \nVdash$, then p_θ is the uniform density function over $\mathbb{M}$. After observing a path $\vec{x}$, using Bayes' rule we obtain the *posterior* density function $p_\theta(\cdot \mid \vec{x})$, which is known to be efficiently computable due to the so-called conjugacy property that holds due to Assumption 1. From the posterior density, we obtain the expected posterior semantic value of φ as: $\mathbb{E}_\theta(\varphi(M) \mid \vec{x}) := \int_{\mathbb{M}} \varphi(M) \cdot p_\theta(M \mid \vec{x}) dM$. The heart of our Bayesian monitor is an efficient incremental computation of $\mathbb{E}_\theta(\varphi(M) \mid \vec{x})$—free from numerical integration. Once we can compute $\mathbb{E}_\theta(\varphi(M) \mid \vec{x})$, we can also compute the posterior variance S^2 of $\varphi(M)$ using the known expression $S^2 = \mathbb{E}_\theta(\varphi^2(M) \mid \vec{x}) - \mathbb{E}_\theta(\varphi(M) \mid \vec{x})$, which enables us to compute a confidence interval for $\varphi(M)$ using the Chebyshev's inequality. In the following, we summarize our procedure for estimating $\mathbb{E}_\theta(\varphi(M) \mid \vec{x})$.

5.1 The Main Principle

The incremental computation of $\mathbb{E}_\theta(\varphi(M) \mid \vec{x})$ is implemented in `BayesExpMonitor`. We first transform the expression φ into the polynomial form $\varphi' = \sum_l \kappa_l \xi_l$, where $\{\kappa_l\}_l$ are the weights and $\{\xi_l\}_l$ are monomials. If the size of φ is n then the size of φ' is $\mathcal{O}(n2^{\frac{n}{2}})$. Then we can use linearity to compute the overall expectation as the weighted sum of expectations of the individual monomials: $\mathbb{E}_\theta(\varphi(M) \mid \vec{x}) = \mathbb{E}_\theta(\varphi'(M) \mid \vec{x}) = \sum_l \kappa_l \mathbb{E}_\theta(\xi_l(M) \mid \vec{x})$. In the following, we summarize the procedure for estimating $\mathbb{E}_\theta(\xi(M) \mid \vec{x})$ for every monomial ξ.

Let ξ be a monomial, and let $\vec{x}ab \in Q^*$ be a sequence of states. We use d_{ij} to store the exponent of the variable v_{ij} in the monomial ξ, and define $d_a := \sum_{j \in [1..N]} d_{aj}$. Also, we record the sets of (i,j)-s and i-s with positive and negative d_{ij} and d_i entries: $D_i^+ := \{j \mid d_{ij} > 0\}$, $D_i^- := \{j \mid d_{ij} < 0\}$, $D^+ := \{i \mid d_i > 0\}$, and $D^- := \{i \mid d_i < 0\}$.

For any given word $\vec{w} \in Q^*$, let $c_{ij}(\vec{w})$ denote the number of ij-s in $\vec{w}$ and let $c_i(\vec{w}) := \sum_{j \in Q} c_{ij}(\vec{w})$. Define $\overline{c}_i(\vec{w}) := c_i(\vec{w}) + \sum_{j \in [1..N]} \theta_{ij}$ and $\overline{c}_{ij}(\vec{w}) := c_{ij}(\vec{w}) + \theta_{ij}$. Let $\mathcal{H} \colon Q^* \to \mathbb{R}$ be defined as:

$$\mathcal{H}(\vec{w}) := \frac{\prod_{i=1}^{N} \prod_{j \in D_i^+} {}^{n}P_{(\overline{c}_{ij}(\vec{w})-1)+|d_{ij}|}|d_{ij}|}{\prod_{i \in D^+} {}^{n}P_{(\overline{c}_i(\vec{w})-1)+|d_i|}|d_i|} \cdot \frac{\prod_{i \in D^-} {}^{n}P_{(\overline{c}_i(\vec{w})-1)}|d_i|}{\prod_{i=1}^{N} \prod_{j \in D_i^-} {}^{n}P_{(\overline{c}_{ij}(\vec{w})-1)}|d_{ij}|}, \tag{5}$$

where ${}^{n}P_n k := \frac{n!}{(n-k)!}$ is the number of permutations of $k > 0$ items from $n > 0$ objects, for $k \leq n$, and we use the convention that for $S = \emptyset$, $\prod_{s \in S} \cdots = 1$. Below, in Lemma 1, we establish that $\mathbb{E}_\theta(\xi(M) \mid \vec{w}) = \mathcal{H}(\vec{w})$, and present an efficient incremental scheme to compute $\mathbb{E}_\theta(\xi(M) \mid \vec{x}ab)$ from $\mathbb{E}_\theta(\xi(M) \mid \vec{x}a)$.

Lemma 1 (Incremental computation of $\mathbb{E}(\cdot \mid \cdot)$). *If the following consistency condition*

$$\forall i,j \in [1..N] \,.\, \overline{c}_{ij}(\vec{w}) + d_{ij} > 0 \tag{6}$$

is met, then the following holds:

$$\mathbb{E}(\xi(M) \mid \vec{x}ab) = \mathcal{H}(\vec{x}ab) = \mathcal{H}(\vec{x}a) \cdot \frac{\overline{c}_{ab}(\vec{x}) + d_{ab}}{\overline{c}_{ab}(\vec{x})} \cdot \frac{\overline{c}_a(\vec{x})}{\overline{c}_a(\vec{x}) + d_a}. \tag{7}$$

Algorithm 3. `BayesExpMonitor`

```
Parameters: $Q, \varphi = \sum_{l=1}^{p} \kappa_l \xi_l, \theta$
Output: $E$
1: function $Init(\sigma = 1)$
2:     for $v_{ij} \in V_\varphi$ do
3:         $\overline{c}_{ij} \leftarrow \theta_{ij}$
4:         $\overline{c}_i \leftarrow \sum_{j \in [1..N]} \theta_{ij}$
5:         $m_{ij} \leftarrow \min_{l \in [1..p]} d^l_{ij}$          ▷cache
6:     active $\leftarrow$ $false$                                    ▷Eq. 6 not true
7:     $\sigma \leftarrow \sigma$                                     ▷prev. state
8:     $E \leftarrow \perp$                                           ▷expect. val.

1: function $Next(\sigma')$
2:     $\overline{c}_\sigma \leftarrow \overline{c}_\sigma + 1$                    ▷update counters
3:     $\overline{c}_{\sigma\sigma'} \leftarrow \overline{c}_{\sigma\sigma'} + 1$
4:     if active $= false$ then
5:         if $(\forall v_{ij} \in V_\varphi \ . \ \overline{c}_{ij} + m_{ij} > 0)$ then
6:             active $\leftarrow$ $true$                                  ▷Eq. 6 is true
7:             for $l \in [1 \,..\, p]$ do                                 ▷Eq. 5
8:                 $h^l \leftarrow \mathcal{H}^l\left(\{\overline{c}_{ij}\}_{i,j}, \{\overline{c}_i\}_i\right)$
9:     else
10:        for $l \in [1 \,..\, p]$ do                                     ▷Eq. 7
11:            $h^l \leftarrow h^l \cdot \frac{\overline{c}_{\sigma\sigma'} - 1 + d^l_{\sigma\sigma'}}{\overline{c}_{\sigma\sigma'} - 1} \cdot \frac{\overline{c}_\sigma - 1}{\overline{c}_\sigma - 1 + d^l_\sigma}$
12:    if active $= true$ then
13:        $E \leftarrow \sum_{l=1}^{p} \kappa_l \cdot h^l$                ▷overall expect.
14:    $\sigma \leftarrow \sigma'$
15:    return $E$
```

Condition (6) guarantees that the permutations in (5) are well-defined. The first equality in (7) follows from Marchal et al. [51], and the rest uses the conjugacy of the prior. Lemma 1 forms the basis of the efficient update of our Bayesian monitor. Observe that on any given path, once (6) holds, it continues to hold forever. Thus, initially the monitor keeps updating $\mathcal{H}$ internally without outputting anything. Once (6) holds, it keeps outputting $\mathcal{H}$ from then on.

5.2 Implementation of the Bayesian Monitor

We present the Bayesian monitor implementation in `BayesConfIntMonitor` (Algorithm 4), which invokes `BayesExpMonitor` (Algorithm 3) as subroutine. `BayesExpMonitor` computes the expected semantic value of an expression φ in polynomial form, by computing the individual expected value of each monomial using Propostion 1, and combining them using the linearity property. We drop the arguments from $\overline{c}_i(\cdot)$ and $\overline{c}_{ij}(\cdot)$ and simply write $\overline{c}_i$ and $\overline{c}_{ij}$ as constants associated to appropriate words. The symbol m_{ij} in Line 5 of *Init* is used as a bookkeeping variable for quickly checking the consistency condition (Eq. 6) in Line 5 of *Next*. In `BayesConfIntMonitor`, we compute the expected value and the variance of φ, by invoking `BayesExpMonitor` on φ and φ^2 respectively, and then compute the confidence interval using the Chebyshev's inequality. It can be observed in the *Next* subroutines of `BayesConfIntMonitor` and `BayesExpMonitor` that a deterministic transition function suffices for the Bayesian monitors.

Theorem 4 (Correctness). *Let (Q, φ, δ) be a problem instance, and p_θ be given as the prior distribution which satisfies Assumption 1. Algorithm 4 produces a monitor for (Q, φ, δ) that solves Problem 2.*

Theorem 5 Computational resources). *Let (Q, φ, δ) be a problem instance and $\mathcal{A}$ be the monitor computed using the* `BayesConfIntMonitor` *routine of*

Algorithm 4. `BayesConfIntMonitor`

```
Parameters: $Q, \varphi, \theta$
Output: $\Lambda$
function $Init(\sigma = 1)$
    $\overline{\varphi} \xleftarrow{\text{polyn.}} \varphi$, $\overline{\varphi^2} \xleftarrow{\text{polyn.}} \varphi^2$        ▷polyn. form
    $EXP \leftarrow$ BayesExpMonitor$(Q, \overline{\varphi}, \theta)$
    $EXP2 \leftarrow$ BayesExpMonitor$(Q, \overline{\varphi^2}, \theta)$
    $EXP.Init(\sigma)$
    $EXP2.Init(\sigma)$
    $\Lambda \leftarrow \bot$

function $Next(\sigma')$
    $E \leftarrow EXP.Next(\sigma')$
    $E2 \leftarrow EXP2.Next(\sigma')$
    if $E \neq \bot$ and $E2 \neq \bot$ then
        $S \leftarrow E2 - E^2$        ▷variance
        $\Lambda \leftarrow \left[E \pm \sqrt{\frac{S}{\delta}}\right]$        ▷Chebysh.
    return $\Lambda$
```

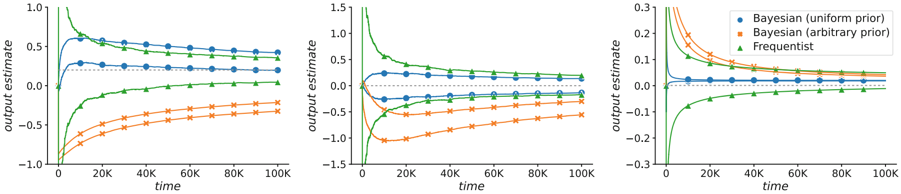
Algorithm 4. Suppose the size of φ is n. The monitor $\mathcal{A}$ requires $\mathcal{O}(n^2 2^n)$ registers, and takes $\mathcal{O}(n^2 2^n)$ time to update its output after receiving a new input symbol. For the special case of φ being in polynomial form, $\mathcal{A}$ requires only $\mathcal{O}(n^2)$ registers, and takes only $\mathcal{O}(n^2)$ time to update its output after receiving a new input symbol.

A bound on the convergence speed of the Bayesian monitor is left open. This would require a bound on the change in variance with respect to the length of the observed path, which is not known for the general case of PSEs. Note that the efficient (quadratic) cases are different for the frequentist and Bayesian monitors, suggesting the use of different monitors for different specifications.

6 Experiments

We implemented our frequentist and Bayesian monitors in a tool written in Rust, and used the tool to design monitors for the lending and the college admission examples taken from the literature [48,54] (described in Sect. 1.1). The generators are modeled as Markov chains (see Fig. 1)—unknown to the monitors—capturing the sequential interactions between the decision-makers (i.e., the bank or the college) and their respective environments (i.e., the loan applicants or the students), as described by D'Amour et al. [16]. The setup of the experiments is as follows: We created a multi-threaded wrapper program, where one thread simulates one long run of the Markov chain, and a different thread executes the monitor. Every time a new state is visited by the Markov chain on the first thread, the information gets transmitted to the monitor on the second thread, which then updates the output. The experiments were run on a Macbook Pro 2017 equipped with a 2,3 GHz Dual-Core Intel Core i5 processor and 8GB RAM. The tool can be downloaded from the following url, where we have also included the scripts to reproduce our experiments: https://github.com/ista-fairness-monitoring/fmlib.

We summarize the experimental results in Fig. 3, and, from the table, observe that both monitors are extremely lightweight: they take less than a millisecond per update and small numbers of registers to operate. From the plots, we observe that the frequentist monitors' outputs are always centered around the ground truth values of the properties, empirically showing that they are always objectively correct. On the other hand, the Bayesian monitors' outputs can vary drastically for different choices of the prior, empirically showing that the correctness of outputs is subjective. It may be misleading that the outputs of the Bayesian monitors are wrong as they often do not contain the ground truth values. We reiterate that from the Bayesian perspective, the ground truth does not exist. Instead, we only have a probability distribution over the true values that gets updated after observing the generated sequence of events. The choice of the type of monitor ultimately depends on the application requirements.



Scenario	Size of expression	Av. comp. time/step		# registers	
		Freq.	Bayes.	Freq.	Bayes.
Lending (bias) + dem. par.	1	13.0µs	29.3µs	15	17
Lending (fair) + eq. opp.	5	21.6µs	31.0µs	29	27
Admission + soc. burden	19	53.8µs	184.6µs	46	102

Fig. 3. The plots show the 95% confidence intervals estimated by the monitors over time, averaged over 10 different sample paths, for the lending with demographic parity (left), lending with equalized opportunity (middle), and the college admission with social burden (right) problems. The horizontal dotted lines are the ground truth values of the properties, obtained by analyzing the Markov chains used to model the systems (unknown to the monitors). The table summarizes various performance metrics.

7 Conclusion

We showed how to monitor algorithmic fairness properties on a Markov chain with unknown transition probabilities. Two separate algorithms are presented, using the frequentist and the Bayesian approaches to statistics. The performances of both approaches are demonstrated, both theoretically and empirically.

Several future directions exist. Firstly, more expressive classes of properties need to be investigated to cover a broader range of algorithmic fairness criteria. We believe that boolean logical connectives, as well as min and max operators can be incorporated straightforwardly using ideas from the related literature [3]. This also adds support for absolute values, since $|x| = \max\{x, -x\}$. On the other hand, properties that require estimating how often a state is visited would require more information about the dynamics of the Markov chain, including its mixing time. Monitoring statistical hyperproperties [18] is another important direction, which will allow us to encode individual fairness properties [21]. Secondly, more liberal assumptions on the system model will be crucial for certain practical applications. In particular, hidden Markov models, time-inhomogeneous Markov models, Markov decision processes, etc., are examples of system models with widespread use in real-world applications. Finally, better error bounds tailored for specific algorithmic fairness properties can be developed through a deeper mathematical analysis of the underlying statistics, which will sharpen the conservative bounds obtained through off-the-shelf concentration inequalities.

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. (TOMACS) **28**(1), 1–39 (2018)
2. Albarghouthi, A., D'Antoni, L., Drews, S., Nori, A.V.: Fairsquare: probabilistic verification of program fairness. Proc. ACM Program. Lang. **1**(OOPSLA), 1–30 (2017)
3. Albarghouthi, A., Vinitsky, S.: Fairness-aware programming. In: Proceedings of the Conference on Fairness, Accountability, and Transparency, pp. 211–219 (2019)
4. Ashok, P., Křetínský, J., Weininger, M.: PAC statistical model checking for Markov decision processes and stochastic games. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 497–519. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_29
5. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Softw. Eng. **29**(6), 524–541 (2003). https://doi.org/10.1109/TSE.2003.1205180
6. Balunovic, M., Ruoss, A., Vechev, M.: Fair normalizing flows. In: International Conference on Learning Representations (2021)
7. Bartocci, E., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
8. Bartocci, E., Falcone, Y.: Lectures on Runtime Verification. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-75632-5
9. Bastani, O., Zhang, X., Solar-Lezama, A.: Probabilistic verification of fairness properties via concentration. Proc. ACM Program. Lang. **3**(OOPSLA), 1–27 (2019)
10. Bellamy, R.K., et al.: Ai fairness 360: an extensible toolkit for detecting and mitigating algorithmic bias. IBM J. Res. Dev. **63**(4/5), 4–1 (2019)
11. Berk, R., et al.: A convex framework for fair regression. arXiv preprint arXiv:1706.02409 (2017)
12. Bird, S., et al.: Fairlearn: a toolkit for assessing and improving fairness in ai. Microsoft, Technical Report. MSR-TR-2020-32 (2020)
13. Chouldechova, A.: Fair prediction with disparate impact: a study of bias in recidivism prediction instruments. Big Data **5**(2), 153–163 (2017)
14. Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 1–12. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_1
15. Corbett-Davies, S., Pierson, E., Feller, A., Goel, S., Huq, A.: Algorithmic decision making and the cost of fairness. In: Proceedings of the 23rd ACM Sigkdd International Conference on Knowledge Discovery and Data Mining, pp. 797–806 (2017)

16. D'Amour, A., Srinivasan, H., Atwood, J., Baljekar, P., Sculley, D., Halpern, Y.: Fairness is not static: deeper understanding of long term fairness via simulation studies. In: Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency, FAT* 2020, pp. 525–534 (2020)
17. David, A., Du, D., Guldstrand Larsen, K., Legay, A., Mikučionis, M.: Optimizing control strategy using statistical model checking. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 352–367. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_24
18. Dimitrova, R., Finkbeiner, B., Torfah, H.: Probabilistic hyperproperties of markov decision processes (2020). https://doi.org/10.48550/ARXIV.2005.03362, https://arxiv.org/abs/2005.03362
19. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9
20. Dressel, J., Farid, H.: The accuracy, fairness, and limits of predicting recidivism. Sci. Adv. **4**(1), eaao5580 (2018)
21. Dwork, C., Hardt, M., Pitassi, T., Reingold, O., Zemel, R.: Fairness through awareness. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 214–226 (2012)
22. Dwork, C., Ilvento, C.: Individual fairness under composition. In: Proceedings of Fairness, Accountability, Transparency in Machine Learning (2018)
23. Ensign, D., Friedler, S.A., Neville, S., Scheidegger, C., Venkatasubramanian, S.: Runaway feedback loops in predictive policing. In: Conference on Fairness, Accountability and Transparency, pp. 160–171. PMLR (2018)
24. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. arXiv preprint arXiv:1711.03829 (2017)
25. Feldman, M., Friedler, S.A., Moeller, J., Scheidegger, C., Venkatasubramanian, S.: Certifying and removing disparate impact. In: proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 259–268 (2015)
26. Ferrère, T., Henzinger, T.A., Kragl, B.: Monitoring event frequencies. In: Fernández, M., Muscholl, A. (eds.) 28th EACSL Annual Conference on Computer Science Logic (CSL 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 152, pp. 20:1–20:16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2020). https://doi.org/10.4230/LIPIcs.CSL.2020.20, https://drops.dagstuhl.de/opus/volltexte/2020/11663
27. Ferrère, T., Henzinger, T.A., Saraç, N.E.: A theory of register monitors. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 394–403 (2018)
28. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.: Collecting statistics over runtime executions. Electron. Notes Theor. Comput. Sci. **70**(4), 36–54 (2002)
29. Ghosh, B., Basu, D., Meel, K.S.: Justicia: a stochastic sat approach to formally verify fairness. arXiv preprint arXiv:2009.06516 (2020)
30. Ghosh, B., Basu, D., Meel, K.S.: Algorithmic fairness verification with graphical models. arXiv preprint arXiv:2109.09447 (2021)
31. Gómez-Corral, A., Insua, D.R., Ruggeri, F., Wiper, M.: Bayesian inference of markov processes. In: Wiley StatsRef: Statistics Reference Online, pp. 1–15 (2014)
32. Hardt, M., Price, E., Srebro, N.: Equality of opportunity in supervised learning. Adv. Neural Inf. Process. Syst. **29** (2016)

33. Henzinger, T.A., Karimi, M., Kueffner, K., Mallik, K.: Monitoring algorithmic fairness. arXiv preprint arXiv:2305.15979 (2023)
34. Henzinger, T.A., Karimi, M., Kueffner, K., Mallik, K.: Runtime monitoring of dynamic fairness properties. arXiv preprint arXiv:2305.04699 (2023). to appear in FAccT '23
35. Henzinger, T.A., Saraç, N.E.: Monitorability under assumptions. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 3–18. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_1
36. Henzinger, T.A., Saraç, N.E.: Quantitative and approximate monitoring. In: 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–14. IEEE (2021)
37. Insua, D., Ruggeri, F., Wiper, M.: Bayesian Analysis of Stochastic Process Models. John Wiley & Sons, Hoboken (2012)
38. Jagielski, M., et al.: Differentially private fair learning. In: International Conference on Machine Learning, pp. 3000–3008. PMLR (2019)
39. John, P.G., Vijaykeerthy, D., Saha, D.: Verifying individual fairness in machine learning models. In: Conference on Uncertainty in Artificial Intelligence, pp. 749–758. PMLR (2020)
40. Junges, S., Torfah, H., Seshia, S.A.: Runtime monitors for markov decision processes. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 553–576. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_26
41. Kamiran, F., Calders, T.: Data preprocessing techniques for classification without discrimination. Knowl. Inf. Syst. **33**(1), 1–33 (2012)
42. Kearns, M., Neel, S., Roth, A., Wu, Z.S.: Preventing fairness gerrymandering: auditing and learning for subgroup fairness. In: International Conference on Machine Learning, pp. 2564–2572. PMLR (2018)
43. Kleinberg, J., Mullainathan, S., Raghavan, M.: Inherent trade-offs in the fair determination of risk scores. In: Papadimitriou, C.H. (ed.) 8th Innovations in Theoretical Computer Science Conference (ITCS 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 67, pp. 43:1–43:23. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2017). https://doi.org/10.4230/LIPIcs.ITCS.2017.43, http://drops.dagstuhl.de/opus/volltexte/2017/8156
44. Knight, K.: Mathematical Statistics. CRC Press, Boca Raton (1999)
45. Konstantinov, N.H., Lampert, C.: Fairness-aware pac learning from corrupted data. J. Mach. Learn. Res. **23** (2022)
46. Kusner, M.J., Loftus, J., Russell, C., Silva, R.: Counterfactual fairness. Adv. Neural Inf. Process. Syst. **30** (2017)
47. Lahoti, P., Gummadi, K.P., Weikum, G.: ifair: learning individually fair data representations for algorithmic decision making. In: 2019 IEEE 35th International Conference on Data Engineering (icde), pp. 1334–1345. IEEE (2019)
48. Liu, L.T., Dean, S., Rolf, E., Simchowitz, M., Hardt, M.: Delayed impact of fair machine learning. In: International Conference on Machine Learning, pp. 3150–3158. PMLR (2018)
49. Lum, K., Isaac, W.: To predict and serve? Significance **13**(5), 14–19 (2016)
50. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
51. Marchal, O., Arbel, J.: On the sub-gaussianity of the beta and dirichlet distributions. Electron. Commun. Probabil. **22**, 1–14 (2017)

52. Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., Galstyan, A.: A survey on bias and fairness in machine learning. ACM Comput. Surv. (CSUR) **54**(6), 1–35 (2021)
53. Meyer, A., Albarghouthi, A., D'Antoni, L.: Certifying robustness to programmable data bias in decision trees. Adv. Neural Inf. Process. Syst. **34**, 26276–26288 (2021)
54. Milli, S., Miller, J., Dragan, A.D., Hardt, M.: The social cost of strategic classification. In: Proceedings of the Conference on Fairness, Accountability, and Transparency, pp. 230–239 (2019)
55. Obermeyer, Z., Powers, B., Vogeli, C., Mullainathan, S.: Dissecting racial bias in an algorithm used to manage the health of populations. Science **366**(6464), 447–453 (2019)
56. Otop, J., Henzinger, T.A., Chatterjee, K.: Quantitative automata under probabilistic semantics. Logical Methods Comput. Sci. **15** (2019)
57. Scheuerman, M.K., Paul, J.M., Brubaker, J.R.: How computers see gender: an evaluation of gender classification in commercial facial analysis services. In: Proceedings of the ACM on Human-Computer Interaction, vol. 3, no. CSCW, pp. 1–33 (2019)
58. Seyyed-Kalantari, L., Liu, G., McDermott, M., Chen, I.Y., Ghassemi, M.: Chexclusion: fairness gaps in deep chest x-ray classifiers. In: BIOCOMPUTING 2021: Proceedings of the Pacific Symposium, pp. 232–243. World Scientific (2020)
59. Sharifi-Malvajerdi, S., Kearns, M., Roth, A.: Average individual fairness: algorithms, generalization and experiments. Adv. Neural Inf. Process. Syst. **32** (2019)
60. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_15
61. Sun, B., Sun, J., Dai, T., Zhang, L.: Probabilistic verification of neural networks against group fairness. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 83–102. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_5
62. Wachter, S., Mittelstadt, B., Russell, C.: Bias preservation in machine learning: the legality of fairness metrics under eu non-discrimination law. W. Va. L. Rev. **123**, 735 (2020)
63. Wexler, J., Pushkarna, M., Bolukbasi, T., Wattenberg, M., Viégas, F., Wilson, J.: The what-if tool: Interactive probing of machine learning models. IEEE Trans. Vis. Comput. Graph. **26**(1), 56–65 (2019)
64. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_17
65. Zafar, M.B., Valera, I., Gomez-Rodriguez, M., Gummadi, K.P.: Fairness constraints: a flexible approach for fair classification. J. Mach. Learn. Res. **20**(1), 2737–2778 (2019)
66. Zemel, R., Wu, Y., Swersky, K., Pitassi, T., Dwork, C.: Learning fair representations. In: International Conference on Machine Learning, pp. 325–333. PMLR (2013)

nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models

Matthias Cosler[2], Christopher Hahn[1](✉), Daniel Mendoza[1](✉),
Frederik Schmitt[2], and Caroline Trippel[1]

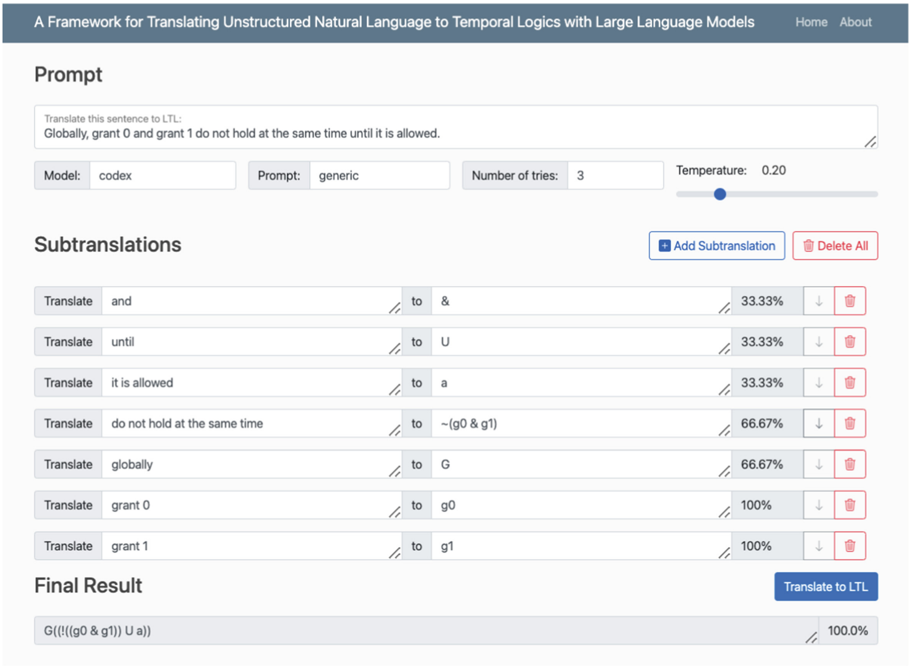
[1] Stanford University, Stanford, CA, USA
`hahn@cs.stanford.edu`, `{dmendo,trippel}@stanford.edu`
[2] CISPA Helmholtz Center for Information Security,
Saarbrücken, Germany
`{matthias.cosler,frederik.schmitt}@cispa.de`

Abstract. A rigorous formalization of desired system requirements is indispensable when performing any verification task. This often limits the application of verification techniques, as writing formal specifications is an error-prone and time-consuming manual task. To facilitate this, we present `nl2spec`, a framework for applying Large Language Models (LLMs) to derive formal specifications (in temporal logics) from unstructured natural language. In particular, we introduce a new methodology to detect and resolve the inherent ambiguity of system requirements in natural language: we utilize LLMs to map subformulas of the formalization back to the corresponding natural language fragments of the input. Users iteratively add, delete, and edit these sub-translations to amend erroneous formalizations, which is easier than manually redrafting the entire formalization. The framework is agnostic to specific application domains and can be extended to similar specification languages and new neural models. We perform a user study to obtain a challenging dataset, which we use to run experiments on the quality of translations. We provide an open-source implementation, including a web-based frontend.

1 Introduction

A rigorous formalization of desired system requirements is indispensable when performing any verification-related task, such as model checking [7], synthesis [6], or runtime verification [20]. Writing formal specifications, however, is an error-prone and time-consuming manual task typically reserved for experts in the field. This paper presents `nl2spec`, a framework, accompanied by a web-based tool, to facilitate and automate writing formal specifications (in LTL [34] and similar temporal logics). The core contribution is a new methodology to decompose the natural language input into *sub-translations* by utilizing Large Language Models (LLMs). The `nl2spec` framework provides an interface to interactively

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 383–396, 2023.
https://doi.org/10.1007/978-3-031-37703-7_18


A Framework for Translating Unstructured Natural Language to Temporal Logics with Large Language Models
Home
About
Add Subtranslation
Delete All
Translate to LTL

Prompt

Translate this sentence to LTL:
Globally, grant 0 and grant 1 do not hold at the same time until it is allowed.

Model: codex

Prompt: generic

Number of tries: 3

Temperature: 0.20

Subtranslations

Translate	and	to	&	33.33%
Translate	until	to	U	33.33%
Translate	it is allowed	to	a	33.33%
Translate	do not hold at the same time	to	~(g0 & g1)	66.67%
Translate	globally	to	G	66.67%
Translate	grant 0	to	g0	100%
Translate	grant 1	to	g1	100%

Final Result

G((!((g0 & g1)) U a)) 100.0%

Fig. 1. A screenshot of the web-interface for `nl2spec`.

add, edit, and delete these *sub-translations* instead of attempting to grapple with the entire formalization at once (a feature that is sorely missing in similar work, e.g., [13,30]).

Figure 1 shows the web-based frontend of `nl2spec`. As an example, we consider the following system requirement given in natural language: "Globally, grant 0 and grant 1 do not hold at the same time until it is allowed". The tool automatically translates the natural language specification correctly into the LTL formula `G((!((g0 & g1)) U a))`. Additionally, the tool generates sub-translations, such as the pair ("do not hold at the same time", `!(g0 & g1)`), which help in verifying the correctness of the translation.

Consider, however, the following ambiguous example: "a holds until b holds or always a holds". Human supervision is needed to resolve the ambiguity on the operator precedence. This can be easily achieved with `nl2spec` by adding or editing a sub-translation using explicit parenthesis (see Sect. 4 for more details and examples). To capture such (and other types of) ambiguity in a benchmark data set, we conducted an expert user study specifically asking for challenging translations of natural language sentences to LTL formulas.

The key insight in the design of `nl2spec` is that the process of translation can be decomposed into many sub-translations automatically via LLMs, and the decomposition into sub-translations allows users to easily resolve ambiguous natural language and erroneous translations through interactively modifying sub-translations. The central goal of `nl2spec` is to keep the human supervision minimal and efficient. To this end, all translations are accompanied by a confidence score. Alternative suggestions for sub-translations can be chosen via a drop-down menu and misleading sub-translations can be deleted before the next loop of the translation. We evaluate the end-to-end translation accuracy of our proposed methodology on the benchmark data set obtained from our expert user study. Note that `nl2spec` can be applied to the user's respective application domain to increase the quality of translation. As proof of concept, we provide additional examples, including an example for STL [31] in the GitHub repository[1].

`nl2spec` is agnostic to machine learning models and specific application domains. We will discuss possible parameterizations and inputs of the tool in Sect. 3. We discuss our sub-translation methodology in more detail in Sect. 3.2 and introduce an interactive few-shot prompting scheme for LLMs to generate them. We evaluate the effectiveness of the tool to resolve erroneous formalizations in Sect. 4 on a data set obtained from conducting an expert user study. We discuss limitations of the framework and conclude in Sect. 5. For additional details, please refer to the complete version [8].

2 Background and Related Work

2.1 Natural Language to Linear-Time Temporal Logic

Linear-time Temporal Logic (LTL) [34] is a temporal logic that forms the basis of many practical specification languages, such as the IEEE property specification language (PSL) [22], Signal Temporal Logic (STL) [31], or System Verilog Assertions (SVA) [43]. By focusing on the prototype temporal logic LTL, we keep the `nl2spec` framework extendable to specification languages in specific application domains. LTL extends propositional logic with temporal modalities $\mathtt{U}$ (until) and $\mathtt{X}$ (next). There are several derived operators, such as $\mathtt{F}\varphi \equiv \mathit{true}\mathtt{U}\varphi$ and $\mathtt{G}\varphi \equiv \neg\mathtt{F}\neg\varphi$. $\mathtt{F}\varphi$ states that φ will *eventually* hold in the future and $\mathtt{G}\varphi$ states that φ holds *globally*. Operators can be nested: $\mathtt{GF}\varphi$, for example, states that φ has to occur infinitely often. LTL specifications describe a systems behavior and its interaction with an environment over time. For example given a process 0 and a process 1 and a shared resource, the formula $\mathtt{G}(r_0 \rightarrow \mathtt{F}g_0) \wedge \mathtt{G}(r_1 \rightarrow \mathtt{F}g_1) \wedge \mathtt{G}\neg(g_0 \wedge g_1)$ describes that whenever a process requests (r_i) access to a shared resource it will eventually be granted (g_i). The subformula $\mathtt{G}\neg(g_0 \wedge g_1)$ ensures that grants given are mutually exclusive.

Early work in translating natural language to temporal logics focused on grammar-based approaches that could handle structured natural language [17, 24]. A survey of earlier research before the advent of deep learning is provided in [4]. Other approaches include an interactive method using SMT solving and semantic parsing [15], or structured temporal aspects in grounded robotics [45] and planning [32]. Neural networks have only recently been used to translate

[1] The tool is available at GitHub: https://github.com/realChrisHahn2/nl2spec.

into temporal logics, e.g., by training a model for STL from scratch [21], fine-tuning language models [19], or an approach to apply GPT-3 [13,30] in a one-shot fashion, where [13] output a restricted set of declare templates [33] that can be translated to a fragment of LTLf [10]. Translating natural langauge to LTL has especially been of interest to the robotics community (see [16] for an overview), where datasets and application domains are, in contrast to our setting, based on structured natural language. Independent of relying on structured data, all previous tools lack a detection and interactive resolving of the inerherent ambiguity of natural language, which is the main contribution of our framework. Related to our approach is recent work [26], where generated code is iteratively refined to match desired outcomes based on human feedback.

2.2 Large Language Models

LLMs are large neural networks typically consisting of up to 176 billion parameters. They are pre-trained on massive amounts of data, such as "The Pile" [14]. Examples of LLMs include the GPT [36] and BERT [11] model families, open-source models, such as T5 [38] and Bloom [39], or commercial models, such as Codex [5]. LLMs are Transformers [42], which is the state of the art neural architecture for natural language proccessing. Additionally, Transformers have shown remarkable performance when being applied to classical problems in verification (e.g., [9,18,25,40]), reasoning (e.g., [28,50]), as well as the auto-formalization [35] of mathematics and formal specifications (e.g., [19,21,49]).

In language modelling, we model the probability of a sequence of tokens in a text [41]. The joint probability of tokens in a text is generally expressed as [39]:

$$p(x) = p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_{<t}) \ ,$$

where x is the sequence of tokens, x_t represents the t-th token, and $x_{<t}$ is the sequence of tokens preceding x_t. We refer to this as an autoregressive language model that iteratively predicts the probability of the next token. Neural network approaches to language modelling have superseded classical approaches, such as n-grams [41]. Especially Transformers [42] were shown to be the most effective architecture at the time of writing [1,23,36].

While fine-tuning neural models on a specific translation task remains a valid approach showing also initial success in generalizing to unstructured natural language when translating to LTL [19], a common technique to obtain high performance with limited amount of labeled data is so-called "few-shot prompting" [3]. The language model is presented a natural language description of the task usually accompanied with a few examples that demonstrate the input-output behavior. The framework presented in this paper relies on this technique. We describe the proposed few-shot prompting scheme in detail in Sect. 3.2.

Currently implemented in the framework and used in the expert-user study are Codex and Bloom, which showed the best performance during testing.

Codex and GPT-3.5-turbo. Codex [5] is a GPT-3 variant that was initially of up to 12B parameters in size and fine-tuned on code. The initial version of GPT-3 itself was trained on variations of Common Crawl,[2] Webtext-2 [37], two internet-based book corpora and Wikipedia [3]. The fine-tuning dataset for the vanilla version Codex was collected in May 2020 from 54 million public software repositories hosted on GitHub, using 159GB of training data for fine-tuning. For our experiments, we used the commercial 2022 version of `code-davinci-002`, which is likely larger (in the 176B range[3]) than the vanilla codex models. GPT-3.5-turbo is the currently available follow-up model of GPT-3.

Bloom. Bloom [39] is an open-source LLM family available in different sizes of up to 176B parameters trained on 46 natural languages and 13 programming languages. It was trained on the `ROOTS` corpus [27], a collection of 498 huggingface [29,48] datasets consisting of 1.61 terabytes of text. For our experiments, we used the 176B version running on the huggingface inference API[4].

3 The `nl2spec` Framework

3.1 Overview

The framework follows a standard frontend-backend implementation. Figure 2 shows an overview of the implementation of `nl2spec`. Parts of the framework that can be extended for further research or usage in practice are highlighted. The framework is implemented in Python 3 and flask [44], a lightweight WSGI web application framework. For the experiments in this paper, we use the OpenAI library and huggingface (transformer) library [47]. We parse the LTL output formulas with a standard LTL parser [12]. The tool can either be run as a command line tool, or with the web-based frontend.

The frontend handles the interaction with a human-in-the-loop. The interface is structured in three views: the "Prompt", "Sub-translations", and "Final Result" view (see Fig. 1). The tool takes a natural language sentence, optional sub-translations, the model temperature, and number of runs as input. It provides sub-translations, a confidence score, alternative sub-translations and the final formalization as output. The frontend then allows for interactively selecting, editing, deleting, or adding sub-translations. The backend implements the handling of the underlying neural models, the generation of the prompt, and the ambiguity resolving, i.e., computing the confidence score including alternative sub-translations and the interactive few-shot prompting algorithm (cf. Sect. 3.2). The framework is designed to have an easy interface to implement new models and write domain-specific prompts. The prompt is a .txt file that can be adjusted to specific domains to increase the quality of translations. To apply the sub-translation refinement methodology, however, the prompt needs to follow our interactive prompting scheme, which we introduce in the next section.

[2] https://commoncrawl.org/.

[3] https://blog.eleuther.ai/gpt3-model-sizes/.

[4] https://huggingface.co/inference-api.

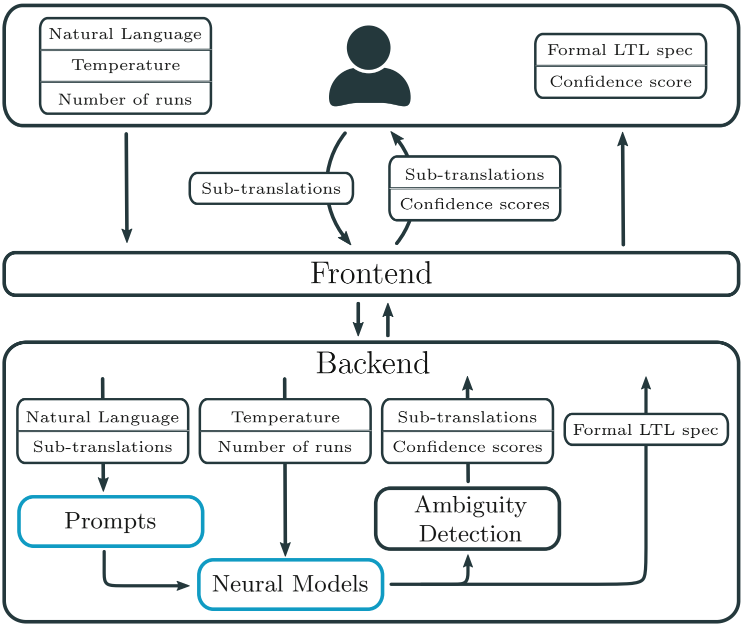


Fig. 2. Overview of the `nl2spec` framework with a human-in-the-loop: highlighted areas indicate parts of the framework that are effortlessly extendable.

3.2 Interactive Few-Shot Prompting

The core of the methodology is the decomposition of the natural language input into sub-translations. We introduce an interactive prompting scheme that generates sub-translations using the underlying neural model and leverages the sub-translations to produce the final translation. Algorithm 1 depicts a high-level overview of the interactive loop. The main idea is to give a human-in-the-loop the options to add, edit, or delete sub-translations and feed them back into the language models as "Given translations" in the prompt (see Fig. 3). After querying a language model M with this prompt F, model specific parameters P and the interactive prompt that is computed in the loop, the model generates a natural language explanation, a dictionary of sub-translations, and the final translation. Notably, the model M can be queried multiple times as specified by the number of runs r, thereby generating multiple possible sub-translations. The confidence score of each sub-translation is computed as votes over multiple queries and by default the sub-translation with the highest confidence score is selected to be used as a given sub-translation in the next iteration. In the frontend, the user may view and select alternative generated sub-translations for each sub-translation via a drop-down menu (see Fig. 1).

Figure 3 shows a generic prompt, that illustrates our methodology. The prompting scheme consists of three parts. The specification language specific part (lines 1–4), the fewshot examples (lines 5–19), and the interactive prompt

```
minimal.txt

 1  Translate the following natural language sentences into an LTL formula and explain your
 2  translation step by step. Remember that X means "next", U means "until", G means
 3  "globally", F means "finally", which means GF means "infinitely often". The formula
 4  should only contain atomic propositions or operators  &, ~, ->, <->, X, U, G, F.
 5  Natural Language: Globally if a holds then c is true until b. Given translations: {}
 6  Explanation: "a holds" from the input translates to the atomic proposition a.
 7  "c is true until b" from the input translates to the subformula c U b. "if x then y"
 8  translates to an implication x -> y, so "if a holds then c is true until b" translates
 9  to an implication a -> c U b. "Globally" from the input translates to the temporal
10  operator G. Explanation dictionary: {"a holds" : "a", "c is true until b" : "c U b",
11  "if a holds then c is true until b" : "a -> c U b", "Globally" : "G"} So the final
12  LTL translation is G a -> c U b.FINISH Natural Language: Every request r is
13  eventually followed by a grant g. Given translations: {} Explanation: "Request r"
14  from the input translates to the atomic proposition r and "grant g" translates to the
15  atomic proposition g. "every" means at every point in time, i.e., globally, "never"
16  means at no point in time, and "eventually" translates to the temporal operator F.
17  "followed by" is the natural language representation of an implication. Explanation
18  dictionary: {"Request r" : "r", "grant g" : "g", "every" : "G", "eventually": "F",
19  "followed by" : "->"} So the final LTL translation is G r -> F g.FINISH
```

Fig. 3. Prompt with minimal domain knowledge of LTL.

including the natural language and sub-translation inputs (not displayed, given as input). The specification language specific part leverages "chain-of-thought" prompt-engineering to elicit reasoning from large language models [46]. The key of nl2spec, however, is the setup of the few-shot examples. This minimal prompt consists of two few-shot examples (lines 5–12 and 12–19). The end of an example is indicated by the "FINISH" token, which is the stop token for the machine learning models. A few-shot example in nl2spec consists of the natural language input (line 5), a dictionary of given translations, i.e., the sub-translations (line 5), an explanation of the translation in natural language (line 6–10), an explanation dictionary, summarizing the sub-translations, and finally, the final LTL formula.

This prompting scheme elicits sub-translations from the model, which serve as a fine-grained explanation of the formalization. Note that sub-translations provided in the prompt are neither unique nor exhaustive, but provide the context for the language model to generate the correct formalization.

4 Evaluation

In this section, we evaluate our framework and prompting methodology on a data set obtained by conducting an expert user study. To show the general applicability of this framework, we use the minimal prompt that includes only minimal domain knowledge of the specification language (see Fig. 3). This prompt has intentionally been written *before* conducting the expert user study. We limited the few-shot examples to two and even provided no few-shot example that includes "given translations". We use the minimal prompt to focus the evaluation on the effectiveness of our interactive sub-translation refinement methodology in

Algorithm 1: Interactive Few-shot Prompting Algorithm

```
Input: Natural language $S$, Few-shot prompt $F$, set of given sub-translations
  $(s, \varphi)$, and language model $M$
Interactions: set of sub-translations $(s, \varphi)$, confidence scores $C$
Set of Model specific parameter $P$: e.g., model-temperature $t$, number of
  runs $r$
Output: LTL formula $\psi$ that formalizes $S$
$\psi$, $(s, \varphi)$ , $C$ = empty
while user not approves LTL formula $\psi$ do
    interactive_prompt = compute_prompt(S, F, $(s, \varphi)$)
    $\psi$, $(s, \varphi)$ , $C$ = query(M, P, interactive_prompt)
    $(s, \varphi)$ = user_interaction($(s, \varphi)$ , $C$)
end while
return $\psi$
```

resolving ambiguity and fixing erroneous translations. In practice, one would like to replace this minimal prompt with domain-specific examples that capture the underlying distribution as closely as possible. As a proof of concept, we elaborate on this in the full version [8].

4.1 Study Setup

To obtain a benchmark dataset of *unstructured* natural language and their formalizations into LTL, we asked five experts in the field to provide examples that the experts thought are challenging for a neural translation approach. Unlike existing datasets that follow strict grammatical and syntatical structure, we posed no such restrictions on the study participants. Each natural language specification was restricted to one sentence and to five atomic propositions a, b, c, d, e. Note that `nl2spec` is not restricted to a specific set of atomic propositions (cf. Fig. 1). Which variable scheme to use can be specified as an initial sub-translation. We elaborate on this in the full version [8]. To ensure unique instances, the experts worked in a shared document, resulting in 36 benchmark instances. We provide three randomly drawn examples for the interested reader:

natural language S	LTL specification ψ
If b holds then, in the next step, c holds until a holds or always c holds	`b -> X ((c U a) || G c)`
If b holds at some point, a has to hold somewhere beforehand	`(F b) -> (!b U (a & !b))`
One of the following aps will hold at all instances: a,b,c	`G( a | b | c)`

The poor performance of existing methods (cf. Table 1) exemplify the difficulty of this data set.

4.2 Results

We evaluated our approach using the `minimal` prompt (if not otherwise stated), with number of runs set to three and with a temperature of 0.2.

Quality of Initial Translation. We analyze the quality of *initial* translations, i.e., translations obtained *before* any human interaction. This experiment demonstrates that the initial translations are of high quality, which is important to ensure an efficient workflow. We compared our approach to fine-tuning language models on structured data [19] and to an approach using GPT-3 or Rasa [2] to translate natural language into a restricted set of declare patterns [13] (which could not handle most of the instances in the benchmark data set, even when replacing the atomic propositions with their used entities). The results of evaluating the accuracy of the initial translations on our benchmark expert set is shown in Table 1.

At the time of writing, using Codex in the backend outperforms GPT-3.5-turbo and Bloom on this task, by correctly translating 44.4% of the instances using the minimal prompt. We only count an instance as correctly translated if it matches the intended meaning of the expert, no alternative translation to ambiguous input was accepted. Additionally to the experiments using the minimal prompt, we conducted experiments on an augmented prompt with in-distribution examples after the user study was conducted by randomly drawing four examples from the expert data set (3 of the examples haven't been solved before, see the GitHub repository or full version for more details). With this in-distribution prompt (ID), the tool translates 21 instances (with the four drawn examples remaining in the set), i.e., 58.3% correctly.

This experiment shows 1) that the initial translation quality is high and can handle unstructured natural language better than previous approaches and 2) that drawing the few-shot examples in distribution only slightly increased translation quality for this data set; making the key contributions of nl2spec, i.e., ambiguity detection and effortless debugging of erroneous formalizations, valuable. Since nl2spec is agnostic to the underlying machine learning models, we expect an even better performance in the future with more fine-tuned models.

Teacher-Student Experiment. In this experiment, we generate an initial set of sub-translations with Codex as the underlying neural model. We then ran the tool with Bloom as a backend, taking these sub-translations as input. There were 11 instances that Codex could solve initially that Bloom was unable to solve. On these instances, Bloom was able to solve 4 more instances, i.e., 36.4% with sub-translations provided by Codex. The four instances that Bloom was able to solve

Table 1. Translation accuracy on the benchmark data set, where B stands for Bloom and C stands for Codex and G for GPT-3.5-Turbo.

nl2ltl [13]	T-5 [19]	nl2spec+B	nl2spec+C	nl2spec+C	nl2spec+C
rasa	fine-tuned	initial	initial	initial+ID	interactive
1/36 (2.7%)	2/36 (5.5%)	5/36 (13.8%)	16/36 (44.4%)	21/36 (58.3%)	31/36 (86.1%)
–	–	–	nl2spec+G	nl2spec+G	nl2spec+G
			initial	initial+ID	interactive
–	–	–	12/36 (33.3%)	17/36 (47.2%)	21/36 (58.3%)

with the help of Codex were: “It is never the case that a and b hold at the same time.”, “Whenever a is enabled, b is enabled three steps later.”, “If it is the case that every a is eventually followed by a b, then c needs to holds infinitely often.”, and “One of the following aps will hold at all instances: a,b,c”. This demonstrates that our sub-translation methodology is a valid appraoch: improving the quality of the sub-translations indeed has a positive effect on the quality of the final formalization. This even holds true when using underperforming neural network models. Note that no supervision by a human was needed in this experiment to improve the formalization quality.

Ambiguity Detection. Out of the 36 instances in the benchmark set, at least 9 of the instances contain ambiguous natural language. We especially observed two classes of ambiguity: 1) ambiguity due to the limits of natural language, e.g., operator precedence, and 2) ambiguity in the semantics of natural language; `nl2spec` can help in resolving both types of ambiguity. Details for the following examples can be found in the full version [8].

An example for the first type of ambiguity from our dataset is the example mentioned in the introduction: “a holds until b holds or always a holds”, which the expert translated into `(a U b) | G a`. Running the tool, however, translated this example into `(a U (b | G(a)))`. By editting the sub-translation of “a holds until b holds” to `(a U b)` through adding explicit parenthesis, the tool translates as intended. An example for the second type of ambiguity is the following instance from our data set: “Whenever a holds, b must hold in the next two steps.” The intended meaning of the expert was `G (a -> (b | X b))`, whereas the tool translated this sentence into `G((a -> X(X(b))))`. After changing the sub-translation of “b must hold in the next two steps” to `b | X b`, the tool translates the input as intended.

Fixing Erroneous Translation. With the inherent ambiguity of natural language and the unstructured nature of the input, the tool's translation cannot be expected to be always correct in the first try. Verifying and debugging sub-translations, however, is significantly easier than redrafting the complete formula from scratch. Twenty instances of the data set were not correctly translated in an initial attempt using Codex and the minimal prompt in the backend (see Table 1). We were able to extract correct translations for 15 instances by performing at most three translation loops (i.e., adding, editing, and removing sub-translations), We were able to get correct results by performing 1.86 translation loops on average. For example, consider the instance, “whenever a holds, b holds as well”, which the tool mistakenly translated to `G(a & b)`. By fixing the sub-translation “b holds as well” to the formula fragment `-> b`, the sentence is translated as intended. Only the remaining five instances that contain highly complex natural language requirements, such as, “once a happened, b won't happen again” were need to be translated by hand.

In total, we correctly translated 31 out of 36 instances, i.e., 86.11% using the `nl2spec` sub-translation methodology by performing only 1.4 translation loops on average (see Table 1).

5 Conclusion

We presented nl2spec, a framework for translating unstructured natural language to temporal logics. A limitation of this approach is its reliance on computational resources at inference time. This is a general limitation when applying deep learning techniques. Both, commercial and open-source models, however, provide easily accessible APIs to their models. Additionally, the quality of initial translations might be influenced by the amount of training data on logics, code, or math that the underlying neural models have seen during pre-training.

At the core of nl2spec lies a methodology to decompose the natural language input into sub-translations, which are mappings of formula fragments to relevant parts of the natural language input. We introduced an interactive prompting scheme that queries LLMs for sub-translations, and implemented an interface for users to interactively add, edit, and delete the sub-translations, which avoids users from manually redrafting the entire formalization to fix erroneous translations. We conducted a user study, showing that nl2spec can be efficiently used to interactively formalize unstructured and ambigous natural language.

Acknowledgements. We thank OpenAI for providing academic access to Codex and Clark Barrett for helpful feedback on an earlier version of the tool.

References

1. Al-Rfou, R., Choe, D., Constant, N., Guo, M., Jones, L.: Character-level language modeling with deeper self-attention. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 3159–3166 (2019)
2. Bocklisch, T., Faulkner, J., Pawlowski, N., Nichol, A.: Rasa: open source language understanding and dialogue management. arXiv preprint arXiv:1712.05181 (2017)
3. Brown, T., et al.: Language models are few-shot learners. Adv. Neural Inf. Process. Syst. **33**, 1877–1901 (2020)
4. Brunello, A., Montanari, A., Reynolds, M.: Synthesis of ltl formulas from natural language texts: state of the art and research directions. In: 26th International Symposium on Temporal Representation and Reasoning (TIME 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
5. Chen, M., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
6. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. J. Symb. Logic **28**(4) (1963)
7. Clarke, E.M.: Model checking. In: Ramesh, S., Sivakumar, G. (eds.) FSTTCS 1997. LNCS, vol. 1346, pp. 54–56. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0058022
8. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: interactively translating unstructured natural language to temporal logics with large language models. arXiv preprint arXiv:2303.04864 (2023)
9. Cosler, M., Schmitt, F., Hahn, C., Finkbeiner, B.: Iterative circuit repair against formal specifications. In: International Conference on Learning Representations (to appear) (2023)

10. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI 2013 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence, pp. 854–860. Association for Computing Machinery (2013)
11. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
12. Fuggitti, F.: LTLf2DFA. Zenodo (2019). https://doi.org/10.5281/ZENODO.3888410, https://zenodo.org/record/3888410
13. Fuggitti, F., Chakraborti, T.: Nl2ltl-a python package for converting natural language (nl) instructions to linear temporal logic (ltl) formulas (2023)
14. Gao, L., et al.: The pile: an 800 gb dataset of diverse text for language modeling. arXiv preprint arXiv:2101.00027 (2020)
15. Gavran, I., Darulova, E., Majumdar, R.: Interactive synthesis of temporal specifications from examples and natural language. Proc. ACM Program. Lang. **4**(OOPSLA), 1–26 (2020)
16. Gopalan, N., Arumugam, D., Wong, L.L., Tellex, S.: Sequence-to-sequence language grounding of non-markovian task specifications. In: Robotics: Science and Systems, vol. 2018 (2018)
17. Grunske, L.: Specification patterns for probabilistic quality properties. In: 2008 ACM/IEEE 30th International Conference on Software Engineering, pp. 31–40. IEEE (2008)
18. Hahn, C., Schmitt, F., Kreber, J.U., Rabe, M.N., Finkbeiner, B.: Teaching temporal logics to neural networks. In: International Conference on Learning Representations (2021)
19. Hahn, C., Schmitt, F., Tillman, J.J., Metzger, N., Siber, J., Finkbeiner, B.: Formal specifications from natural language. arXiv preprint arXiv:2206.01962 (2022)
20. Havelund, K., Roşu, G.: Monitoring java programs with java pathexplorer. Electron. Notes Theor. Comput. Sci. **55**(2), 200–217 (2001)
21. He, J., Bartocci, E., Ničković, D., Isakovic, H., Grosu, R.: Deepstl: from english requirements to signal temporal logic. In: Proceedings of the 44th International Conference on Software Engineering, pp. 610–622 (2022)
22. IEEE-Commission, et al.: IEEE standard for property specification language (PSL). IEEE Std 1850–2005 (2005)
23. Kaplan, J., et al.: Scaling laws for neural language models. arXiv preprint arXiv:2001.08361 (2020)
24. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, pp. 372–381 (2005)
25. Kreber, J.U., Hahn, C.: Generating symbolic reasoning problems with transformer gans. arXiv preprint arXiv:2110.10054 (2021)
26. Lahiri, S.K., et al.: Interactive code generation via test-driven user-intent formalization. arXiv preprint arXiv:2208.05950 (2022)
27. Laurençon, H., et al.: The bigscience roots corpus: a 1.6 tb composite multilingual dataset. In: Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (2022)
28. Lewkowycz, A., et al.: Solving quantitative reasoning problems with language models. arXiv preprint arXiv:2206.14858 (2022)
29. Lhoest, Q., et al.: Datasets: a community library for natural language processing. arXiv preprint arXiv:2109.02846 (2021)

30. Liu, J.X., et al.: Lang2ltl: translating natural language commands to temporal specification with large language models. In: Workshop on Language and Robotics at CoRL 2022
31. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
32. Patel, R., Pavlick, R., Tellex, S.: Learning to ground language to temporal logical form. In: NAACL (2019)
33. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) BPM 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006). https://doi.org/10.1007/11837862_18
34. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57. IEEE (1977)
35. Rabe, M.N., Szegedy, C.: Towards the automatic mathematician. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 25–37. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_2
36. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training (2018)
37. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
38. Raffel, C., et al.: Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. **21**(140), 1–67 (2020). http://jmlr.org/papers/v21/20-074.html
39. Scao, T.L., et al.: Bloom: a 176b-parameter open-access multilingual language model. arXiv preprint arXiv:2211.05100 (2022)
40. Schmitt, F., Hahn, C., Rabe, M.N., Finkbeiner, B.: Neural circuit synthesis from specification patterns. Adv. Neural Inf. Process. Syst. **34**, 15408–15420 (2021)
41. Shannon, C.E.: A mathematical theory of communication. Bell Syst. Tech. J. **27**(3), 379–423 (1948)
42. Vaswani, A., et al.: Attention is all you need. Adv. Neural Inf. Process. Syst. **30** (2017)
43. Vijayaraghavan, S., Ramanathan, M.: A Practical Guide for SystemVerilog Assertions. Springer, Heidelberg (2005). https://doi.org/10.1007/b137011
44. Vyshnavi, V.R., Malik, A.: Efficient way of web development using python and flask. Int. J. Recent Res. Asp **6**(2), 16–19 (2019)
45. Wang, C., Ross, C., Kuo, Y.L., Katz, B., Barbu, A.: Learning a natural-language to ltl executable semantic parser for grounded robotics. arXiv preprint arXiv:2008.03277 (2020)
46. Wei, J., et al.: Chain of thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903 (2022)
47. Wolf, T., et al.: Huggingface's transformers: state-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019)
48. Wolf, T., et al.: Transformers: State-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45 (2020)
49. Wu, Y., et al.: Autoformalization with large language models. arXiv preprint arXiv:2205.12615 (2022)
50. Zelikman, E., Wu, Y., Goodman, N.D.: Star: bootstrapping reasoning with reasoning. arXiv preprint arXiv:2203.14465 (2022)

NNV 2.0: The Neural Network Verification Tool

Diego Manzanas Lopez[1](✉), Sung Woo Choi[2], Hoang-Dung Tran[2], and Taylor T. Johnson[1]

[1] Vanderbilt University, Nashville, USA
diego.manzanas.lopez@vanderbilt.edu
[2] University of Nebraska, Lincoln, USA

Abstract. This manuscript presents the updated version of the Neural Network Verification (NNV) tool. NNV is a formal verification software tool for deep learning models and cyber-physical systems with neural network components. NNV was first introduced as a verification framework for feedforward and convolutional neural networks, as well as for neural network control systems. Since then, numerous works have made significant improvements in the verification of new deep learning models, as well as tackling some of the scalability issues that may arise when verifying complex models. In this new version of NNV, we introduce verification support for multiple deep learning models, including neural ordinary differential equations, semantic segmentation networks and recurrent neural networks, as well as a collection of reachability methods that aim to reduce the computation cost of reachability analysis of complex neural networks. We have also added direct support for standard input verification formats in the community such as VNNLIB (verification properties), and ONNX (neural networks) formats. We present a collection of experiments in which NNV verifies safety and robustness properties of feedforward, convolutional, semantic segmentation and recurrent neural networks, as well as neural ordinary differential equations and neural network control systems. Furthermore, we demonstrate the capabilities of NNV against a commercially available product in a collection of benchmarks from control systems, semantic segmentation, image classification, and time-series data.

Keywords: neural networks · cyber-physical systems · verification · tool

1 Introduction

Deep Learning (DL) models have achieved impressive performance on a wide range of tasks, including image classification [13,24,44], natural language processing [15,25], and robotics [47]. Recently, the usage of these models has expanded into many other areas, including safety-critical domains, such as autonomous vehicles [9,10,85]. However, deep learning models are opaque systems, and it has been demonstrated that their behavior can be unpredictable when small changes are applied to their inputs (i.e., adversarial attacks) [67].

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 397–412, 2023.
https://doi.org/10.1007/978-3-031-37703-7_19

Therefore, for safety-critical applications, it is often necessary to comprehend and analyze the behavior of the whole system, including reasoning about the safety guarantees of the system. To address this challenge, many researches have been developing techniques and tools to verify Deep Neural Networks (DNN) [4,6,22,39,40,48,55,64,65,77,83,84,86,87], as well as learning-enabled Cyber-Physical Systems (CPS) [3,8,12,23,26,34,35,38,50,51]. It is worth noting that despite the growing research interest, the verification of deep learning models still remains a challenging task, as the complexity and non-linearity of these models make them difficult to analyze. Moreover, some verification methods suffer from scalability issues, which limits the applicability of some existing techniques to large-scale and complex models. Another remaining challenge is the extension of existing or new methods for the verification of the extensive collection of layers and architectures existing in the DL area, such as Recurrent Neural Networks (RNN) [37], Semantic Segmentation Neural Networks (SSNN) [58] or Neural Ordinary Differential Equations (ODE) [11].

This work contributes to addressing the latter challenge by introducing version 2.0 of NNV[1] (*N*eural *N*etwork *V*erification)[2], which is a software tool that supports the verification of multiple DL models as well as learning-enabled CPS, also known as Neural Network Control Systems (NNCS) [80]. NNV is a software verification tool with the ability to compute exact and over-approximate reachable sets of feedforward neural networks (FFNN) [75,77,80], Convolutional Neural Networks (CNN) [78], and NNCS [73,80]. In NNV 2.0, we add verification support of 3 main DL models: 1) RNNs [74], 2) SSNNs (encoder-decoder architectures) [79], and 3) neural ODEs [52], as well as several other improvements introduced in Sect. 3, including support for The Verification of Neural Networks Library (VNNLIB) [29] and reachability methods for MaxUnpool and Leaky ReLU layers. Once the reachability computation is completed, NNV is capable of verifying a variety of specifications such as safety or robustness, very commonly used in learning-enabled CPS and classification domains, respectively [50,55]. We demonstrate NNV capabilities through a collection of safety and robustness verification properties, which involve the reachable set computation of feedforward, convolutional, semantic segmentation and recurrent neural networks, as well as neural ordinary differential equations and neural network control systems. Throughout these experiments, we showcase the range of the existing methods, executing up to 6 different star-based reachability methods that we compare against MATLAB's commercially available verification tool [69].

2 Related Work

The area of DNN verification has increasingly grown in recent years, leading to the development of standard input formats [29] as well as friendly competitions [50,55], that help compare and evaluate all the recent methods and tools proposed in the community [4,6,19,22,31,39–41,48,55,59,64,65,77,83,84,

[1] Code available at: https://github.com/verivital/nnv/releases/tag/cav2023.

[2] Archival version: https://doi.org/10.24433/CO.0803700.v1.

86,87]. However, the majority of these methods focus on regression and classification tasks performed by FFNN and CNN. In addition to FFNN and CNN verification, Tran et al. [79] introduced a collection of star-based reachability analysis that also verify SSNNs. Fischer et al. [21] proposed a probabilistic method for the robustness verification of SSNNs based on randomize smoothing [14]. Since then, some of the other recent tools, including Verinet [31], α,β-Crown [84,87], and MN-BaB [20] are also able to verify image segmentation properties as demonstrated in [55]. A less explored area is the verification of RNN. These models have unique "memory units" that enable them to store information for a period of time and learn complex patterns of time-series or sequential data. However, due to their memory units, verifying the robustness of RNNs is challenging. Recent notable state-of-the-art methodologies for verifying RNNs include unrolling the network into an FFNN and then verify it [2], invariant inference [36,62,90], and star-based reachability [74]. Similar to RNNs, neural ODEs are also deep learning models with "memory", which makes them suitable to learn time-series data, but are also applicable to other tasks such as continuous normalizing flows (CNF) and image classification [11,61]. However, existing work is limited to a stochastic reachability approach [27,28], reachability approaches using star and zonotope reachability methods for a general class of neural ODEs (GNODE) with continuous and discrete time layers [52], and GAINS [89], which leverages ODE-solver information to discretize the models using a computation graph that represent all possible trajectories from a given input to accelerate their bound propagation method. However, one of the main challenges is to find a framework that is able to verify several of these models successfully. For example, α,β-Crown was the top performer on last year's NN verification competition [55], able to verify FFNN, CNN and SSNNs, but it lacks support for neural ODEs or NNCS. There exist other tools that focus more on the verification of NNCS such as Verisig [34,35], Juliareach [63], ReachNN [17,33], Sherlock [16], RINO [26], VenMas [1], POLAR [32], and CORA [3,42]. However, their support is limited to NNCS with a linear, nonlinear ODE or hybrid automata as the plant model, and a FFNN as the controller.

Finally, for a more detailed comparison to state-of-the-art methods for the novel features of NNV 2.0, we refer to the comparison and discussion about neural ODEs in [52]. For SSNNs [79], there is a discussion on scalability and conservativeness of methods presented (approx and relax star) for the different layers that may be part of a SSNN [79]. For RNNs, the approach details and a state-of-the-art comparison can be found in [74]. We also refer the reader to two verification competitions, namely VNN-COMP [6,55] and AINNCS ARCH-COMP [38,50], for a comparison on state-of-the-art methods for neural network verification and neural network control system verification, respectively.

3 Overview and Features

NNV is an object-oriented toolbox developed in MATLAB [53] and built on top of several open-source software, including CORA [3] for reachability analysis of nonlinear ordinary differential equations (ODE) [73] and hybrid automata, MPT toolbox [45] for polytope-based operations [76], YALMIP [49] for some optimization problems in addition to MATLAB's Optimization Toolbox [53] and GLPK [56], and MatConvNet [82] for some convolution and pooling operations. NNV also makes use of MATLAB's deep learning toolbox to load the Open Neural Network Exchange (ONNX) format [57,68], and the Hybrid Systems Model Transformation and Translation tool (HyST) [5] for NNCS plant configuration.

NNV consists of two main modules: a *computation engine* and an *analyzer*, as illustrated in Fig. 1. The computation engine module consists of four components: 1) *NN constructor*, 2) *NNCS constructor*, 3) *reachability solvers*, and 4) *evaluator*. The NN constructor takes as an input a neural network, either as a DAGNetwork, dlnetwork, SeriesNetwork (MATLAB built-in formats) [69], or as an ONNX file [57], and generates a NN object suitable for verification. The NNCS constructor takes as inputs the NN object and an ODE or Hybrid Automata (HA) file describing the dynamics of a system, and then creates an NNCS object. Depending on the task to solve, either the NN (or NNCS) object is passed into the reachability solver to compute the reachable set of the system from a given set of initial conditions. Then, the computed set is sent to the analyzer module to verify/falsify a given property, and/or visualize the reachable sets. Given a specification, the verifier can formally reason whether the specification is met by computing the intersection of the define property and the reachable sets. If an exact (sound and complete) method is used, (e.g., exact-star), the analyzer can determine if the property is satisfied or unsatisfied. If an over-approximate (sound and incomplete) method is used, the verifier may also return "*uncertain*" (unknown), in addition to satisfied or unsatisfied.

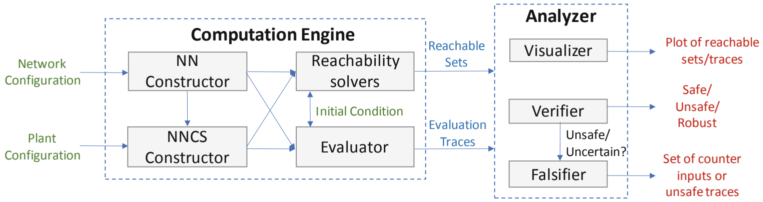


Fig. 1. An overview of NNV and its major modules and components.

3.1 NNV 2.0 vs NNV

Since the introduction of NNV [80], we have added to NNV support for the verification of a larger subset of deep learning models. We have added reachability methods to verify SSNNs [79], and a collection of relax-star reachability methods [79], reachability techniques for Neural ODEs [52] and RNNs [74]. In addition, there have been changes that include the creation of a common NN class that encapsulates previously supported neural network classes (FFNN and CNN) as well as Neural ODEs, SSNNs, and RNNs, which significantly reduces the software complexity and simplifies user experience. We have also added direct support for ONNX [57], as well as a parser for VNN-LIB [29], which describes properties to verify of any class of neural networks. We have also added flexibility to use one of the many solvers supported by YALMIP [49], GLPK [56] or linprog [70]. Table 1 shows a summary of the major features of NNV, highlighting the novel features.

Table 1. Overview of major features available in NNV. Links refer to relevant files/classes in the NNV codebase. BN refers to batch normalization layers, FC to fully-connected layers, AvgPool to average pooling layers, Conv to convolutional layers, and MaxPool to max pooling layers.

Feature	Supported (NNV 2.0 additions in blue)
Neural Network Type	FFNN, CNN, NeuralODE, SSNN, RNN
Layers	MaxPool, Conv, BN, AvgPool, FC, MaxUnpool, TC, DC, NODE
Activation functions	ReLU, Satlin, Sigmoid, Tanh, Leaky ReLU, Satlins
Plant dynamics (NNCS)	Linear ODE, Nonlinear ODE, HA, Continuous & Discrete Time
Set Representation	Polyhedron, Zonotope, Star, ImageStar
Star Reach methods	exact, approx, abs-dom, relax-range, relax-area, relax-random, relax-bound
Reachable set visualization	Yes, exact and over-approximation
Verification	Safety, Robustness, VNNLIB
Miscellaneous	Parallel computing, counterexample generation, ONNX*

*ONNX was partially supported for feedforward neural networks through NNVMT. Support has been extended to other NN types without the need for external libraries.

Semantic Segmentation [79]**.** Semantic segmentation consists on classifying image pixels into one or more classes which are semantically interpretable, like the different objects in an image. This task is common in areas like perception for autonomous vehicles, and medical imaging [71], which is typically accomplished by neural networks, referred to as semantic segmentation neural networks (SSNNs). These are characterized by two major portions, the encoder, or sequence of down-sampling layers to extract important features in the input, and the decoder, or sequence of up-sampling layers, to scale back the data information and classify each pixel into its corresponding class. Thus, the verification of these models is rather challenging, due to the complexity of the layers, and the output space dimensionality. We implement in NNV the collection of reachability methods introduced by Tran et al. [79], that are able to verify the robustness of a SSNNs. This means that we can formally guarantee the robustness value for each pixel, and determine the percentage of pixels that are correctly classified despite the adversarial attack. This was demonstrated using several architectures on two datasets: MNIST and M2NIST [46]. To achieve this, additional support for transposed and dilated convolutional layers was added [79].

Neural Ordinary Differential Equations [52]**.** Continuous deep learning models, referred to as Neural ODEs, have received a growing consideration over the last few years [11]. One of the main reasons for their popularity is due to their memory efficiency and their ability to learn from irregularly sampled data [61]. Similarly to SSNNs, despite their recent popularity, there is very limited work on the formal verification of these models [52]. For this reason, we implemented in NNV the first deterministic verification approach for a general class of neural ODEs (GNODE), which supports GNODEs to be constructed with multiple continuous layers (neural ODEs), linear or nonlinear, as well as any discrete-time layer already supported in NNV, such as ReLU, fully-connected or convolutional layers [52]. NNV demonstrates its capabilities in a series of time-series, control systems and image classification benchmarks, where it significantly outperforms any of the compared tools in the number of benchmarks and architectures supported [52].

Recurrent Neural Networks [74]**.** We implement star-based verification methods for RNNs introduced in [74]. These are able to verify RNNs without unrolling, reducing accumulated over-approximation error by optimized relaxation in the case of approximate reachability. The star set is an efficient technique in the computation of RNN reachable sets due to its advantages in computing affine mapping, the intersection of half-spaces, and Minkowski summation [74]. A new star set representing the reachable set of the current hidden state can be directly and efficiently constructed based on the reachable sets of the previous hidden state and the current input set. As proposed in verifying FFNNs [7,77,78], CNNs [72], and SSNNs [79], tight and efficient over-approximation reachability can be applied to the verification of ReLU RNNs. The triangular over-approximation of ReLU enables a tight over-approximation of the exact reachable set, preventing exponentially increasing the number of star sets during splitting. Estimation of the state bound required for over-approximation can compute state bounds without solving LPs. Furthermore, the relaxed approximate reachability estimates the triangle over-approximation areas to optimize the ranges of state by solving LP optimization. Consequently, the extended exact reachability method is 10× faster, and the over-approximation method is 100× to 5000× faster than existing state-of-the-art methods [74].

Zonotope Pre-filtering Star Set Reachability [78]**.** The star-based reachability methods are improved by using the zonotope pre-filtering approach [7,78]. This improvement consists on equipping the star set with an outer-zonotope, on the reachability analysis of a ReLU layer, to estimate quickly the lower and upper bounds of the star set at each specific neuron to establish if splitting may occur at this neuron without the need to solve any LP problems. The reduction of LP optimizations to solve is critical for the scalability of star-set reachability methods [77]. For the exact analysis, we are able to avoid the use of the zonotope pre-filtering, since we can efficiently construct the new output set with one star, if the zero point is not within the set range, or the union of 2 stars, if the zero point is contained [78]. In the over-approximation star, the range information is required to construct the output set at a specific neuron if and only if the range contains the zero point.

Relax-Star Reachability [79]**.** To tackle some of the scalability problems that may arise when computing the reachable set of complex neural networks such as SSNNs, a collection of four relaxed reachability methods were introduced [79]. The main goal of these methods is to reduce the number of Linear Programming (LP) problems to solve by quickly estimating the bounds or the reachable set, and only solving a fraction of the LP problems, while over-approximating the others. The LPs to solve are determined by the heuristics chosen, which can be random, area-based, bound-based, or range-based. The number of LPs is also determined by the user, who can choose from 0% to 100%. The closer to 100%, the larger number of LPs are skipped and over-approximated, thus the reachable set tends to be a larger over-approximation of the output, which significantly reduces the computation time [79].

Other Updates. In addition to the previous features described, there is a set of changes and additions included in the latest NNV version:

- *Activation Functions.* The star set method is extended to other classes of piecewise activation functions such as saturating linear layer (satlin), saturating linear symmetric layer (satlins), and leaky ReLU. The reachability analysis of each of these functions can be performed similarly to ReLU layers using the zonotope pre-filtering method to find where splits happen.

- *LP solver.* We generalize the use of LP solvers across all methods and optimizations. We allow the user to select the solver to use, which can choose between GLPK [56], linprog [70] (MATLAB's Optimization Toolbox) or any of the solvers supported by YALMIP [49]. We select *linprog* as the default solver, while keeping GLPK as a backup. However, if a different solver is selected that is supported by YALMIP, our implementation of the LP solver abstraction also supports this selection for any reachability method.

- *Standard Input Formats.* In the past few years, the verification community has been working to standardize formats across all tools to facilitate comparison among them. We have improved NNV by replacing the NNVMT tool [81] with a module to load ONNX [57] networks directly from MATLAB, as well as adding support for VNNLIB [29] files to define NN properties.

4 Evaluation

The evaluation is divided into 4 sections: 1) Comparison of FFNN and CNN to MATLAB's commercial toolbox [53,69], 2) Reachability analysis of Neural ODEs [52], 3) Robustness Verification of RNNs [74], and 4) Robustness Verification of SSNNs [79]. The results presented were all performed on a desktop with the following configuration: AMD Ryzen 9 5900X @3.7GHz 12-Core Processor, 64 GB Memory, and 64-bit Microsoft Windows 10 Pro.

4.1 Comparison to MATLAB's Deep Learning Verification Toolbox

In this comparison, we make use of a subset of the benchmarks and properties evaluated in last year's Verification of Neural Network (VNN) [55] competition, in which we demonstrate the capabilities of NNV with respect to the latest commercial product from MATLAB for the verification of neural networks [69].

We compared them on a subset of benchmarks from VNN-COMP'22 [55]: *ACAS Xu*, *Tllverify*, *Oval21* (CIFAR10 [43]), and *RL* benchmarks, which consists on verifying 90 out of 145 properties of the ACAS Xu, where we compare

Table 2. Verification of ACAS Xu properties 3 and 4.

		matlab	approx	relax 25%	relax 50%	relax 75%	relax 100%	exact (8)
prop 3 (45)	*SAT*	3	3	3	2	0	0	3
	UNSAT	10	29	8	2	1	0	42
	time (s)	0.1383	0.6368	0.6192	0.5714	0.3843	0.0276	521.9
prop 4 (45)	*SAT*	1	3	3	2	0	0	3
	UNSAT	2	32	6	1	1	0	42
	time (s)	0.1387	0.6492	0.6420	0.5682	0.3568	0.0261	89.85

Table 3. Verification results of the RL, tllverify and oval21 benchmarks. We selected 50 random specifications from the RL benchmarks, 10 from tllverify and all 30 from oval21. - means that the benchmark is not supported.

	RL (50)			Tllverify (10)			Oval21 (30)		
	SAT	*UNSAT*	*time (s)*	*SAT*	*UNSAT*	*time (s)*	*SAT*	*UNSAT*	*time (s)*
matlab	20	11	0.0504	0	0	0.1947	–	–	–
NNV	32	14	0.0822	0	0	13.57	0	11	136.5

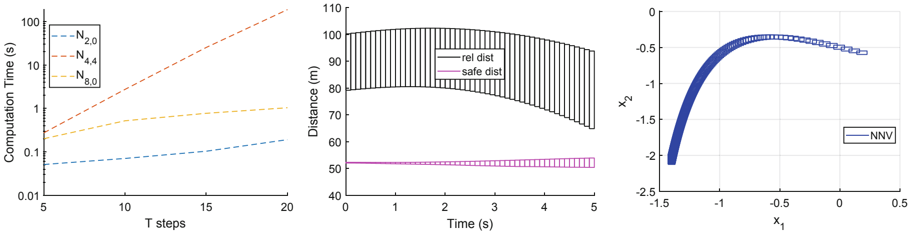
MATLAB's methods, approx-star, exact (parallel, 8 cores) and 4 relax-star methods. From the other 3 benchmarks, we select a total of 90 properties to verify, from which we limit the comparison to the approx-star and MATLAB's method. In this section, we demonstrate NNV is able to verify fully-connected layers, ReLU layers, flatten layers, and convolutional layers. The results of this comparison are described in Table 2. We can observe that MATLAB's computation time is faster than NNV star methods, except for the relax star with 100% relaxation. However, NNV's exact and approx methods significantly outperform MATLAB's framework by verifying 100% and 74% of the properties respectively, compared to 18% from MATLAB's. The remainder of the comparison is described in Table 3, which shows a similar trend: MATLAB's computation is faster, while NNV is able to verify a larger fraction of the properties.

4.2 Neural Ordinary Differential Equations

We exhibit the reachability analysis of GNODEs with three tasks: dynamical system modeling of a Fixed Point Attractor (FPA) [52,54], image classification of MNIST [46], and an adaptive cruise control (ACC) system [73].

Dynamical Systems. For the FPA, we compute the reachable set for a time horizon of 10 s, given a perturbation of ± 0.01 on all 5 input dimensions. The results of this example are illustrated in Fig. 2c, with a computation time of 3.01 s. The FPA model consists of one nonlinear neural ODE, no discrete-time layers are part of this model [52].

Classification. For the MNIST benchmark, we evaluate the robustness of two GNODEs with convolutional, fully-connected, ReLU and neural ODE layers, corresponding to CNODE$_S$ and CNODE$_M$ models introduced in [52]. We verify the robustness of 5 random images under an L$_\infty$ attack with a perturbation



(a) RNN Computation Times

(b) Neural ODE, NNCS Nonlinear ACC

(c) Neural ODE FPA

Fig. 2. Verification of RNN and neural ODE results. Figure 2a shows the verification time of the 3 RNNs evaluated. Figure 2b depicts the safety verification of the ACC, and Fig. 2c shows the reachability results of the FPA benchmark.

value of ± 0.5 on all the pixels. We are able to prove the robustness of both models on 100% of images, with an average computation time of 16.3 s for the CNODE_S, and 119.9 s for the CNODE_M.

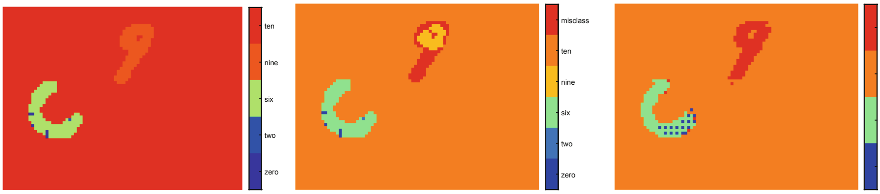
Control Systems. We verify an NNCS of an adaptive cruise control (ACC) system, where the controller is a FFNN with 5 ReLU layers with 20 neurons each, and one output linear layer, and the plant is a nonlinear neural ODE [52]. The verification results are illustrated in Fig. 2b, showing the current distance between the ego and lead cars and the safety distance allowed. We can observe that there is no intersection between the two, guaranteeing its safety.

4.3 Recurrent Neural Networks

For the RNN evaluation, we evaluate of three RNNs trained on the speaker recognition VCTK dataset [88]. Each network has an input layer of 40 neurons, two hidden layers with 2,4, or 8 memory units, followed by 5 ReLU layers with 32 neurons, and an output layer of 20 neurons. For each of the networks, we use the same 5 input points (40-dimensional time-independent vectors) for comparison. The robustness verification consists on proving that the output label after T $\in \{5, 10, 15, 20\}$ steps in the sequence is still the same, given an adversarial attack perturbation of $\epsilon = \pm\ 0.01$. We compute the reachable sets of all reachability instances using the approx-star method, which was able to prove the robustness of *19* out of 20 on $\text{N}_{2,0}$, and $\text{N}_{4,4}$ networks, and 18 for the $\text{N}_{8,0}$ network. We show the average reachability time per T value in Fig. 2a.

4.4 Semantic Segmentation

We demonstrate the robustness verification of two SSNNs, one with dilated convolutional layers and the other one with transposed convolutional layers, in addition to average pooling, convolutional and ReLU layers, which correspond to N_4 and N_5 introduced in Table 1 by Tran et al. [79]. We evaluate them on one random image of M2NIST [18] by attacking each image using an UBAA brightening attack [79]. One of the main differences of this evaluation with respect to the robustness analysis of other classification is the evaluation metrics used. For these networks, we evaluate the average robustness values (percentage of pixels correctly classified), sensitivity (number of not robust pixels over number of attacked pixels), and IoU (intersection over union) of the SSNNs. The computation time for the dilated example, shown in Fig. 3, is 54.52 s, with a robustness value of 97.2%, a sensitivity of 3.04, and a IoU of 57.8%. For the equivalent example with the transposed network, the robustness value is 98.14%, sensitivity of 2, IoU of 72.8%, and a computation time of 7.15 s.



(a) Target Image (b) Transposed SSNN (c) Dilated SSNN

Fig. 3. Robustness verification of the dilated and transposed SSNN under a UBAA brightening attack to 150 random pixels in the input image.

5 Conclusions

We presented version 2.0 of NNV, the updated version of the Neural Network Verification (NNV) tool [80], a software tool for the verification of deep learning models and learning-enabled CPS. To the best of our knowledge, NNV is the most comprehensive verification tool in terms of the number of tasks and neural networks architectures supported, including the verification of feedforward, convolutional, semantic segmentation, and recurrent neural networks, neural ODEs and NNCS. With the recent additions to NNV, we have demonstrated that NNV can be a one-stop verification tool for users with a diverse problem set, where verification of multiple neural network types is needed. In addition, NNV supports zonotope, polyhedron based methods, and up to 6 different star-based reachability methods to handle verification tradeoffs for the verification problem of neural networks, ranging from the exact-star, which is sound and complete, but computationally expensive, to the relax-star methods, which are significantly faster but more conservative. We have also shown that NNV outperforms a commercially available product from MATLAB, which computes the reachable sets of feedforward neural networks using the zonotope reachability method presented in [66]. In the future, we plan to ensure support for other deep learning models such as ResNets [30] and UNets [60].

Acknowledgments. The material presented in this paper is based upon work supported by the National Science Foundation (NSF) through grant numbers 1910017, 2028001, 2220418, 2220426 and 2220401, and the NSF Nebraska EPSCoR under grant OIA-2044049, the Defense Advanced Research Projects Agency (DARPA) under contract number FA8750-18-C-0089 and FA8750-23-C-0518, and the Air Force Office of Scientific Research (AFOSR) under contract number FA9550-22-1-0019 and FA9550-23-1-0135. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of AFOSR, DARPA, or NSF.

References

1. Akintunde, M.E., Botoeva, E., Kouvaros, P., Lomuscio, A.: Formal verification of neural agents in non-deterministic environments. In: Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), IFAAMAS 2020, . ACM, Auckland (2020)
2. Akintunde, M.E., Kevorchian, A., Lomuscio, A., Pirovano, E.: Verification of rnn-based neural agent-environment systems. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 6006–6013 (2019)
3. Althoff, M.: An introduction to cora 2015. In: Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
4. Bak, S.: nnenum: verification of ReLU neural networks with optimized abstraction refinement. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 19–36. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_2
5. Bak, S., Bogomolov, S., Johnson, T.T.: Hyst: a source transformation and translation tool for hybrid automaton models. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, pp. 128–133. ACM (2015)
6. Bak, S., Liu, C., Johnson, T.T.: The second international verification of neural networks competition (VNN-COMP 2021): Summary and results. CoRR abs/2109.00498 (2021)
7. Bak, S., Tran, H.-D., Hobbs, K., Johnson, T.T.: Improved geometric path enumeration for verifying ReLU neural networks. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 66–96. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_4
8. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: Juliareach: a toolbox for set-based reachability. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 39–44 (2019)
9. Bojarski, M., et al.: End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316 (2016)
10. Chen, C., Seff, A., Kornhauser, A., Xiao, J.: Deepdriving: learning affordance for direct perception in autonomous driving. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 2722–2730 (2015)
11. Chen, R.T.Q., Rubanova, Y., Bettencourt, J., Duvenaud, D.: Neural ordinary differential equations. Adv. Neural Inf. Process. Syst. (2018)
12. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18

13. Cireşan, D., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. arXiv preprint arXiv:1202.2745 (2012)
14. Cohen, J., Rosenfeld, E., Kolter, Z.: Certified adversarial robustness via randomized smoothing. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 97, pp. 1310–1320. PMLR (2019)
15. Collobert, R., Weston, J.: A unified architecture for natural language processing: Deep neural networks with multitask learning. In: Proceedings of the 25th International Conference on Machine Learning, pp. 160–167. ACM (2008)
16. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks, pp. 121–138 (2018)
17. Fan, J., Huang, C., Chen, X., Li, W., Zhu, Q.: ReachNN*: a tool for reachability analysis of neural-network controlled systems. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 537–542. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_30
18. (farhanhubble), F.A.: M2NIST, MNIST of semantic segmentation. https://www.kaggle.com/datasets/farhanhubble/multimnistm2nist
19. Ferlez, J., Khedr, H., Shoukry, Y.: Fast BATLLNN: fast box analysis of two-level lattice neural networks. In: Bartocci, E., Putot, S. (eds.) HSCC '22: 25th ACM International Conference on Hybrid Systems: Computation and Control, Milan, Italy, 4–6 May 2022. pp. 23:1–23:11. ACM (2022)
20. Ferrari, C., Müller, M.N., Jovanovic, N., Vechev, M.T.: Complete verification via multi-neuron relaxation guided branch-and-bound. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, 25–29 April 2022. OpenReview.net (2022)
21. Fischer, M., Baader, M., Vechev, M.: Scalable certified segmentation via randomized smoothing. In: Meila, M., Zhang, T. (eds.) Proceedings of the 38th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 139, pp. 3340–3351. PMLR (2021)
22. Fischer, M., Sprecher, C., Dimitrov, D.I., Singh, G., Vechev, M.: Shared certificates for neural network verification. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification, pp. 127–148. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_7
23. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
24. Gatys, L.A., Ecker, A.S., Bethge, M.: Image style transfer using convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2414–2423 (2016)
25. Goldberg, Y.: A primer on neural network models for natural language processing. J. Artif. Intell. Res. **57**, 345–420 (2016)
26. Goubault, E., Putot, S.: Rino: Robust inner and outer approximated reachability of neural networks controlled systems. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification, pp. 511–523. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_25
27. Gruenbacher, S., Hasani, R.M., Lechner, M., Cyranka, J., Smolka, S.A., Grosu, R.: On the verification of neural odes with stochastic guarantees. In: AAAI (2021)
28. Gruenbacher, S., et al.: Gotube: scalable stochastic verification of continuous-depth models (2021)
29. Guidotti, D., Demarchi, S., Tacchella, A., Pulina, L.: The Verification of Neural Networks Library (VNN-LIB) (2022). https://www.vnnlib.org

30. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
31. Henriksen, P., Hammernik, K., Rueckert, D., Lomuscio, A.: Bias field robustness verification of large neural image classifiers. In: Proceedings of the 32nd British Machine Vision Conference (BMVC21). BMVA Press (2021)
32. Huang, C., Fan, J., Chen, X., Li, W., Zhu, Q.: Polar: a polynomial arithmetic framework for verifying neural-network controlled systems (2021). https://doi.org/10.48550/ARXIV.2106.13867
33. Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q.: Reachnn: reachability analysis of neural-network controlled systems. arXiv preprint arXiv:1906.10654 (2019)
34. Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G., Lee, I.: Verisig 2.0: verification of neural network controllers using taylor model preconditioning. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 249–262. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_11
35. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Hybrid Systems: Computation and Control (HSCC) (2019)
36. Jacoby, Y., Barrett, C., Katz, G.: Verifying recurrent neural networks using invariant inference. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 57–74. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_3
37. Jain, L.C., Medsker, L.R.: Recurrent neural networks: design and applications (1999)
38. Johnson, T.T., et al.: Arch-comp21 category report: artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21). EPiC Series in Computing, vol. 80, pp. 90–119. EasyChair (2021). https://doi.org/10.29007/kfk9
39. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
40. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
41. Khedr, H., Ferlez, J., Shoukry, Y.: PEREGRiNN: penalized-relaxation greedy neural network verifier. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 287–300. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_13
42. Kochdumper, N., Schilling, C., Althoff, M., Bak, S.: Open- and closed-loop neural network verification using polynomial zonotopes (2022)
43. Krizhevsky, A.: Learning multiple layers of features from tiny images, pp. 32–33 (2009)
44. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
45. Kvasnica, M., Grieder, P., Baotić, M., Morari, M.: Multi-parametric toolbox (MPT). In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 448–462. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_30
46. LeCun, Y.: The mnist database of handwritten digits (1998). http://yann.lecun.com/exdb/mnist/

47. Lenz, I.: Deep learning for robotics. Ph.D. thesis, Cornell University (2016)
48. Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. Found. Trends Optim. **4**(3–4), 244–404 (2021). https://doi.org/10.1561/2400000035
49. Löfberg, J.: Yalmip : A toolbox for modeling and optimization in MATLAB. In: Proceedings of the CACSD Conference, Taipei, Taiwan (2004). http://users.isy.liu.se/johanl/yalmip
50. Lopez, D.M., et al.: Arch-comp22 category report: artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M., Schoitsch, E., Guiochet, J. (eds.) Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22). EPiC Series in Computing, vol. 90, pp. 142–184. EasyChair (2022)
51. Lopez, D.M., Johnson, T.T., Bak, S., Tran, H.D., Hobbs, K.: Evaluation of neural network verification methods for air to air collision avoidance. AIAA J. Air Transp. (JAT) (2022)
52. Manzanas Lopez, D., Musau, P., Hamilton, N., Johnson, T.: Reachability analysis of a general class of neural ordinary differential equation. In: Proceedings of the 20th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2022), Co-Located with CONCUR, FMICS, and QEST as part of CONFEST 2022, Warsaw, Poland (2022)
53. MATLAB: Update 3, (R2022b). The MathWorks Inc., Natick, Massachusetts (2022)
54. Musau, P., Johnson, T.T.: Continuous-time recurrent neural networks (ctrnns) (benchmark proposal). In: 5th Applied Verification for Continuous and Hybrid Systems Workshop (ARCH), Oxford, UK (2018). https://doi.org/10.29007/6czp
55. Müller, M.N., Brix, C., Bak, S., Liu, C., Johnson, T.T.: The third international verification of neural networks competition (vnn-comp 2022): Summary and results (2022)
56. Oki, E.: Glpk (gnu linear programming kit) (2012)
57. (ONNX), O.N.N.E.: https://github.com/onnx/
58. O'Shea, K., Nash, R.: An introduction to convolutional neural networks. CoRR abs/1511.08458 (2015). http://dblp.uni-trier.de/db/journals/corr/corr1511.html#OSheaN15
59. Prabhakar, P., Rahimi Afzal, Z.: Abstraction based output range analysis for neural networks. Adv. Neural Inf. Process. Syst. **32** (2019)
60. Ronneberger, O., Fischer, P., Brox, T.: U-Net: convolutional networks for biomedical image segmentation. In: Navab, N., Hornegger, J., Wells, W.M., Frangi, A.F. (eds.) MICCAI 2015. LNCS, vol. 9351, pp. 234–241. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24574-4_28
61. Rubanova, Y., Chen, R.T.Q., Duvenaud, D.K.: Latent ordinary differential equations for irregularly-sampled time series. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc. (2019)
62. Ryou, W., Chen, J., Balunovic, M., Singh, G., Dan, A., Vechev, M.: Scalable polyhedral verification of recurrent neural networks. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 225–248. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_10
63. Schilling, C., Forets, M., Guadalupe, S.: Verification of neural-network control systems by integrating Taylor models and zonotopes. In: AAAI, pp. 8169–8177. AAAI Press (2022). https://doi.org/10.1609/aaai.v36i7.20790

64. Shriver, D., Elbaum, S., Dwyer, M.B.: DNNV: a framework for deep neural network verification. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 137–150. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_6
65. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. In: Advances in Neural Information Processing Systems, pp. 10825–10836 (2018)
66. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 41 (2019)
67. Szegedy, C., et al.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
68. The MathWorks, I.: Deep Learning Toolbox Converter for ONNX Model Format. Natick, Massachusetts, United State (2022). https://www.mathworks.com/matlabcentral/fileexchange/67296-deep-learning-toolbox-converter-for-onnx-model-format
69. The MathWorks, I.: Deep Learning Toolbox Verification Library. Natick, Massachusetts, United State (2022). https://www.mathworks.com/matlabcentral/fileexchange/118735-deep-learning-toolbox-verification-library
70. The MathWorks, I.: Optimization Toolbox. Natick, Massachusetts, United State (2022). https://www.mathworks.com/products/optimization.html
71. Thoma, M.: A survey of semantic segmentation (2016)
72. Tran, H.-D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 18–42. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_2
73. Tran, H.D., Cei, F., Lopez, D.M., Johnson, T.T., Koutsoukos, X.: Safety verification of cyber-physical systems with reinforcement learning control. In: ACM SIGBED International Conference on Embedded Software (EMSOFT 2019). ACM (2019)
74. Tran, H.D., Choi, S., Yamaguchi, T., Hoxha, B., Prokhorov, D.: Verification of recurrent neural networks using star reachability. In: The 26th ACM International Conference on Hybrid Systems: Computation and Control (HSCC) (2023)
75. Tran, H.D., et al.: Parallelizable reachability analysis algorithms for feed-forward neural networks. In: Proceedings of the 7th International Workshop on Formal Methods in Software Engineering (FormaliSE 2019), pp. 31–40. IEEE Press, Piscataway (2019). https://doi.org/10.1109/FormaliSE.2019.00012
76. Tran, H.D., et al.: Parallelizable reachability analysis algorithms for feed-forward neural networks. In: 7th International Conference on Formal Methods in Software Engineering (FormaliSE2019), Montreal, Canada (2019)
77. Tran, H.-D., et al.: Star-based reachability analysis of deep neural networks. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 670–686. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_39
78. Tran, H.D., et al.: Verification of piecewise deep neural networks: a star set approach with zonotope pre-filter. Formal Asp. Comput. **33**(4), 519–545 (2021)
79. Tran, H.-D., et al.: Robustness verification of semantic segmentation neural networks using relaxed reachability. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 263–286. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_12
80. Tran, H.D., et al.: NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: 32nd International Conference on Computer-Aided Verification (CAV) (2020)

81. Transformation, N.N.V.M.: https://github.com/verivital/nnvmt
82. Vedaldi, A., Lenc, K.: Matconvnet: convolutional neural networks for matlab. In: Proceedings of the 23rd ACM International Conference on Multimedia, pp. 689–692. ACM (2015)
83. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th {USENIX} Security Symposium ({USENIX} Security 2018), pp. 1599–1614 (2018)
84. Wang, S., et al.: Beta-CROWN: efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. Adv. Neural Inf. Process. Syst. **34** (2021)
85. Wu, B., Iandola, F.N., Jin, P.H., Keutzer, K.: Squeezedet: unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In: CVPR Workshops, pp. 446–454 (2017)
86. Xiang, W., Tran, H.D., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5777–5783 (2018)
87. Xu, K., et al.: Fast and Complete: enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In: International Conference on Learning Representations (2021). https://openreview.net/forum?id=nVZtXBI6LNn
88. Yamagishi, J., Veaux, C., MacDonald, K.: Cstr vctk corpus: English multi-speaker corpus for cstr voice cloning toolkit (version 0.92). In: University of Edinburgh. The Centre for Speech Technology Research (CSTR) (2019). https://doi.org/10.7488/ds/2645
89. Zeqiri, M., Mueller, M.N., Fischer, M., Vechev, M.: Efficient robustness verification of neural ordinary differential equations. In: The Symbiosis of Deep Learning and Differential Equations II (2022)
90. Zhang, H., Shinn, M., Gupta, A., Gurfinkel, A., Le, N., Narodytska, N.: Verification of recurrent neural networks for cognitive tasks via reachability analysis. In: ECAI 2020, pp. 1690–1697. IOS Press (2020)

QEBVerif: Quantization Error Bound Verification of Neural Networks

Yedi Zhang[1], Fu Song[1,2,3](✉), and Jun Sun[4]

[1] ShanghaiTech University, Shanghai 201210, China
songfu@shanghaitech.edu.cn
[2] Institute of Software, Chinese Academy of Sciences and University of Chinese Academy of Sciences, Beijing 100190, China
[3] Automotive Software Innovation Center, Chongqing 400000, China
[4] Singapore Management University, Singapore 178902, Singapore

Abstract. To alleviate the practical constraints for deploying deep neural networks (DNNs) on edge devices, quantization is widely regarded as one promising technique. It reduces the resource requirements for computational power and storage space by quantizing the weights and/or activation tensors of a DNN into lower bit-width fixed-point numbers, resulting in quantized neural networks (QNNs). While it has been empirically shown to introduce minor accuracy loss, critical verified properties of a DNN might become invalid once quantized. Existing verification methods focus on either individual neural networks (DNNs or QNNs) or quantization error bound for *partial* quantization. In this work, we propose a quantization error bound verification method, named QEBVerif, where both weights and activation tensors are quantized. QEBVerif consists of two parts, i.e., a differential reachability analysis (DRA) and a mixed-integer linear programming (MILP) based verification method. DRA performs difference analysis between the DNN and its quantized counterpart layer-by-layer to compute a tight quantization error interval efficiently. If DRA fails to prove the error bound, then we encode the verification problem into an equivalent MILP problem which can be solved by off-the-shelf solvers. Thus, QEBVerif is sound, complete, and reasonably efficient. We implement QEBVerif and conduct extensive experiments, showing its effectiveness and efficiency.

1 Introduction

In the past few years, the development of deep neural networks (DNNs) has grown at an impressive pace owing to their outstanding performance in solving various complicated tasks [23,28]. However, modern DNNs are often large in size and contain a great number of 32-bit floating-point parameters to achieve competitive performance. Thus, they often result in high computational costs and excessive storage requirements, hindering their deployment on resource-constrained embedded devices, e.g., edge devices. A promising solution is to quantize the weights and/or activation tensors as fixed-point numbers of lower

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 413–437, 2023.
https://doi.org/10.1007/978-3-031-37703-7_20

bit-width [17,21,25,35]. For example, TensorFlow Lite [18] supports quantization of weights and/or activation tensors to reduce the model size and latency, and Tesla FSD-chip [61] stores all the data and weights of a network in the form of 8-bit integers.

In spite of the empirically impressive results which show there is only minor accuracy loss, quantization does not necessarily preserve properties such as robustness [16]. Even worse, input perturbation can be amplified by quantization [11,36], worsening the robustness of quantized neural networks (QNNs) compared to their DNN counterparts. Indeed, existing neural network quantization methods focus on minimizing its impact on model accuracy (e.g., by formulating it as an optimization problem that aims to maximize the accuracy [27,43]). However, they cannot guarantee that the final quantization error is always lower than a given error bound, especially when some specific safety-critical input regions are concerned. This is concerning as such errors may lead to catastrophes when the quantized networks are deployed in safety-critical applications [14,26]. Furthermore, analyzing (in particular, quantifying) such errors can also help us understand how quantization affect the network behaviors [33], and provide insights on, for instance, how to choose appropriate quantization bit sizes without introducing too much error. Therefore, a method that soundly quantifies the errors between DNNs and their quantized counterparts is highly desirable.

There is a large and growing body of work on developing verification methods for DNNs [2,12,13,15,19,24,29,30,32,37,38,51,54,55,58–60,62] and QNNs [1,3,16,22,46,66,68], aiming to establish a formal guarantee on the network behaviors. However, all the above-mentioned methods focus exclusively on verifying individual neural networks. Recently, Paulsen et al. [48,49] proposed differential verification methods, aimed to establish formal guarantees on the difference between two DNNs. Specifically, given two DNNs $\mathcal{N}_1$ and $\mathcal{N}_2$ with the same network topology and inputs, they try to prove that $|\mathcal{N}_1(\mathbf{x}) - \mathcal{N}_2(\mathbf{x})| < \epsilon$ for all possible inputs $\mathbf{x} \in \mathcal{X}$, where $\mathcal{X}$ is the interested input region. They presented fast and sound difference propagation techniques followed by a refinement of the input region until the property can be successfully verified, i.e., the property is either proved or falsified by providing a counterexample. This idea has been extended to handle recurrent neural networks (RNNs) [41] though the refinement is not considered therein. Although their methods [41,48,49] can be used to analyze the error bound introduced by quantizing weights (called *partially* QNNs), they are not complete and cannot handle the cases where both the weights and activation tensors of a DNN are quantized to lower bit-width fixed-point numbers (called *fully* QNNs). We remark that fully QNN can significantly reduce the energy-consumption (floating-point operations consume much more energy than integer-only operations) [61].

Main Contributions. We propose a sound and complete **Q**uantization **E**rror **B**ound **Verif**ication method (QEBVerif) to efficiently and effectively verify if the quantization error of a *fully* QNN w.r.t. an input region and its original DNN is always lower than an error bound (a.k.a. robust error bound [33]). QEBVerif first conducts a novel reachability analysis to quantify the quantization errors, which is referred to as *differential reachability analysis* (DRA). Such an analysis yields two results: (1) *Proved*, meaning that the quantization error is proved to be always less than the given error bound; or (2) *Unknown*, meaning that it fails to prove the error bound, possibly due to a conservative approximation of the quantization error. If the outcome is *Unknown*, we further encode this quantization error bound verification problem into an equivalent mixed-integer linear programming (MILP) problem, which can be solved by off-the-shelf solvers.

There are two main technical challenges that must be addressed for DRA. First, the activation tensors in a fully QNN are discrete values and contribute additional rounding errors to the final quantization errors, which are hard to propagate symbolically and make it difficult to establish relatively accurate difference intervals. Second, much more activation-patterns (i.e., $3 \times 6 = 18$) have to consider in a forward propagation, while 9 activation-patterns are sufficient in [48,49], where an activation-pattern indicates the status of the output range of a neuron. A neuron in a DNN under an input region has 3 patterns: always-active (i.e., output ≥ 0), always-inactive (i.e., output < 0), or both possible. A neuron in a QNN has 6 patterns due to the clamp function (cf. Definition 2). We remark that handling these different combinations efficiently and soundly is highly nontrivial. To tackle the above challenges, we propose sound transformations for the affine and activation functions to propagate quantization errors of two networks layer-by-layer. Moreover, for the affine transformation, we provide two alternative solutions: *interval-based* and *symbolic-based*. The former directly computes sound difference intervals via interval analysis [42], while the latter leverages abstract interpretation [10] to compute sound and symbolic difference intervals, using the polyhedra abstract domain. In comparison, the symbolic-based one is usually more accurate but less efficient than the interval-based one. Note that though existing tools can obtain quantization error intervals by independently computing the output intervals of two networks followed by interval subtractions, such an approach is often too conservative.

To resolve those problems that cannot be proved via our DRA, we resort to the sound and complete MILP-based verification method. Inspired by the MILP encoding of DNN and QNN verification [39,40,68], we propose a novel MILP encoding for verifying quantization error bounds. QEBVerif represents both the computations of the QNN and the DNN in mixed-integer linear constraints which are further simplified using their own output intervals. Moreover, we also encode the output difference intervals of hidden neurons from our DRA as mixed-integer linear constraints to boost the verification.

We implement our method as an end-to-end tool and use Gurobi [20] as our back-end MILP solver. We extensively evaluate it on a large set of verification tasks using neural networks for ACAS Xu [26] and MNIST [31], where the number of neurons varies from 310 to 4890, the number of bits for quantizing weights and activation tensors ranges from 4 to 10 bits, and the number of bits for quantizing inputs is fixed to 8 bits. For DRA, we compare QEBVerif with a naive method that first independently computes the output intervals of DNNs and QNNs using the existing state-of-the-art (symbolic) interval analysis [22,55], and then conducts an interval subtraction. The experimental results show that both our interval- and symbolic-based approaches are much more accurate and can successfully verify much more tasks without the MILP-based verification. We also find that the quantization error interval returned by DRA is getting tighter with the increase of the quantization bit size. The experimental results also confirm the effectiveness of our MILP-based verification method, which can help verify many tasks that cannot be solved by DRA solely. Finally, our results also allow us to study the potential correlation of quantization errors and robustness for QNNs using QEBVerif.

We summarize our contributions as follows:

- We introduce the first sound, complete, and reasonably efficient quantization error bound verification method QEBVerif for fully QNNs by cleverly combining novel DRA and MILP-based verification methods;
- We propose a novel DRA to compute sound and tight quantization error intervals accompanied by an abstract domain tailored to QNNs, which can significantly and soundly tighten the quantization error intervals;
- We implement QEBVerif as an end-to-end open-source tool [64] and conduct an extensive evaluation on various verification tasks, demonstrating its effectiveness and efficiency.

The source code of our tool and benchmarks are available at https://github.com/S3L-official/QEBVerif. Missing proofs, more examples, and experimental results can be found in [65].

2 Preliminaries

We denote by $\mathbb{R}, \mathbb{Z}, \mathbb{N}$ and $\mathbb{B}$ the sets of real-valued numbers, integers, natural numbers, and Boolean values, respectively. Let $[n]$ denote the integer set $\{1, \ldots, n\}$ for given $n \in \mathbb{N}$. We use **BOLD UPPERCASE** (e.g., $\mathbf{W}$) and **bold lowercase** (e.g., $\mathbf{x}$) to denote matrices and vectors, respectively. We denote by $\mathbf{W}_{i,j}$ the j-entry in the i-th row of the matrix $\mathbf{W}$, and by $\mathbf{x}_i$ the i-th entry of the vector $\mathbf{x}$. Given a matrix $\mathbf{W}$ and a vector $\mathbf{x}$, we use $\widehat{\mathbf{W}}$ and $\hat{\mathbf{x}}$ (resp. $\widetilde{\mathbf{W}}$ and $\tilde{\mathbf{x}}$) to denote their quantized/integer (resp. fixed-point) counterparts.

2.1 Neural Networks

A deep neural network (DNN) consists of a sequence of layers, where the first layer is the *input layer*, the last layer is the *output layer* and the others are called *hidden layers*. Each layer contains one or more neurons. A DNN is *feed-forward* if all the neurons in each non-input layer only receive inputs from the neurons in the preceding layer.

Definition 1 (Feed-forward Deep Neural Network). *A feed-forward DNN $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^s$ with d layers can be seen as a composition of d functions such that $\mathcal{N} = l_d \circ l_{d-1} \circ \cdots \circ l_1$. Then, given an input $\mathbf{x} \in \mathbb{R}^n$, the output of the DNN $\mathbf{y} = \mathcal{N}(\mathbf{x})$ can be obtained by the following recursive computation:*

- *Input layer* $l_1 : \mathbb{R}^n \to \mathbb{R}^{n_1}$ *is the identity function, i.e.,* $\mathbf{x}^1 = l_1(\mathbf{x}) = \mathbf{x}$*;*
- *Hidden layer* $l_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$ *for* $2 \le i \le d-1$ *is the function such that* $\mathbf{x}^i = l_i(\mathbf{x}^{i-1}) = \phi(\mathbf{W}^i\mathbf{x}^{i-1} + \mathbf{b}^i)$*;*
- *Output layer* $l_d : \mathbb{R}^{n_{d-1}} \to \mathbb{R}^s$ *is the function such that* $\mathbf{y} = \mathbf{x}^d = l_d(\mathbf{x}^{d-1}) = \mathbf{W}^d\mathbf{x}^{d-1} + \mathbf{b}^d$*.*

where $n_1 = n$*,* $\mathbf{W}^i$ *and* $\mathbf{b}^i$ *are the weight matrix and bias vector in the i-th layer, and* $\phi(\cdot)$ *is the activation function which acts element-wise on an input vector.*

In this work, we focus on feed-forward DNNs with the most commonly used activation functions: the rectified linear unit (ReLU) function, defined as $\text{ReLU}(x) = \max(x, 0)$. We also use n_d to denote the output dimension s.

A quantized neural network (QNN) is structurally similar to its real-valued counterpart, except that all the parameters, inputs of the QNN, and outputs of all the hidden layers are quantized into integers according to the given quantization scheme. Then, the computation over real-valued arithmetic in a DNN can be replaced by the computation using integer arithmetic, or equally, fixed-point arithmetic. In this work, we consider the most common quantization scheme, i.e., symmetric uniform quantization [44]. We first give the concept of quantization configuration which effectively defines a quantization scheme.

A *quantization configuration* $\mathcal{C}$ is a tuple $\langle \tau, Q, F \rangle$, where Q and F are the total bit size and the fractional bit size allocated to a value, respectively, and $\tau \in \{+, \pm\}$ indicates if the quantized value is unsigned or signed. Given a real number $x \in \mathbb{R}$ and a quantization configuration $\mathcal{C} = \langle \tau, Q, F \rangle$, its quantized integer counterpart $\hat{x}$ and the fixed-point counterpart $\tilde{x}$ under the symmetric uniform quantization scheme are:

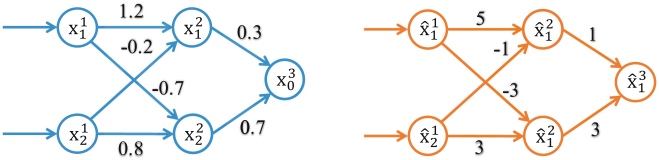
$$\hat{x} = \text{clamp}(\lfloor 2^F \cdot x \rceil,\ \mathcal{C}^{\text{lb}},\ \mathcal{C}^{\text{ub}}) \quad \text{and} \quad \tilde{x} = \hat{x}/2^F$$

where $\mathcal{C}^{\text{lb}} = 0$ and $\mathcal{C}^{\text{ub}} = 2^Q - 1$ if $\tau = +$, $\mathcal{C}^{\text{lb}} = -2^{Q-1}$ and $\mathcal{C}^{\text{ub}} = 2^{Q-1} - 1$ otherwise, and $\lfloor \cdot \rceil$ is the round-to-nearest integer operator. The *clamping function* $\text{clamp}(x, a, b)$ with a lower bound a and an upper bound b is defined as:

$$\text{clamp}(x, a, b) = \begin{cases} a, & \text{if } x < a; \\ x, & \text{if } a \le x \le b; \\ b, & \text{if } x > b. \end{cases}$$

Definition 2 (Quantized Neural Network). *Given quantization configurations for the weights, biases, output of the input layer and each hidden layer as* $\mathcal{C}_w = \langle \tau_w, Q_w, F_w \rangle$*,* $\mathcal{C}_b = \langle \tau_b, Q_b, F_b \rangle$*,* $\mathcal{C}_{in} = \langle \tau_{in}, Q_{in}, F_{in} \rangle$*,* $\mathcal{C}_h = \langle \tau_h, Q_h, F_h \rangle$*, the quantized version (i.e., QNN) of a DNN* $\mathcal{N}$ *with* d *layers is a function* $\widehat{\mathcal{N}} : \mathbb{Z}^n \to \mathbb{R}^s$ *such that* $\widehat{\mathcal{N}} = \hat{l}_d \circ \hat{l}_{d-1} \circ \cdots \circ \hat{l}_1$*. Then, given a quantized input* $\hat{\mathbf{x}} \in \mathbb{Z}^n$*, the output of the QNN* $\hat{\mathbf{y}} = \widehat{\mathcal{N}}(\hat{\mathbf{x}})$ *can be obtained by the following recursive computation:*

- *Input layer* $\hat{l}_1 : \mathbb{Z}^n \to \mathbb{Z}^{n_1}$ *is the identity function, i.e.,* $\hat{\mathbf{x}}^1 = \hat{l}_1(\hat{\mathbf{x}}) = \hat{\mathbf{x}}$*;*
- *Hidden layer* $\hat{l}_i : \mathbb{Z}^{n_{i-1}} \to \mathbb{Z}^{n_i}$ *for* $2 \le i \le d-1$ *is the function such that for each* $j \in [n_i]$*,*



(a) DNN $\mathcal{N}_e$.

(b) QNN $\widehat{\mathcal{N}}_e$.

Fig. 1. A 3-layer DNN $\mathcal{N}_e$ and its quantized version $\widehat{\mathcal{N}}_e$.

$$\hat{\mathbf{x}}_j^i = \text{clamp}(\lfloor 2^{F_i}\widehat{\mathbf{W}}_{j,:}^i \cdot \hat{\mathbf{x}}^{i-1} + 2^{F_h - F_b}\hat{\mathbf{b}}_j^i \rceil, 0,\ \mathcal{C}_h^{\text{ub}}),$$

where F_i is $F_h - F_w - F_{in}$ if $i = 2$, and $-F_w$ otherwise;

- *Output layer $\hat{l}_d : \mathbb{Z}^{n_{d-1}} \to \mathbb{R}^s$ is the function such that $\hat{\mathbf{y}} = \hat{\mathbf{x}}^d = \hat{l}_d(\hat{\mathbf{x}}^{d-1}) = 2^{-F_w}\widehat{\mathbf{W}}^d\hat{\mathbf{x}}^{d-1} + 2^{F_h - F_b}\hat{\mathbf{b}}^d$.*

where for every $2 \leq i \leq d$ and $k \in [n_{i-1}]$, $\widehat{\mathbf{W}}_{j,k}^i = \text{clamp}(\lfloor 2^{F_w}\mathbf{W}_{j,k}^i \rceil, \mathcal{C}_w^{\text{lb}}, \mathcal{C}_w^{\text{ub}})$ is the quantized weight and $\hat{\mathbf{b}}_j^i = \text{clamp}(\lfloor 2^{F_b}\mathbf{b}_j^i \rceil, \mathcal{C}_b^{\text{lb}}, \mathcal{C}_b^{\text{ub}})$ is the quantized bias.

We remark that 2^{F_i} and $2^{F_h - F_b}$ in Definition 2 are used to align the precision between the inputs and outputs of hidden layers, and F_i for $i = 2$ and $i > 2$ because quantization bit sizes for the outputs of the input layer and hidden layers can be different.

2.2 Quantization Error Bound and Its Verification Problem

We now give the formal definition of the quantization error bound verification problem considered in this work as follows.

Definition 3 (Quantization Error Bound). *Given a DNN $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^s$, the corresponding QNN $\widehat{\mathcal{N}} : \mathbb{Z}^n \to \mathbb{R}^s$, a quantized input $\hat{\mathbf{x}} \in \mathbb{Z}^n$, a radius $r \in \mathbb{N}$ and an error bound $\epsilon \in \mathbb{R}$. The QNN $\widehat{\mathcal{N}}$ has a quantization error bound of ϵ w.r.t. the input region $R(\hat{\mathbf{x}}, r) = \{\hat{\mathbf{x}}' \in \mathbb{Z}^n \mid ||\hat{\mathbf{x}}' - \hat{\mathbf{x}}||_\infty \leq r\}$ if for every $\hat{\mathbf{x}}' \in R(\hat{\mathbf{x}}, r)$, we have $||2^{-F_h}\widehat{\mathcal{N}}(\hat{\mathbf{x}}') - \mathcal{N}(\mathbf{x}')||_\infty < \epsilon$, where $\mathbf{x}' = \hat{\mathbf{x}}'/(\mathcal{C}_{in}^{\text{ub}} - \mathcal{C}_{in}^{\text{lb}})$.*

Intuitively, quantization-error-bound is the bound of the output difference of the DNN and its quantized counterpart for all the inputs in the input region. In this work, we obtain the input for DNN via dividing $\hat{\mathbf{x}}'$ by $(\mathcal{C}_{in}^{\text{ub}} - \mathcal{C}_{in}^{\text{lb}})$ to allow input normalization. Furthermore, 2^{-F_h} is used to align the precision between the outputs of QNN and DNN.

Example 1. Consider the DNN $\mathcal{N}_e$ with 3 layers (one input layer, one hidden layer, and one output layer) given in Fig. 1, where weights are associated with the edges and all the biases are 0. The quantization configurations for the weights, the output of the input layer and hidden layer are $\mathcal{C}_w = \langle \pm, 4, 2 \rangle$, $\mathcal{C}_{in} = \langle +, 4, 4 \rangle$ and $\mathcal{C}_h = \langle +, 4, 2 \rangle$. Its QNN $\widehat{\mathcal{N}}_e$ is shown in Fig. 1.

Given a quantized input $\hat{\mathbf{x}} = (9, 6)$ and a radius $r = 1$, the input region for QNN $\widehat{\mathcal{N}}_e$ is $R((9, 6), 1) = \{(x, y) \in \mathbb{Z}^2 \mid 8 \leq x \leq 10, 5 \leq y \leq 7\}$. Since $\mathcal{C}_{in}^{ub} = 15$ and $\mathcal{C}_{in}^{lb} = 0$, by Definitions 1, 2, and 3, we have the maximum quantization error as $\max(2^{-2}\widehat{\mathcal{N}}_e(\hat{\mathbf{x}}') - \mathcal{N}_e(\hat{\mathbf{x}}'/15)) = 0.067$ for $\hat{\mathbf{x}}' \in R((9, 6), 1)$. Then, $\widehat{\mathcal{N}}_e$ has a quantization error bound of ϵ w.r.t. input region $R((9, 6), 1)$ for any $\epsilon > 0.067$.

We remark that if only weights are quantized and the activation tensors are floating-point numbers, the maximal quantization error of $\widehat{\mathcal{N}}_e$ for the input region $R((9, 6), 1)$ is 0.04422, which implies that existing methods [48,49] cannot be used to analyze the error bound for a fully QNN.

In this work, we focus on the quantization error bound verification problem for classification tasks. Specifically, for a classification task, we only focus on the output difference of the predicted class instead of all the classes. Hence, given a DNN $\mathcal{N}$, a corresponding QNN $\widehat{\mathcal{N}}$, a quantized input $\hat{\mathbf{x}}$ which is classified to class g by the DNN $\mathcal{N}$, a radius r and an error bound ϵ, the quantization error bound property $P(\mathcal{N}, \widehat{\mathcal{N}}, \hat{\mathbf{x}}, r, \epsilon)$ for a classification task can be defined as follows:

$$\bigwedge_{\hat{\mathbf{x}}' \in R(\hat{\mathbf{x}}, r)} \left(|2^{-F_h}\widehat{\mathcal{N}}(\hat{\mathbf{x}}')_g - \mathcal{N}(\mathbf{x}')_g| < \epsilon \right) \wedge \left(\mathbf{x}' = \hat{\mathbf{x}}'/(\mathcal{C}_{in}^{\text{ub}} - \mathcal{C}_{in}^{\text{lb}}) \right)$$

Note that $\mathcal{N}(\cdot)_g$ denotes the g-th entry of the vector $\mathcal{N}(\cdot)$.

2.3 DeepPoly

We briefly recap DeepPoly [55], which will be leveraged in this work for computing the output of each neuron in a DNN.

The core idea of DeepPoly is to give each neuron an abstract domain in the form of a linear combination of the variables preceding the neuron. To achieve this, each hidden neuron $\mathbf{x}_j^i$ (the j-th neuron in the i-th layer) in a DNN is seen as two nodes $\mathbf{x}_{j,0}^i$ and $\mathbf{x}_{j,1}^i$, such that $\mathbf{x}_{j,0}^i = \sum_{k=1}^{n_{i-1}} \mathbf{W}_{j,k}^i \mathbf{x}_{k,1}^{i-1} + \mathbf{b}_j^i$ (affine function) and $\mathbf{x}_{j,1}^i = \text{ReLU}(\mathbf{x}_{j,0}^i)$ (ReLU function). Then, the affine function is characterized as an abstract transformer using an upper polyhedral computation and a lower polyhedral computation in terms of the variables $\mathbf{x}_{k,1}^{i-1}$. Finally, it recursively substitutes the variables in the upper and lower polyhedral computations with the corresponding upper/lower polyhedral computations of the variables until they only contain the input variables from which the concrete intervals are computed.

Formally, the abstract element $\mathcal{A}_{j,s}^i$ for the node $\mathbf{x}_{j,s}^i$ ($s \in \{0, 1\}$) is a tuple $\mathcal{A}_{j,s}^i = \langle \mathbf{a}_{j,s}^{i,\leq}, \mathbf{a}_{j,s}^{i,\geq}, l_{j,s}^i, u_{j,s}^i \rangle$, where $\mathbf{a}_{j,s}^{i,\leq}$ and $\mathbf{a}_{j,s}^{i,\geq}$ are respectively the lower and upper polyhedral computations in the form of a linear combination of the variables $\mathbf{x}_{k,1}^{i-1}$'s if $s = 0$ or $\mathbf{x}_{k,0}^i$'s if $s = 1$, $l_{j,s}^i \in \mathbb{R}$ and $u_{j,s}^i \in \mathbb{R}$ are the concrete lower and upper bound of the neuron. Then, the concretization of the abstract element $\mathcal{A}_{j,s}^i$ is $\Gamma(\mathcal{A}_{j,s}^i) = \{x \in \mathbb{R} \mid \mathbf{a}_{j,s}^{i,\leq} \leq x \wedge x \leq \mathbf{a}_{j,s}^{i,\geq}\}$.

Concretely, $\mathbf{a}_{j,0}^{i,\leq}$ and $\mathbf{a}_{j,0}^{i,\geq}$ are defined as $\mathbf{a}_{j,0}^{i,\leq} = \mathbf{a}_{j,0}^{i,\geq} = \sum_{k=1}^{n_{i-1}} \mathbf{W}_{j,k}^i \mathbf{x}_{k,1}^{i-1} + \mathbf{b}_j^i$. Furthermore, we can repeatedly substitute every variable in $\mathbf{a}_{j,0}^{i,\leq}$ (resp. $\mathbf{a}_{j,0}^{i,\geq}$) with its lower (resp. upper) polyhedral computation according to the coefficients until no further substitution is possible. Then, we can get a sound lower (resp. upper) bound in the form of a linear combination of the input variables based on which $l^i_{j,0}$ (resp. $u^i_{j,0}$) can be computed immediately from the given input region.

For ReLU function $\mathbf{x}^i_{j,1} = \text{ReLU}(\mathbf{x}^i_{j,0})$, there are three cases to consider of the abstract element $\mathcal{A}^i_{j,1}$:

- If $u^i_{j,0} \leq 0$, then $\mathbf{a}^{i,\leq}_{j,1} = \mathbf{a}^{i,\geq}_{j,1} = 0$, $l^i_{j,1} = u^i_{j,1} = 0$;
- If $l^i_{j,0} \geq 0$, then $\mathbf{a}^{i,\leq}_{j,1} = \mathbf{a}^{i,\leq}_{j,0}$, $\mathbf{a}^{i,\geq}_{j,1} = \mathbf{a}^{i,\geq}_{j,0}$, $l^i_{j,1} = l^i_{j,0}$ and $u^i_{j,1} = u^i_{j,0}$;
- If $l^i_{j,0} < 0 \wedge u^i_{j,0} > 0$, then $\mathbf{a}^{i,\geq}_{j,1} = \frac{u^i_{j,0}(\mathbf{x}^i_{j,0} - l^i_{j,0})}{u^i_{j,0} - l^i_{j,0}}$, $\mathbf{a}^{i,\leq}_{j,1} = \lambda \mathbf{x}^i_{j,0}$ where $\lambda \in \{0, 1\}$ such that the area of resulting shape by $\mathbf{a}^{i,\leq}_{j,1}$ and $\mathbf{a}^{i,\geq}_{j,1}$ is minimal, $l^i_{j,1} = \lambda l^i_{j,0}$ and $u^i_{j,1} = u^i_{j,0}$.

Note that DeepPoly also introduces transformers for other functions, such as sigmoid, tanh, and maxpool functions. In this work, we only consider DNNs with only ReLU as non-linear operators.

3 Methodology of QEBVerif

In this section, we first give an overview of our quantization error bound verification method, QEBVerif, and then give the detailed design of each component.

3.1 Overview of QEBVerif

An overview of QEBVerif is shown in Fig. 2. Given a DNN $\mathcal{N}$, its QNN $\widehat{\mathcal{N}}$, a quantization error bound ϵ and an input region consisting of a quantized input $\hat{\mathbf{x}}$ and a radius r, to verify the quantization error bound property $P(\mathcal{N}, \widehat{\mathcal{N}}, \hat{\mathbf{x}}, r, \epsilon)$, QEBVerif first performs a differential reachability analysis (DRA) to compute a sound output difference interval for the two networks. Note that, the difference intervals of all the neurons are also recorded for later use. If the output difference interval of the two networks is contained in $[-\epsilon, \epsilon]$, then the property is proved and QEBVerif outputs "Proved". Otherwise, QEBVerif leverages our MILP-based quantization error bound verification method by encoding the problem into an equivalent mixed integer linear programming (MILP) problem which can be solved by off-the-shelf solvers. To reduce the size of mixed integer linear constraints and boost the verification, QEBVerif independently applies symbolic interval analysis on the two networks based on which some activation patterns could be omitted. We further encode the difference intervals of all the neurons from DRA as mixed integer linear constraints and add them to the MILP problem. Though it increases the number of mixed integer linear constraints, it is very helpful for solving hard verification tasks. Therefore, the whole verification process is sound, complete yet reasonably efficient. We remark that the MILP-based verification method is often more time-consuming and thus the first step allows us to quickly verify many tasks first.

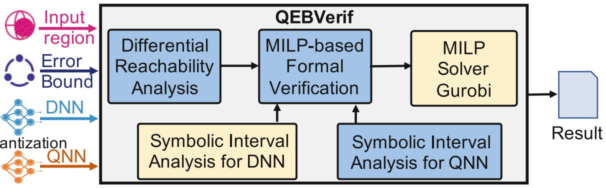

Input region
Error Bound
DNN
Quantization
QNN
QEBVerif
Differential Reachability Analysis
MILP-based Formal Verification
MILP Solver Gurobi
Symbolic Interval Analysis for DNN
Symbolic Interval Analysis for QNN
Result

Fig. 2. An overview of QEBVerif.

3.2 Differential Reachability Analysis

Naively, one could use an existing verification tool in the literature to independently compute the output intervals for both the QNN and the DNN, and then compute their output difference directly by interval subtraction. However, such an approach would be ineffective due to the significant precision loss.

Recently, Paulsen et al. [48] proposed ReluDiff and showed that the accuracy of output difference for two DNNs can be greatly improved by propagating the difference intervals layer-by-layer. For each hidden layer, they first compute the output difference of affine functions (before applying the ReLU), and then they use a ReLU transformer to compute the output difference after applying the ReLU functions. The reason why ReluDiff outperforms the naive method is that ReluDiff first computes part of the difference before it accumulates. ReluDiff is later improved to tighten the approximated difference intervals [49]. However, as mentioned previously, they do not support *fully* quantified neural networks. Inspired by their work, we design a difference propagation algorithm for our setting. We use $S^{in}(\mathbf{x}_j^i)$ (resp. $S^{in}(\hat{\mathbf{x}}_j^i)$) to denote the interval of the j-th neuron in the i-th layer in the DNN (resp. QNN) before applying the ReLU function (resp. clamp function), and use $S(\mathbf{x}_j^i)$ (resp. $S(\hat{\mathbf{x}}_j^i)$) to denote the output interval after applying the ReLU function (resp. clamp function). We use δ_i^{in} (resp. δ_i) to denote the difference interval for the i-th layer before (resp. after) applying the activation functions, and use $\delta_{i,j}^{in}$ (resp. $\delta_{i,j}$) to denote the interval for the j-th neuron of the i-th layer. We denote by $\mathrm{LB}(\cdot)$ and $\mathrm{UB}(\cdot)$ the concrete lower and upper bounds accordingly.

Based on the above notations, we give our difference propagation in Algorithm 1. It works as follows. Given a DNN $\mathcal{N}$, a QNN $\widehat{\mathcal{N}}$ and a quantized input region $R(\hat{\mathbf{x}}, r)$, we first compute intervals $S^{in}(\mathbf{x}_j^i)$ and $S(\mathbf{x}_j^i)$ for neurons in $\mathcal{N}$ using *symbolic* interval analysis DeepPoly, and compute interval $S^{in}(\hat{\mathbf{x}}_j^i)$ and $S(\hat{\mathbf{x}}_j^i)$ for neurons in $\widehat{\mathcal{N}}$ using concrete interval analysis method [22]. Remark that no symbolic interval analysis for QNNs exists. By Definition 3, for each quantized input $\hat{\mathbf{x}}'$ for QNN, we obtain the input for DNN as $\mathbf{x}' = \hat{\mathbf{x}}'/(\mathcal{C}_{in}^{\mathrm{ub}} - \mathcal{C}_{in}^{\mathrm{lb}})$. After precision alignment, we get the input difference as $2^{-F_{in}}\hat{\mathbf{x}}' - \mathbf{x}' = (2^{-F_{in}} - 1/(\mathcal{C}_{in}^{\mathrm{ub}} - \mathcal{C}_{in}^{\mathrm{lb}}))\hat{\mathbf{x}}'$. Hence, given an input region, we get the output difference of the input layer: $\delta_1 = (2^{-F_{in}} - 1/(\mathcal{C}_{in}^{\mathrm{ub}} - \mathcal{C}_{in}^{\mathrm{lb}}))S(\hat{\mathbf{x}}^1)$. Then, we compute the output difference δ_i of each hidden layer iteratively by applying the affine transformer and activation transformer given in Algorithm 2 and Algorithm 3.

Algorithm 1: Forward Difference Propagation

```
Input  : DNN $\mathcal{N}$, QNN $\widehat{\mathcal{N}}$, input region $R(\hat{x}, r)$
output: Output difference interval $\delta$
Compute $S^{in}(\mathbf{x}^i_j)$ and $S(\mathbf{x}^i_j)$ for $i \in [d-1]$, $j \in [n_i]$ using DeepPoly;
Compute $S^{in}(\hat{\mathbf{x}}^i_j)$ and $S(\hat{\mathbf{x}}^i_j)$ for $i \in [d-1]$, $j \in [n_i]$ by applying interval analysis [22];
Initialize the difference: $\delta_1 = (2^{-F_{in}} - 1/(\mathcal{C}^{\text{ub}}_{in} - \mathcal{C}^{\text{lb}}_{in}))S(\hat{\mathbf{x}}^1)$;
for $i$ in $2, \ldots, d-1$ do  // propagate in hidden layers
    for $j$ in $1, \ldots, n_i$ do
        $\Delta\mathbf{b}^i_j = 2^{-F_b}\hat{\mathbf{b}}^i_j - \mathbf{b}^i_j$;  $\xi = 2^{-F_h-1}$;
        $\delta^{in}_{i,j} = \textsc{AffTrs}(\mathbf{W}^i_{j,:}, 2^{-F_w}\widehat{\mathbf{W}}^i_{j,:}, \Delta\mathbf{b}^i_j, S(\mathbf{x}^{i-1}), \delta_{i-1}, \xi)$;
        $\delta_{i,j} = \textsc{ActTrs}(\delta^{in}_{i,j}, S^{in}(\mathbf{x}^i_j), 2^{-F_h}S^{in}(\hat{\mathbf{x}}^i_j))$;
// propagate in the output layer
for $j$ in $1, \ldots, n_d$ do
    $\Delta\mathbf{b}^d_j = 2^{-F_b}\hat{\mathbf{b}}^d_j - \mathbf{b}^d_j$;
    $\delta_{d,j} = \delta^{in}_{d,j} = \textsc{AffTrs}(\mathbf{W}^d_{j,:}, 2^{-F_w}\widehat{\mathbf{W}}^d_{j,:}, \Delta\mathbf{b}^d_j, S(\mathbf{x}^{d-1}), \delta_{d-1}, 0)$;
return $(\delta_{i,j})_{2 \le i \le d, 1 \le j \le n_d}$;
```

Algorithm 2: AffTrs Function

```
Input  : Weight vector $\mathbf{W}^i_{j,:}$, weight vector $\widetilde{\mathbf{W}}^i_{j,:}$, bias difference $\Delta\mathbf{b}^i_j$, neuron interval
         $S(\mathbf{x}^{i-1})$, difference interval $\delta_{i-1}$, rounding error $\xi$
output: Difference interval $\delta^{in}_{i,j}$
$lb = \text{LB}\big(\widetilde{\mathbf{W}}^i_{j,:}\delta_{i-1} + (\widetilde{\mathbf{W}}^i_{j,:} - \mathbf{W}^i_{j,:})S(\mathbf{x}^{i-1})\big) + \Delta\mathbf{b}^i_j - \xi$;
$ub = \text{UB}\big(\widetilde{\mathbf{W}}^i_{j,:}\delta_{i-1} + (\widetilde{\mathbf{W}}^i_{j,:} - \mathbf{W}^i_{j,:})S(\mathbf{x}^{i-1})\big) + \Delta\mathbf{b}^i_j + \xi$;
return $[lb, ub]$;
```

Finally, we get the output difference for the output layer using only the affine transformer.

Affine Transformer. The difference before applying the activation function for the j-th neuron in the i-th layer is: $\delta^{in}_{i,j} = 2^{-F_h}\lfloor 2^{F_i}\widehat{\mathbf{W}}^i_{j,:}S(\hat{\mathbf{x}}^{i-1}) + 2^{F_h-F_b}\hat{\mathbf{b}}^i_j\rceil - \mathbf{W}^i_{j,:}S(\mathbf{x}^{i-1}) - \mathbf{b}^i_j$ where 2^{-F_h} is used to align the precision between the outputs of the two networks (cf. Sect. 2). Then, we soundly remove the rounding operators and give constraints for upper/lower bounds of $\delta^{in}_{i,j}$ as follows:

$$\text{UB}(\delta^{in}_{i,j}) \le \text{UB}(2^{-F_h}(2^{F_i}\widehat{\mathbf{W}}^i_{j,:}S(\hat{\mathbf{x}}^{i-1}) + 2^{F_h-F_b}\hat{\mathbf{b}}^i_j + 0.5) - \mathbf{W}^i_{j,:}S(\mathbf{x}^{i-1}) - \mathbf{b}^i)$$
$$\text{LB}(\delta^{in}_{i,j}) \ge \text{LB}(2^{-F_h}(2^{F_i}\widehat{\mathbf{W}}^i_{j,:}S(\hat{\mathbf{x}}^{i-1}) + 2^{F_h-F_b}\hat{\mathbf{b}}^i_j - 0.5) - \mathbf{W}^i_{j,:}S(\mathbf{x}^{i-1}) - \mathbf{b}^i)$$

Finally, we have $\text{UB}(\delta^{in}_{i,j}) \le \text{UB}\big(\widetilde{\mathbf{W}}^i_{j,:}S(\tilde{\mathbf{x}}^{i-1}) - \mathbf{W}^i_{j,:}S(\mathbf{x}^{i-1})\big) + \Delta\mathbf{b}^i_j + \xi$ and $\text{LB}(\delta^{in}_{i,j}) \ge \text{LB}\big(\widetilde{\mathbf{W}}^i_{j,:}S(\tilde{\mathbf{x}}^{i-1}) - \mathbf{W}^i_{j,:}S(\mathbf{x}^{i-1})\big) + \Delta\mathbf{b}^i_j - \xi$, which can be further reformulated as follows:

$$\text{UB}(\delta^{in}_{i,j}) \le \text{UB}\big(\widetilde{\mathbf{W}}^i_{j,:}\delta_{i-1} + \Delta\mathbf{W}^i_{j,:}S(\mathbf{x}^{i-1})\big) + \Delta\mathbf{b}^i_j + \xi$$
$$\text{LB}(\delta^{in}_{i,j}) \ge \text{LB}\big(\widetilde{\mathbf{W}}^i_{j,:}\delta_{i-1} + \Delta\mathbf{W}^i_{j,:}S(\mathbf{x}^{i-1})\big) + \Delta\mathbf{b}^i_j - \xi$$

where $S(\tilde{\mathbf{x}}^{i-1}) = 2^{-F_{in}}S(\hat{\mathbf{x}}^{i-1})$ if $i = 2$, and $2^{-F_h}S(\hat{\mathbf{x}}^{i-1})$ otherwise. $\widetilde{\mathbf{W}}^i_{j,:} = 2^{-F_w}\widehat{\mathbf{W}}^i_{j,:}$, $\Delta\mathbf{W}^i_{j,:} = \widetilde{\mathbf{W}}^i_{j,:} - \mathbf{W}^i_{j,:}$, $\Delta\mathbf{b}^i_j = 2^{-F_b}\hat{\mathbf{b}}^i_j - \mathbf{b}^i_j$ and $\xi = 2^{-F_h-1}$.

Algorithm 3: ActTrs function

```
Input : Difference interval $\delta^{in}_{i,j}$, neuron interval $S^{in}(\mathbf{x}^i_j)$, neuron interval $S^{in}(\tilde{\mathbf{x}}^i_j)$, clamp
        upper bound $t$
output: Difference interval $\delta_{i,j}$
if $\mathsf{UB}(S^{in}(\mathbf{x}^i_j)) \leq 0$ then $lb = \text{clamp}(\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)), 0, t)$; $ub = \text{clamp}(\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)), 0, t)$;
else if $\mathsf{LB}(S^{in}(\mathbf{x}^i_j)) \geq 0$ then
    if $\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \leq t$ and $\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \geq 0$ then $lb = \mathsf{LB}(\delta^{in}_{i,j})$; $ub = \mathsf{UB}(\delta^{in}_{i,j})$;
    else if $\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \geq t$ or $\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \leq 0$ then
        $lb = \text{clamp}(\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)), 0, t) - \mathsf{UB}(S^{in}(\mathbf{x}^i_j))$;
        $ub = \text{clamp}(\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)), 0, t) - \mathsf{LB}(S^{in}(\mathbf{x}^i_j))$;
    else if $\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \leq t$ then
        $lb = \max(-\mathsf{UB}(S^{in}(\mathbf{x}^i_j)), \mathsf{LB}(\delta^{in}_{i,j}))$; $ub = \max(-\mathsf{LB}(S^{in}(\mathbf{x}^i_j)), \mathsf{UB}(\delta^{in}_{i,j}))$;
    else if $\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \geq 0$ then
        $lb = \min(t - \mathsf{UB}(S^{in}(\mathbf{x}^i_j)), \mathsf{LB}(\delta^{in}_{i,j}))$; $ub = \min(t - \mathsf{LB}(S^{in}(\mathbf{x}^i_j)), \mathsf{UB}(\delta^{in}_{i,j}))$;
    else
        $lb = \max(-\mathsf{UB}(S^{in}(\mathbf{x}^i_j)), \min(t - \mathsf{UB}(S^{in}(\mathbf{x}^i_j)), \mathsf{LB}(\delta^{in}_{i,j})))$;
        $ub = \max(-\mathsf{LB}(S^{in}(\mathbf{x}^i_j)), \min(t - \mathsf{LB}(S^{in}(\mathbf{x}^i_j)), \mathsf{UB}(\delta^{in}_{i,j})))$;
else
    if $\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \leq t$ and $\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \geq 0$ then
        $lb = \min(\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)), \mathsf{LB}(\delta^{in}_{i,j}))$; $ub = \min(\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)), \mathsf{UB}(\delta^{in}_{i,j}))$;
    else if $\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \geq t$ or $\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \leq 0$ then
        $lb = \text{clamp}(\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)), 0, t) - \mathsf{UB}(S^{in}(\mathbf{x}^i_j))$; $ub = \text{clamp}(\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)), 0, t)$;
    else if $\mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \leq t$ then
        $lb = \max(\mathsf{LB}(\delta^{in}_{i,j}), -\mathsf{UB}(S^{in}(\mathbf{x}^i_j)))$; $ub = \min(\mathsf{UB}(\delta^{in}_{i,j}), \mathsf{UB}(S^{in}(\tilde{\mathbf{x}}^i_j)))$;
        if $\mathsf{UB}(\delta^{in}_{i,j}) \leq 0$ then $ub = 0$;
        if $\mathsf{LB}(\delta^{in}_{i,j}) \geq 0$ then $lb = 0$;
    else if $\mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)) \geq 0$ then
        $lb = \min(\mathsf{LB}(\delta^{in}_{i,j}), \mathsf{LB}(S^{in}(\tilde{\mathbf{x}}^i_j)), t - \mathsf{UB}(S^{in}(\mathbf{x}^i_j)))$; $ub = \min(\mathsf{UB}(\delta^{in}_{i,j}), t)$;
    else
        $lb = \min(t - \mathsf{UB}(S^{in}(\mathbf{x}^i_j)), 0, \max(\mathsf{LB}(\delta^{in}_{i,j}), -\mathsf{UB}(S^{in}(\mathbf{x}^i_j))))$;
        $ub = \text{clamp}(\mathsf{UB}(\delta^{in}_{i,j}), 0, t)$;
return $[lb, ub] \cap \big((S^{in}(\tilde{\mathbf{x}}^i_j) \cap [0, t]) - (S^{in}(\mathbf{x}^i_j) \cap [0, +\infty))\big)$;
```

Activation Transformer. Now we give our activation transformer in Algorithm 3 which computes the difference interval $\delta_{i,j}$ from the difference interval $\delta^{in}_{i,j}$. Note that, the neuron interval $S(\hat{\mathbf{x}}^i_j)$ for the QNN has already been converted to the fixed-point counterpart $S(\tilde{\mathbf{x}}^i_j) = 2^{-F_h} S(\hat{\mathbf{x}}^i_j)$ as an input parameter, as well as the clamping upper bound ($t = 2^{-F_h} \mathcal{C}^{\text{ub}}_h$). Different from ReluDiff [48] which focuses on the subtraction of two ReLU functions, here we investigate the subtraction of the clamping function and ReLU function.

Theorem 1. *If $\tau_h = +$, then Algorithm 1 is sound.*

Example 2. We exemplify Algorithm 1 using the networks $\mathcal{N}_e$ and $\widehat{\mathcal{N}}_e$ shown in Fig. 1. Given quantized input region $R((9, 6), 3)$ and the corresponding real-valued input region $R((0.6, 0.4), 0.2)$, we have $S(\hat{\mathbf{x}}^1_1) = [6, 12]$ and $S(\hat{\mathbf{x}}^1_2) = [3, 9]$.

First, we get $S^{in}(\mathbf{x}^2_1) = S(\mathbf{x}^2_1) = [0.36, 0.92]$, $S^{in}(\mathbf{x}^2_2) = [-0.4, 0.2]$, $S(\mathbf{x}^2_2) = [0, 0.2]$ based on DeepPoly and $S^{in}(\hat{\mathbf{x}}^2_1) = S(\hat{\mathbf{x}}^2_1) = [1, 4]$, $S^{in}(\hat{\mathbf{x}}^2_2) = [-2, 1]$, $S(\hat{\mathbf{x}}^2_2) = [0, 1]$ via interval analysis: $\text{LB}(S^{in}(\hat{\mathbf{x}}^2_1)) = \lfloor(5\text{LB}(\hat{\mathbf{x}}^1_1) - \text{UB}(\hat{\mathbf{x}}^1_2))/2^{-4}\rceil = 1$, $\text{UB}(S^{in}(\hat{\mathbf{x}}^2_1)) = \lfloor(5\text{UB}(\hat{\mathbf{x}}^1_1) - \text{LB}(\hat{\mathbf{x}}^1_2))/2^{-4}\rceil = 4$, $\text{LB}(S^{in}(\hat{\mathbf{x}}^2_2)) = \lfloor(-3\text{UB}(\hat{\mathbf{x}}^1_1) +$

$3\mathrm{LB}(\hat{\mathbf{x}}_2^1))/2^{-4}\rceil = -2$, and $\mathrm{UB}(S^{in}(\hat{\mathbf{x}}_2^2)) = \lfloor(-3\mathrm{LB}(\hat{\mathbf{x}}_1^1) + 3\mathrm{UB}(\hat{\mathbf{x}}_2^1))/2^{-4}\rceil = 1$. By Line 3 in Algorithm 1, we have $\delta_{1,1} = -\frac{1}{16\times 15}S(\hat{\mathbf{x}}_1^1) = [-0.05, -0.025]$, $\delta_{1,2} = -\frac{1}{16\times 15}S(\hat{\mathbf{x}}_2^1) = [-0.0375, -0.0125]$.

Then, we compute the difference interval before the activation functions. The rounding error is $\xi = 2^{-F_h-1} = 0.125$. We obtain the difference intervals $\delta_{2,1}^{in} = [-0.194375, 0.133125]$ and $\delta_{2,2}^{in} = [-0.204375, 0.123125]$ as follows based on Algorithm 2:

- $\mathrm{LB}(\delta_{2,1}^{in}) = \mathrm{LB}(\widetilde{\mathbf{W}}_{1,1}^1\delta_{1,1} + \widetilde{\mathbf{W}}_{1,2}^1\delta_{1,2} + \Delta\mathbf{W}_{1,1}^1 S(\mathbf{x}_1^1) + \Delta\mathbf{W}_{1,2}^1 S(\mathbf{x}_2^1)) - \xi = 1.25\times\mathrm{LB}(\delta_{1,1}) - 0.25\times\mathrm{UB}(\delta_{1,2}) + (1.25-1.2)\times\mathrm{LB}(S(\mathbf{x}_1^1)) + (-0.25+0.2)\times\mathrm{UB}(S(\mathbf{x}_2^1)) - 0.125$, $\mathrm{UB}(\delta_{2,1}^{in}) = \mathrm{UB}(\widetilde{\mathbf{W}}_{1,1}^1\delta_{1,1} + \widetilde{\mathbf{W}}_{1,2}^1\delta_{1,2} + \Delta\mathbf{W}_{1,1}^1 S(\mathbf{x}_1^1) + \Delta\mathbf{W}_{1,2}^1 S(\mathbf{x}_2^1)) + \xi = 1.25\times\mathrm{UB}(\delta_{1,1}) - 0.25\times\mathrm{LB}(\delta_{1,2}) + (1.25-1.2)\times\mathrm{UB}(S(\mathbf{x}_1^1)) + (-0.25+0.2)\times\mathrm{LB}(S(\mathbf{x}_2^1)) + 0.125$;
- $\mathrm{LB}(\delta_{2,2}^{in}) = \mathrm{LB}(\widetilde{\mathbf{W}}_{2,1}^1\delta_{1,1} + \widetilde{\mathbf{W}}_{2,2}^1\delta_{1,2} + \Delta\mathbf{W}_{2,1}^1 S(\mathbf{x}_1^1) + \Delta\mathbf{W}_{2,2}^1 S(\mathbf{x}_2^1)) - \xi = -0.75\times\mathrm{UB}(\delta_{1,1}) + 0.75\times\mathrm{LB}(\delta_{1,2}) + (-0.75+0.7)\times\mathrm{UB}(S(\mathbf{x}_1^1)) + (0.75-0.8)\times\mathrm{UB}(S(\mathbf{x}_2^1)) - 0.125$, $\mathrm{UB}(\delta_{2,2}^{in}) = \mathrm{UB}(\widetilde{\mathbf{W}}_{2,1}^1\delta_{1,1} + \widetilde{\mathbf{W}}_{2,2}^1\delta_{1,2} + \Delta\mathbf{W}_{2,1}^1 S(\mathbf{x}_1^1) + \Delta\mathbf{W}_{2,2}^1 S(\mathbf{x}_2^1)) + \xi = -0.75\times\mathrm{LB}(\delta_{1,1}) + 0.75\times\mathrm{UB}(\delta_{1,2}) + (-0.75+0.7)\times\mathrm{LB}(S(\mathbf{x}_1^1)) + (0.75-0.8)\times\mathrm{LB}(S(\mathbf{x}_2^1)) + 0.125$.

By Lines 20∼22 in Algorithm 3, we get the difference intervals after the activation functions for the hidden layer as: $\delta_{2,1} = \delta_{2,1}^{in} = [-0.194375, 0.133125]$, $\delta_{2,1} = [\max\big(\mathrm{LB}(\delta_{2,2}^{in}), -\mathrm{UB}(S^{in}(\mathbf{x}_2^2))\big), \min\big(\mathrm{UB}(\delta_{2,2}^{in}), \mathrm{UB}(S^{in}(\tilde{\mathbf{x}}_2^2))\big)] = [-0.2, 0.123125]$.

Next, we compute the output difference interval of the networks using Algorithm 2 again but with $\xi = 0$: $\mathrm{LB}(\delta_{3,1}^{in}) = \mathrm{LB}(\widetilde{\mathbf{W}}_{1,1}^2\delta_{2,1} + \widetilde{\mathbf{W}}_{1,2}^2\delta_{2,2} + \Delta\mathbf{W}_{1,1}^2 S(\mathbf{x}_1^2) + \Delta\mathbf{W}_{1,2}^2 S(\mathbf{x}_2^2)) = 0.25\times\mathrm{LB}(\delta_{2,1}) + 0.75\times\mathrm{LB}(\delta_{2,2}) + (0.25-0.3)\times\mathrm{UB}(S(\mathbf{x}_1^2)) + (0.75-0.7)\times\mathrm{LB}(S(\mathbf{x}_2^2))$, $\mathrm{UB}(\delta_{3,2}^{in}) = \mathrm{UB}(\widetilde{\mathbf{W}}_{1,1}^2\delta_{2,1} + \widetilde{\mathbf{W}}_{1,2}^2\delta_{2,2} + \Delta\mathbf{W}_{1,1}^2 S(\mathbf{x}_1^2) + \Delta\mathbf{W}_{1,2}^2 S(\mathbf{x}_2^2)) = 0.25\times\mathrm{UB}(\delta_{2,1}) + 0.75\times\mathrm{UB}(\delta_{2,2}) + (0.25-0.3)\times\mathrm{LB}(S(\mathbf{x}_1^2)) + (0.75-0.7)\times\mathrm{UB}(S(\mathbf{x}_2^2))$. Finally, the quantization error interval is [-0.24459375, 0.117625].

3.3 MILP Encoding of the Verification Problem

If DRA fails to prove the property, we encode the problem as an equivalent MILP problem. Specifically, we encode both the QNN and DNN as sets of (mixed integer) linear constraints, and quantize the input region as a set of integer linear constraints. We adopt the MILP encodings of DNNs [39] and QNNs [40] to transform the DNN and QNN into a set of linear constraints. We use (symbolic) intervals to further reduce the size of linear constraints similar to [39] while [40] did not. We suppose that the sets of constraints encoding the QNN, DNN, and quantized input region are $\Theta_{\widehat{\mathcal{N}}}$, $\Theta_{\mathcal{N}}$, and Θ_R, respectively. Next, we give the MILP encoding of the robust error bound property.

Recall that, given a DNN $\mathcal{N}$, an input region $R(\hat{\mathbf{x}}, r)$ such that $\mathbf{x}$ is classified to class g by $\mathcal{N}$, a QNN $\widehat{\mathcal{N}}$ has a quantization error bound ϵ w.r.t. $R(\hat{\mathbf{x}}, r)$ if for every $\hat{\mathbf{x}}' \in R(\hat{\mathbf{x}}, r)$, we have $|2^{-F_h}\widehat{\mathcal{N}}(\hat{\mathbf{x}}')_g - \mathcal{N}(\mathbf{x}')_g| < \epsilon$. Thus, it suffices to check if $|2^{-F_h}\widehat{\mathcal{N}}(\hat{\mathbf{x}}')_g - \mathcal{N}(\mathbf{x}')_g| \geq \epsilon$ for some $\hat{\mathbf{x}}' \in R(\hat{\mathbf{x}}, r)$.

Let $\hat{\mathbf{x}}_g^d$ (resp. $\mathbf{x}_g^d$) be the g-th output of $\widehat{\mathcal{N}}$ (resp. $\mathcal{N}$). We introduce a real-valued variable η and a Boolean variable v such that $\eta = \max(2^{-F_h}\hat{\mathbf{x}}_g^d - \mathbf{x}_g^d, 0)$ can be encoded by the set Θ_g of constraints with an extremely large number $\mathbf{M}$: $\Theta_g = \big\{\eta \geq 0,\ \eta \geq 2^{-F_h}\hat{\mathbf{x}}_g^d - \mathbf{x}_g^d,\ \eta \leq \mathbf{M}\cdot v,\ \eta \leq 2^{-F_h}\hat{\mathbf{x}}_g^d - \mathbf{x}_g^d + \mathbf{M}\cdot(1-v)\big\}$. As a result, $|2^{-F_h}\hat{\mathbf{x}}_g^d - \mathbf{x}_g^d| \geq \epsilon$ iff the set of linear constraints $\Theta_\epsilon = \Theta_g \cup \{2\eta - (2^{-F_h}\hat{\mathbf{x}}_g^d - \mathbf{x}_g^d) \geq \epsilon\}$ holds.

Finally, the quantization error bound verification problem is equivalent to the solving of the constraints: $\Theta_P = \Theta_{\widehat{\mathcal{N}}} \cup \Theta_{\mathcal{N}} \cup \Theta_R \cup \Theta_\epsilon$. Remark that the output difference intervals of hidden neurons obtained from Algorithm 1 can be encoded as linear constraints which are added into the set Θ_P to boost the solving.

4 An Abstract Domain for Symbolic-Based DRA

While Algorithm 1 can compute difference intervals, the affine transformer explicitly adds a concrete rounding error interval to each neuron, which accumulates into a significant precision loss over the subsequent layers. To alleviate this problem, we introduce an abstract domain based on DeepPoly which helps to compute sound symbolic approximations for the lower and upper bounds of each difference interval, hence computing tighter difference intervals.

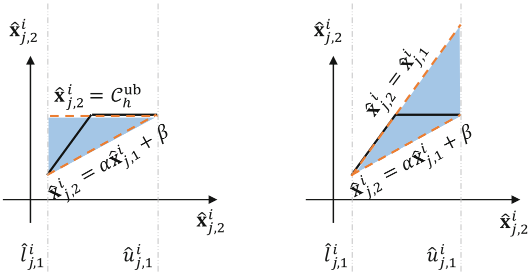
4.1 An Abstract Domain for QNNs

We first introduce transformers for affine transforms with rounding operators and clamp functions in QNNs. Recall that the activation function in a QNN $\widehat{\mathcal{N}}$ is also a min-ReLU function: $\min(\text{ReLU}(\lfloor\cdot\rceil), \mathcal{C}_h^{\text{ub}})$. Thus, we regard each hidden neuron $\hat{\mathbf{x}}_j^i$ in a QNN as three nodes $\hat{\mathbf{x}}_{j,0}^i$, $\hat{\mathbf{x}}_{j,1}^i$, and $\hat{\mathbf{x}}_{j,2}^i$ such that $\hat{\mathbf{x}}_{j,0}^i = \lfloor 2^{F_i}\sum_{k=1}^{n_{i-1}}\widehat{\mathbf{W}}_{j,k}^i\hat{\mathbf{x}}_{k,2}^{i-1} + 2^{F_h-F_b}\hat{\mathbf{b}}_j^i\rceil$ (affine function), $\hat{\mathbf{x}}_{j,1}^i = \max(\hat{\mathbf{x}}_{j,0}^i, 0)$ (ReLU function) and $\hat{\mathbf{x}}_{j,2}^i = \min(\hat{\mathbf{x}}_{j,1}^i, \mathcal{C}_h^{\text{ub}})$ (min function). We now give the abstract domain $\widehat{\mathcal{A}}_{j,p}^i = \langle\hat{\mathbf{a}}_{j,p}^{i,\leq}, \hat{\mathbf{a}}_{j,p}^{i,\geq}, \hat{l}_{j,p}^i, \hat{u}_{j,p}^i\rangle$ for each neuron $\hat{\mathbf{x}}_{j,p}^i$ ($p \in \{0,1,2\}$) in a QNN as follows.

Following DeepPoly, $\hat{\mathbf{a}}_{j,0}^{i,\leq}$ and $\hat{\mathbf{a}}_{j,0}^{i,\geq}$ for the affine function of $\hat{\mathbf{x}}_{j,0}^i$ with rounding operators are defined as $\hat{\mathbf{a}}_{j,0}^{i,\leq} = 2^{F_i}\sum_{k=1}^{n_{i-1}}\widehat{\mathbf{W}}_{j,k}^i\hat{\mathbf{x}}_{k,2}^{i-1} + 2^{F_h-F_b}\hat{\mathbf{b}}_j^i - 0.5$ and $\hat{\mathbf{a}}_{j,0}^{i,\geq} = 2^{F_i}\sum_{k=1}^{n_{i-1}}\widehat{\mathbf{W}}_{j,k}^i\hat{\mathbf{x}}_{k,2}^{i-1} + 2^{F_h-F_b}\hat{\mathbf{b}}_j^i + 0.5$. We remark that $+0.5$ and -0.5 here are added to soundly encode the rounding operators and have no effect on the perseverance of invariant since the rounding operators will add/subtract 0.5 at most to round each floating-point number into its nearest integer. The abstract transformer for the ReLU function $\mathbf{x}_{j,1}^i = \text{ReLU}(\mathbf{x}_{j,0}^i)$ is defined the same as DeepPoly.

For the min function $\hat{\mathbf{x}}_{j,2}^i = \min(\hat{\mathbf{x}}_{j,1}^i, \mathcal{C}_h^{\text{ub}})$, there are three cases for $\widehat{\mathcal{A}}_{j,2}^i$:

- If $\hat{l}_{j,1}^i \geq \mathcal{C}_h^{\text{ub}}$, then $\hat{\mathbf{a}}_{j,2}^{i,\leq} = \hat{\mathbf{a}}_{j,2}^{i,\geq} = \mathcal{C}_h^{\text{ub}}$, $\hat{l}_{j,2}^i = \hat{u}_{j,2}^i = \mathcal{C}_h^{\text{ub}}$;
- If $\hat{u}_{j,1}^i \leq \mathcal{C}_h^{\text{ub}}$, then $\hat{\mathbf{a}}_{j,2}^{i,\leq} = \hat{\mathbf{a}}_{j,1}^{i,\leq}$, $\hat{\mathbf{a}}_{j,2}^{i,\geq} = \hat{\mathbf{a}}_{j,1}^{i,\geq}$, $\hat{l}_{j,2}^i = \hat{l}_{j,1}^i$ and $\hat{u}_{j,2}^i = \hat{u}_{j,1}^i$;


$\hat{\mathbf{x}}^i_{j,2}$
$\hat{\mathbf{x}}^i_{j,2} = \mathcal{C}_h^{\text{ub}}$
$\hat{\mathbf{x}}^i_{j,2} = \alpha\hat{\mathbf{x}}^i_{j,1} + \beta$
$\hat{\mathbf{x}}^i_{j,2}$
$\hat{l}^i_{j,1}$
$\hat{u}^i_{j,1}$
$\hat{\mathbf{x}}^i_{j,2}$
$\hat{\mathbf{x}}^i_{j,2} = \hat{\mathbf{x}}^i_{j,1}$
$\hat{\mathbf{x}}^i_{j,2} = \alpha\hat{\mathbf{x}}^i_{j,1} + \beta$
$\hat{\mathbf{x}}^i_{j,2}$
$\hat{l}^i_{j,1}$
$\hat{u}^i_{j,1}$

(a) $(\gamma, \mu) = (0, \mathcal{C}_h^{\text{ub}})$. (b) $(\gamma, \mu) = (1, 0)$.

Fig. 3. Convex approximation for the min function in QNNs, where Fig. 3(a) and Fig. 3(b) show the two ways where $\alpha = \frac{\mathcal{C}_h^{\text{ub}} - \hat{l}^i_{j,1}}{\hat{u}^i_{j,1} - \hat{l}^i_{j,1}}$ and $\beta = \frac{(\hat{u}^i_{j,1} - \mathcal{C}_h^{\text{ub}})}{\hat{u}^i_{j,1} - \hat{l}^i_{j,1}}$.

- If $\hat{l}^i_{j,1} < \mathcal{C}_h^{\text{ub}} \wedge \hat{u}^i_{j,1} > \mathcal{C}_h^{\text{ub}}$, then $\hat{\mathbf{a}}^{i,\geq}_{j,2} = \lambda\hat{\mathbf{x}}^i_{j,1} + \mu$ and $\hat{\mathbf{a}}^{i,\leq}_{j,2} = \frac{\mathcal{C}_h^{\text{ub}} - \hat{l}^i_{j,1}}{\hat{u}^i_{j,1} - \hat{l}^i_{j,1}}\hat{\mathbf{x}}^i_{j,1} + \frac{(\hat{u}^i_{j,1} - \mathcal{C}_h^{\text{ub}})}{\hat{u}^i_{j,1} - \hat{l}^i_{j,1}}\hat{l}^i_{j,1}$, where $(\lambda, \mu) \in \{(0, \mathcal{C}_h^{\text{ub}}), (1, 0)\}$ such that the area of resulting shape by $\hat{\mathbf{a}}^{i,\leq}_{j,2}$ and $\hat{\mathbf{a}}^{i,\geq}_{j,2}$ is minimal, $\hat{l}^i_{j,2} = \hat{l}^i_{j,1}$ and $\hat{u}^i_{j,2} = \lambda\hat{u}^i_{j,1} + \mu$. We show the two ways of approximation in Fig. 3.

Theorem 2. *The min abstract transformer preserves the following invariant:* $\Gamma(\widehat{\mathcal{A}}^i_{j,2}) \subseteq [\hat{l}^i_{j,2}, \hat{u}^i_{j,2}]$.

From our abstract domain for QNNs, we get a symbolic interval analysis, similar to the one for DNNs using DeepPoly, to replace Line 2 in Algorithm 1.

4.2 Symbolic Quantization Error Computation

Recall that to compute tight bounds of QNNs or DNNs via symbolic interval analysis, variables in upper and lower polyhedral computations are recursively substituted with the corresponding upper/lower polyhedral computations of variables until they only contain the input variables from which the concrete intervals are computed. This idea motivates us to design a symbolic difference computation approach for differential reachability analysis based on the abstract domain DeepPoly for DNNs and our abstract domain for QNNs.

Consider two hidden neurons $\mathbf{x}^i_{j,s}$ and $\hat{\mathbf{x}}^i_{j,s}$ from the DNN $\mathcal{N}$ and the QNN $\widehat{\mathcal{N}}$. Let $\mathcal{A}^{i,*}_{j,s} = \langle \mathbf{a}^{i,\leq,*}_{j,s}, \mathbf{a}^{i,\geq,*}_{j,s}, l^{i,*}_{j,s}, u^{i,*}_{j,s} \rangle$ and $\widehat{\mathcal{A}}^i_{j,p} = \langle \hat{\mathbf{a}}^{i,\leq,*}_{j,p}, \hat{\mathbf{a}}^{i,\geq,*}_{j,p}, \hat{l}^{i,*}_{j,p}, \hat{u}^{i,*}_{j,p} \rangle$ be their abstract elements, respectively, where all the polyhedral computations are linear combinations of the input variables of the DNN and QNN, respectively, i.e.,

- $\mathbf{a}^{i,\leq,*}_{j,s} = \sum_{k=1}^m \mathbf{w}^{l,*}_k \mathbf{x}^1_k + \mathbf{b}^{l,*}_j$, $\mathbf{a}^{i,\geq,*}_{j,s} = \sum_{k=1}^m \mathbf{w}^{u,*}_k \mathbf{x}^1_k + \mathbf{b}^{u,*}_j$;
- $\hat{\mathbf{a}}^{i,\leq,*}_{j,p} = \sum_{k=1}^m \hat{\mathbf{w}}^{l,*}_k \hat{\mathbf{x}}^1_k + \hat{\mathbf{b}}^{l,*}_j$, $\hat{\mathbf{a}}^{i,\geq,*}_{j,p} = \sum_{k=1}^m \hat{\mathbf{w}}^{u,*}_k \hat{\mathbf{x}}^1_k + \hat{\mathbf{b}}^{u,*}_j$.

Then, the sound lower bound $\Delta l^{i,*}_{j,s}$ and upper $\Delta u^{i,*}_{j,s}$ bound of the difference can be derived as follows, where $p = 2s$:

Table 1. Benchmarks for QNNs and DNNs on MNIST.

Arch	#Paras	QNNs				DNNs
		$Q = 4$	$Q = 6$	$Q = 8$	$Q = 10$	
P1: 1blk_100	≈ 79.5k	96.38%	96.79%	96.77%	96.74%	96.92%
P2: 2blk_100	≈ 89.6k	96.01%	97.04%	97.00%	97.02%	97.07%
P3: 3blk_100	≈ 99.7k	95.53%	96.66%	96.59%	96.68%	96.71%
P4: 2blk_512	≈ 669.7k	96.69%	97.41%	97.35%	97.36%	97.36%
P5: 4blk_1024	≈ 3,963k	97.71%	98.05%	98.01%	98.04%	97.97%

- $\varDelta l^{i,*}_{j,s} = \mathrm{LB}(2^{-F_h}\hat{\mathbf{x}}^i_{j,p} - \mathbf{x}^i_{j,s}) = 2^{-F_h}\hat{\mathbf{a}}^{i,\leq,*}_{j,p} - \mathbf{a}^{i,\geq,*}_{j,s}$;
- $\varDelta u^{i,*}_{j,s} = \mathrm{UB}(2^{-F_h}\hat{\mathbf{x}}^i_{j,p} - \mathbf{x}^i_{j,s}) = 2^{-F_h}\hat{\mathbf{a}}^{i,\geq,*}_{j,p} - \mathbf{a}^{i,\leq,*}_{j,s}$.

Given a quantized input $\hat{\mathbf{x}}$ of the QNN $\widehat{\mathcal{N}}$, the input difference of two networks is $2^{-F_{in}}\hat{\mathbf{x}} - \mathbf{x} = (2^{-F_{in}}\mathcal{C}^{\text{ub}}_h - 1)\mathbf{x}$. Therefore, we have $\varDelta^1_k = \tilde{\mathbf{x}}^1_k - \mathbf{x}^1_k = 2^{-F_{in}}\hat{\mathbf{x}}^1_k - \mathbf{x}^1_k = (2^{-F_{in}}\mathcal{C}^{\text{ub}}_h - 1)\mathbf{x}$. Then, the lower bound of difference can be reformulated as follows which only contains the input variables of DNN $\mathcal{N}$: $\varDelta l^{i,*}_{j,s} = \varDelta \mathbf{b}^{l,*}_j + \sum_{k=1}^m(-\mathbf{w}^{u,*}_k + 2^{-F_{in}}\mathcal{C}^{\text{ub}}_h\tilde{\mathbf{w}}^{l,*}_k)\mathbf{x}^1_k$, where $\varDelta \mathbf{b}^{l,*}_j = 2^{-F_h}\hat{\mathbf{b}}^{l,*}_j - \mathbf{b}^{u,*}_j$, $F^* = F_{in} - F_h$, $\varDelta^1_k = \tilde{\mathbf{x}}^1_k - \mathbf{x}^1_k$ and $\tilde{\mathbf{w}}^{l,*}_k = 2^{F^*}\hat{\mathbf{w}}^{l,*}_k$.

Similarly, we can reformulated the upper bound $\varDelta u^{i,*}_{j,s}$ as follows using the input variables of the DNN: $\varDelta u^{i,*}_{j,s} = \varDelta \mathbf{b}^{u,*}_j + \sum_{k=1}^m(-\mathbf{w}^{l,*}_k + 2^{-F_{in}}\mathcal{C}^{\text{ub}}_h\tilde{\mathbf{w}}^{u,*}_k)\mathbf{x}^1_k$, where $\varDelta \mathbf{b}^{u,*}_j = 2^{-F_h}\hat{\mathbf{b}}^{u,*}_j - \mathbf{b}^{l,*}_j$, $F^* = F_{in} - F_h$, and $\tilde{\mathbf{w}}^{u,*}_k = 2^{F^*}\hat{\mathbf{w}}^{u,*}_k$.

Finally, we compute the concrete input difference interval $\delta^{in}_{i,j}$ based on the given input region as $\delta^{in}_{i,j} = [\mathrm{LB}(\varDelta l^{i,*}_{j,0}), \mathrm{UB}(\varDelta u^{i,*}_{j,0})]$, with which we can replace the AffTrs functions in Algorithm 1 directly. An illustrating example is given in [65].

5 Evaluation

We have implemented our method QEBVerif as an end-to-end tool written in Python, where we use Gurobi [20] as our back-end MILP solver. All floating-point numbers used in our tool are 32-bit. Experiments are conducted on a 96-core machine with Intel(R) Xeon(R) Gold 6342 2.80 GHz CPU and 1 TB main memory. We allow Gurobi to use up to 24 threads. The time limit for each verification task is 1 h.

Benchmarks. We first build 45 * 4 QNNs from the 45 DNNs of ACAS Xu [26], following a *post-training quantization scheme* [44] and using quantization configurations $\mathcal{C}_{in} = \langle \pm, 8, 8 \rangle$, $\mathcal{C}_w = \mathcal{C}_b = \langle \pm, Q, Q-2 \rangle$, $\mathcal{C}_h = \langle +, Q, Q-2 \rangle$, where $Q \in \{4, 6, 8, 10\}$. We then train 5 DNNs with different architectures using the MNIST dataset [31] and build 5 * 4 QNNs following the same quantization scheme and quantization configurations except that we set $\mathcal{C}_{in} = \langle +, 8, 8 \rangle$ and $\mathcal{C}_w = \langle \pm, Q, Q-1 \rangle$ for each DNN trained on MNIST. Details on the networks trained on the MNIST dataset are presented in Table 1. Column 1 gives the name and architecture of each DNN, where Ablk_B means that the network has A hidden layers with each hidden layer size B neurons, Column 2 gives the number of parameters in each DNN, and Columns 3–7 list the accuracy of these networks. Hereafter, we denote by Px-y (resp. Ax-y) the QNN using the architecture Px (using the x-th DNN) and quantization bit size $Q = y$ for MNIST (resp. ACAS Xu), and by Px-Full (resp. Ax-Full) the DNN of architecture Px for MNIST (resp. the x-th DNN in ACAS Xu).

5.1 Effectiveness and Efficiency of DRA

We first implement a naive method using existing state-of-the-art reachability analysis methods for QNNs and DNNs. Specifically, we use the symbolic interval analysis of DeepPoly [55] to compute the output intervals for a DNN, and use interval analysis of [22] to compute the output intervals for a QNN. Then, we compute quantization error intervals via interval subtraction. Note that no existing methods can directly verify quantization error bounds and the methods in [48,49] are not applicable. Finally, we compare the quantization error intervals computed by the naive method against DRA in QEBVerif, using DNNs Ax-Full, Py-Full and QNNs Ax-z, Py-z for $x = 1$, $y \in \{1, 2, 3, 4, 5\}$ and $z \in \{4, 6, 8, 10\}$. We use the same adversarial input regions (5 input points with radius $r = \{3, 6, 13, 19, 26\}$ for each point) as in [29] for ACAS Xu, and set the quantization error bound $\epsilon \in \{0.05, 0.1, 0.2, 0.3, 0.4\}$, i.e., resulting 25 tasks for each radius. For MNIST, we randomly select 30 input samples from the test set of MNIST and set radius $r = 3$ for each input sample and quantization error bound $\epsilon \in \{1, 2, 4, 6, 8\}$, resulting in a total of 150 tasks for each pair of DNN and QNN of same architecture for MNIST.

Table 2 reports the analysis results for ACAS Xu (above) and MNIST (below). Column 2 lists different analysis methods, where QEBVerif (Int) is Algorithm 1 and QEBVerif (Sym) uses a symbolic-based method for the affine transformation in Algorithm 1 (cf. Sect. 4.2). Columns (H_Diff) (resp. O_Diff) averagely give the sum ranges of the difference intervals of all the hidden neurons (resp. output neurons of the predicted class) for the 25 verification tasks for ACAS Xu and 150 verification tasks for MNIST. Columns (#S/T) list the number of tasks (#S) successfully proved by DRA and average computation time (T) in seconds, respectively, where the best ones (i.e., solving the most tasks) are highlighted in blue. Note that Table 2 only reports the number of true propositions proved by DRA while the exact number is unknown.

Unsurprisingly, QEBVerif (Sym) is less efficient than the others but is still in the same order of magnitude. However, we can observe that QEBVerif (Sym) solves the most tasks for both ACAS Xu and MNIST and produces the most accurate difference intervals of both hidden neurons and output neurons for almost all the tasks in MNIST, except for P1-8 and P1-10 where QEBVerif (Int) performs better on the intervals for the output neurons. We also find that QEBVerif (Sym) may perform worse than the naive method when the quantization bit size is small for ACAS Xu. It is because: (1) the rounding error added into

Table 2. Differential Reachability Analysis on ACAS Xu and MNIST.

Q	Method	$r = 3$			$r = 6$			$r = 13$			$r = 19$			$r = 26$		
		H_Diff	O_Diff	#S/T	H_Diff	O_Diff	#S/T	H_Diff	O_Diff	#S/T	H_Diff	O_Diff	#S/T	H_Diff	O_Diff	#S/T
4	Naive	270.5	0.70	15/0.47	423.7	0.99	9/0.52	1,182	4.49	0/0.67	6,110	50.91	0/0.79	18,255	186.6	0/0.81
	QEBVerif (Int)	270.5	0.70	15/0.49	423.4	0.99	9/0.53	1,181	4.46	0/0.70	6,044	50.91	0/0.81	17,696	186.6	0/0.85
	QEBVerif (Sym)	749.4	145.7	0/2.02	780.9	150.2	0/2.11	1,347	210.4	0/2.24	6,176	254.7	0/2.35	18,283	343.7	0/2.39
6	Naive	268.3	1.43	5/0.47	557.2	4.00	0/0.51	1,258	6.91	0/0.67	6,145	53.29	0/0.77	18,299	189.0	0/0.82
	QEBVerif (Int)	268.0	1.41	5/0.50	555.0	3.98	0/0.54	1,245	6.90	0/0.69	6,125	53.28	0/0.80	18,218	189.0	0/0.83
	QEBVerif (Sym)	299.7	2.58	10/1.48	365.1	3.53	9/1.59	1,032	7.65	5/1.91	5,946	85.46	4/2.15	18,144	260.5	0/2.27
8	Naive	397.2	3.57	0/0.47	587.7	5.00	0/0.51	1,266	7.90	0/0.67	6,160	54.27	0/0.78	18,308	190.0	0/0.81
	QEBVerif (Int)	388.4	3.56	0/0.49	560.1	5.00	0/0.53	1,222	7.89	0/0.69	6,103	54.27	0/0.79	18,212	190.0	0/0.83
	QEBVerif (Sym)	35.75	0.01	24/1.10	93.78	0.16	18/1.19	845.2	5.84	8/1.65	5,832	58.73	5/1.97	18,033	209.6	5/2.12
10	Naive	394.5	3.67	0/0.49	591.4	5.17	0/0.51	1,268	8.04	0/0.68	6,164	54.42	0/0.78	18,312	190.1	0/0.80
	QEBVerif (Int)	361.9	3.67	0/0.50	546.2	5.17	0/0.54	1,209	8.04	0/0.68	6,083	54.42	0/0.79	18,182	190.1	0/0.83
	QEBVerif (Sym)	15.55	0.01	25/1.04	54.29	0.06	22/1.15	764.6	4.53	9/1.52	5,780	57.21	5/1.91	18,011	228.7	5/2.08

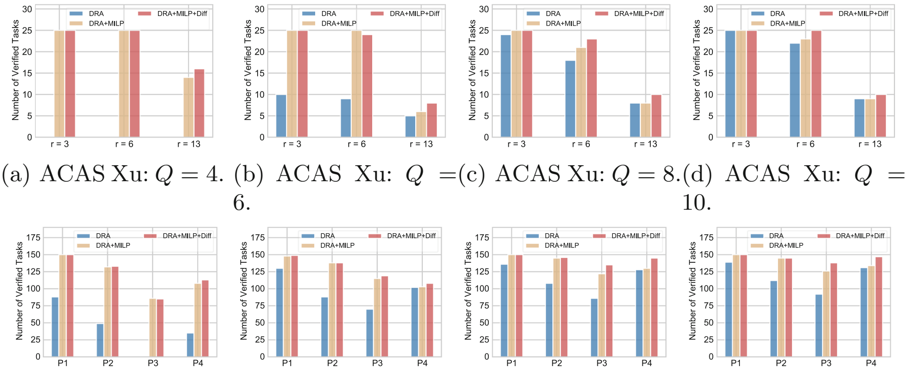
Q	Method	P1			P2			P3			P4			P5		
		H_Diff	O_Diff	#S/T	H_Diff	O_Diff	#S/T	H_Diff	O_Diff	#S/T	H_Diff	O_Diff	#S/T	H_Diff	O_Diff	#S/T
4	Naive	64.45	7.02	61/0.77	220.9	20.27	0/1.53	551.6	47.75	0/2.38	470.1	22.69	2/11.16	5,336	140.4	0/123.0
	QEBVerif (Int)	32.86	6.65	63/0.78	194.8	20.27	0/1.54	530.9	47.75	0/2.40	443.3	22.69	2/11.23	5,275	140.4	0/123.4
	QEBVerif (Sym)	32.69	3.14	88/1.31	134.9	7.11	49/2.91	313.8	14.90	1/5.08	365.2	11.11	35/22.28	1,864	50.30	1/310.2
6	Naive	68.94	7.89	66/0.77	249.5	24.25	0/1.52	616.2	54.66	0/2.38	612.2	31.67	1/11.18	7,399	221.0	0/125.4
	QEBVerif (Int)	10.33	2.19	115/0.78	89.66	12.81	14/1.54	466.0	52.84	0/2.39	307.6	20.22	5/11.28	7,092	221.0	0/125.1
	QEBVerif (Sym)	10.18	1.46	130/1.34	55.73	3.11	88/2.85	131.3	5.33	70/4.72	158.5	3.99	102/21.85	861.9	12.67	22/279.9
8	Naive	69.15	7.95	64/0.77	251.6	24.58	0/1.52	623.1	55.42	0/2.38	620.6	32.43	1/11.29	7,542	226.1	0/125.3
	QEBVerif (Int)	4.27	0.89	135/0.78	38.87	5.99	66/1.54	320.1	40.84	0/2.39	134.0	8.99	50/11.24	7,109	226.1	0/125.7
	QEBVerif (Sym)	4.13	1.02	136/1.35	34.01	2.14	108/2.82	82.90	3.48	86/4.61	96.26	2.39	128/21.45	675.7	6.20	27/273.6
10	Naive	69.18	7.96	65/0.77	252.0	24.63	0/1.52	624.0	55.55	0/2.36	620.4	32.40	1/11.19	7,559	226.9	0/124.2
	QEBVerif (Int)	2.72	0.56	139/0.78	25.39	4.15	79/1.53	260.9	34.35	0/2.40	84.12	5.75	73/11.26	7,090	226.9	0/125.9
	QEBVerif (Sym)	2.61	0.92	139/1.35	28.59	1.91	112/2.82	71.33	3.06	92/4.56	81.08	2.01	131/21.48	646.5	5.68	31/271.5

the abstract domain of the affine function in each hidden layer of QNNs is large due to the small bit size, and (2) such errors can accumulate and magnify layer by layer, in contrast to the naive approach where we directly apply the interval subtraction. We remark that symbolic-based reachability analysis methods for DNNs become less accurate as the network gets deeper and the input region gets larger. It means that for a large input region, the output intervals of hidden/output neurons computed by symbolic interval analysis for DNNs can be very large. However, the output intervals of their quantized counterparts are always limited by the quantization grid limit, i.e., $[0, \frac{2^Q-1}{2^{Q-2}}]$. Hence, the difference intervals computed in Table 2 can be very conservative for large input regions and deeper networks.

5.2 Effectiveness and Efficiency of QEBVerif

We evaluate QEBVerif on QNNs Ax-z, Py-z for $x = 1$, $y \in \{1,2,3,4\}$ and $z \in \{4,6,8,10\}$, as well as DNNs correspondingly. We use the same input regions and error bounds as in Sect. 5.1 except that we consider $r \in \{3,6,13\}$ for each input point for ACAS Xu. Note that, we omit the other two radii for ACAS Xu and use medium-sized QNNs for MNIST as our evaluation benchmarks of this experiment for the sake of time and computing resources.

Figure 4 shows the verification results of QEBVerif within 1 h per task, which gives the number of successfully verified tasks with three methods. Note that only the number of successfully proved tasks is given in Fig. 4 for DRA due to its incompleteness. The blue bars show the results using only the symbolic



(a) ACAS Xu: $Q = 4$. (b) ACAS Xu: $Q = 6$. (c) ACAS Xu: $Q = 8$. (d) ACAS Xu: $Q = 10$.

(e) MNIST: $Q = 4$. (f) MNIST: $Q = 6$. (g) MNIST: $Q = 8$. (h) MNIST: $Q = 10$.

Fig. 4. Verification Results of QEBVerif on ACAS Xu and MNIST.

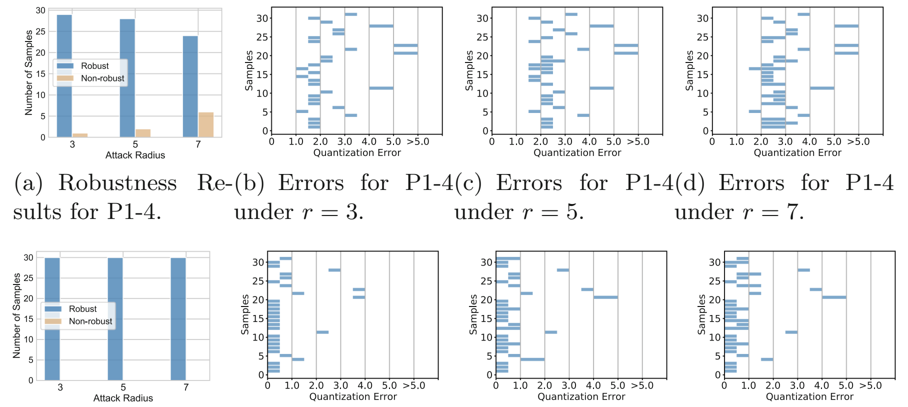
differential reachability analysis, i.e., QEBVerif (Sym). The yellow bars give the results by a full verification process in QEBVerif as shown in Fig. 2, i.e., we first use DRA and then use MILP solving if DRA fails. The red bars are similar to the yellow ones except that linear constraints of the difference intervals of hidden neurons got from DRA are added into the MILP encoding.

Overall, although DRA successfully proved most of the tasks (60.19% with DRA solely), our MILP-based verification method can help further verify many tasks on which DRA fails, namely, 85.67% with DRA+MILP and 88.59% with DRA+MILP+Diff. Interestingly, we find that the effectiveness of the added linear constraints of the difference intervals varies on the MILP solving efficiency on different tasks. Our conjecture is that there are some heuristics in the Gurobi solving algorithm for which the additional constraints may not always be helpful. However, those difference linear constraints allow the MILP-based verification method to verify more tasks, i.e., 79 tasks more in total.

5.3 Correlation of Quantization Errors and Robustness

We use QEBVerif to verify a set of properties $\mathit{\Psi} = \{P(\mathcal{N}, \widehat{\mathcal{N}}, \hat{\mathbf{x}}, r, \epsilon)\}$, where $\mathcal{N}$ = P1-Full, $\widehat{\mathcal{N}} \in$ {P1-4, P1-8}, $\hat{\mathbf{x}} \in \mathcal{X}$ and $\mathcal{X}$ is the set of the 30 samples from MNIST as above, $r \in \{3, 5, 7\}$ and $\epsilon \in \mathit{\Omega} = \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 5.0\}$. We solve all the above tasks and process all the results to obtain the tightest range of quantization error bounds $[a, b]$ for each input region such that $a, b \in \mathit{\Omega}$. It allows us to obtain intervals that are tighter than those obtained via DRA. Finally, we implemented a robustness verifier for QNNs in a way similar to [40] to check the robustness of P1-4 and P1-8 w.r.t. the input regions given in $\mathit{\Psi}$.

Figure 5 gives the experimental results. The blue (resp. yellow) bars in Figs. 5(a) and 5(e) show the number of robust (resp. non-robust) samples among the 30 verification tasks, and blue bars in the other 6 figures demonstrate the



(a) Robustness Results for P1-4.
(b) Errors for P1-4 under $r = 3$.
(c) Errors for P1-4 under $r = 5$.
(d) Errors for P1-4 under $r = 7$.

(e) Robustness Results for P1-8.
(f) Errors for P1-8 under $r = 3$.
(g) Errors for P1-8 under $r = 5$.
(h) Errors for P1-8 under $r = 7$.

Fig. 5. Distribution of (non-)robust samples and Quantization Errors under radius r and quantization bits Q.

quantization error interval for each input region. By comparing the results of P1-8 and P1-4, we observe that P1-8 is more robust than P1-4 w.r.t. the 90 input regions and its quantization errors are also generally much smaller than that of P1-4. Furthermore, we find that P1-8 remains consistently robust as the radius increases, and its quantization error interval changes very little. However, P1-4 becomes increasingly less robust as the radius increases and its quantization error also increases significantly. Thus, we speculate that there may be some correlation between network robustness and quantization error in QNNs. Specifically, as the quantization bit size decreases, the quantization error increases and the QNN becomes less robust. The reason we suspect "the fewer bits, the less robust" is that with fewer bits, a perturbation may easily cause significant change on hidden neurons (i.e., the change is magnified by the loss of precision) and consequently the output. Furthermore, the correlation between the quantization error bound and the empirical robustness of the QNN suggests that it is indeed possible to apply our method to compute the quantization error bound and use it as a guide for identifying the best quantization scheme which balances the size of the model and its robustness.

6 Related Work

While there is a large and growing body of work on quality assurance techniques for neural networks including testing (e.g., [4–7,47,50,56,57,63,69]) and formal verification (e.g., [2,8,12,13,15,19,24,29,30,32,34,37,38,51,54,55,58–60, 62,70]). Testing techniques are often effective in finding violations, but they cannot prove their absence. While formal verification can prove their absence, existing methods typically target real-valued neural networks, i.e., DNNs, and are not effective in verifying quantization error bound [48]. In this section, we mainly discuss the existing verification techniques for QNNs.

Early work on formal verification of QNNs typically focuses on 1-bit quantized neural networks (i.e., BNNs) [3,9,46,52,53,66,67]. Narodytska et al. [46] first proposed to reduce the verification problem of BNNs to a satisfiability problem of a Boolean formula or an integer linear programming problem. Baluta et al. [3] proposed a PAC-style quantitative analysis framework for BNNs via approximate SAT model-counting solvers. Shih et al. proposed a quantitative verification framework for BNNs [52,53] via a BDD learning-based method [45]. Zhang et al. [66,67] proposed a BDD-based verification framework for BNNs, which exploits the internal structure of the BNNs to construct BDD models instead of BDD-learning. Giacobbe et al. [16] pushed this direction further by introducing the first formal verification for multiple-bit quantized DNNs (i.e., QNNs) by encoding the robustness verification problem into an SMT formula based on the first-order theory of quantifier-free bit-vector. Later, Henzinger et al. [22] explored several heuristics to improve the efficiency and scalability of [16]. Very recently, [40,68] proposed an ILP-based method and an MILP-based verification method for QNNs, respectively, and both outperform the SMT-based verification approach [22]. Though these works can directly verify QNNs or BNNs, they cannot verify quantization error bounds.

There are also some works focusing on exploring the properties of two neural networks which are most closely related to our work. Paulsen et al. [48,49] proposed differential verification methods to verify two DNNs with the same network topology. This idea has been extended to handle recurrent neural networks [41]. The difference between [41,48,49] and our work has been discussed throughout this work, i.e., they focus on quantized weights and cannot handle quantized activation tensors. Moreover, their methods are not complete, thus would fail to prove tighter error bounds. Semi-definite programming was used to analyze the different behaviors of DNNs and *fully* QNNs [33]. Different from our work focusing on verification, they aim at generating an upper bound for the worst-case error induced by quantization. Furthermore, [33] only scales tiny QNNs, e.g., 1 input neuron, 1 output neuron, and 10 neurons per hidden layer (up to 4 hidden layers). In comparison, our differential reachability analysis scales to much larger QNNs, e.g., QNN with 4890 neurons.

7 Conclusion

In this work, we proposed a novel quantization error bound verification method QEBVerif which is sound, complete, and arguably efficient. We implemented it as an end-to-end tool and conducted thorough experiments on various QNNs with different quantization bit sizes. Experimental results showed the effectiveness and the efficiency of QEBVerif. We also investigated the potential correlation between robustness and quantization errors for QNNs and found that as the quantization error increases the QNN might become less robust. For further work, it would be interesting to investigate the verification method for other activation functions and network architectures, towards which this work makes a significant step.

Acknowledgements. This work is supported by the National Key Research Program (2020AAA0107800), National Natural Science Foundation of China (62072309), CAS Project for Young Scientists in Basic Research (YSBR-040), ISCAS New Cultivation Project (ISCAS-PYFX-202201), and the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the Ministry of Education, Singapore.

References

1. Amir, G., Wu, H., Barrett, C.W., Katz, G.: An SMT-based approach for verifying binarized neural networks. In: Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 203–222 (2021)
2. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 731–744 (2019)
3. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1249–1264 (2019)
4. Bu, L., Zhao, Z., Duan, Y., Song, F.: Taking care of the discretization problem: a comprehensive study of the discretization problem and a black-box adversarial attack in discrete integer domain. IEEE Trans. Dependable Secur. Comput. **19**(5), 3200–3217 (2022)
5. Carlini, N., Wagner, D.A.: Towards evaluating the robustness of neural networks. In: Proceedings of the 2017 IEEE Symposium on Security and Privacy, pp. 39–57 (2017)
6. Chen, G., et al.: Who is real Bob? Adversarial attacks on speaker recognition systems. In: Proceedings of the 42nd IEEE Symposium on Security and Privacy, pp. 694–711 (2021)
7. Chen, G., Zhao, Z., Song, F., Chen, S., Fan, L., Liu, Y.: AS2T: arbitrary source-to-target adversarial attack on speaker recognition systems. IEEE Trans. Dependable Secur. Comput., 1–17 (2022)
8. Chen, G., et al.: Towards understanding and mitigating audio adversarial examples for speaker recognition. IEEE Trans. Dependable Secur. Comput., 1–17 (2022)
9. Choi, A., Shi, W., Shih, A., Darwiche, A.: Compiling neural networks into tractable Boolean circuits. In: Proceedings of the AAAI Spring Symposium on Verification of Neural Networks (2019)
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, pp. 238–252 (1977)
11. Duncan, K., Komendantskaya, E., Stewart, R., Lones, M.: Relative robustness of quantized neural networks against adversarial attacks. In: Proceedings of the International Joint Conference on Neural Networks, pp. 1–8 (2020)
12. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis, pp. 269–286 (2017)

13. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: Proceedings of the 32nd International Conference on Computer Aided Verification, pp. 43–65 (2020)
14. Eykholt, K., et al.: Robust physical-world attacks on deep learning visual classification. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1625–1634 (2018)
15. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI^2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 3–18 (2018)
16. Giacobbe, M., Henzinger, T.A., Lechner, M.: How many bits does it take to quantize your neural network? In: TACAS 2020. LNCS, vol. 12079, pp. 79–97. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_5
17. Gong, R., et al.: Differentiable soft quantization: bridging full-precision and low-bit neural networks. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 4851–4860 (2019)
18. Google: Tensorflow lite (2022). https://www.tensorflow.org/lite
19. Guo, X., Wan, W., Zhang, Z., Zhang, M., Song, F., Wen, X.: Eager falsification for accelerating robustness verification of deep neural networks. In: Proceedings of the 32nd IEEE International Symposium on Software Reliability Engineering, pp. 345–356 (2021)
20. Gurobi: a most powerful mathematical optimization solver (2018). https://www.gurobi.com/
21. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding. In: Proceedings of the 4th International Conference on Learning Representations (2016)
22. Henzinger, T.A., Lechner, M., Zikelic, D.: Scalable verification of quantized neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, pp. 3787–3795 (2021)
23. Hinton, G., et al.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Sig. Process. Mag. **29**(6), 82–97 (2012)
24. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings of the 29th International Conference on Computer Aided Verification, pp. 3–29 (2017)
25. Jacob, B., et al.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2704–2713 (2018)
26. Julian, K.D., Kochenderfer, M.J., Owen, M.P.: Deep neural network compression for aircraft collision avoidance systems. J. Guid. Control. Dyn. **42**(3), 598–608 (2019)
27. Jung, S., et al.: Learning to quantize deep networks by optimizing quantization intervals with task loss. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4350–4359 (2019)
28. Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., Li, F.: Large-scale video classification with convolutional neural networks. In: Proceedings of 2014 IEEE Conference on Computer Vision and Pattern Recognition, pp. 1725–1732 (2014)
29. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Proceedings of the 29th International Conference on Computer Aided Verification, pp. 97–117 (2017)

30. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Proceedings of the 31st International Conference on Computer Aided Verification, pp. 443–452 (2019)
31. LeCun, Y., Cortes, C.: MNIST handwritten digit database (2010)
32. 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: Chang, B.-Y.E. (ed.) SAS 2019. LNCS, vol. 11822, pp. 296–319. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32304-2_15
33. Li, J., Drummond, R., Duncan, S.R.: Robust error bounds for quantised and pruned neural networks. In: Proceedings of the 3rd Annual Conference on Learning for Dynamics and Control, pp. 361–372 (2021)
34. Li, R., et al.: Prodeep: a platform for robustness verification of deep neural networks. In: Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1630–1634 (2020)
35. Lin, D.D., Talathi, S.S., Annapureddy, V.S.: Fixed point quantization of deep convolutional networks. In: Proceedings of the 33nd International Conference on Machine Learning, pp. 2849–2858 (2016)
36. Lin, J., Gan, C., Han, S.: Defensive quantization: when efficiency meets robustness. In: Proceedings of the International Conference on Learning Representations (2019)
37. Liu, J., Xing, Y., Shi, X., Song, F., Xu, Z., Ming, Z.: Abstraction and refinement: towards scalable and exact verification of neural networks. CoRR abs/2207.00759 (2022)
38. Liu, W., Song, F., Zhang, T., Wang, J.: Verifying ReLU neural networks from a model checking perspective. J. Comput. Sci. Technol. **35**(6), 1365–1381 (2020)
39. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward ReLU neural networks. CoRR abs/1706.07351 (2017)
40. Mistry, S., Saha, I., Biswas, S.: An MILP encoding for efficient verification of quantized deep neural networks. IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst. (Early Access) (2022)
41. Mohammadinejad, S., Paulsen, B., Deshmukh, J.V., Wang, C.: DiffRNN: differential verification of recurrent neural networks. In: Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems, pp. 117–134 (2021)
42. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis, vol. 110. SIAM (2009)
43. Nagel, M., Amjad, R.A., Van Baalen, M., Louizos, C., Blankevoort, T.: Up or down? Adaptive rounding for post-training quantization. In: Proceedings of the International Conference on Machine Learning, pp. 7197–7206 (2020)
44. Nagel, M., Fournarakis, M., Amjad, R.A., Bondarenko, Y., van Baalen, M., Blankevoort, T.: A white paper on neural network quantization. arXiv preprint arXiv:2106.08295 (2021)
45. Nakamura, A.: An efficient query learning algorithm for ordered binary decision diagrams. Inf. Comput. **201**(2), 178–198 (2005)
46. Narodytska, N., Kasiviswanathan, S.P., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 6615–6624 (2018)
47. Odena, A., Olsson, C., Andersen, D.G., Goodfellow, I.J.: TensorFuzz: debugging neural networks with coverage-guided fuzzing. In: Proceedings of the 36th International Conference on Machine Learning, pp. 4901–4911 (2019)

48. Paulsen, B., Wang, J., Wang, C.: ReluDiff: differential verification of deep neural networks. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 714–726. IEEE (2020)
49. Paulsen, B., Wang, J., Wang, J., Wang, C.: NeuroDiff: scalable differential verification of neural networks using fine-grained approximation. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 784–796 (2020)
50. Pei, K., Cao, Y., Yang, J., Jana, S.: DeepXplore: automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 1–18 (2017)
51. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Proceedings of the 22nd International Conference on Computer Aided Verification, pp. 243–257 (2010)
52. Shih, A., Darwiche, A., Choi, A.: Verifying binarized neural networks by Angluin-style learning. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 354–370. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_25
53. Shih, A., Darwiche, A., Choi, A.: Verifying binarized neural networks by local automaton learning. In: Proceedings of the AAAI Spring Symposium on Verification of Neural Networks (2019)
54. Singh, G., Ganvir, R., Püschel, M., Vechev, M.T.: Beyond the single neuron convex barrier for neural network certification. In: Proceedings of the Annual Conference on Neural Information Processing Systems, pp. 15072–15083 (2019)
55. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. (POPL) **3**, 41:1–41:30 (2019)
56. Song, F., Lei, Y., Chen, S., Fan, L., Liu, Y.: Advanced evasion attacks and mitigations on practical ml-based phishing website classifiers. Int. J. Intell. Syst. **36**(9), 5210–5240 (2021)
57. Tian, Y., Pei, K., Jana, S., Ray, B.: DeepTest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering, pp. 303–314 (2018)
58. Tran, H.-D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using ImageStars. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 18–42. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_2
59. Tran, H., et al.: Star-based reachability analysis of deep neural networks. In: Proceedings of the 3rd World Congress on Formal Methods, pp. 670–686 (2019)
60. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of the 27th USENIX Security Symposium, pp. 1599–1614 (2018)
61. WikiChip: FSD chip - tesla. https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip. Accessed 30 Apr 2022
62. Yang, P., et al.: Improving neural network verification through spurious region guided refinement. In: Groote, J.F., Larsen, K.G. (eds.) Proceedings of 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 389–408 (2021)
63. Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine learning testing: survey, landscapes and horizons. IEEE Trans. Software Eng. **48**(2), 1–36 (2022)
64. Zhang, Y., Song, F., Sun, J.: QEBVerif (2023). https://github.com/S3L-official/QEBVerif
65. Zhang, Y., Song, F., Sun, J.: QEBVerif: quantization error bound verification of neural networks. CoRR abs/2212.02781 (2023)

66. Zhang, Y., Zhao, Z., Chen, G., Song, F., Chen, T.: BDD4BNN: a BDD-based quantitative analysis framework for binarized neural networks. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 175–200. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_8
67. Zhang, Y., Zhao, Z., Chen, G., Song, F., Chen, T.: Precise quantitative analysis of binarized neural networks: a BDD-based approach. ACM Trans. Software Eng. Methodol. **32**(3) (2023)
68. Zhang, Y., et al.: QVIP: an ILP-based formal verification approach for quantized neural networks. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 82:1–82:13 (2023)
69. Zhao, Z., Chen, G., Wang, J., Yang, Y., Song, F., Sun, J.: Attack as defense: characterizing adversarial examples using robustness. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 42–55 (2021)
70. Zhao, Z., Zhang, Y., Chen, G., Song, F., Chen, T., Liu, J.: CLEVEREST: accelerating CEGAR-based neural network verification via adversarial attacks. In: Singh, G., Urban, C. (eds.) Proceedings of the 29th International Symposium on Static Analysis, pp. 449–473 (2022). https://doi.org/10.1007/978-3-031-22308-2_20

Verifying Generalization in Deep Learning

Guy Amir(✉), Osher Maayan, Tom Zelazny, Guy Katz, and Michael Schapira

The Hebrew University of Jerusalem, Jerusalem, Israel
{guyam,osherm,tomz,guykatz,schapiram}@cs.huji.ac.il

Abstract. Deep neural networks (DNNs) are the workhorses of deep learning, which constitutes the state of the art in numerous application domains. However, DNN-based decision rules are notoriously prone to poor *generalization*, i.e., may prove inadequate on inputs not encountered during training. This limitation poses a significant obstacle to employing deep learning for mission-critical tasks, and also in real-world environments that exhibit high variability. We propose a novel, verification-driven methodology for identifying DNN-based decision rules that generalize well to new input domains. Our approach quantifies generalization to an input domain by the extent to which decisions reached by *independently trained* DNNs are in agreement for inputs in this domain. We show how, by harnessing the power of DNN verification, our approach can be efficiently and effectively realized. We evaluate our verification-based approach on three deep reinforcement learning (DRL) benchmarks, including a system for Internet congestion control. Our results establish the usefulness of our approach. More broadly, our work puts forth a novel objective for formal verification, with the potential for mitigating the risks associated with deploying DNN-based systems in the wild.

1 Introduction

Over the past decade, deep learning [35] has achieved state-of-the-art results in natural language processing, image recognition, game playing, computational biology, and many additional fields [4,18,21,45,50,84,85]. However, despite its impressive success, deep learning still suffers from severe drawbacks that limit its applicability in domains that involve mission-critical tasks or highly variable inputs.

One such crucial limitation is the notorious difficulty of deep neural networks (DNNs) to *generalize* to new input domains, i.e., their tendency to perform poorly on inputs that significantly differ from those encountered while training. During training, a DNN is presented with input data sampled from a specific distribution over some input domain ("*in-distribution*" inputs). The induced DNN-based rules may fail in generalizing to inputs not encountered during training due to (1) the DNN being invoked "out-of-distribution" (OOD), i.e., when there is a mismatch between the distribution over inputs in the training data and in

G. Amir and O. Maayan—Contributed equally.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 438–455, 2023.
https://doi.org/10.1007/978-3-031-37703-7_21

the DNN's operational data; (2) some inputs not being sufficiently represented in the finite training data (e.g., various low-probability corner cases); and (3) "overfitting" the decision rule to the training data.

A notable example of the importance of establishing the generalizability of DNN-based decisions lies in recently proposed applications of deep reinforcement learning (DRL) [56] to real-world systems. Under DRL, an *agent*, realized as a DNN, is trained by repeatedly interacting with its environment to learn a decision-making *policy* that attains high performance with respect to a certain objective ("*reward*"). DRL has recently been applied to many real-world challenges [20,44,54,55,64–67,96,108]. In many application domains, the learned policy is expected to perform well across a daunting breadth of operational environments, whose diversity cannot possibly be captured in the training data. Further, the cost of erroneous decisions can be dire. Our discussion of DRL-based Internet congestion control (see Sect. 4.3) illustrates this point.

Here, we present a methodology for identifying DNN-based decision rules that generalize well to *all possible distributions* over an input domain of interest. Our approach hinges on the following key observation. DNN training in general, and DRL policy training in particular, incorporate multiple stochastic aspects, such as the initialization of the DNN's weights and the order in which inputs are observed during training. Consequently, even when DNNs with *the same* architecture are trained to perform an *identical* task on *the same* data, somewhat different decision rules will typically be learned. Paraphrasing Tolstoy's Anna Karenina [93], we argue that "successful decision rules are all alike; but every unsuccessful decision rule is unsuccessful in its own way". Differently put, when examining the decisions by several *independently trained* DNNs on a certain input, these are likely to agree only when their (similar) decisions yield high performance.

In light of the above, we propose the following heuristic for generating DNN-based decision rules that generalize well to *an entire* given domain of inputs: independently train multiple DNNs, and then seek a subset of these DNNs that are in strong agreement across *all* possible inputs in the considered input domain (implying, by our hypothesis, that these DNNs' learned decision rules generalize well to all probability distributions over this domain). Our evaluation demonstrates (see Sect. 4) that this methodology is extremely powerful and enables distilling from a collection of decision rules the few that indeed generalize better to inputs within this domain. Since our heuristic seeks DNNs whose decisions are in agreement for *each and every* input in a specific domain, the decision rules reached this way achieve robustly high generalization across different possible distributions over inputs in this domain.

Since our methodology involves contrasting the outputs of different DNNs over possibly *infinite* input domains, using formal verification is natural. To this end, we build on recent advances in formal verification of DNNs [2,12,14,16,27,60,78,86,102]. DNN verification literature has focused on establishing the local adversarial robustness of DNNs, i.e., seeking small input perturbations that result in misclassification by the DNN [31,36,61]. Our approach broadens the applicability of DNN verification by demonstrating, for the first time (to the best of our knowledge), how it can also be used to identify DNN-based decision rules that generalize well. More specifically, we show how, for a given input domain, a DNN verifier can be utilized to assign a score to a DNN reflecting its level of agreement with other DNNs across the entire input domain. This enables iteratively pruning the set of candidate DNNs, eventually keeping only those in strong agreement, which tend to generalize well.

To evaluate our methodology, we focus on three popular DRL benchmarks: (i) *Cartpole*, which involves controlling a cart while balancing a pendulum; (ii) *Mountain Car*, which involves controlling a car that needs to escape a valley; and (iii) *Aurora*, an Internet congestion controller.

Aurora is a particularly compelling example for our approach. While Aurora is intended to tame network congestion across a vast diversity of real-world Internet environments, Aurora is trained only on synthetically generated data. Thus, to deploy Aurora in the real world, it is critical to ensure that its policy is sound for numerous scenarios not captured by its training inputs.

Our evaluation results show that, in all three settings, our verification-driven approach is successful at ranking DNN-based DRL policies according to their ability to generalize well to out-of-distribution inputs. Our experiments also demonstrate that formal verification is superior to gradient-based methods and predictive uncertainty methods. These results showcase the potential of our approach. Our code and benchmarks are publicly available as an artifact accompanying this work [8].

The rest of the paper is organized as follows. Section 2 contains background on DNNs, DRLs, and DNN verification. In Sect. 3 we present our verification-based methodology for identifying DNNs that successfully generalize to OOD inputs. We present our evaluation in Sect. 4. Related work is covered in Sect. 5, and we conclude in Sect. 6.

2 Background

Deep Neural Networks (DNNs) [35] are directed graphs that comprise several layers. Upon receiving an assignment of values to the nodes of its first (input) layer, the DNN propagates these values, layer by layer, until ultimately reaching the assignment of the final (output) layer. Computing the value for each node is performed according to the type of that node's layer. For example, in weighted-sum layers, the node's value is an affine combination of the values of the nodes in the preceding layer to which it is connected. In *rectified linear unit* (*ReLU*) layers, each node y computes the value $y = \text{ReLU}(x) = \max(x, 0)$, where x is a single node from the preceding layer. For additional details on DNNs and their

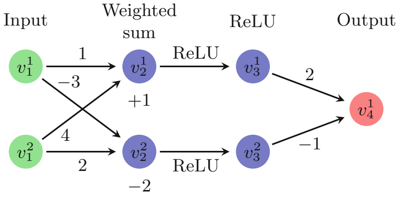


Fig. 1. A toy DNN.

training see [35]. Figure 1 depicts a toy DNN. For input $V_1 = [1, 2]^T$, the second layer computes the (weighted sum) $V_2 = [10, -1]^T$. The ReLU functions are subsequently applied in the third layer, and the result is $V_3 = [10, 0]^T$. Finally, the network's single output is $V_4 = [20]$.

Deep Reinforcement Learning (DRL) [56] is a machine learning paradigm, in which a DRL agent, implemented as a DNN, interacts with an *environment* across discrete time-steps $t \in 0, 1, 2....$ At each time-step, the agent is presented with the environment's *state* $s_t \in \mathcal{S}$, and selects an *action* $N(s_t) = a_t \in \mathcal{A}$. The environment then transitions to its next state s_{t+1}, and presents the agent with the *reward* r_t for its previous action. The agent is trained through repeated interactions with its environment to maximize the *expected cumulative discounted reward* $R_t = \mathbb{E}\big[\sum_t \gamma^t \cdot r_t\big]$ (where $\gamma \in \big[0, 1\big]$ is termed the *discount factor*) [38, 82, 90, 91, 97, 107].

DNN and DRL Verification. A sound DNN verifier [46] receives as input (i) a *trained* DNN N; (ii) a precondition P on the DNN's inputs, limiting the possible assignments to a domain of interest; and (iii) a postcondition Q on the DNN's outputs, limiting the possible outputs of the DNN. The verifier can reply in one of two ways: (i) `SAT`, with a concrete input x' for which $P(x') \wedge Q(N(x'))$ is satisfied; or (ii) `UNSAT`, indicating there does not exist such an x'. Typically, Q encodes the *negation* of N's desirable behavior for inputs that satisfy P. Thus, a `SAT` result indicates that the DNN errs, and that x' triggers a bug; whereas an `UNSAT` result indicates that the DNN performs as intended. An example of this process appears in Appendix B of our extended paper [7]. To date, a plethora of verification approaches have been proposed for general, feed-forward DNNs [3, 31, 41, 46, 61, 99], as well as DRL-based agents that operate within reactive environments [5, 9, 15, 22, 28].

3 Quantifying Generalizability via Verification

Our approach for assessing how well a DNN is expected to generalize on out-of-distribution inputs relies on the "Karenina hypothesis": while there are many (possibly infinite) ways to produce *incorrect results*, correct outputs are likely to be fairly similar. Hence, to identify DNN-based decision rules that generalize well to new input domains, we advocate training multiple DNNs and scoring the learned decision models according to how well their outputs are aligned with those of the other models for the considered input domain. These scores can be computed using a backend DNN verifier. We show how, by iteratively filtering out models that tend to disagree with the rest, DNNs that generalize well can be effectively distilled.

We begin by introducing the following definitions for reasoning about the extent to which two DNN-based decision rules are in agreement over an input domain.

Definition 1 (Distance Function). *Let $\mathcal{O}$ be the space of possible outputs for a DNN. A* distance function *for $\mathcal{O}$ is a function $d : \mathcal{O} \times \mathcal{O} \mapsto \mathbb{R}^+$.*

Intuitively, a distance function (e.g., the L_1 norm) allows us to quantify the level of (dis)agreement between the decisions of two DNNs on the same input. We elaborate on some choices of distance functions that may be appropriate in various domains in Appendix B of our extended paper [7].

Definition 2 (Pairwise Disagreement Threshold). *Let N_1, N_2 be DNNs with the same output space $\mathcal{O}$, let d be a distance function, and let Ψ be an input domain. We define the* pairwise disagreement threshold *(PDT) of N_1 and N_2 as:*

$$\alpha = PDT_{d,\Psi}(N_1, N_2) \triangleq \min\left\{\alpha' \in \mathbb{R}^+ \mid \forall x \in \Psi\colon d(N_1(x), N_2(x)) \leq \alpha'\right\}$$

The definition captures the notion that for *any* input in Ψ, N_1 and N_2 produce outputs that are at most α-distance apart. A small α value indicates that the outputs of N_1 and N_2 are close for all inputs in Ψ, whereas a high value indicates that there exists an input in Ψ for which the decision models diverge significantly.

To compute PDT values, our approach employs verification to conduct a binary search for the maximum distance between the outputs of two DNNs; see Algorithm 1.

Algorithm 1. Pairwise Disagreement Threshold

Input: DNNs $(N_i,\ N_j)$, distance func. d, input domain Ψ, max. disagreement $\texttt{M} > 0$
Output: PDT(N_i, N_j)

```
low ← 0, high ← M
while (low < high) do
    $\alpha \leftarrow \frac{1}{2} \cdot (\texttt{low} + \texttt{high})$
    query ← SMT SOLVER $\langle P \leftarrow \Psi, [N_i; N_j], Q \leftarrow d(N_i, N_j) \geq \alpha\rangle$
    if query is SAT then: low ← $\alpha$
    else if query is UNSAT then: high ← $\alpha$
end while
return $\alpha$
```

Pairwise disagreement thresholds can be aggregated to measure the disagreement between a decision model and a *set* of other decision models, as defined next.

Definition 3 (Disagreement Score). *Let $\mathcal{N} = \{N_1, N_2, \ldots, N_k\}$ be a set of k DNN-induced decision models, let d be a distance function, and let Ψ be an input domain. A model's* disagreement score *(DS) with respect to $\mathcal{N}$ is defined as:*

$$DS_{\mathcal{N},d,\Psi}(N_i) = \frac{1}{|\mathcal{N}| - 1} \sum_{j \in [k], j \neq i} PDT_{d,\Psi}(N_i, N_j)$$

Intuitively, the disagreement score measures how much a single decision model tends to disagree with the remaining models, on average.

Using disagreement scores, our heuristic employs an iterative scheme for selecting a subset of models that generalize to OOD scenarios—as encoded by inputs in Ψ (see Algorithm 2). First, a set of k DNNs $\{N_1, N_2, \ldots, N_k\}$ are *independently* trained on the training data. Next, a backend verifier is invoked to calculate, for each of the $\binom{k}{2}$ DNN-based model pairs, their respective pairwise-disagreement threshold (up to some ϵ accuracy). Next, our algorithm iteratively: (i) calculates the disagreement score for each model in the remaining subset of models; (ii) identifies the models with the (relative) highest DS scores; and (iii) removes them (Line 9 in Algorithm 2). The algorithm terminates after exceeding a user-defined number of iterations (Line 3 in Algorithm 2), or when the remaining models "agree" across the input domain, as indicated by nearly identical disagreement scores (Line 7 in Algorithm 2). We note that the algorithm is also given an upper bound ($\mathtt{M}$) on the maximum difference, informed by the user's domain-specific knowledge.

Algorithm 2. Model Selection

```
Input: Set of models $\mathcal{N} = \{N_1, \ldots, N_k\}$, max disagreement M, number of ITERATIONS
Output: $\mathcal{N}' \subseteq \mathcal{N}$
 1: PDT ← Pairwise Disagreement Thresholds($\mathcal{N}, d, \Psi$, M)   ▷ table with all PDTs
 2: $\mathcal{N}' \leftarrow \mathcal{N}$
 3: for $l = 1$...ITERATIONS do
 4:     for $N_i \in \mathcal{N}'$ do
 5:         currentDS[$N_i$] ← $DS_{\mathcal{N}'}(N_i$, PDT)   ▷ based on definition 3
 6:     end for
 7:     if modelScoresAreSimilar(currentDS) then: break
 8:     modelsToRemove ← findModelsWithHighestDS(currentDS)
 9:     $\mathcal{N}' \leftarrow \mathcal{N}' \setminus$ modelsToRemove   ▷ remove models that tend to disagree
10: end for
11: return $\mathcal{N}'$
```

DS Removal Threshold. Different criteria are possible for determining the DS threshold above for which models are removed, and how many models to remove in each iteration (Line 8 in Algorithm 2). A natural and simple approach, used in our evaluation, is to remove the p% models with the *highest* disagreement scores, for some choice of p (25% in our evaluation). Due to space constraints, a thorough discussion of additional filtering criteria (all of which proved successful) is relegated to Appendix C of our extended paper [7].

4 Evaluation

We extensively evaluated our method using three DRL benchmarks. As discussed in the introduction, verifying the generalizability of DRL-based systems is important since such systems are often expected to provide robustly high performance across a broad range of environments, whose diversity is not captured by the training data. Our evaluation spans two classic DRL settings, Cartpole [17] and Mountain Car [68], as well as the recently proposed Aurora congestion controller for Internet traffic [44]. Aurora is a particularly compelling example for a fairly complex DRL-based system that addresses a crucial real-world challenge and must generalize to real-world conditions not represented in its training data.

Setup. For each of the three DRL benchmarks, we first trained multiple DNNs with the same architecture, where the training process differed only in the random seed used. We then removed from this set of DNNs all but the ones that achieved high reward values in-distribution (to eliminate the possibility that a decision model generalizes poorly simply due to poor training). Next, we defined out-of-distribution input domains of interest for each specific benchmark, and used Algorithm 2 to select the models most likely to generalize well on those domains according to our framework. To establish the ground truth for how well different models actually generalize in practice, we then applied the models to OOD inputs drawn from the considered domain and ranked them based on their empirical performance (average reward). To investigate the robustness of our results, the last step was conducted for varying choices of probability distributions over the inputs in the domain. All DNNs used have a feed-forward architecture comprised of two hidden layers of ReLU activations, and include 32-64 neurons in the first hidden layer, and 16 neurons in the second hidden layer.

The results indicate that models selected by our approach are likely to perform *significantly better* than the rest. Below we describe the gist of our evaluation; extensive additional information is available in [7].

4.1 Cartpole

Cartpole [33] is a well-known RL benchmark in which an agent controls the movement of a cart with an upside-down pendulum ("pole") attached to its top. The cart moves on a platform and the agent's goal is to keep the pole balanced for as long as possible (see Fig. 2).

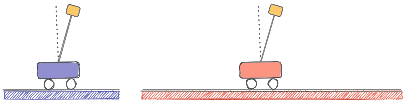


Fig. 2. Cartpole: in-distribution setting (blue) and OOD setting (red). (Color figure online)

Agent and Environment. The agent's inputs are $s = (x, v_x, \theta, v_\theta)$, where x represents the cart's location on the platform, θ represents the pole's angle (i.e., $|\theta| \approx 0$ for a balanced pole, $|\theta| \approx 90°$ for an unbalanced pole), v_x represents the cart's horizontal velocity and v_θ represents the pole's angular velocity.

In-Distribution Inputs. During training, the agent is incentivized to balance the pole, while staying within the platform's boundaries. In each iteration, the agent's single output indicates the cart's acceleration (sign and magnitude) for the next step. During training, we defined the platform's bounds to be $[-2.4, 2.4]$, and the cart's initial position as near-static, and close to the center of the platform (left-hand side of Fig. 2). This was achieved by drawing the cart's initial state vector values uniformly from the range $[-0.05, 0.05]$.

(OOD) Input Domain. We consider an input domain with larger platforms than the ones used in training. To wit, we now allow the x coordinate of the input vectors to cover a wider range of $[-10, 10]$. For the other inputs, we used the same bounds as during the training. See [7] for additional details.

Evaluation. We trained $k = 16$ models, all of which achieved high rewards during training on the short platform. Next, we ran Algorithm 2 until convergence (7 iterations, in our experiments) on the aforementioned input domain, resulting in a set of 3 models. We then tested all 16 original models using (OOD) inputs drawn from the new domain, such that the generated distribution encodes a novel setting: the cart is now placed at the center of a much longer, shifted platform (see the red cart in Fig. 2).

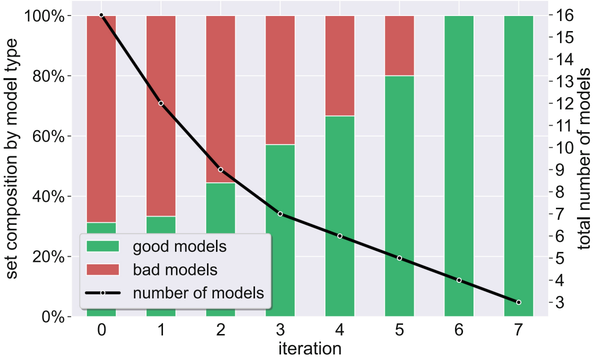


Fig. 3. Cartpole: Algorithm 2's results, per iteration: the bars reflect the ratio between the good/bad models (left y-axis) in the surviving set of models, and the curve indicates the number of surviving models (right y-axis).

All other parameters in the OOD environment were identical to those used for the original training. Figure 9 (in [7]) depicts the results of evaluating the models using 20,000 OOD instances. Of the original 16 models, 11 scored a low-to-mediocre average reward, indicating their poor ability to generalize to this new distribution. Only 5 models obtained high reward values, including the 3 models identified by Algorithm 2; thus implying that our method was able to effectively remove all 11 models that would have otherwise performed poorly in this OOD setting (see Fig. 3). For additional information, see [7].

4.2 Mountain Car

For our second experiment, we evaluated our method on the Mountain Car [79] benchmark, in which an agent controls a car that needs to learn how to escape a valley and reach a target. As in the Cartpole experiment, we selected a set of models that performed well in-distribution and applied our method to identify a subset of models that make similar decisions in a predefined input domain. We again generated OOD inputs (relative to the training) from within this domain, and observed that the models selected by our algorithm indeed generalize significantly better than their peers that were iteratively removed. Detailed information about this benchmark can be found in Appendix E of our extended paper [7].

4.3 Aurora Congestion Controller

In our third benchmark, we applied our method to a complex, real-world system that implements a policy for Internet congestion control. The goal of congestion control is to determine, for each traffic source in a communication network, the pace at which data packets should be sent into the network. Congestion control is a notoriously difficult and fundamental challenge in computer networking [59,69]; sending packets too fast might cause network congestion, leading to data loss and delays. Conversely, low sending rates might under-utilize available network bandwidth. *Aurora* [44] is a DRL-based congestion controller that is the subject of recent work on DRL verification [9,28]. In each time-step, an Aurora agent observes statistics regarding the network and decides the packet sending rate for the following time-step. For example, if the agent observes excellent network conditions (e.g., no packet loss), we expect it to increase the packet sending rate to better utilize the network. We note that Aurora handles a much harder task than classical RL benchmarks (e.g., Cartpole and Mountain Car): congestion controllers must react gracefully to various possible events based on nuanced signals, as reflected by Aurora's inputs. Here, unlike in the previous benchmarks, it is not straightforward to characterize the optimal policy.

Agent and Environment. Aurora's inputs are t vectors $v_1, \ldots, v_t$, representing observations from the t previous time-steps. The agent's single output value indicates the change in the packet sending rate over the next time-step. Each vector $v_i \in \mathbb{R}^3$ includes three distinct values, representing statistics that reflect the network's condition (see details in Appendix F of [7]). In line with previous work [9,28,44], we set $t = 10$ time-steps, making Aurora's inputs of size $3t = 30$. The reward function is a linear combination of the data sender's throughput, latency, and packet loss, as observed by the agent (see [44] for additional details).

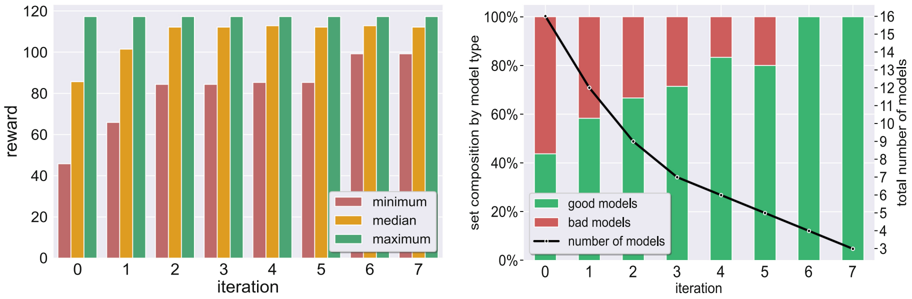
In-Distribution Inputs. Aurora's training applies the congestion controller to simple network scenarios where a *single* sender sends traffic towards a *single* receiver across a *single* network link. Aurora is trained across varying choices of initial sending rate, link bandwidth, link packet-loss rate, link latency, and size of the link's packet buffer. During training, packets are initially sent by Aurora at a rate corresponding to $0.3 - 1.5$ times the link's bandwidth.

(OOD) Input Domain. In our experiments, the input domain encoded a link with a *shallow packet buffer*, implying that only a few packets can accumulate in the network (while most excess traffic is discarded), causing the link to exhibit a volatile behavior. This is captured by the initial sending rate being up to 8 times the link's bandwidth, to model the possibility of a dramatic decrease in available bandwidth (e.g., due to competition, traffic shifts, etc.). See [7] for additional details.

Evaluation. We ran our algorithm and scored the models based on their disagreement upon this large domain, which includes inputs they had not encountered during training, representing the aforementioned novel link conditions.

Experiment (1): High Packet Loss. In this experiment, we trained over 100 Aurora agents in the original (in-distribution) environment. Out of these, we selected $k = 16$ agents that achieved a high average reward in-distribution (see Fig. 20a in [7]). Next, we evaluated these agents on OOD inputs that are included in the previously described domain. The main difference between the training distribution and the new (OOD) ones is the possibility of extreme packet loss rates upon initialization.

Our evaluation over the OOD inputs, within the domain, indicates that although all 16 models performed well in-distribution, only 7 agents could successfully handle such OOD inputs (see Fig. 20b in [7]). When we ran Algorithm 2 on the 16 models, it was able to filter out *all* 9 models that generalized poorly on the OOD inputs (see Fig. 4). In particular, our method returned model $\{16\}$, which is the best-performing model according to our simulations. We note that in the first iterations, the four models to be filtered out were models $\{1, 2, 6, 13\}$, which are indeed the four worst-performing models on the OOD inputs (see Appendix F of [7]).



(a) *Reward statistics of remaining models* (b) *Ratio between good/bad models*

Fig. 4. Aurora: Algorithm 2's results, per iteration.

Experiment (2): Additional Distributions over OOD Inputs. To further demonstrate that, in the specified input domain, our method is indeed likely to keep better-performing models while removing bad models, we reran the previous Aurora experiments for additional distributions (probability density functions) over the OOD inputs. Our evaluation reveals that all models removed by Algorithm 2 achieved low reward values also for these additional distributions. These results highlight an important advantage of our approach: it applies to all inputs within the considered domain, and so it applies to *all distributions over these inputs.*

Additional Experiments. We also generated a new set of Aurora models by altering the training process to include significantly longer interactions. We then repeated the aforementioned experiments. The results (summarized in [7]) demonstrate that our approach (again) successfully selected a subset of models that generalizes well to distributions over the OOD input domain.

4.4 Comparison to Additional Methods

Gradient-based methods [40,53,62,63] are optimization algorithms capable of finding DNN inputs that satisfy prescribed constraints, similarly to verification methods. These algorithms are extremely popular due to their simplicity and scalability. However, this comes at the cost of being inherently incomplete and not as precise as DNN verification [11,101]. Indeed, when modifying our algorithm to calculate PDT scores with gradient-based methods, the results (summarized in Appendix G of [7]) reveal that, in our context, the verification-based approach is superior to the gradient-based ones. Due to the incompleteness of gradient-based approaches [101], they often computed sub-optimal PDT values, resulting in models that generalize poorly being retained.

Predictive uncertainty methods [1,74] are *online* methods for assessing uncertainty with respect to observed inputs, to determine whether an encountered input is drawn from the training distribution. We ran an experiment comparing our approach to uncertainty-prediction-based model selection: we generated ensembles [23,30,51] of our original models, and used a variance-based metric (motivated by [58]) to identify subsets of models with low output variance on OOD-sampled inputs. Similar to gradient-based methods, predictive-uncertainty techniques proved fast and scalable, but lacked the precision afforded by verification-driven model selection and were unable to discard poorly generalizing models. For example, when ranking Cartpole models by their uncertainty on OOD inputs, the three models with the lowest uncertainty included also "bad" models, which had been filtered out by our approach.

5 Related Work

Recently, a plethora of approaches and tools have been put forth for ensuring DNN correctness [2,6,10,15,19,24–27,29,31,32,34,36,37,41–43,46–49,52, 57,61,70,76,81,83,86,87,89,92,94,95,98,100,102,104,106], including techniques for DNN shielding [60], optimization [14,88], quantitative verification [16], abstraction [12,13,73,78,86,105], size reduction [77], and more. Non-verification techniques, including runtime-monitoring [39], ensembles [71,72,80,103] and additional methods [75] have been utilized for OOD input detection.

In contrast to the above approaches, we aim to establish *generalization guarantees* with respect to an *entire input domain* (spanning all distributions across this domain). In addition, to the best of our knowledge, ours is the first attempt to exploit variability across models for distilling a subset thereof, with improved *generalization* capabilities. In particular, it is also the first approach to apply formal verification for this purpose.

6 Conclusion

This work describes a novel, verification-driven approach for identifying DNN models that generalize well to an input domain of interest. We presented an iterative scheme that employs a backend DNN verifier, allowing us to score models based on their ability to produce similar outputs on the given domain. We demonstrated extensively that this approach indeed distills models capable of good generalization. As DNN verification technology matures, our approach will become increasingly scalable, and also applicable to a wider variety of DNNs.

Acknowledgements. The work of Amir, Zelazny, and Katz was partially supported by the Israel Science Foundation (grant number 683/18). The work of Amir was supported by a scholarship from the Clore Israel Foundation. The work of Maayan and Schapira was partially supported by funding from Huawei.

References

1. Abdar, M., et al.: A review of uncertainty quantification in deep learning: techniques, applications and challenges. Inf. Fusion **76**, 243–297 (2021)
2. Alamdari, P., Avni, G., Henzinger, T., Lukina, A.: Formal methods with a touch of magic. In: Proceedings 20th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 138–147 (2020)
3. Albarghouthi, A.: Introduction to Neural Network Verification (2021). verifieddeeplearning.com
4. AlQuraishi, M.: AlphaFold at CASP13. Bioinformatics **35**(22), 4862–4865 (2019)
5. Amir, G., et al.: Verifying learning-based robotic navigation systems. In: Sankaranarayanan, S., Sharygina, N. (eds.) Proceedings 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 607–627. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_31
6. Amir, G., Freund, Z., Katz, G., Mandelbaum, E., Refaeli, I.: veriFIRE: verifying an industrial, learning-based wildfire detection system. In: Proceedings 25th International Symposium on Formal Methods (FM), pp. 648–656. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_38
7. Amir, G., Maayan, O., Zelazny, O., Katz, G., Schapira, M.: Verifying generalization in deep learning. Technical report (2023). https://arxiv.org/abs/2302.05745
8. Amir, G., Maayan, O., Zelazny, T., Katz, G., Schapira, M.: Verifying generalization in deep learning: artifact (2023). https://zenodo.org/record/7884514#.ZFAz_3ZBy3B
9. Amir, G., Schapira, M., Katz, G.: Towards scalable verification of deep reinforcement learning. In: Proceedings 21st Internationl Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 193–203 (2021)
10. Amir, G., Wu, H., Barrett, C., Katz, G.: An SMT-based approach for verifying binarized neural networks. In: TACAS 2021. LNCS, vol. 12652, pp. 203–222. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_11
11. Amir, G., Zelazny, T., Katz, G., Schapira, M.: Verification-aided deep ensemble selection. In: Proceedings 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 27–37 (2022)

12. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In: Proceedings 40th ACM SIGPLAN Conference on Programming Languages Design and Implementations (PLDI), pp. 731–744 (2019)
13. Ashok, P., Hashemi, V., Kretinsky, J., Mohr, S.: DeepAbstract: neural network abstraction for accelerating verification. In: Proceedings 18th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 92–107 (2020)
14. Avni, G., Bloem, R., Chatterjee, K., Henzinger, T.A., Könighofer, B., Pranger, S.: Run-time optimization for learned controllers through quantitative games. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 630–649. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_36
15. Bacci, E., Giacobbe, M., Parker, D.: Verifying reinforcement learning up to infinity. In: Proceedings 30th International Joint Conference on Artificial Intelligence (IJCAI) (2021)
16. Baluta, T., Shen, S., Shinde, S., Meel, K., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1249–1264 (2019)
17. Barto, A., Sutton, R., Anderson, C.: Neuronlike adaptive elements that can solve difficult learning control problems. In: Proceedings of IEEE Systems Man and Cybernetics Conference (SMC), pp. 834–846 (1983)
18. Bojarski, M., et al.: End to end learning for self-driving cars. Technical report (2016). http://arxiv.org/abs/1604.07316
19. Bunel, R., Turkaslan, I., Torr, P., Kohli, P., Mudigonda, P.: A unified view of piecewise linear neural network verification. In: Proceedings 32nd Conference on Neural Information Processing Systems (NeurIPS), pp. 4795–4804 (2018)
20. Chen, W., Xu, Y., Wu, X.: Deep reinforcement learning for multi-resource multi-machine job scheduling. Technical report (2017). http://arxiv.org/abs/1711.07440
21. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. J. Mach. Learn. Res. (JMLR) **12**, 2493–2537 (2011)
22. Corsi, D., Marchesini, E., Farinelli, A.: Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. In: Proceedings 37th Conference on Uncertainty in Artificial Intelligence (UAI), pp. 333–343 (2021)
23. Dietterich, T.: Ensemble methods in machine learning. In: Proceedings 1st International Workshop on Multiple Classifier Systems (MCS), pp. 1–15 (2020)
24. Dong, G., Sun, J., Wang, J., Wang, X., Dai, T.: Towards repairing neural networks correctly. Technical report (2020). http://arxiv.org/abs/2012.01872
25. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: Proceedings 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 157–168 (2019)
26. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Learning and verification of feedback control systems using feedforward neural networks. IFAC-PapersOnLine **51**(16), 151–156 (2018)
27. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Proceedings 15th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 269–286 (2017)

28. Eliyahu, T., Kazak, Y., Katz, G., Schapira, M.: Verifying learning-augmented systems. In: Proceedings Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pp. 305–318 (2021)
29. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. In: Proceedings 32nd AAAI Conference on Artificial Intelligence (AAAI) (2018)
30. Ganaie, M., Hu, M., Malik, A., Tanveer, M., Suganthan, P.: Ensemble deep learning: a review. Eng. Appl. Artif. Intell. **115**, 105151 (2022)
31. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, E., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings 39th IEEE Symposium on Security and Privacy (S&P) (2018)
32. Geng, C., Le, N., Xu, X., Wang, Z., Gurfinkel, A., Si, X.: Toward reliable neural specifications. Technical report (2022). https://arxiv.org/abs/2210.16114
33. Geva, S., Sitte, J.: A cartpole experiment benchmark for trainable controllers. IEEE Control Syst. Mag. **13**(5), 40–51 (1993)
34. Goldberger, B., Adi, Y., Keshet, J., Katz, G.: Minimal modifications of deep neural networks using verification. In: Proceedings 23rd Proceedings Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), pp. 260–278 (2020)
35. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
36. Gopinath, D., Katz, G., Păsăreanu, C., Barrett, C.: DeepSafe: a data-driven approach for assessing robustness of neural networks. In: Proceedings 16th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 3–19 (2018)
37. Goubault, E., Palumby, S., Putot, S., Rustenholz, L., Sankaranarayanan, S.: Static analysis of ReLU neural networks with tropical Polyhedra. In: Proceedings 28th International Symposium on Static Analysis (SAS), pp. 166–190 (2021)
38. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: Proceedings Conference on Machine Learning, pp. 1861–1870. PMLR (2018)
39. Hashemi, V., Křetínsky, J., Rieder, S., Schmidt, J.: Runtime monitoring for out-of-distribution detection in object detection neural networks. Technical report (2022). http://arxiv.org/abs/2212.07773
40. Huang, S., Papernot, N., Goodfellow, I., Duan, Y., Abbeel, P.: Adversarial attacks on neural network policies. Technical report (2017). https://arxiv.org/abs/1702.02284
41. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings 29th International Conference on Computer Aided Verification (CAV), pp. 3–29 (2017)
42. Isac, O., Barrett, C., Zhang, M., Katz, G.: Neural network verification with proof production. In: Proceedings 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 38–48 (2022)
43. Jacoby, Y., Barrett, C., Katz, G.: Verifying recurrent neural networks using invariant inference. In: Proceedings 18th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 57–74 (2020)
44. Jay, N., Rotman, N., Godfrey, B., Schapira, M., Tamar, A.: A deep reinforcement learning perspective on internet congestion control. In: Proceedings 36th International Conference on Machine Learning (ICML), pp. 3050–3059 (2019)

45. Julian, K., Lopez, J., Brush, J., Owen, M., Kochenderfer, M.: Policy compression for aircraft collision avoidance systems. In: Proceedings 35th Digital Avionics Systems Conference (DASC), pp. 1–10 (2016)
46. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Proceedings 29th International Conference on Computer Aided Verification (CAV), pp. 97–117 (2017)
47. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Reluplex: a calculus for reasoning about deep neural networks. Formal Methods Syst. Des. (FMSD) (2021)
48. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Proceedings 31st International Conference on Computer Aided Verification (CAV), pp. 443–452 (2019)
49. Könighofer, B., Lorber, F., Jansen, N., Bloem, R.: Shield synthesis for reinforcement learning. In: Proceedings International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), pp. 290–306 (2020)
50. Krizhevsky, A., Sutskever, I., Hinton, G.: ImageNet classification with deep convolutional neural networks. In: Proceedings 26th Conference on Neural Information Processing Systems (NeurIPS), pp. 1097–1105 (2012)
51. Krogh, A., Vedelsby, J.: Neural network ensembles, cross validation, and active learning. In: Proceedings 7th Conference on Neural Information Processing Systems (NeurIPS), pp. 231–238 (1994)
52. Kuper, L. Katz, G., Gottschlich, J., Julian, K., Barrett, C., Kochenderfer, M.: Toward scalable verification for safety-critical deep networks. Technical report (2018). https://arxiv.org/abs/1801.05950
53. Kurakin, A., Goodfellow, I., Bengio, S.: Adversarial examples in the physical world. Technical report (2016). http://arxiv.org/abs/1607.02533
54. Lekharu, A., Moulii, K., Sur, A., Sarkar, A.: Deep learning based prediction model for adaptive video streaming. In: Proceedings 12th International Conference on Communication Systems & Networks (COMSNETS), pp. 152–159. IEEE (2020)
55. Li, W., Zhou, F., Chowdhury, K.R., Meleis, W.: QTCP: adaptive congestion control with reinforcement learning. IEEE Trans. Netw. Sci. Eng. **6**(3), 445–458 (2018)
56. Li, Y.: Deep reinforcement learning: an overview. Technical report (2017). http://arxiv.org/abs/1701.07274
57. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward ReLU neural networks. Technical report (2017). http://arxiv.org/abs/1706.07351
58. Loquercio, A., Segu, M., Scaramuzza, D.: A general framework for uncertainty estimation in deep learning. In: Proceedings International Conference on Robotics and Automation (ICRA), pp. 3153–3160 (2020)
59. Low, S., Paganini, F., Doyle, J.: Internet congestion control. IEEE Control Syst. Mag. **22**(1), 28–43 (2002)
60. Lukina, A., Schilling, C., Henzinger, T.A.: Into the unknown: active monitoring of neural networks. In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 42–61. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_3
61. Lyu, Z., Ko, C.Y., Kong, Z., Wong, N., Lin, D., Daniel, L.: Fastened crown: tightened neural network robustness certificates. In: Proceedings 34th AAAI Conference on Artificial Intelligence (AAAI), pp. 5037–5044 (2020)
62. Ma, J., Ding, S., Mei, Q.: Towards more practical adversarial attacks on graph neural networks. In: Proceedings 34th Conference on Neural Information Processing Systems (NeurIPS) (2020)

63. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. Technical report (2017). http://arxiv.org/abs/1706.06083
64. Mammadli, R., Jannesari, A., Wolf, F.: Static neural compiler optimization via deep reinforcement learning. In: Proceedings 6th IEEE/ACM Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar), pp. 1–11 (2020)
65. Mao, H., Alizadeh, M., Menache, I., Kandula, S.: Resource management with deep reinforcement learning. In: Proceedings 15th ACM Workshop on Hot Topics in Networks (HotNets), pp. 50–56 (2016)
66. Mao, H., Netravali, R., Alizadeh, M.: Neural adaptive video streaming with Pensieve. In: Proceedings Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pp. 197–210 (2017)
67. Mnih, V., et al.: Playing Atari with deep reinforcement learning. Technical report (2013). https://arxiv.org/abs/1312.5602
68. Moore, A.: Efficient Memory-based Learning for Robot Control. University of Cambridge (1990)
69. Nagle, J.: Congestion control in IP/TCP internetworks. ACM SIGCOMM Comput. Commun. Rev. **14**(4), 11–17 (1984)
70. Okudono, T., Waga, M., Sekiyama, T., Hasuo, I.: Weighted automata extraction from recurrent neural networks via regression on state spaces. In: Proceedings 34th AAAI Conference on Artificial Intelligence (AAAI), pp. 5037–5044 (2020)
71. Ortega, L., Cabañas, R., Masegosa, A.: Diversity and generalization in neural network ensembles. In: Proceedings 25th International Conference on Artificial Intelligence and Statistics (AISTATS), pp. 11720–11743 (2022)
72. Osband, I., Aslanides, J., Cassirer, A.: Randomized prior functions for deep reinforcement learning. In: Proceedings 31st International Conference on Neural Information Processing Systems (NeurIPS), pp. 8617–8629 (2018)
73. Ostrovsky, M., Barrett, C., Katz, G.: An abstraction-refinement approach to verifying convolutional neural networks. In Proceedings 20th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 391–396 (2022)
74. Ovadia, Y., et al.: Can you trust your model's uncertainty? Evaluating predictive uncertainty under dataset shift. In: Proceedings 33rd Conference on Neural Information Processing Systems (NeurIPS), pp. 14003–14014 (2019)
75. Packer, C., Gao, K., Kos, J., Krähenbühl, P., Koltun, V., Song, D.: Assessing generalization in deep reinforcement learning. Technical report (2018). https://arxiv.org/abs/1810.12282
76. Polgreen, E., Abboud, R., Kroening, D.: Counterexample guided neural synthesis. Technical report (2020). https://arxiv.org/abs/2001.09245
77. Prabhakar, P.: Bisimulations for neural network reduction. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 285–300. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_14
78. Prabhakar, P., Afzal, Z.: Abstraction based output range analysis for neural networks. Technical report (2020). https://arxiv.org/abs/2007.09527
79. Riedmiller, M.: Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 317–328. Springer, Heidelberg (2005). https://doi.org/10.1007/11564096_32

80. Rotman, N., Schapira, M., Tamar, A.: Online safety assurance for deep reinforcement learning. In: Proceedings 19th ACM Workshop on Hot Topics in Networks (HotNets), pp. 88–95 (2020)
81. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: Proceedings 27th International Joint Conference on Artificial Intelligence (IJCAI) (2018)
82. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. Technical report (2017). http://arxiv.org/abs/1707.06347
83. Seshia, S., et al.: Formal specification for deep neural networks. In: Proceedings 16th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 20–34 (2018)
84. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
85. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. Technical report (2014). http://arxiv.org/abs/1409.1556
86. Singh, G., Gehr, T., Puschel, M., Vechev, M.: An abstract domain for certifying neural networks. In: Proceedings 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (2019)
87. Sotoudeh, M., Thakur, A.: Correcting deep neural networks with small, generalizing patches. In: Workshop on Safety and Robustness in Decision Making (2019)
88. Strong, C., et al.: Global optimization of objective functions represented by ReLU networks. J. Mach. Learn., 1–28 (2021)
89. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Proceedings 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC) (2019)
90. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press (2018)
91. Sutton, R., McAllester, D., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Proceedings 12th Conference on Neural Information Processing Systems (NeurIPS) (1999)
92. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. Technical report (2017). http://arxiv.org/abs/1711.07356
93. Tolstoy, L.: Anna Karenina. The Russian Messenger (1877)
94. Urban, C., Christakis, M., Wüstholz, V., Zhang, F.: Perfectly parallel fairness certification of neural networks. In: Proceedings ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 1–30 (2020)
95. Usman, M., Gopinath, D., Sun, Y., Noller, Y., Păsăreanu, C.: NNrepair: constraint-based repair of neural network classifiers. Technical report (2021). http://arxiv.org/abs/2103.12535
96. Valadarsky, A., Schapira, M., Shahaf, D., Tamar, A.: Learning to route with deep RL. In: NeurIPS Deep Reinforcement Learning Symposium (2017)
97. van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double Q-learning. In: Proceedings 30th AAAI Conference on Artificial Intelligence (AAAI) (2016)
98. Vasić, M., Petrović, A., Wang, K., Nikolić, M., Singh, R., Khurshid, S.: MoËT: mixture of expert trees and its application to verifiable reinforcement learning. Neural Netw. **151**, 34–47 (2022)

99. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings 27th USENIX Security Symposium, pp. 1599–1614 (2018)
100. Wu, H., et al.: Parallelization techniques for verifying neural networks. In: Proceedings 20th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 128–137 (2020)
101. Wu, H., Zeljić, A., Katz, K., Barrett, C.: Efficient neural network analysis with sum-of-infeasibilities. In: Proceedings 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 143–163 (2022)
102. Xiang, W., Tran, H., Johnson, T.: Output reachable set estimation and verification for multi-layer neural networks. IEEE Trans. Neural Netw. Learn. Syst. (TNNLS) (2018)
103. Yang, J., Zeng, X., Zhong, S., Wu, S.: Effective neural network ensemble approach for improving generalization performance. IEEE Trans. Neural Netw. Learn. Syst. (TNNLS) **24**(6), 878–887 (2013)
104. Yang, X., Yamaguchi, T., Tran, H., Hoxha, B., Johnson, T., Prokhorov, D.: Neural network repair with reachability analysis. In: Proceedings 20th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS), pp. 221–236 (2022)
105. Zelazny, T., Wu, H., Barrett, C., Katz, G.: On reducing over-approximation errors for neural network verification. In: Proceedings 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 17–26 (2022)
106. Zhang, H., Shinn, M., Gupta, A., Gurfinkel, A., Le, N., Narodytska, N.: Verification of recurrent neural networks for cognitive tasks via reachability analysis. In: Proceedings 24th European Conference on Artificial Intelligence (ECAI), pp. 1690–1697 (2020)
107. Zhang, J., Kim, J., O'Donoghue, B., Boyd, S.: Sample efficient reinforcement learning with REINFORCE. Technical report (2020). https://arxiv.org/abs/2010.11364
108. Zhang, J., et al.: An end-to-end automatic cloud database tuning system using deep reinforcement learning. In: Proceedings of the 2019 International Conference on Management of Data (SIGMOD), pp. 415–432 (2019)

Correction to: CoqCryptoLine: A Verified Model Checker with Certified Results

Ming-Hsien Tsai, Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi,
Bow-Yaw Wang, and Bo-Yin Yang

Correction to:
Chapter "CoqCryptoLine: A Verified Model Checker with Certified Results" in: C. Enea and A. Lal (Eds.): *Computer Aided Verification*, LNCS 13965, https://doi.org/10.1007/978-3-031-37703-7_11

The originally published version of chapter 11 contained mistakes in authors affiliation information and typographical errors in the article main text. The mistakes in authors affiliation information and typographical errors have been corrected.

The updated original version of this chapter can be found at
https://doi.org/10.1007/978-3-031-37703-7_11

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, p. C1, 2023.
https://doi.org/10.1007/978-3-031-37703-7_22

Author Index

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 457–460, 2023.
https://doi.org/10.1007/978-3-031-37703-7