Arie Gurfinkel
Vijay Ganesh (Eds.)

LNCS 14683

Computer Aided Verification

36th International Conference, CAV 2024
Montreal, QC, Canada, July 24–27, 2024
Proceedings, Part III

3 Part III

Springer

Lecture Notes in Computer Science 14683

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.

Arie Gurfinkel · Vijay Ganesh
Editors

Computer Aided Verification

36th International Conference, CAV 2024
Montreal, QC, Canada, July 24–27, 2024
Proceedings, Part III

Springer

Editors

Arie Gurfinkel
University of Waterloo
Waterloo, ON, Canada

Vijay Ganesh
Georgia Institute of Technology
Atlanta, GA, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-65632-3 ISBN 978-3-031-65633-0 (eBook)
https://doi.org/10.1007/978-3-031-65633-0

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

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

If disposing of this product, please recycle the paper.

Preface

It was our privilege to serve as the program chairs for CAV 2024, the 36th International Conference on Computer-Aided Verification. CAV 2024 was held in Montreal, Canada, on July 24–27, 2024, and the pre-conference workshops were held on July 22–23, 2024.

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. This year's program is a reflection of this commitment.

CAV 2024 received 317 submissions. We accepted 16 tool papers, 2 case-study papers, and 51 regular papers, which amounts to an acceptance rate of roughly 26% in each category. 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 Noriko Arai (National Institute of Informatics, Japan), Leonardo de Moura (Amazon Web Services, USA), and Erika Abraham (RWTH Aachen University, Germany). In addition to the contributed talks, CAV 2024 also hosted the CAV Award ceremony, and a report from the Synthesis Competition (SYNTCOMP) chairs. Furthermore, we continued the tradition of Logic Lounge, a series of discussions on computer science topics targeting a general audience. This year's Logic Lounge speaker was Scott J. Shapiro (Yale Law School) who spoke about topics at the intersection of formal methods and the law.

In addition to the main conference, CAV 2024 hosted the following workshops: Verification Mentoring Workshop (VMW), Correct Data Compression (CoDaC), Workshop on Synthesis (SYNT), Workshop on Verification of Probabilistic Programs (VeriProP), Developing an Open-Source, State-of-the-Art Symbolic Model-Checking Framework for the Model-Checking Research Community (OSSyM), Formal Reasoning in Distributed Algorithms (FRIDA), Workshop on Hyperproperties: Advances in Theory and Practice (HYPER), Symposium on AI Verification (SAIV), Deep Learning-aided Verification (DAV), and International Workshop on Satisfiability Modulo Theories (SMT).

Organizing a flagship conference like CAV requires a great deal of effort from the community. The Program Committee for CAV 2024 consisted of 90 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 900 reviews while investing significant effort to maintain and ensure the high quality of the conference program. We are grateful to the CAV 2024 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 54 artifact submissions, all of which received at least one badge. The Artifact Evaluation Committee consisted of 92 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 2024 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 2024 a success. We would like to thank Mirco Giacobbe and Milan Ceska for chairing the Artifact Evaluation Committee. We also thank Temegshen Kahsai for chairing the workshop organization. Norine Coenen and Hadar Frenkel for leading publicity efforts, Eric Koskinen and Grigory Fedyukovich as the fellowship chairs, Grigory Fedyukovich as sponsorship chair, and John (Zhengyang) Lu as the website chair. Hari Govind V. K. helped prepare the proceedings. We also thank Grigory Fedyukovich, Eric Koskinen, Umang Mathur, Yoni Zohar, and Jingbo Wang for organizing the Verification 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 2024.

We hope that you will find the proceedings of CAV 2024 scientifically interesting and thought-provoking!

June 2024

Arie Gurfinkel
Vijay Ganesh

Organization

Steering Committee

Aarti Gupta	Princeton University
Daniel Kroening	University of Oxford
Kenneth McMillan	University of Texas at Austin
Ornal Grumberg	Technion

Conference Co-chairs

Arie Gurfinkel	University of Waterloo
Vijay Ganesh	Georgia Institute of Technology

Artifact Evaluation Co-chairs

Mirco Giacobbe	University of Birmingham
Milan Ceska	Brno University of Technology

Local Chair

Xujie Si	University of Toronto

Area Chairs

Alexandra Silva	Cornell University
Anthony Widjaja Lin	Technical University of Kaiserslautern
Borzoo Bonakdarpour	Michigan State University
Corina Pasareanu	NASA
Kristin Yvonne Rozier	Iowa State University
Laura Kovacs	TU Wien

Workshop Chair

Temesghen Kahsai	Amazon

Fellowship Chairs

Grigory Fedyukovich	Florida State University
Eric Koskinen	Stevens Institute of Technology

Publicity Chairs

Norine Coenen	CISPA Helmholtz Center for Information Security
Hadar Frenkel	CISPA Helmholtz Center for Information Security

Publication Chair

Hari Govind V. K.	University of Waterloo

Website Chair

John (Zhengyang) Lu	University of Waterloo

Program Committee

Aditya Thakur	University of California, Davis
Ahmed Bouajjani	IRIF
Aina Niemetz	Stanford University
Akash Lal	Microsoft Research
Alan Hu	University of British Columbia
Alessandro Cimatti	Fondazione Bruno Kessler
Alexander Nadel	Technion & Intel
Alexandra Silva	Cornell University
Amir Goharshady	Hong Kong University of Science and Technology
Anastasia Mavridou	KBR Inc.
Andrew Reynolds	University of Iowa
Anna Slobodova	Intel

Anthony Widjaja Lin	Technical University of Kaiserslautern
Azadeh Farzan	University of Toronto
B. Srivathsan	Chennai Mathematical Institute
Benjamin Kaminski	Saarland University
Bernd Finkbeiner	CISPA Helmholtz Center for Information Security
Bettina Könighofer	Graz University of Technology
Bor-Yuh Evan Chang	University of Colorado
Borzoo Bonakdarpour	Michigan State University
Caterina Urban	Inria
Cezara Dragoi	Inria
Christopher Hahn	Google
Constantin Enea	Ecole Polytechnique
Corina Pasareanu	NASA
Deepak D'Souza	Indian Institute of Science
Dejan Jovanović	Amazon
Elizabeth Polgreen	University of Edinburgh
Elvira Albert	Universidad Complutense de Madrid
Erika Abraham	RWTH Aachen University
Eunsuk Kang	Carnegie Mellon University
Florin Manea	University of Göttingen
Gagandeep Singh	University of Illinois Urbana-Champaign
Grigory Fedyukovich	Florida State University
Guy Amir	Hebrew University of Jerusalem
Hadar Frenkel	CISPA Helmholtz Center for Information Security
Hongce Zhang	Hong Kong University of Science and Technology, China
Ichiro Hasuo	National Institute of Informatics
Isil Dillig	University of Texas at Austin
Jana Hofmann	Azure Research, Microsoft
Jianwen Li	East China Normal University
Jingbo Wang	University of Southern California
Jorge A. Navas	Certora
Ken McMillan	University of Texas at Austin
Kristin Yvonne Rozier	Iowa State University
Kshitij Bansal	Google
Kuldeep Meel	University of Toronto
Kumar Madhukar	Indian Institute of Technology Delhi
Laura Kovacs	TU Wien
Liana Hadarean	Amazon
Loris D'Antoni	University of Wisconsin-Madison
Mathias Preiner	Stanford University
Matthias Heizmann	University of Freiburg

Mihaela Sighireanu	Université Paris-Saclay
Mirco Giacobbe	University of Birmingham
Naijun Zhan	Chinese Academy of Sciences
Natasha Sharygina	University of Lugano
Nathalie Sznajder	Sorbonne Université
Nikolaj Bjørner	Microsoft Research
Ning Luo	Northwestern University
Oded Padon	VMware Research
Orna Grumberg	Technion
Pascal Fontaine	Université de Liège
Peter Schrammel	University of Sussex
Qirun Zhang	Georgia Institute of Technology
Ranjit Jhala	University of California, San Diego
Ravi Mangal	Carnegie Mellon University
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Rohit Dureja	Advanced Micro Devices, Inc.
Roland Yap	National University of Singapore
Rose Bohrer	Worcester Polytechnic Institute
Ruzica Piskac	Yale University
S. Akshay	Indian Institute of Technology Bombay
Sebastian Junges	Radboud University
Serdar Tasiran	Amazon
Sharon Shoham	Tel Aviv University
Shuvendu Lahiri	Microsoft Research
Sorav Bansal	Indian Institute of Technology Delhi
Sriram Sankaranarayanan	University of Colorado Boulder
Subhajit Roy	Indian Institute of Technology Kanpur
Subodh Sharma	Indian Institute of Technology Delhi
Suguman Bansal	Georgia Institute of Technology
Supratik Chakraborty	Indian Institute of Technology Bombay
Temesghen Kahsai	Amazon
Umang Mathur	National University of Singapore
Xujie Si	University of Toronto
Yakir Vizel	Technion
Yann Thierry-Mieg	LIP6
Yu-Fang Chen	Academia Sinica
Zvonimir Rakamaric	Amazon

Artifact Evaluation Committee

Abhinandan Pal	University of Birmingham
Adwait Godbole	UC Berkeley
Akshatha Shenoy	Tata Consultancy Services Ltd.
Alejandro Hernández-Cerezo	Complutense University of Madrid
Alvin George	IISc Bangalore
Ameer Hamza	Florida State University
Andreas Katis	KBR Inc. at NASA Ames Research Center
Anna Becchi	Fondazione Bruno Kessler
Benjamin Mikek	Georgia Institute of Technology
Bohua Zhan	Institute of Software, Chinese Academy of Sciences
Chenyu Zhou	University of Southern California
Daniel Dietsch	University Freiburg
Daniel Riley	Florida State University
Diptarko Roy	University of Oxford
Edoardo Manino	University of Manchester
Ennio Visconti	TU Wien
Enrico Magnago	Amazon Web Services
Filip Cano	Graz University of Technology
Filip Macák	Brno University of Technology
Florian Renkin	IRIF
Francesco Parolini	Sorbonne Université
Francesco Pontiggia	TU Wien
Gianluca Redondi	Fondazione Bruno Kessler
Giulio Garbi	University of Molise
Haoze Wu	Stanford University
Jacqueline Mitchell	University of Southern California
Jialuo Chen	Zhejiang University
Jie An	National Institute of Informatics
Jiong Yang	National University of Singapore
Julia Klein	University of Konstanz
Kartik Nagar	IIT Madras
Kaushik Mallik	Institute of Science and Technology Austria
Kazuki Watanabe	National Institute of Informatics, Tokyo
Kevin Cheang	Amazon Web Services
Konstantin Kueffner	Institute of Science and Technology Austria
Lelio Brun	National Institute of Informatics
Lorenz Leutgeb	Max Planck Institute for Informatics
Luca Arnaboldi	University of Birmingham
Lucas Zavalia	Florida State University

Malinda Dilhara	University of Colorado Boulder
Marcel Moosbrugger	TU Wien
Marck van der Vegt	Radboud University
Marco Casadio	Heriot-Watt University
Marco Lewis	Newcastle University
Marek Chalupa	Institute of Science and Technology Austria
Mário Pereira	NOVA University Lisbon
Marius Mikučionis	Aalborg University
Mathias Fleury	University of Freiburg
Matteo Marescotti	Meta Platforms
Matthias Schlaipfer	Amazon Web Services
Maximilian Weininger	Institute of Science and Technology Austria
Mertcan Temel	Intel Corporation
Mihir Mehta	University of Texas at Austin
N. Ege Saraç	Institute of Science and Technology Austria
Natasha Jeppu	Amazon Web Services
Neea Rusch	Augusta University
Neta Elad	Tel Aviv University
Nham Le	University of Waterloo
Oliver Markgraf	Max Planck Institute Kaiserslautern
Omar Inverso	Gran Sasso Science Institute
Omri Isac	Hebrew University of Jerusalem
Oyendrila Dobe	Michigan State University
P. Habeeb	Indian Institute of Science
Patrick Trentin	Amazon Web Services
Philippe Heim	CISPA Helmholtz Center for Information Security
Po-Chun Chien	LMU Munich
Ranadeep Biswas	Informal Systems
Remi Desmartin	Heriot-Watt University
Roman Andriushchenko	Brno University of Technology
Samuel Pastva	Institute of Science and Technology Austria
Sayan Mukherjee	Université libre de Bruxelles
Shengping Xiao	East China Normal University
Shubham Ugare	University of Illinois Urbana-Champaign
Shufang Zhu	University of Oxford
Shuo Ding	Georgia Institute of Technology
Siddharth Priya	University of Waterloo
Sidi Mohamed Beillahi	University of Toronto
Stefan Pranger	Graz University of Technology
Tobias Meggendorfer	Lancaster University Leipzig
Tobias Winkler	RWTH Aachen University
Tzu-Han Hsu	Michigan State University

Wael-Amine Boutglay	Université Paris Cité and Mohammed VI Polytechnic University
Xidan Song	University of Manchester
Xindi Zhang	Institute of Software, Chinese Academy of Sciences
Xiyue Zhang	University of Oxford
Yannan Li	Oracle
Yannik Schnitzer	University of Oxford
Yizhak Elboher	Hebrew University of Jerusalem
Yuzhou Fang	University of Southern California
Zhe Tao	University of California, Davis
Zhendong Ang	National University of Singapore
Zhiwei Zhang	Rice University

Additional Reviewers

Albarghouthi, Aws
Amarilli, Antoine
Ang, Zhendong
Antal, László
Banerjee, Subarno
Batz, Kevin
Becchi, Anna
Ben Shimon, Yoav
Biagiola, Matteo
Blicha, Martin
Bossut, Camille
Britikov, Konstantin
Campion, Marco
De Palma, Alessandro
Ding, Shuo
Dobe, Oyendrila
Eeralla, Ajay
Elad, Neta
Elboher, Yizhak
Emmi, Michael
Frenkel, Eden
Georgiou, Pamina
Gerlach, Carolina
Gürtler, Tobias
Hartmanns, Arnd
Hoad, Stuart
Hong, Chih-Duo
Hsu, Tzu-Han
Hunt, Warren
Hyvärinen, Antti
Ivrii, Alexander
Karmarkar, Hrishikesh
Koll, Charles
Labbaf, Faezeh
Lester, Martin Mariusz
Lotan, Raz
Luo, Ziyan
Magnago, Enrico
Metta, Ravindra
Metzger, Niklas
Mikek, Benjamin
Moosbrugger, Marcel
Morris, Jason
Mover, Sergio
Mukhopadhyay, Diganta
Nalbach, Jasper
Otoni, Rodrigo
Pailoor, Shankara
Patterson, Zachary
Piskachev, Goran
Promies, Valentin
Quatmann, Tim
Rappoport, Omer
Ravitch, Tristan

Rawson, Michael
Ritzert, Martin
Saatcioglu, Goktug
Shenoy, Akshatha
Shetty, Abhishek
Shi, Zheng
Tarrach, Thorsten
Trivedi, Ashutosh
Tunç, Hünkar Can
Verscht, Lena
Visconti, Ennio
Winkler, Tobias
Zhang, Minjian
Kaivola, Roope
Kaufmann, Daniela
Kolárik, Tomáš
Le, Nham
Li, Yong
Lu, Zhengyang
Löding, Christof

Invited Talks

How to Solve Math Problems Without Talent

Noriko Arai

National Institute of Informatics, Japan

The desire to solve mathematical problems without inherent talent has been a longstanding aspiration of humanity since ancient times. In this lecture, we delve into the complexity theory of proofs, examining the relationship between talent and the cost of proof. Additionally, we discuss the possibilities and limitations of using a fusion of computational methods, including computer algebra and natural language processing, to solve mathematical problems with machines. Join us as we explore the frontier of machine-enabled mathematical problem-solving, reflecting on its potential and boundaries in fulfilling this age-old human ambition.

Bridging Formal Mathematics and Software Verification

Leonardo de Moura

Amazon Web Services, USA

This talk will explore the dual applications of Lean 4, the latest iteration of the Lean proof assistant and programming language, in advancing formal mathematics and software verification. We begin with an overview of its design and implementation. We will detail how Lean 4 enables the formalization of complex mathematical theories and proofs, thereby enhancing collaboration and reliability in mathematical research. This endeavor is supported by a philosophy that promotes decentralized innovation, empowering a diverse community of researchers, developers, and enthusiasts to collaboratively push the boundaries of mathematical practice. Simultaneously, we will discuss software verification applications using Lean 4 at AWS. By leveraging Lean's dual capabilities as both a proof assistant and a functional programming language, we achieve a cohesive approach to software development and verification. Additionally, the talk will outline future directions for Lean 4, including efforts to expand its user community, enhance user experience, and further integrate formal methods into both academic research and industrial applications.

The Art of SMT Solving

Erika Ábrahám

RWTH Aachen University, Germany

Satisfiability Modulo Theories (SMT) solving [3, 4, 9] is a technology for the fully automated solution of logical formulas. SMT solvers can be used as general-purpose off-the-shelf tools. Due to their impressive efficiency, they are nowadays frequently used in a wide variety of applications [2]. A typical application encodes real-world problems as logical formulas, whose solutions can be decoded to solutions of the original real-world problem.

Besides its unquestionable practical impact, SMT solving has another great merit: it inspired truly elegant ideas, which do not only enable the construction of efficient software tools, but provide also interesting theoretical insights.

For *propositional logic* where each formula has a finite number of Boolean variables, we could enumerate and check all possible variable assignments, but due to its bad average complexity, this exploration approach is not applicable in practice. Alternatively, the proof system of Boolean resolution can be applied, but the applicability of this method is also restricted to rather small problems. However, in the 90s, *SAT solvers* succeeded to become impressively powerful due to an elegant combination of these two methods, where the proof construction is guided by an exploration of the assignment space equipped with a smart look-ahead mechanism [5, 6, 10].

The effectivity of SAT solvers gave motivation to extend the scope of solver technologies to formulas of *quantifier-free first-order logic over different theories.* On the one hand, *eager SMT solving* approaches have been proposed for certain theories to transform their formulas to propositional logic and use SAT solving to check the result for satisfiability. On the other hand, *(full/less) lazy SMT solving* uses SAT solving to explore the Boolean structure of the formula, and employs theory solvers to check the consistency of Boolean assignments in the theory domains.

Recently, the idea of symbiotic combination of exploration and proof construction has been also generalized to theories, most notably quantifier-free real algebra [7], in the framework of the *model constructing satisfiability calculus (MCSAT)* [11]. In this approach, exploration-guided proof construction is designed to run *both* in the Boolean space and in the theory domain, simultaneously in a consistent manner.

Both the SAT and the MCSAT approaches are based on the generalization of "wrong guesses", made during exploration, into pieces of a proof, which are collected and used to synthesize a global proof during the solving process. While being one of the currently best approaches, for large or complex formulas, a large number of "proof pieces" cause high effort for their processing and restrict scalability.

Thus the question comes up whether there are also other ways to store such information in a more structured way, allowing a less costly processing. This idea is taken up by the *cylindrical algebraic covering* method [1, 8], developed for the satisfiability check of conjunctions of polynomial constraints.

In this talk we give an introduction to the mechanisms of SAT and SMT solving, discuss the above ideas, and illustrate the usage of SMT solvers on a few application examples.

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. Log. Algebraic Methods Program. **119**, 100633 (2021). https://doi.org/10.1016/j.jlamp.2020.100633
2. Ábrahám, E., Kovács, J., Remke, A.: SMT: something you must try. In: Herber, P., Wijs, A. (eds) iFM 2023. LNCS, vol. 14300, pp. 3–18. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-47705-8_1
3. Ábrahám, E., Kremer, G.: SMT solving for arithmetic theories: theory and tool support. In: Proceedings SYNASC 2017, pp. 1–8. IEEE (2017). https://doi.org/10.1109/SYNASC.2017.00009
4. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 26, pp. 825–885. IOS Press (2009)
5. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
6. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962). https://doi.org/10.1145/368273.368557
7. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 339–354. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_27
8. Kremer, G., Ábrahám, E., England, M., Davenport, J.H.: On the implementation of cylindrical algebraic coverings for satisfiability modulo theories solving. In: Proceedings SYNASC 2021, pp. 37–39. IEEE (2021). https://doi.org/10.1109/SYNASC54541.2021.00018
9. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-662-50497-0
10. Moskewicz, M., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings 38th Design Automation Conference (2001)
11. de Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_1

Contents – Part III

Synthesis and Repair

Learning

Cyberphysical and Hybrid Systems

Probabilistic Systems

Quantum Systems

Synthesis and Repair

Syntax-Guided Automated Program Repair for Hyperproperties

Raven Beutner[1](✉), Tzu-Han Hsu[2], Borzoo Bonakdarpour[2], and Bernd Finkbeiner[1]

[1] CISPA Helmholtz Center for Information Security,
Saarbrücken, Germany
{raven.beutner,finkbeiner}@cispa.de

[2] Michigan State University, East Lansing, MI, USA
{tzuhan,borzoo}@msu.edu

Abstract. We study the problem of automatically repairing infinite-state software programs w.r.t. temporal hyperproperties. As a first step, we present a repair approach for the temporal logic HyperLTL based on symbolic execution, constraint generation, and syntax-guided synthesis of repair expression (SyGuS). To improve the repair quality, we introduce the notation of a *transparent* repair that aims to find a patch that is as close as possible to the original program. As a practical realization, we develop an *iterative* repair approach. Here, we search for a sequence of repairs that are closer and closer to the original program's behavior. We implement our method in a prototype and report on encouraging experimental results using off-the-shelf SyGuS solvers.

1 Introduction

Hyperproperties and *program repair* are two popular topics within the formal methods community. Hyperproperties [14] relate multiple executions of a system and occur, e.g., in information-flow control [56], robustness [12], and concurrent data structures [10]. Traditionally, automated program repair (APR) [25,28] attempts to repair the *functional* behavior of a program. In this paper, we, for the first time, tackle the challenging combination of APR and hyperproperties: given an (infinite-state) software program $\mathbb{P}$ and a violated hyperproperty φ, repair $\mathbb{P}$ such that φ is satisfied.

As a motivating example, consider the data leak in the EDAS conference manager [1] (simplified in Fig. 1). The function `display` is given the current phase of the review process (`phase`), paper title (`title`), session (`session`), and acceptance decision (`decision`), and computes a string (`print`) that will be displayed to the author(s). As usual in a conference management system, the displayed string should not leak information other than the `title`, unless the review process has been concluded. We can specify this *non-interference* policy as a hyperproperty in HyperLTL [13] as follows:

$$\forall \pi_1. \forall \pi_2. \big(\mathtt{phase}_{\pi_1} \neq \texttt{"Done"} \land \mathtt{phase}_{\pi_2} \neq \texttt{"Done"} \land \mathtt{title}_{\pi_1} = \mathtt{title}_{\pi_2}\big) \rightarrow \bigcirc \big(\mathtt{print}_{\pi_1} = \mathtt{print}_{\pi_2}\big). \qquad (\varphi_{\mathsf{edas}})$$

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 3–26, 2024.
https://doi.org/10.1007/978-3-031-65633-0_1

That is, for any *two* execution traces π_1, π_2 of `display` that, initially (i.e., at the first `observe` statement in line 3), have not reached the `"Done"` phase (i.e., `phase` $\neq$ `"Done"`) and agree on the title, should, at the second `observe` in line 10, agree on the value of `print`. It is straightforward to observe that function `display` violates φ_{edas}. The code *implicitly* leaks the acceptance decision by printing the session iff the paper is accepted. A natural question to ask is whether it is possible to automatically *repair* the `display` function such that φ_{edas} is satisfied.

```
1  display(string phase, string title,
2   string session, string decision) {
3     observe
4     decision = decision
5     if (decision == "Accept") {
6        print = title + session
7     } else {
8        print = title
9     }
10    observe
11 }
```

Fig. 1. Information leak in EDAS conference management system.

Constraint-Based Repair for Hyperproperties. As a first contribution, we propose a constraint-based APR approach for HyperLTL. Similar to existing constraint-based APR methods for functional properties [46], we rely on fault localization to identify potential repair locations (e.g., line 4 of our example in Fig. 1). We then replace the repair locations with a fresh function symbol; use symbolic execution to explore symbolic paths of the program; and generate repair constraints on the inserted function symbols. We show that we can use the *syntax-guided synthesis* (SyGuS) framework [2] to express (and solve) the repair constraints for HyperLTL properties with an *arbitrary* quantifier prefix.

Many Solutions. The main challenge in APR for hyperproperties lies in the large number of possible repair patches; a problem that already exists when repairing against functional properties [50] but is even more amplified when targeting hyperproperties. Different from functional specification, hyperproperties do not reason about the concrete functional (trace-level) behavior of a program, and rather express abstract relations between multiple computation traces. For example, information-flow policies such as observational determinism [56] can be checked and applied to arbitrary programs, regardless of their functional behavior. In contrast to functional trace properties, we thus cannot partition the set of all program executions into “correct” executions (i.e., executions that already satisfy the trace property and should be preserved in the repair) and “incorrect” executions. Instead, we need to alter the *set* of all program executions such that the executions together satisfy the hyperproperty, leading to an even larger space of potential repairs. Moreover, within this large space, many repairs trivially satisfy the hyperproperty by severely changing the functional behavior of the program, which is usually not desirable.

In our concrete example, the φ_{edas} property implicitly reasons about the (in)dependence between `phase`, `title`, and `print` but does not impose *how* the (in)dependence is realized functionally. If we apply our basic SyGuS-based repair approach, i.e., search for *some* repair of line 4 that satisfies φ_{edas}, it will immediately return a trivial repair patch: `decision ="Reject"`. This repair

```
if (phase == "Done"){
   decision = decision
} else {
   decision = "Reject"
}
```

(a)

```
if ((phase == "Done") or (decision != "Accept")){
   decision = decision
} else {
   decision = "Reject"
}
```

(b)

Fig. 2. Repair candidates discovered by our iterative repair.

simply sets the `decision` to some string not equal to `"Accept"` (we use `"Reject"` here for easier presentation). While this certainly satisfies our information-flow requirement, it does not yield a desirable implementation of `display` because the session is never displayed.

Transparent Repair. To tackle this issue, we strengthen our repair constraints using the concept of *transparency* (borrowed from the runtime enforcement literature [45]). Intuitively, we search for a repair that not only satisfies the hyperproperty but preserves as much functional behavior of the original program as possible. We show that we can integrate this within our SyGuS-based repair constraints. In the extreme, *full transparency* states that a repair is only allowed to deviate from the original program's behavior if absolutely necessary, i.e., only when the original behavior is part of a violation of the hyperproperty.

Iterative Repair. In the setting of hyperproperties, full transparency is often not particularly useful. It strictly dictates what traces can be changed by a repair, potentially resulting in the absence of a repair (within a given search space). In other instances (including the EDAS example), many paths (in the EDAS example, *all* paths) take part in some violation of the hyperproperty, allowing the repair to intervene arbitrarily. We introduce a more practical repair methodology that follows the same objective as (full) transparency (i.e., preserve as much original program behavior as possible). Our method, which we call *iterative repair*, approximates the global search for an optimal repair by a step-wise search for repairs of increasing quality. Concretely, starting from some initial repair, we iteratively try to find repair patches that preserve *more* original program behavior than our previous repair candidate. We show that we can effectively encode this into SyGuS constraints, and existing off-the-shelf SyGuS solvers can handle the resulting queries in many challenging instances. Notably, while some APR approaches (for functional properties) also try to find repairs that are close to the original program, they often do so heuristically. In contrast, our iterative repair constraints *guarantee* that the repair candidates strictly improve in each iteration. See Sect. 7 for more discussion.

Iterative Repair in Action. Coming back to our initial EDAS example, we can use iterative repair to improve upon the naïve repair `decision = "Reject"`. When using our iterative encoding, we find the improved repair solution in Fig. 2a that (probably) best mirrors the intuition of a programmer (cf. [47]):

This repair patch only overwrites the decision in cases where the phase does not equal `"Done"`. In particular, note how our iterative repair finds the *explicit* dependence of `decision` on `phase` (in the form of a conditional) even though this is only specified *implicitly* in φ_{edas}. In a third iteration, we can find an even closer repair, displayed in Fig. 2b: This repair only changes the decision if the review process is not completed *and* the decision equals `"Accept"`.

Implementation. We implement our repair approach in a prototype named `HyRep` and evaluate `HyRep` on a set of repair instances, including k-safety properties from the literature and challenging information-flow requirements.

Structure. Section 2 presents basic preliminaries, including our simple programming language and the formal specification language for hyperproperties targeted by our repair. Section 3 introduces our basic SyGuS-based repair approach, and we discuss our transparent and iterative extensions in Sects. 4 and 5, respectively. We present our experimental evaluation in Sect. 6 and discuss related work in Sect. 7.

2 Preliminaries

Given a set Y, we write Y^* for the set of finite sequences over Y, Y^ω for the set of infinite sequences, and $Y^\star := Y^* \cup Y^\omega$ for the set of finite and infinite sequences. For $t \in Y^\star$, we define $|t| \in \mathbb{N} \cup \{\infty\}$ as the length of t.

Programs. Let X be a fixed set of program variables. We write $\mathcal{E}_\mathbb{Z}$ and $\mathcal{E}_\mathbb{B}$ for the set of all arithmetic (integer-valued) and Boolean expressions over X, respectively. We consider a simple (integer-valued) programming language

$$\mathbb{P}, \mathbb{Q} := \texttt{skip} \mid x = e \mid \texttt{if}(b, \mathbb{P}, \mathbb{Q}) \mid \texttt{while}(b, \mathbb{P}) \mid \mathbb{P} \fatsemi \mathbb{Q} \mid \texttt{observe}$$

where $x \in X$, $e \in \mathcal{E}_\mathbb{Z}$, and $b \in \mathcal{E}_\mathbb{B}$. Most statements behave as expected. Notably, our language includes a dedicated `observe` statement, which we will use to express *asynchronous* (hyper)properties [5,11,29]. Intuitively, each `observe` statement causes an observation in our temporal formula, and we skip over unobserved (intermediate) computation steps (see also [7]).

Semantics. Programs manipulate (integer-valued) stores $\sigma : X \to \mathbb{Z}$, and we define $\mathit{Stores} := \{\sigma \mid \sigma : X \to \mathbb{Z}\}$ as the set of all stores. Our (small-step) semantics operates on configurations $C = \langle \mathbb{P}, \sigma \rangle$, where $\mathbb{P}$ is a program and $\sigma \in \mathit{Stores}$. Reduction steps have the form $C \xrightarrow{\mu} C'$, where $\mu \in \mathit{Stores} \cup \{\epsilon\}$. Most program steps have the form $C \xrightarrow{\epsilon} C'$ and model a transition without observation. Every execution of an `observe` statement induces a transition $C \xrightarrow{\sigma} C'$, modeling a transition in which we observe the current store σ. Figure 3 depicts a selection of reduction rules. For a program $\mathbb{P}$ and store σ, there exists a unique *maximal* execution $\langle \mathbb{P}, \sigma \rangle \xrightarrow{\mu_1} \langle \mathbb{P}_1, \sigma_1 \rangle \xrightarrow{\mu_2} \langle \mathbb{P}_2, \sigma_2 \rangle \xrightarrow{\mu_3} \cdots$, where

$$\frac{}{\langle x = e, \sigma\rangle \xrightarrow{\epsilon} \langle \texttt{skip}, \sigma[x \mapsto [\![e]\!]_\sigma]\rangle} \qquad\qquad \frac{}{\langle \texttt{observe}, \sigma\rangle \xrightarrow{\sigma} \langle \texttt{skip}, \sigma\rangle}$$

$$\frac{[\![b]\!]_\sigma = \mathit{true}}{\langle \texttt{if}(b, \mathbb{P}, \mathbb{Q}), \sigma\rangle \xrightarrow{\epsilon} \langle \mathbb{P}, \sigma\rangle} \qquad \frac{}{\langle \texttt{skip} \fatsemi \mathbb{P}, \sigma\rangle \xrightarrow{\epsilon} \langle \mathbb{P}, \sigma\rangle} \qquad \frac{\langle \mathbb{P}, \sigma\rangle \xrightarrow{\mu} \langle \mathbb{P}', \sigma'\rangle}{\langle \mathbb{P} \fatsemi \mathbb{Q}, \sigma\rangle \xrightarrow{\mu} \langle \mathbb{P}' \fatsemi \mathbb{Q}, \sigma'\rangle}$$

Fig. 3. Selection of small-step reduction rules. We write $[\![e]\!]_\sigma \in \mathbb{Z}$ and $[\![b]\!]_\sigma \in \mathbb{B}$ for the value of expression e and b in store σ, respectively.

$\mu_1, \mu_2, \mu_3, \dots \in \mathit{Stores} \cup \{\epsilon\}$. Note that this execution can be finite or infinite. We define $\mathit{obs}(\mathbb{P}, \sigma) := \mu_1\mu_2\mu_3 \cdots \in \mathit{Stores}^\star$ as the (finite or infinite) *observation sequence* along this execution (obtained by removing all ϵs). We write $\mathit{Traces}(\mathbb{P}) := \{\mathit{obs}(\mathbb{P}, \sigma) \mid \sigma \in \mathit{Stores}\} \subseteq \mathit{Stores}^\star$ for the set of all traces generated by $\mathbb{P}$. We say a program $\mathbb{P}$ is *terminating*, if all its executions are finite.

Syntax-Guided Synthesis. A Syntax-Guided Synthesis (SyGuS) problem is a triple $\varXi = (\{\tilde{f}_1, \dots, \tilde{f}_n\}, \varrho, \{G_1, \dots, G_n\})$, where $\tilde{f}_1, \dots, \tilde{f}_n$ are function symbols, ϱ is an SMT constraint over the function symbols $\tilde{f}_1, \dots, \tilde{f}_n$, and $G_1, \dots, G_n$ are grammars [2]. A solution for $\varXi$ is a vector of terms $\boldsymbol{e} = (e_1, \dots, e_n)$ such that each e_i is generated by grammar G_i, and $\varrho[\tilde{f}_1/e_1, \dots, \tilde{f}_n/e_n]$ holds (i.e., we replace each function symbol $\tilde{f}_i$ with expression e_i).

Example 1. Consider the SyGuS problem $\varXi = (\{\tilde{f}\}, \varrho, \{G\})$, where

$$\varrho := \forall x, y.\, \tilde{f}(x, y) \geq x \land \tilde{f}(x, y) \geq y \land (\tilde{f}(x, y) = x \lor \tilde{f}(x, y) = y)$$

$$G := \begin{cases} I \to x \mid y \mid 0 \mid 1 \mid I + I \mid I - I \mid \mathit{ite}(B, I, I) \\ B \to B \land B \mid B \lor B \mid \neg B \mid I = I \mid I \leq I \mid I \geq I. \end{cases}$$

This SyGuS problem constrains $\tilde{f}$ to be the function that returns the maximum of its arguments, and the grammar admits arbitrary piece-wise linear functions. A possible solution to $\varXi$ would be $\tilde{f}(x, y) := \mathit{ite}(x \leq y, y, x)$. △

HyperLTL. As the basic specification language for hyperproperties, we use HyperLTL, an extension of LTL with explicit quantification over execution traces [13]. Let $\mathcal{V} = \{\pi_1, \dots, \pi_n\}$ be a set of *trace variables*. For a trace variable $\pi_j \in \mathcal{V}$, we define $X_{\pi_j} := \{x_{\pi_j} \mid x \in X\}$ as a set of indexed program variables and $\boldsymbol{X} := X_{\pi_1} \cup \dots \cup X_{\pi_n}$. We include predicates from an arbitrary first-order theory $\mathfrak{T}$ to reason about the infinite variable domains in programs (cf. [7]), and denote satisfaction in $\mathfrak{T}$ with $\models^{\mathfrak{T}}$. We write $\mathcal{F}_{\boldsymbol{X}}$ for the set of first-order predicates over variables $\boldsymbol{X}$. HyperLTL formulas are generated by the following grammar:

$$\varphi := \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \psi$$
$$\psi := \theta \mid \psi \land \psi \mid \psi \lor \psi \mid \bigcirc \psi \mid \psi \mathcal{W} \psi$$

where $\pi \in \mathcal{V}$, $\theta \in \mathcal{F}_{\boldsymbol{X}}$, and $\bigcirc$ and $\mathcal{W}$ are the *next* and *weak-until* operator, respectively. W.l.o.g., we assume that all variables in $\mathcal{V}$ occur in the prefix exactly once. We use the usual derived constants and connectives $\mathit{true}, \mathit{false}, \to$, and $\leftrightarrow$.

Remark 1. We only allow negation within the atomic predicates, effectively ensuring that the LTL-like body denotes a *safety property* [38]. The reason for this is simple: In our program semantics, we specifically allow for both infinite and *finite* executions. Our repair approach is thus applicable to reactive systems but also handles (classical) programs that terminate. By requiring that the body denotes a safety property, we can easily handle arbitrary combinations of finite and infinite executions. Note that our logic supports *arbitrary* quantifier alternations, so we can still express hyperliveness properties such as GNI. △

Let $\mathcal{T} \subseteq \mathit{Stores}^\star$ be a set of traces. For $t \in \mathcal{T}$ and $i < |t|$, we write $t(i)$ for the ith store in t. A trace assignment is a partial mapping $\Pi : \mathcal{V} \rightharpoonup \mathcal{T}$ from trace variables to traces. We write $\Pi_{(i)}$ for the assignment $\boldsymbol{X} \to \mathbb{Z}$ given by $\Pi_{(i)}(x_\pi) := \Pi(\pi)(i)(x)$, i.e., the value of x_π, is the value of x in the ith step on the trace bound to π. We define the semantics inductively as:

$$\begin{array}{lll}
\Pi, i \models_{\mathcal{T}} \psi & \text{if} & \exists \pi \in \mathcal{V}.\, |\Pi(\pi)| \leq i \\
\Pi, i \models_{\mathcal{T}} \theta & \text{if} & \Pi_{(i)} \models^{\mathfrak{T}} \theta \\
\Pi, i \models_{\mathcal{T}} \psi_1 \land \psi_2 & \text{if} & \Pi, i \models_{\mathcal{T}} \psi_1 \text{ and } \Pi, i \models_{\mathcal{T}} \psi_2 \\
\Pi, i \models_{\mathcal{T}} \psi_1 \lor \psi_2 & \text{if} & \Pi, i \models_{\mathcal{T}} \psi_1 \text{ or } \Pi, i \models_{\mathcal{T}} \psi_2 \\
\Pi, i \models_{\mathcal{T}} \bigcirc \psi & \text{if} & \Pi, i+1 \models_{\mathcal{T}} \psi \\
\Pi, i \models_{\mathcal{T}} \psi_1 \mathcal{W} \psi_2 & \text{if} & \big(\exists j \geq i.\, \Pi, j \models_{\mathcal{T}} \psi_2 \text{ and } \forall i \leq k < j.\, \Pi, k \models_{\mathcal{T}} \psi_1\big) \text{ or} \\
 & & \quad (\forall j \geq i.\, \Pi, j \models_{\mathcal{T}} \psi_1) \\
\Pi, i \models_{\mathcal{T}} \exists \pi.\, \varphi & \text{if} & \exists t \in \mathcal{T}.\, \Pi[\pi \mapsto t], i \models_{\mathcal{T}} \varphi \\
\Pi, i \models_{\mathcal{T}} \forall \pi.\, \varphi & \text{if} & \forall t \in \mathcal{T}.\, \Pi[\pi \mapsto t], i \models_{\mathcal{T}} \varphi
\end{array}$$

As we deal with safety formulas (cf. Remark 1), we let Π, i satisfy any formula ψ as soon as we have moved past the length of the shortest trace in Π (i.e., $\exists \pi \in \mathcal{V}.\, |\Pi(\pi)| \leq i$). A program $\mathbb{P}$ satisfies φ, written $\mathbb{P} \models \varphi$, if $\emptyset, 0 \models_{\mathit{Traces}(\mathbb{P})} \varphi$, where $\emptyset$ denotes the trace assignment with an empty domain.

NSA. A *nondeterministic safety automaton* (NSA) over alphabet Σ is a tuple $\mathcal{A} = (Q, Q_0, \delta)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, and $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation. A *run* of $\mathcal{A}$ on a word $u \in \Sigma^\star$ is a sequence $q_0 q_1 \cdots \in Q^\star$ such that $q_0 \in Q_0$ and for every $i < |u|$, $(q_i, u(i), q_{i+1}) \in \delta$. We write $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\star$ for the set of words on which $\mathcal{A}$ has *some* run.

3 Program Repair by Symbolic Execution

In our repair setting, we are given a pair $(\mathbb{P}, \varphi)$ such that $\mathbb{P} \not\models \varphi$, and try to construct a repaired program $\mathbb{Q}$ with $\mathbb{Q} \models \varphi$. In particular, we repair w.r.t. a *formal specification* instead of a set of input-output examples. The reason for this lies within the nature of the properties we want to repair against: When repairing against trace properties (i.e., functional specifications), it is often intuitive to write input-output examples that test a program's functional behavior.

$$\frac{}{\langle x = e, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathtt{skip}, \nu[x \mapsto [\![e]\!]_\nu], \alpha, \beta\rangle} \qquad \frac{}{\langle \mathtt{observe}, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathtt{skip}, \nu, \alpha, \beta \cdot \nu\rangle}$$

$$\frac{}{\langle \mathtt{skip} \fatsemi \mathbb{P}, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P}, \nu, \alpha, \beta\rangle} \qquad \frac{}{\langle \mathtt{if}(b, \mathbb{P}, \mathbb{Q}), \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P}, \nu, \alpha \wedge [\![b]\!]_\nu, \beta\rangle}$$

$$\frac{\langle \mathbb{P}, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P}', \nu', \alpha', \beta'\rangle}{\langle \mathbb{P} \fatsemi \mathbb{Q}, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P}' \fatsemi \mathbb{Q}, \nu', \alpha', \beta'\rangle} \qquad \frac{}{\langle \mathtt{if}(b, \mathbb{P}, \mathbb{Q}), \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{Q}, \nu, \alpha \wedge \neg[\![b]\!]_\nu, \beta\rangle}$$

$$\frac{}{\langle \mathtt{while}(b, \mathbb{P}), \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P} \fatsemi \mathtt{while}(b, \mathbb{P}), \nu, \alpha \wedge [\![b]\!]_\nu, \beta\rangle}$$

$$\frac{}{\langle \mathtt{while}(b, \mathbb{P}), \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathtt{skip}, \nu, \alpha \wedge \neg[\![b]\!]_\nu, \beta\rangle}$$

Fig. 4. Small-step reduction rules for symbolic execution.

In contrast, hyperproperties do not directly reason about concrete functional behavior but rather about the abstract relation between multiple computations. For example, information-flow properties such as non-interference can be applied to arbitrary programs; independent of the program's functional behavior. Perhaps counter-intuitively, in our hyper-setting, formal specifications are thus often easier to construct than input-output examples.

3.1 Symbolic Execution

The first step in our repair pipeline is the computation of a mathematical summary of (parts of) the program's executions using symbolic execution (SE) [37]. In SE, we execute the program using symbolic placeholders instead of concrete values for variables, and explore all symbolic paths of a program (recording conditions that a concrete store needs to satisfy to take any given branch). A *symbolic store* is a function $\nu : X \to \mathcal{E}_\mathbb{Z}$ that maps each variable to an expression, and we write $\mathit{SymStores} := \{\nu \mid \nu : X \to \mathcal{E}_\mathbb{Z}\}$ for the set of all symbolic stores. A *symbolic configuration* is then a tuple $\langle \mathbb{P}, \nu, \alpha, \beta\rangle$, where $\mathbb{P}$ is a program, $\nu \in \mathit{SymStores}$ is a symbolic store, $\alpha \in \mathcal{F}_X$ is a first-order formula over X that records which conditions the current path should satisfy (called the *path condition*), and $\beta \in \mathit{SymStores}^*$ is a sequence of symbolic stores recording the observations. For $e \in \mathcal{E}_\mathbb{Z}$ and $\nu \in \mathit{SymStores}$, we write $[\![e]\!]_\nu$ for the expression obtained by replacing each variable x in e with $\nu(x)$. For example, if $\nu = [x \mapsto x - 1, y \mapsto z * y]$, we have $[\![x + y]\!]_\nu = (x - 1) + (z * y)$. We give the symbolic execution relation $\xrightarrow{sym}$ in Fig. 4. We start the symbolic execution in symbolic store $\nu_0 := \big[x \mapsto x\big]_{x \in X}$ that maps each variable to itself, path condition $\alpha_0 := \mathit{true}$, and an empty observation sequence $\beta_0 := \epsilon$. Given a program $\mathbb{P}$, a symbolic execution is a *finite* sequence of symbolic configurations

$$\rho = \langle \mathbb{P}, \nu_0, \alpha_0, \beta_0\rangle \xrightarrow{sym} \langle \mathbb{P}_1, \nu_1, \alpha_1, \beta_1\rangle \xrightarrow{sym} \cdots \xrightarrow{sym} \langle \mathbb{P}_m, \nu_m, \alpha_m, \beta_m\rangle \qquad (1)$$

We say execution ρ is *maximal* if $\mathbb{P}_m = \mathtt{skip}$, i.e., we cannot perform any more execution steps. Given a symbolic execution ρ, we are interested in the

acc^{ψ}_{Δ}

For $\Delta : \{\pi_1, \dots, \pi_n\} \to \mathit{SymStores}^*$, define $acc^{\psi}_{\Delta} := \bigvee_{q \in Q_{0,\psi}} acc^{q,0}_{\Delta}$, where $acc^{q,i}_{\Delta} := \mathit{true}$ iff $\exists \pi.\, |\Delta(\pi)| \leq i$, and otherwise

$$\begin{aligned} acc^{q,i}_{\Delta} := \bigvee_{(q,\iota,q') \in \delta_\psi} \Big(& acc^{q',i+1}_{\Delta} \wedge \bigwedge_{\theta \in F \mid \theta \in \iota} \theta\Big[x_{\pi_j} / \big(\Delta(\pi_j)(i)(x)\big[y / y_{\pi_j}\big]\big)\Big] \wedge \\ & \bigwedge_{\theta \in F \mid \theta \notin \iota} \neg\theta\Big[x_{\pi_j} / \big(\Delta(\pi_j)(i)(x)\big[y / y_{\pi_j}\big]\big)\Big]\Big) \end{aligned}$$

Fig. 5. Encoding for acceptance of ψ.

path condition α_m (to ensure that we follow an actual program path), and the observation sequence β_m (to evaluate the HyperLTL property). We define a *symbolic path* as a pair in $\mathcal{F}_X \times \mathit{SymStores}^*$, recording the path condition and symbolic observation sequence. Each execution ρ of the form in (1), yields a symbolic path (α_m, β_m). We call the symbolic path (α_m, β_m) *maximal* if ρ is maximal, and *satisfiable* if α_m is satisfiable (i.e., some actual program execution can take a path summarized by ρ). We write $\mathit{SymPaths}(\mathbb{P}) \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$ for the set of all satisfiable symbolic paths of $\mathbb{P}$ and $\mathit{SymPaths}_{\mathit{max}}(\mathbb{P}) \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$ for the set of all satisfiable maximal symbolic paths.

Remark 2. An interesting class of programs are those that are terminating and where $\mathit{SymPaths}_{\mathit{max}}(\mathbb{P})$ is *finite*. This is either the case when the program is loop-free or has some upper bound on the number of loop executions (and thus control paths). Crucially, if $\mathit{SymPaths}_{\mathit{max}}(\mathbb{P})$ is finite, it provides a precise and complete mathematical summary of the program's executions. △

3.2 Symbolic Paths and Safety Automata

We can use symbolic paths to approximate the HyperLTL semantics by explicitly considering path combinations. Let $\varphi = \mathbb{Q}_1 \pi_1 \dots \mathbb{Q}_n \pi_n.\, \psi$ be a fixed HyperLTL formula, where $\mathbb{Q}_1, \dots, \mathbb{Q}_n \in \{\forall, \exists\}$ are quantifiers, and ψ is the LTL body of φ. Further, let $F \subseteq \mathcal{F}_{\boldsymbol{X}}$ be the *finite* set of predicates used in ψ. Due to our syntactic safety restriction on LTL formulas, we can construct an NSA $\mathcal{A}_\psi = (Q_\psi, Q_{0,\psi}, \delta_\psi)$ over alphabet 2^F accepting exactly the words that satisfy ψ [38].

Assume $\Delta : \{\pi_1, \dots, \pi_n\} \to \mathit{SymStores}^*$ is a function that assigns each path variable $\pi_1, \dots, \pi_n$ a symbolic observation sequence. We design a formula acc^{ψ}_{Δ}, which encodes that the symbolic observation sequences in Δ have an accepting prefix in $\mathcal{A}_\psi$, given in Fig. 5. The intermediate formula $acc^{q,i}_{\Delta}$ encodes that the observations in Δ have some run from state q in the ith step. For all steps i, longer than the shortest trace in Δ, we accept (i.e., $acc^{q,i}_{\Delta} := \mathit{true}$, similar to our HyperLTL semantics). Otherwise, we require some transition $(q, \iota, q') \in \delta_\psi$ such

$enc^{\varphi}_{\mathcal{P}}$

For $\mathcal{P} \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$, define $enc^{\varphi}_{\mathcal{P}} := enc^{\varphi}_{\mathcal{P},\emptyset}$, where

$$enc^{\psi}_{\mathcal{P},\Delta} := acc^{\psi}_{\Delta}$$

$$enc^{\exists \pi_j . \varphi'}_{\mathcal{P},\Delta} := \exists_{x_{\pi_j} \in X_{\pi_j}} x_{\pi_j} . \bigvee_{(\alpha,\beta)\in\mathcal{P}} \Big(\alpha[x/x_{\pi_j}] \wedge enc^{\varphi'}_{\mathcal{P},\Delta[\pi_j \mapsto \beta]}\Big)$$

$$enc^{\forall \pi_j . \varphi'}_{\mathcal{P},\Delta} := \forall_{x_{\pi_j} \in X_{\pi_j}} x_{\pi_j} . \bigwedge_{(\alpha,\beta)\in\mathcal{P}} \Big(\alpha[x/x_{\pi_j}] \rightarrow enc^{\varphi'}_{\mathcal{P},\Delta[\pi_j \mapsto \beta]}\Big)$$

Fig. 6. Encoding of the HyperLTL semantics on symbolic paths $\mathcal{P}$.

that $acc^{q',i+1}_{\Delta}$ holds, and the label $\iota \in 2^F$ holds in step i. To encode the latter, we use the symbolic observation sequences in Δ: For every predicate $\theta \in F$, we require that $\theta \in \iota$ iff $\theta\big[x_{\pi_j} / \big(\Delta(\pi_j)(i)(x)[y/y_{\pi_j}]\big)\big]$. That is, we replace variable x_{π_j} with the expression $\Delta(\pi_j)(i)(x)[y/y_{\pi_j}]$, i.e., we look up the expression bound to variable x in the ith step on $\Delta(\pi_j)$, and – within this expression – index all variables with π_j (i.e., replace each variable $y \in X$ with $y_{\pi_j} \in X_{\pi_j}$).

3.3 Encoding for HyperLTL

Let $\mathcal{P} \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$ be a finite set of symbolic paths and consider the formula $enc^{\varphi}_{\mathcal{P}}$ in Fig. 6. Intuitively, the formula encodes the satisfaction of φ on the symbolic paths in $\mathcal{P}$. For this, we maintain a *partial* mapping $\Delta : \{\pi_1, \dots, \pi_n\} \rightharpoonup \mathit{SymStores}^*$, and for each subformula φ' we define an intermediate formula $enc^{\varphi'}_{\mathcal{P},\Delta}$. If we reach the LTL body ψ, we define $enc^{\psi}_{\mathcal{P},\Delta} := acc^{\psi}_{\Delta}$, stating that the symbolic observation sequences in Δ satisfy ψ (cf. Fig. 5). Each trace quantifier is then resolved on the symbolic paths in $\mathcal{P}$. Concretely, for a subformula $\exists \pi_j . \varphi'$, we existentially quantify over variables X_{π_j} and *disjunctively* pick a symbolic path $(\alpha, \beta) \in \mathcal{P}$. We require that path condition α holds (after replacing each variable x with x_{π_j}), and that the remaining formula φ' is satisfied if we bind observation sequence β to π_j (i.e., $enc^{\varphi'}_{\mathcal{P},\Delta[\pi_j \mapsto \beta]}$).

Proposition 1. *If $\mathbb{Q}$ is a terminating program and $\mathit{SymPaths}_{max}(\mathbb{Q})$ is finite, then $\mathbb{Q} \models \varphi$ if and only if $enc^{\varphi}_{\mathit{SymPaths}_{max}(\mathbb{Q})}$.*

The above proposition essentially states that we can use SE to verify a program (with finitely-many symbolic paths) against HyperLTL formulas with *arbitrary* quantifier alternations. This is in sharp contrast to existing SE-based approaches, which only apply to k-safety properties (i.e., $\forall^*$ HyperLTL formulas) [17,18,22,51,52]. To the best of our knowledge, ours is the first approach that can check properties containing arbitrary alternations on fragments of infinite-state software programs. Previous methods either focus on finite-state systems [6,8,16,24,31,32] or only consider restricted quantifier structures [7,23,34,49,53].

Alternation-Free Formulas. In many situations, we cannot explore *all* symbolic paths of a program $\mathbb{Q}$ (i.e., $\mathit{SymPaths}_{max}(\mathbb{Q})$ is infinite). However, even by just exploring a subset of paths, our encoding still allows us to draw conclusions about the full program as long as the formula is *alternation-free.*

Proposition 2. *Assume φ is a $\exists^*$ formula and $\mathcal{P} \subseteq \mathit{SymPaths}_{max}(\mathbb{Q})$ is a finite set of maximal symbolic paths. If $\mathit{enc}^{\varphi}_{\mathcal{P}}$, then $\mathbb{Q} \models \varphi$.*

Proposition 3. *Assume φ is a $\forall^*$ formula and $\mathcal{P} \subseteq \mathit{SymPaths}(\mathbb{Q})$ is a finite set of (not necessarily maximal) symbolic paths. If $\neg \mathit{enc}^{\varphi}_{\mathcal{P}}$, then $\mathbb{Q} \not\models \varphi$.*

In particular, we can use Proposition 3 for our repair approach for $\forall^*$ properties (which captures many properties of interest, such as non-interference, cf. φ_{edas}). If we symbolically execute a program to some fixed depth (and thus capture a subset of the symbolic paths), any possible repair must satisfy the bounded property described in $\mathit{enc}^{\varphi}_{\mathcal{P}}$ (cf. Sect. 3.4). Note that this does not ensure that the repair patch that fulfills $\mathit{enc}^{\varphi}_{\mathcal{P}}$ is correct on the entire program; $\mathit{enc}^{\varphi}_{\mathcal{P}}$ merely describes a *necessary* condition any possible repair needs to satisfy. In our experiments (cf. Sect. 6), we (empirically) found that the repair for the bounded version also serves as a repair for the full program in many instances.

3.4 Program Repair Using SyGuS

Using SE and our encoding, we can now outline our basic SyGuS-based repair approach. Assume $\mathbb{P} \not\models \varphi$ is the program that should be repaired. As in other semantic-analysis-based repair frameworks [44,46], we begin our repair by predicting fault locations [54] within the program, i.e., locations that are likely to be responsible for the violation of φ. In our later experiments, we assume that these locations are provided by the user. After we have identified a set of n repair locations, we instrument $\mathbb{P}$ by replacing the expressions in all repair locations with fresh *function symbols.* That is, if we want to repair statement $x = e$, $\texttt{if}(b, \mathbb{P}_1, \mathbb{P}_2)$, or $\texttt{while}(b, \mathbb{P})$, we replace the statement with $x = \tilde{f}(x_1, \ldots, x_m)$, $\texttt{if}(\tilde{f}(x_1, \ldots, x_m), \mathbb{P}_1, \mathbb{P}_2)$, or $\texttt{while}(\tilde{f}(x_1, \ldots, x_m), \mathbb{P})$, respectively, for some fresh function symbol $\tilde{f}$ and program variables $x_1, \ldots, x_m \in X$ (inferred using a lightweight dependency analysis). Let $\mathbb{Q}$ be the resulting program, which contains function symbols, $\tilde{f}_1, \ldots, \tilde{f}_n$. We symbolically execute $\mathbb{Q}$, leading to a set of symbolic paths $\mathcal{P}$ containing $\tilde{f}_1, \ldots, \tilde{f}_n$, and define the SyGuS problem $\varXi_{\mathcal{P}} := (\{\tilde{f}_1, \ldots, \tilde{f}_n\}, \mathit{enc}^{\varphi}_{\mathcal{P}}, \{G_1, \ldots, G_n\})$. Here, we fix a grammar G_i for each function symbol $\tilde{f}_i$, based on the type and context of each repair location. Note that $\mathit{enc}^{\varphi}_{\mathcal{P}}$ now constitutes an SMT constraint over $\tilde{f}_1, \ldots, \tilde{f}_n$. Any solution for $\varXi_{\mathcal{P}}$ thus defines concrete expressions for $\tilde{f}_1, \ldots, \tilde{f}_n$ such that the symbolic paths in $\mathcal{P}$ satisfy φ. Concretely, let $\boldsymbol{e} = (e_1, \ldots, e_n)$ be a solution to $\varXi_{\mathcal{P}}$. Define $\mathbb{Q}[\boldsymbol{e}] := \mathbb{Q}[\tilde{f}_1/e_1, \ldots, \tilde{f}_n/e_n]$, i.e., we replace each function symbol $\tilde{f}_i$ by expression e_i. As $\boldsymbol{e}$ is a solution to $\varXi_{\mathcal{P}}$, we directly obtain that $\mathbb{Q}[\boldsymbol{e}]$ satisfies φ; at least restricted to the executions captured by the symbolic paths in $\mathcal{P}$. Afterward, we can *verify* that $\mathbb{Q}[\boldsymbol{e}]$ indeed satisfies φ (even on paths not explored in $\mathcal{P}$), using existing hyperproperty verification techniques [7,23,34,49,53].

Example 2. Consider the EDAS program $\mathbb{P}$ in Fig. 1, and let $\mathbb{Q}$ be the modified program where the assignment in line 4 is replaced with a fresh function symbol $\tilde{f}$. Define $X := \{\texttt{phase}, \texttt{title}, \texttt{session}, \texttt{decision}\}$. If we perform SE on $\mathbb{Q}$, we get two symbolic paths $\mathcal{P}_{\mathbb{Q}} = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2)\}$, where $\alpha_1 = (\tilde{f}(X) = \texttt{"Accept"})$, $\alpha_2 = (\tilde{f}(X) \neq \texttt{"Accept"})$, $\beta_1 = \big[[\ldots], [\texttt{print} \mapsto \texttt{title + session}, \texttt{decision} \mapsto \tilde{f}(X), \ldots]\big]$, and $\beta_2 = \big[[\ldots], [\texttt{print} \mapsto \texttt{title}, \texttt{decision} \mapsto \tilde{f}(X), \ldots]\big]$. For illustration, we consider the simple trace property $\varphi_{trace} = \forall \pi. \bigcirc(\texttt{print}_\pi = \texttt{title}_\pi)$. If we construct $enc_{\mathcal{P}_{\mathbb{Q}}}^{\varphi_{trace}}$, we get

$$\forall_{x_\pi \in X_\pi} x_\pi. \left(\tilde{f}(X_\pi) = \texttt{"Accept"} \rightarrow \texttt{title}_\pi \texttt{ + } \texttt{session}_\pi = \texttt{title}_\pi\right) \land \left(\tilde{f}(X_\pi) \neq \texttt{"Accept"} \rightarrow \texttt{title}_\pi = \texttt{title}_\pi\right),$$

allowing the simple SyGuS solution $\tilde{f}(X_\pi) := \texttt{"Reject"}$. △

4 Transparent Repair

As argued in Sect. 1, searching for *any* repair (as in Sect. 3) often returns a patch that severely changes the functional behavior of the program. In this paper, we study a principled constraint-based approach on how to guide the search towards a useful repair without requiring extensive additional specifications. Our method is based on the simple idea that the repair should be somewhat *close* to the original program. Crucially, we define “closeness” via *rigorous* systems of (SyGuS) constraints, guiding our constraint-based repair towards minimal patches, with *guaranteed* quality. In this section, we introduce the concept of a (fully) transparent repair. In Sect. 5, we adapt this idea and present a more practical adaption in the form of iterative repair.

4.1 Transparency

Our transparent repair approach is motivated by ideas from the *enforcement* literature [45]. In enforcement, we do not repair the program (i.e., we do not manipulate its source code) but rather let an enforcer run alongside the program and intervene on unsafe behavior (by, e.g., overwriting the output). The obvious enforcement strategy would thus always intervene, effectively overwriting all program behaviors with some dummy (but safe) behavior. To avoid such trivial enforcement, researchers have developed the notion of *transparency* (also called *precision* [45]). Transparency states that the enforcer should not intervene unless an intervention is *absolutely necessary* to satisfy the safety specification, i.e., a safe prefix of the program execution should never trigger the enforcer.

Transparent Repair. The original transparency definition is specific to program enforcement and refers to the *time step* in which the enforcer intervenes. We propose an adoption to the repair setting based on the idea of preserving as

$\mathit{trans}^{\varphi}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$

For $\mathcal{P}_{\mathbb{P}}, \mathcal{P}_{\mathbb{Q}} \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$, define $\mathit{trans}^{\varphi}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$ as

$$\forall_{x \in X} x. \Bigg(\Bigg[\bigvee_{(\alpha_{\mathbb{P}},\beta_{\mathbb{P}}) \in \mathcal{P}_{\mathbb{P}}} \bigvee_{(\alpha_{\mathbb{Q}},\beta_{\mathbb{Q}}) \in \mathcal{P}_{\mathbb{Q}}} \alpha_{\mathbb{P}} \wedge \alpha_{\mathbb{Q}} \wedge \bigvee_{i=0}^{\min(|\beta_{\mathbb{P}}|,|\beta_{\mathbb{Q}}|)-1} \bigvee_{x \in X_{out}} \beta_{\mathbb{P}}(i)(x) \neq \beta_{\mathbb{Q}}(i)(x)\Bigg] \rightarrow$$
$$\exists_{x_{\pi_1} \in X_{\pi_1}} x_{\pi_1}. \cdots \exists_{x_{\pi_n} \in X_{\pi_n}} x_{\pi_n}. \bigvee_{(\alpha_{\pi_1},\beta_{\pi_1}) \in \mathcal{P}_{\mathbb{P}}} \cdots \bigvee_{(\alpha_{\pi_n},\beta_{\pi_n}) \in \mathcal{P}_{\mathbb{P}}}$$
$$\Big(\bigwedge_{j=1}^{n} \alpha_{\pi_j}[x/x_{\pi_j}]\Big) \wedge \Big(\bigvee_{j=1}^{n} \bigwedge_{x \in X} x = x_{\pi_j}\Big) \wedge \neg \mathit{acc}^{\psi}_{[\pi_1 \mapsto \beta_{\pi_1}, \ldots, \pi_n \mapsto \beta_{\pi_n}]}\Bigg)$$

Fig. 7. Encoding for (fully) transparent repair.

much input-output behavior of the original program as possible. Let $X_{out} \subseteq X$ be a set of program variables defining the output. For two stores $\sigma, \sigma' \in \mathit{Stores}$, we write $\sigma \neq_{X_{out}} \sigma'$ if $\sigma(x) \neq \sigma'(x)$ for some $x \in X_{out}$, and extend $\neq_{X_{out}}$ position-wise to sequences of stores.

Definition 1 (Fully Transparent Repair). *Assume $\varphi = \forall \pi_1 \ldots \forall \pi_n. \psi$ is a $\forall^*$ HyperLTL formula and $\mathbb{P}, \mathbb{Q}$ are programs. We say $\mathbb{Q}$ is a fully transparent repair of $(\mathbb{P}, \varphi)$, if (1) $\mathbb{Q} \models \varphi$, and (2) for every store $\sigma \in \mathit{Stores}$ where $\mathit{obs}(\mathbb{P}, \sigma) \neq_{X_{out}} \mathit{obs}(\mathbb{Q}, \sigma)$, there exist stores $\sigma_1, \ldots, \sigma_n \in \mathit{Stores}$ such that $\big[\pi_j \mapsto \mathit{obs}(\mathbb{P}, \sigma_j)\big]_{j=1}^{n}, 0 \not\models \psi$, and $\sigma = \sigma_j$ for some $1 \leq j \leq n$.*

Our definition reasons about inputs σ on which the output behavior of $\mathbb{Q}$ differs from the original program $\mathbb{P}$. Any such input σ must take part in a violation of φ on the original program $\mathbb{P}$. Phrased differently, the repair may only change $\mathbb{P}$'s behavior on executions that take part in a combination of n traces that violate φ. Note that, similar to enforcement approaches [15,45], our transparency definition only applies to $\forall^*$ formulas. As soon as the property includes existential quantification, we can no longer formalize when some execution is "part of a violation of φ". We will extend the central idea underpinning transparency to arbitrary HyperLTL formulas in Sect. 5.

4.2 Encoding for Transparent Repair

Given two finite sets of symbolic paths $\mathcal{P}_{\mathbb{P}}, \mathcal{P}_{\mathbb{Q}} \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$, we define formula $\mathit{trans}^{\varphi}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$ in Fig. 7. The premise states that X defines some input on which $\mathbb{P}$ and $\mathbb{Q}$ differ in their output. That is, for some symbolic paths $(\alpha_{\mathbb{P}}, \beta_{\mathbb{P}}) \in \mathcal{P}_{\mathbb{P}}$ and $(\alpha_{\mathbb{Q}}, \beta_{\mathbb{Q}}) \in \mathcal{P}_{\mathbb{Q}}$, the path conditions $\alpha_{\mathbb{P}}$ and $\alpha_{\mathbb{Q}}$ hold, but the symbolic observation sequences yield some different values for some $x \in X_{out}$. In this case, we require that there exist n symbolic paths $(\alpha_{\pi_1}, \beta_{\pi_1}), \ldots, (\alpha_{\pi_n}, \beta_{\pi_n}) \in \mathcal{P}_{\mathbb{P}}$ and concrete inputs $X_{\pi_1}, \ldots, X_{\pi_n}$, such that (1) the path conditions $\alpha_{\pi_1}, \ldots, \alpha_{\pi_n}$

hold; (2) the assignment to some X_{π_j} equals X; and (3) the symbolic observation sequences $\beta_{\pi_1}, \ldots, \beta_{\pi_n}$ violate ψ (cf. Fig. 5).

Proposition 4. *If* $\mathbb{P}$, $\mathbb{Q}$ *are terminating and* $\mathit{SymPaths}_{max}(\mathbb{P})$, $\mathit{SymPaths}_{max}(\mathbb{Q})$ *are finite, then* $\mathbb{Q}$ *is a fully transparent repair of* $(\mathbb{P}, \varphi)$ *if and only if*

$$\mathit{enc}^{\varphi}_{\mathit{SymPaths}_{max}(\mathbb{Q})} \wedge \mathit{trans}^{\varphi}_{\mathit{SymPaths}_{max}(\mathbb{P}),\mathit{SymPaths}_{max}(\mathbb{Q})}.$$

Example 3. We illustrate transparent repairs using Example 2. If we set $X_{out} := \{\texttt{decision}\}$, and compute $\mathit{trans}^{\varphi_{trace}}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$, we get

$$\forall_{x \in X} x.\ (\texttt{decision} \neq \tilde{f}(X)) \rightarrow \Big((\texttt{decision} = \texttt{"Accept"} \wedge \texttt{title + session} \neq \texttt{title}) \vee (\texttt{decision} \neq \texttt{"Accept"} \wedge \texttt{title} \neq \texttt{title})\Big).$$

For simplicity, we directly resolved the existentially quantified variables X_π with X and summarized all path constraints in the premise. The naïve solution $\tilde{f}(X) := \texttt{"Reject"}$ from Example 2 no longer satisfies $\mathit{trans}^{\varphi_{trace}}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$. Instead, a possible SyGuS solution for $\mathit{enc}^{\varphi_{trace}}_{\mathcal{P}_{\mathbb{Q}}} \wedge \mathit{trans}^{\varphi_{trace}}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$ is

$$\tilde{f}(X) := \mathit{ite}(\texttt{decision} = \texttt{"Accept"} \wedge \texttt{session} \neq \texttt{""}, \texttt{"Reject"}, \texttt{decision}).$$

This solution only changes the decision if the `decision` is `"Accept"` *and* the `session` does not equal the empty string, i.e., it changes the program's `decision` on exactly those traces that violate $\varphi_{trace} = \forall \pi. \bigcirc(\texttt{print}_\pi = \texttt{title}_\pi)$. △

5 Iterative Repair

Our full transparency definition only applies to $\forall^*$ properties, and, even on $\forall^*$ formulas, might yield undesirable results: In some instances, Definition 1 limits which traces may be changed by a repair, potentially resulting in the absence of any repair. In other instances (including the EDAS example), many paths (in the EDAS example, *all* paths) take part in *some* violation of the hyperproperty, so full transparency does not impose any additional constraints. In the EDAS example, this would again allow the naïve repair `decision = "Reject"`. To alleviate this, we introduce an *iterative repair* approach that follows the same philosophical principle as (full) transparency (i.e., search for repairs that are close to the original program), but allows for the *iterative* discovery of better and better repair patches.

Definition 2. *Assume* φ *is a HyperLTL formula and* $\mathbb{P}$, $\mathbb{Q}$, *and* $\mathbb{S}$ *are programs. We say repair* $\mathbb{Q}$ *is a better repair than* $\mathbb{S}$ *w.r.t.* $(\mathbb{P}, \varphi)$ *if (1)* $\mathbb{Q} \models \varphi$, *(2) for every* $\sigma \in \mathit{Stores}$, *where* $\mathit{obs}(\mathbb{P}, \sigma) \neq_{X_{out}} \mathit{obs}(\mathbb{Q}, \sigma)$, *we have* $\mathit{obs}(\mathbb{P}, \sigma) \neq_{X_{out}} \mathit{obs}(\mathbb{S}, \sigma)$, *and (3) for some* $\sigma \in \mathit{Stores}$, *we have* $\mathit{obs}(\mathbb{P}, \sigma) \neq_{X_{out}} \mathit{obs}(\mathbb{S}, \sigma)$ *but* $\mathit{obs}(\mathbb{P}, \sigma) =_{X_{out}} \mathit{obs}(\mathbb{Q}, \sigma)$.

Intuitively, $\mathbb{Q}$ is better than $\mathbb{S}$ if it preserves at least all those behaviors of $\mathbb{P}$ already preserved by $\mathbb{S}$, i.e., $\mathbb{Q}$ is only allowed to deviate from $\mathbb{P}$ on inputs where $\mathbb{S}$ already deviates. Moreover, it must be strictly better than $\mathbb{S}$, i.e., preserve at least one additional behavior.

$\mathit{iter}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{S}},\mathcal{P}_{\mathbb{Q}}}$

For $\mathcal{P}_{\mathbb{P}}, \mathcal{P}_{\mathbb{S}}, \mathcal{P}_{\mathbb{Q}} \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$, define $\mathit{iter}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{S}},\mathcal{P}_{\mathbb{Q}}}$ as

$$\begin{aligned}
&\Bigg(\mathop{\forall}_{x \in X} x. \bigwedge_{(\alpha_{\mathbb{P}},\beta_{\mathbb{P}}) \in \mathcal{P}_{\mathbb{P}}} \bigwedge_{(\alpha_{\mathbb{S}},\beta_{\mathbb{S}}) \in \mathcal{P}_{\mathbb{S}}} \bigwedge_{(\alpha_{\mathbb{Q}},\beta_{\mathbb{Q}}) \in \mathcal{P}_{\mathbb{Q}}} \\
&\qquad \Bigg[\alpha_{\mathbb{P}} \wedge \alpha_{\mathbb{S}} \wedge \alpha_{\mathbb{Q}} \wedge \bigvee_{i=0}^{\min(|\beta_{\mathbb{P}}|,|\beta_{\mathbb{Q}}|)-1} \bigvee_{x \in X_{out}} \mathit{obs}_{\mathbb{P}}(i)(x) \neq \mathit{obs}_{\mathbb{Q}}(i)(x) \Bigg] \rightarrow \\
&\qquad \Bigg[\bigvee_{i=0}^{\min(|\beta_{\mathbb{P}}|,|\beta_{\mathbb{S}}|)-1} \bigvee_{x \in X_{out}} \mathit{obs}_{\mathbb{P}}(i)(x) \neq \mathit{obs}_{\mathbb{S}}(i)(x) \Bigg] \Bigg) \wedge \\
&\Bigg(\mathop{\exists}_{x \in X} x. \bigvee_{(\alpha_{\mathbb{P}},\beta_{\mathbb{P}}) \in \mathcal{P}_{\mathbb{Q}}} \bigvee_{(\alpha_{\mathbb{S}},\beta_{\mathbb{S}}) \in \mathcal{P}_{\mathbb{S}}} \bigvee_{(\alpha_{\mathbb{Q}},\beta_{\mathbb{Q}}) \in \mathcal{P}_{\mathbb{Q}}} \alpha_{\mathbb{P}} \wedge \alpha_{\mathbb{S}} \wedge \alpha_{\mathbb{Q}} \wedge \\
&\qquad \Bigg[\bigvee_{i=0}^{\min(|\beta_{\mathbb{P}}|,|\beta_{\mathbb{S}}|)-1} \bigvee_{x \in X_{out}} \mathit{obs}_{\mathbb{P}}(i)(x) \neq \mathit{obs}_{\mathbb{S}}(i)(x) \Bigg] \wedge \\
&\qquad \Bigg[\bigwedge_{i=0}^{\min(|\beta_{\mathbb{P}}|,|\beta_{\mathbb{Q}}|)-1} \bigwedge_{x \in X_{out}} \mathit{obs}_{\mathbb{P}}(i)(x) = \mathit{obs}_{\mathbb{Q}}(i)(x) \Bigg] \Bigg)
\end{aligned}$$

Fig. 8. Encoding for iterative repair.

5.1 Encoding for Iterative Repair

As before, we show that we can encode Definition 2 via a repair constraint. Let $\mathcal{P}_{\mathbb{P}}$, $\mathcal{P}_{\mathbb{S}}, \mathcal{P}_{\mathbb{Q}} \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$ be finite sets of symbolic paths, and define $\mathit{iter}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{S}},\mathcal{P}_{\mathbb{Q}}}$ as in Fig. 8.

Proposition 5. *If* $\mathbb{P}$*,* $\mathbb{Q}$*, and* $\mathbb{S}$ *are terminating programs and* $\mathit{SymPaths}_{max}(\mathbb{P})$*,* $\mathit{SymPaths}_{max}(\mathbb{S})$*, and* $\mathit{SymPaths}_{max}(\mathbb{Q})$ *are finite, then* $\mathbb{Q}$ *is a better repair than* $\mathbb{S}$*, w.r.t.,* $(\mathbb{P}, \varphi)$ *if and only if*

$$\mathit{enc}^{\varphi}_{\mathit{SymPaths}_{max}(\mathbb{Q})} \wedge \mathit{iter}_{\mathit{SymPaths}_{max}(\mathbb{P}),\mathit{SymPaths}_{max}(\mathbb{S}),\mathit{SymPaths}_{max}(\mathbb{Q})}.$$

5.2 Iterative Repair Loop

We sketch our iterative repair algorithm in Algorithm 1. In line 2, we infer the locations that we want to repair from user annotations. We leave the exploration of automated fault localization techniques specific for hyperproperties as future work, and, in our experiments, assume that the user marks potential repair locations. In line 3, we instrument $\mathbb{P}$ by replacing all repair locations in *locs* with fresh function symbols. At the same time, we record the original expression at all those locations as a vector $\boldsymbol{e}_{\mathbb{P}}$. Subsequently, we perform symbolic execution on the skeleton program $\mathbb{Q}$ (i.e., the program that contains fresh function symbols), yielding a set of symbolic paths $\mathcal{P}$ containing function symbols (line 4). Initially, we now search for *some* repair of φ by using the SyGuS constraint $enc^{\varphi}_{\mathcal{P}}$, giving us an initial repair patch in the form of some expression vector $\boldsymbol{e}$ (line 5). Afterward, we try to iteratively improve upon the repair solution $\boldsymbol{e}$ found previously. For this, we consider the SyGuS constraint $enc^{\varphi}_{\mathcal{P}} \land iter_{\mathcal{P}[\boldsymbol{e}_{\mathbb{P}}],\mathcal{P}[\boldsymbol{e}],\mathcal{P}}$ where we replaced each function symbol in $\mathcal{P}$ with $\boldsymbol{e}_{\mathbb{P}}$ to get the symbolic paths of the original program (denoted $\mathcal{P}[\boldsymbol{e}_{\mathbb{P}}]$), and with $\boldsymbol{e}$ to get the symbolic paths of the previous repair (denoted $\mathcal{P}[\boldsymbol{e}]$) (line 7). If this SyGuS constraint admits a solution $\boldsymbol{e}'$, we set $\boldsymbol{e}$ to $\boldsymbol{e}'$ and repeat with a further improvement iteration (line 11). If the SyGuS constraint is unsatisfiable (or, e.g., a timeout is reached, or the number of iterations is bounded) (written $\boldsymbol{e}' = \bot$), we return the last solution we found, i.e., the program $\mathbb{Q}[\boldsymbol{e}]$ (line 9). By using a single set of symbolic paths $\mathcal{P}$ of the skeleton program $\mathbb{Q}$, we can optimize our query construction. For example, in $iter_{\mathcal{P}[\boldsymbol{e}_{\mathbb{P}}],\mathcal{P}[\boldsymbol{e}],\mathcal{P}}$, we consider all 3 tuples of symbolic paths leading to a potentially large SyGuS query. As we use a common set of paths $\mathcal{P}$ we can prune many path combinations. For example, on fragments preceding a repair location, we never have to combine contradicting branch conditions.

6 Implementation and Evaluation

We have implemented our repair techniques from Sects. 3 to 5 in a proof-of-concept prototype called HyRep, which takes as input a HyperLTL formula and a program in a minimalist C-like language featuring Booleans, integers, and strings. We use spot [20] to translate LTL formulas to NSAs. HyRep can use any solver supporting the SyGuS input format [2]; we use cvc5 (version 1.0.8) [4] as the default solver in all experiments. In HyRep, the user can determine what SyGuS grammar to use, guiding the solver towards a particular (potentially domain-specific) solution. By default, HyRep repairs integer and Boolean expressions using piece-wise linear functions (similar to Example 1), and string-valued expressions by a grammar allowing selected string constants and concatenation of string variables. All results in this paper were obtained using a Docker container of HyRep running on an Apple M1 Pro CPU and 32 GB of memory.

Algorithm 1. Iterative repair algorithm

```
1  def iterativeRepair($\mathbb{P}$,$\varphi$):
2    $locs$ := faultLocalization($\mathbb{P}$,$\varphi$)
3    $\mathbb{Q}$,$\boldsymbol{e}_{\mathbb{P}}$ := instrument($\mathbb{P}$,$locs$)
4    $\mathcal{P}$ := symbolicExecution($\mathbb{Q}$)
5    $\boldsymbol{e}$ := SyGuS($enc^{\varphi}_{\mathcal{P}}$)
6    repeat:
7      $\boldsymbol{e}'$ := SyGuS($enc^{\varphi}_{\mathcal{P}} \land iter_{\mathcal{P}[\boldsymbol{e}_{\mathbb{P}}],\mathcal{P}[\boldsymbol{e}],\mathcal{P}}$)
8      if ($\boldsymbol{e}'$ = $\bot$) then
9        return $\mathbb{Q}[\boldsymbol{e}]$
10     else
11       $\boldsymbol{e}$ := $\boldsymbol{e}'$
```

Scalability Limitations. As we repair for hyperproperties, we necessarily need to reason about the combination of paths, requiring us to analyze multiple paths simultaneously. Unsurprisingly, this limits the scalability of our repair. Consequently, we cannot tackle programs with hundreds of LoC, where existing

```
login(int password, bool attack) {
  if (password == 366) {
    if (attack == true) {
      request = 2 // hidden request (unsafe)
    } else {
      request = 1 // user request (safe)
    }
  } else {
    request = 0 // empty request
  }
  request = request
  observe
}
```

```
request = 0
```

(a)

```
if (password == 366) {
  request = 1
} else {
  request = request
}
```

(b)

Fig. 9. A CSRF attack and repair candidates by `HyRep`.

(*functional*) APR approaches collect a small summary that only depends on the number of input-output examples (see, e.g., *angelic forests* [44]). However, our experiments with `HyRep` attest that – while we can only handle small programs – our approach can find *complex* repair solutions that go beyond previous repair approaches for hyperproperties (cf. Sect. 7).

6.1 Iterative Repair for Hyperproperties

We first focus on `HyRep`'s ability to find, often non-trivial, repair solutions using its iterative repair approach. Table 1 depicts an overview of the 5 benchmark families we consider (explained in the following). For some of the benchmarks, we also consider small variants by adding additional complexity to the program.

EDAS. As already discussed in Sect. 1, `HyRep` is able to repair (a simplified integer-based version of) the EDAS example in Fig. 1 and derive the repairs in Fig. 2.

Table 1. We depict the number of improvement iterations, the number repair locations, and the repair time (in seconds).

Instance	#Iter	#Locations	*t*
EDAS	2	1	2.5
CSRF	2	1	17.9
LOG	1	1	0.9
LOG′	1	1	1.0
LOG″	1	1	7.4
ATM	3	2	4.2
REVIEWS	3	2	18.5
REVIEWS′	3	2	151.6

CSRF. Cross Site Request Forgery (CSRF) [35] attacks target web session integrity. As an abstract example, consider the simple login program as shown in Fig. 9(left), where we leave out intermediate instructions that are not necessary to understand the subsequent repair. If the user attempts to log in and enters the correct password, we either set `request = 1` (modeling a login on the original page), or `request = 2` (modeling an attack, i.e., a login request

```
1  log(string password, string username,
2    string date) {
3    if(password == userPassword){
4      // password flows to credentials
5      credentials = username + password
6    } else {
7      credentials = username
8    }
9    // then flows to info
10   info = date + credentials
11   // then flows to LOG
12   LOG = info
13   observe
14 }
```

```
LOG = ""
```

(a)

```
LOG = date + username
```

(b)

Fig. 10. Privacy leakage by logging and repair candidates by HyRep.

at some untrusted website). We specify that the request should only depend on the (correctness of the) password. When repairing line 11, HyRep first discoverers the trivial repair that always overwrites request with a fixed constant (Fig. 9a). However, in the second improvement iteration, HyRep finds a better repair (Fig. 9b), where the request is only overwritten after a successful login. The potential attack request (request = 2) is thus deterministically overwritten.

LOG. We investigate privacy leaks induced by *logging* of credentials. We depict a simplified code snipped in Fig. 10. Crucially, in case of a successful login, the secret password flows into the public LOG (via credentials and info). We specify that the LOG may only depend on public information (i.e., everything except the password) and use HyRep to overwrite the final value of LOG (i.e., to repair line 12). As shown in Fig. 10a, HyRep first finds a trivial repair that does not log anything. In the first improvement iteration, HyRep automatically finds the more accurate repair in Fig. 10b. That is, it automatically infers that LOG can contain the date and username (as in the original program) but not the password.

```
atm(int balance, int amount) {
  if (balance < amount){
    ErrorLog = "overdraft"
  } else {
    balance = balance - amount
    TransactionLog = "success"
  }
  ErrorLog = ErrorLog
  TransactionLog = TransactionLog
  observe
}
```

Fig. 11. An ATM that leaks the balance to ErrorLog and TransactionLog.

ATM. Many cases require repairing *multiple* lines of code simultaneously. We use cases derived from open-source security benchmarks [26,30,41] and mark multiple repair locations in the input programs. For example, consider the ATM

```
1  reviews(int reviewerAid, int reviewerBid,
2    string reviewA, string reviewB) {
3    notification = "Your␣CAV24␣reviews:"
4    if (reviewerAid <= reviewerBid){
5      order = 1
6    } else {
7      order = 2
8    }
9    order = order
10   if (order == 1) {
11     notification = notification + reviewA
12     notification = notification + reviewB
13   } else {
14     notification = notification + reviewB
15     notification = notification + reviewA
16   }
17   observe
18 }
```

```
order = 0
```

(a)

```
if (reviewerAid < 2) {
  order = order
} else {
  order = 2
}
```

(b)

Fig. 12. A review system that leaks the reviewer ids via the review order and repair candidates by HyRep.

program in Fig. 11. Depending on whether the withdraw amount is greater than balance (secret), different messages will be logged (public). To repair it, we need to repair both ErrorLog and TransactionLog under different conditions (i.e., do not update ErrorLog in the if-clause and do not update TransactionLog in the else-clause). By indicating lines 8 and 9 as two repair locations, HyRep is able to synthesize the correct multiline repair.

REVIEWS. We also investigate the review system depicted in Fig. 12(left). Here the id of each reviewer determines in which order the reviews are displayed to the author. We assume that the PC chair always has the fixed ID 1 (so if he/she submits a review, it will always be displayed first). We want to avoid that the author can infer which review was potentially written by the PC chair. When asked to repair line 9, HyRep produces the repair patches displayed in Figs. 12a and b. In particular, the last repair infers that if reviewerAid < 2 (i.e., reviewer A is the PC chair), we can leave the order; otherwise, we use some fixed constant.

6.2 Scalability in Solution Size

Most modern SyGuS solvers rely on a (heavily optimized) *enumeration* of solution candidates [3,19,33,48]. The synthesis time, therefore, naturally scales in the size of the smallest solutions. Our above experiments empirically show that most repairs can be achieved by small patches. Nevertheless, to test the scalability in the solution size, we have designed a benchmark family that only admits large solutions. Concretely, we consider a program that computes the conjunction of

Table 2. In Table 2a, we evaluate HyRep's scalability in the SyGuS solution size. The timeout (denoted "'-") is 120 s. In Table 2b, we repair a selection of k-safety instances from [7,23,49,53]. In Table 2c, we evaluate on a selection of functional repair instances from [27,44]. All times are given in seconds.

(a)

n	#Iter	t	Size
0	0	0.8	1
1	0	0.8	1
2	1	1.1	3
3	2	1.4	5
4	3	1.8	7
5	4	5.1	9
6	5	89.8	11
7	-	-	-

(b)

Instance	t
CollItemSym	1.4
CounterDet	4.9
DoubleSquareNiFF	4.2
DoubleSquareNi	2.9
Exp1x3	1.1
Fig2	2.4
Fig3	1.1
MultEquiv	2.0

(c)

Instance	t
Assignment	0.7
Deletion	0.7
Guard	0.6
Long-Output	0.7
Multiline	0.8
Not-Equal	0.6
SimpleExample	1.0
OffByOne	2.1

n Boolean inputs $i_1, \ldots, i_n$. We repair against a simple $\forall^2$ HyperLTL property which states that the output may not depend on the last input, guiding the repair towards the optimal solution $i_1 \land \cdots \land i_{n-1}$. We display the number of improvement iterations, the run time, and the solution size (measured in terms of AST nodes) in Table 2a. We note that one of the main features of SyGuS is the flexibility in the input grammar. When using a less permissive (domain-specific) grammar, HyRep scales to even larger repair solutions.

6.3 Evaluation on *k*-Safety Instances

To demonstrate that HyRep can tackle the repair problem in the size-range supported by current *verification* approaches for hyperproperties, we collected a small set of k-safety verification instances from [7,23,49,53]. We modify each program such that the k-safety property is violated and use HyRep's plain (non-iterative) SyGuS constraints to find a repair. The results in Table 2b demonstrate that (1) existing off-the-shelf SyGuS solver can repair programs of the complexity studied in the context of k-safety *verification*, and (2) even in the presence of loops (which are included in all instances in Table 2b), finite unrolling often suffices to generate repair constraints that yield repair patches that work for the full program.

6.4 Evaluation on Functional Properties

While we cannot handle the large programs supported by existing APR approaches for functional properties, we can evaluate HyRep on (very) small test cases. We sample instances from Angelix [44] and GenProg [27], and apply HyRep's direct (non-iterative) repair. We report the run times in Table 2c.

7 Related Work

APR. Existing APR approaches for functional properties can be grouped into search-based and constraint-based [25,28]. Approaches in the former category use a heuristic to explore a set of possible patch candidates. Examples include `GenProg` [27] and `PAR` [36], `SPR` [42], `TBar` [40], or machine-learning-based approaches [21,57]. These approaches typically scale to large code bases, but might fail to find a solution (due to the large solution space). Our approach falls within the latter (constraint-based) category. This approach was pioneered by `SemFix` [46] and later refined by `DirectFix` [43], `Angelix` [44], and `S3` [39]. To the best of our knowledge, we are the first to employ the (more general) SyGuS framework for APR, which leaves the exact search to an external solver. Most APR approaches rely on a finite set of input-output examples. To avoid overfitting [50] these approaches either use heuristics (to, e.g., infer variables that a repair should depend on [55]) or employ richer (e.g., MaxSMT-based) constraints [44]. Crucially, these approaches are *local*, whereas our repair constraints reason about the entire (*global*) program execution by utilizing the entire symbolic path. Any repair sequence generated by our iterative repair is thus guaranteed to increase in quality, i.e., preserve more behavior of the original program.

APR for Hyperproperties. Coenen et al. [15] study *enforcement* of alternation-free hyperproperties. Different from our approach, enforcement does not provide guarantees on the functional behavior of the enforced system. Bonakdarpour and Finkbeiner [9] study the repair-complexity of hyperproperties in *finite*-state transition systems. In their setting, a repair consists of a substructure, i.e., a system obtained by removing some of the transitions of the system, so the repair problem is trivially decidable. Polikarpova et al. [47] present `Lifty`, and encoding of information-flow properties using refinement types. `Lifty` can automatically patch a program to satisfy an information-flow requirement by assigning *all* private variables some public dummy default constant. In contrast, our approach can repair against complex temporal hyperproperties (possibly involving quantifier alternations), and our repair often goes beyond insertion of constants.

8 Conclusion

We have studied the problem of automatically repairing an (infinite-state) software program against a temporal hyperproperty, using SyGuS-based constraint generation. To enhance our basic SyGuS-based approach, we have introduced an iterative repair approach inspired by the notion of transparency. Our approach interprets "closeness" rigorously, encodes it within our constraint system for APR, and can consequently derive non-trivial repair patches.

Acknowledgments. This work was partially supported by the European Research Council (ERC) Grant HYPER (101055412), by the German Research Foundation (DFG) as part of TRR 248 (389792660), and by the United States NSF SaTC Awards 210098 and 2245114.

References

1. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: Computer Security Foundations Symposium, CSF 2016 (2016). https://doi.org/10.1109/CSF.2016.24
2. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013 (2013)
3. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017 (2017). https://doi.org/10.1007/978-3-662-54577-5_18
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. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: International Conference on Computer Aided Verification, CAV 2021 (2021). https://doi.org/10.1007/978-3-030-81685-8_33
6. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: Computer Security Foundations Symposium, CSF 2022 (2022). https://doi.org/10.1109/CSF54842.2022.9919658
7. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13185-1_17
8. Beutner, R., Finkbeiner, B.: AutoHyper: explicit-state model checking for HyperLTL. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023. LNCS, vol. 13993, pp. 145–163. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_8
9. Bonakdarpour, B., Finkbeiner, B.: Program repair for hyperproperties. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 423–441. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_25
10. Bonakdarpour, B., Sanchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 8–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_2
11. Bozzelli, L., Peron, A., Sánchez, C.: Asynchronous extensions of HyperLTL. In: Symposium on Logic in Computer Science, LICS 2021 (2021). https://doi.org/10.1109/LICS52264.2021.9470583
12. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM **55**(8) (2012). https://doi.org/10.1145/2240236.2240262
13. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference on Principles of Security and Trust, POST 2014 (2014). https://doi.org/10.1007/978-3-642-54792-8_15
14. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Computer Security Foundations Symposium, CSF 2008 (2008). https://doi.org/10.1109/CSF.2008.7
15. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J., Schillo, Y.: Runtime enforcement of hyperproperties. In: International Symposium on Automated Technology for Verification and Analysis, ATVA 2021 (2021). https://doi.org/10.1007/978-3-030-88885-5_19

16. 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
17. Daniel, L., Bardin, S., Rezk, T.: Binsec/Rel: efficient relational symbolic execution for constant-time at binary-level. In: Symposium on Security and Privacy, SP 2020 (2020). https://doi.org/10.1109/SP40000.2020.00074
18. Daniel, L., Bardin, S., Rezk, T.: Hunting the haunter - efficient relational symbolic execution for Spectre with haunted RelSE. In: Annual Network and Distributed System Security Symposium, NDSS 2021 (2021)
19. Ding, Y., Qiu, X.: Enhanced enumeration techniques for syntax-guided synthesis of bit-vector manipulations. Proc. ACM Program. Lang. (POPL) (2024). https://doi.org/10.1145/3632913
20. Duret-Lutz, A., et al.: From spot 2.0 to spot 2.10: what's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification, CAV 2022. LNCS, vol. 13372, pp. 174–187. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_9
21. Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., Tan, S.H.: Automated repair of programs from large language models. In: International Conference on Software Engineering, ICSE 2023 (2023). https://doi.org/10.1109/ICSE48619.2023.00128
22. Farina, G.P., Chong, S., Gaboardi, M.: Relational symbolic execution. In: International Symposium on Principles and Practice of Programming Languages, PPDP 2019 (2019). https://doi.org/10.1145/3354166.3354175
23. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 200–218. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_11
24. 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
25. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: a survey. IEEE Trans. Softw. Eng. **45**(1) (2019). https://doi.org/10.1109/TSE.2017.2755013
26. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in DroidSafe. In: Annual Network and Distributed System Security Symposium, NDSS 2015 (2015)
27. Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering, ICSE 2012 (2012). https://doi.org/10.1109/ICSE.2012.6227211
28. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12) (2019). https://doi.org/10.1145/3318162
29. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Automata and fixpoints for asynchronous hyperproperties. Proc. ACM Program. Lang. (POPL) (2021). https://doi.org/10.1145/3434319
30. Hamann, T., Herda, M., Mantel, H., Mohr, M., Schneider, D., Tasch, M.: A uniform information-flow security benchmark suite for source code and bytecode. In: Nordic Conference on Secure IT Systems, NordSec 2018 (2018). https://doi.org/10.1007/978-3-030-03638-6_27
31. Hsu, T.-H., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: TACAS 2021. LNCS, vol. 12651, pp. 94–112. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_6

32. Hsu, T., Sánchez, C., Sheinvald, S., Bonakdarpour, B.: Efficient loop conditions for bounded model checking hyperproperties. In: Sankaranarayanan, S., Sharygina, N. (eds.) International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023. LNCS, vol. 13993, pp. 66–84. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_4
33. Huang, K., Qiu, X., Shen, P., Wang, Y.: Reconciling enumerative and deductive program synthesis. In: International Conference on Programming Language Design and Implementation, PLDI 2020 (2020). https://doi.org/10.1145/3385412.3386027
34. Itzhaky, S., Shoham, S., Vizel, Y.: Hyperproperty verification as CHC satisfiability. In: Weirich, S. (eds.) European Symposium on Programming Languages and Systems, ESOP 2024. LNCS, vol. 14577, pp. 212–241. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57267-8_9
35. Khan, W., Calzavara, S., Bugliesi, M., De Groef, W., Piessens, F.: Client side web session integrity as a non-interference property. In: Prakash, A., Shyamasundar, R. (eds.) ICISS 2014. LNCS, vol. 8880, pp. 89–108. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13841-1_6
36. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: International Conference on Software Engineering, ICSE 2013 (2013). https://doi.org/10.1109/ICSE.2013.6606626
37. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976). https://doi.org/10.1145/360248.360252
38. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 172–183. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_17
39. Le, X.D., Chu, D., Lo, D., Goues, C.L., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017 (2017). https://doi.org/10.1145/3106237.3106309
40. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: TBar: revisiting template-based automated program repair. In: International Symposium on Software Testing and Analysis, ISSTA 2019 (2019). https://doi.org/10.1145/3293882.3330577
41. Livshits, B.: SecuriBench Micro (2014). https://github.com/too4words/securibench-micro
42. Long, F., Rinard, M.C.: Automatic patch generation by learning correct code. In: Symposium on Principles of Programming Languages, POPL 2016 (2016). https://doi.org/10.1145/2837614.2837617
43. Mechtaev, S., Yi, J., Roychoudhury, A.: DirectFix: looking for simple program repairs. In: International Conference on Software Engineering, ICSE 2015 (2015). https://doi.org/10.1109/ICSE.2015.63
44. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: International Conference on Software Engineering, ICSE 2016 (2016). https://doi.org/10.1145/2884781.2884807
45. Ngo, M., Massacci, F., Milushev, D., Piessens, F.: Runtime enforcement of security policies on black box reactive programs. In: Symposium on Principles of Programming Languages, POPL 2015 (2015). https://doi.org/10.1145/2676726.2676978
46. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: International Conference on Software Engineering, ICSE 2013 (2013). https://doi.org/10.1109/ICSE.2013.6606623
47. Polikarpova, N., Stefan, D., Yang, J., Itzhaky, S., Hance, T., Solar-Lezama, A.: Liquid information flow control. Proc. ACM Program. Lang. (ICFP) (2020). https://doi.org/10.1145/3408987

48. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: cvc4sy: smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 74–83. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_5
49. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 161–179. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_9
50. Smith, E.K., Barr, E.T., Goues, C.L., Brun, Y.: Is the cure worse than the disease? Overfitting in automated program repair. In: Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015 (2015). https://doi.org/10.1145/2786805.2786825
51. Tiraboschi, I., Rezk, T., Rival, X.: Sound symbolic execution via abstract interpretation and its application to security. In: International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2023 (2023). https://doi.org/10.1007/978-3-031-24950-1_13
52. Tsoupidi, R., Balliu, M., Baudry, B.: Vivienne: relational verification of cryptographic implementations in WebAssembly. In: Secure Development Conference, SecDev 2021 (2021). https://doi.org/10.1109/SECDEV51306.2021.00029
53. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 742–766. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_35
54. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**(8) (2016). https://doi.org/10.1109/TSE.2016.2521368
55. Xiong, Y., et al.: Precise condition synthesis for program repair. In: International Conference on Software Engineering, ICSE 2017 (2017). https://doi.org/10.1109/ICSE.2017.45
56. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Computer Security Foundations Workshop, CSFW 2003 (2003). https://doi.org/10.1109/CSFW.2003.1212703
57. Zhu, Q., et al.: A syntax-guided edit decoder for neural program repair. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021 (2021). https://doi.org/10.1145/3468264.3468544

The SemGuS Toolkit

Keith J. C. Johnson[1(✉)], Andrew Reynolds[2], Thomas Reps[1], and Loris D'Antoni[1]

[1] University of Wisconsin–Madison, Madison, USA
keithj@cs.wisc.edu
[2] University of Iowa, Iowa City, USA

Abstract. Semantics-Guided Synthesis (SemGuS) is a programmable framework for defining synthesis problems in a domain- and solver-agnostic way. This paper presents the standardized SemGuS format, together with an open-source toolkit that providesa parser, a verifier, and enumerative SemGuS solvers. The paper also describes an initial set of SemGuS benchmarks, which form the basis for comparing SemGuS solvers, and presents an evaluation of the baseline enumerative solvers.

1 Introduction

The field of program synthesis aims to create tools that can automatically create a program from a specification of desired behavior. Synthesis holds the promise of easing the burden on programmers (e.g., by finding solutions to tricky special cases automatically), and allowing non-programmers to create programs merely by indicating the outcome that they want the program to produce.

While program synthesis has seen successes in many industrial applications [9,23], these successes have typically been achieved using domain-specific synthesizers that take advantage of the structure of the specific domain.

To apply synthesis beyond specific domains, synthesis frameworks and tools should allow one to customize the search space and specifications of a synthesis problem in a programmable way that is agnostic of a specific domain or synthesis solver. To address the problem of making synthesis "programmable", Kim et al. [15] proposed the SemGuS framework, which enables one to specify synthesis problems in a solver-agnostic and domain-agnostic way [7].

The SemGuS framework allows one to specify an arbitrary synthesis problem by defining a programming language via (i) a grammar (the syntax), and (ii) a set of Constrained Horn Clauses (CHCs) (the semantics). Once one has described the language, one can define synthesis problems over that language by providing a specification as a formula. Solving the synthesis problem means finding a program in the language that satisfies the specification. Building solvers for general SemGuS problems can be difficult due to the framework's flexibility [7].

This paper presents the SemGuS toolkit, which provides an open-source implementation of the components needed for researchers to get started building SemGuS solvers. The toolkit consists of the following components.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 27–40, 2024.
https://doi.org/10.1007/978-3-031-65633-0_2

SemGuS Format 1.0: The first standardized format for SemGuS, which is built on top of the SMT-LIB and SyGuS formats [3,21], thus making it expressible, extensible, modular, and easy to integrate with existing constraint solvers (e.g., to build SemGuS verifiers). We provide an open-source parser (Sect. 2).

Baseline Verifier and Solvers: The flexibility of SemGuS makes verifying whether a term is a solution to a SemGuS problem undecidable. Furthermore, because the semantics of the user-provided programming language is expressed declaratively using CHCs, it is even challenging to efficiently execute programs in the language. Our implementation provides a compiler that, given a term t in the user-specified language, can extract efficiently executable semantics for t from the declarative one provided by the user, as well as an incomplete SMT-based verifier that can construct constraints for checking whether t matches a specification φ. We also provide implementations of top-down and bottom-up example-based enumerative solvers that are integrated with these verifiers and can thus produce solutions to SemGuS problems (Sect. 3).

Benchmarks: We provide 431 SemGuS benchmarks from different domains. Our solvers can only solve 161/431 benchmarks, and we hope this toolkit will energize the community to build solvers for the remaining challenging problems and to provide additional benchmarks (Sect. 4).

2 The SemGuS Format 1.0

We refer the reader to the original SemGuS paper [15] for a more formal definition of the SemGuS framework, but in this section we show how each component is expressed in our proposed standard format. The SemGuS parser (https://github.com/semgus-git/Semgus-Parser) can translate the textual SemGuS format into two intermediate representations: a JSON format and a declarative S-expression format, which is then used by solvers and other tools.

Figures 1 and 2 give an example specification of a SemGuS problem, which we describe in detail in this section. In this example, the goal is to synthesize an imperative program (with loops) that multiplies two numbers through iterative addition. We choose this example because it illustrates how SemGuS can describe synthesis problems involving complex programming constructs and is thus strictly more expressive than limited synthesis frameworks, such as SyGuS [1].

Term Universe. SemGuS problems define a universe of terms with a modified SMT datatype declaration using the command `declare-term-types` (lines 1–8 of Fig. 1). This command defines the syntax of the programming language over which one can specify synthesis problems. The term universe L is intentionally separated from the sub-universe (defined by a grammar) from which the answer is to be synthesized, and from the constraints on the answer (Fig. 2). The user defines L and its semantics once and for all, and can reuse those definitions for different synthesis problems. This separation enables both *(i)* building specialized SemGuS solvers for important languages (e.g., L = SQL), and *(ii)* instantiating more restricted synthesis problems by confining the search space to just the terms generated by a grammar.

Semantics as CHCs. The semantics of our term language is given by the SMT-LIB command `define-funs-rec` (lines 9–43). In a nutshell, this command defines a set of Constrained Horn Clauses (CHCs) inductively over terms in the universe. A CHC is a first-order formula of the form:

$$\forall \bar{x}_1, \ldots, \bar{x}_n, \bar{x}.\ \phi \wedge R_1(\bar{x}_1) \wedge \cdots \wedge R_n(\bar{x}_n) \Rightarrow H(\bar{x})$$

where $R_1, \ldots, R_n$ and H are uninterpreted relations, $\bar{x}_1, \ldots, \bar{x}_n$ and $\bar{x}$ are (vectors/tuples of) variables, and ϕ is a quantifier-free constraint over the variables within some first-order theory. In the specification, one provides the names and types of the semantic relations used in the semantic definitions (lines 11–16), and then CHCs that define such relations (lines 17–43). To better align with the fact that CHCs are used to define the semantics of programs inductively (i.e., as an interpreter), in the SemGuS format we encode CHCs as a set of mutually-recursive SMT functions, taking the term to be evaluated, input variables, and output variables as arguments, and returning a Boolean. A function is provided for every non-terminal, and `match` statements are used to dispatch on the term constructors for which one is defining the semantics. The match statement must match on all productions for the given term type (i.e., the corresponding non-terminal). The match statement can also be annotated with which variables are inputs and outputs in the specific semantics (note that some semantics, e.g., a term-rewriting system, do not necessarily have inputs and outputs). Each match on a production starts with an optional `exists` block, which specifies auxiliary variables, followed by the CHC body as a conjunction. Some productions, such as the *while* production in Fig. 1 (lines 29–38), have two associated CHCs. For example, the `While-false` CHC can be logically written as

$$\frac{B.Sem(tb, xi, yi, ri, b) \quad b = \mathit{false} \wedge xo = xi \wedge yo = yi \wedge ro = ri}{S.Sem((\texttt{\$while}\ tb\ ts), xi, yi, ri, xo, yo, ro)}\ \texttt{While-false}$$

The signature at the bottom of the CHC—i.e., the particular variables names used in this relation instance—is the one defined in Fig. 1 (line 13).

As discussed in the original SemGuS paper [15], many synthesizers have achieved scalability by exploiting alternative semantics that either underapproximate the actual semantics of the programming language (to speed up evaluation and enable constraint solving) or overapproximate it (which sometimes makes it possible to prune the search space of programs). While such previous work “hardcodes” and takes advantage of such semantics in the solver itself, SemGuS allows one to write such semantics directly in the SemGuS file. In fact, there is no limit on how many semantic relations one can define in a SemGuS file. For example, one might define a semantic relation that associates costs to programs (but does not evaluate programs) and a semantic relation that captures program evaluation. The specification can then require finding a program that (i)

```
 1 ;; Nonterminals
 2 (declare-term-types ((F 0) (S 0) (E 0) (B 0))
 3 ;; Term Universe (i.e., language syntax)
 4 ((($function S E))                             ;; F
 5  (($x<- E) ($y<- E) ($r<- E)
 6   ($noop) ($seq S S) ($while B S))             ;; S
 7  (($r) ($0) ($1) ($x) ($y) ($+ E E) ($- E E))  ;; E
 8  (($< E E))))                                  ;; B
 9 ;; Constrained Horn Clauses (i.e., language semantics)
10 (define-funs-rec
11  ;; Types of semantic relations
12  ((F.Sem ((t F) (x Int) (y Int) (ret Int)) Bool)
13   (S.Sem ((t S) (xi Int) (yi Int) (ri Int) (xo Int)
14           (yo Int) (ro Int)) Bool)
15   (E.Sem ((t E) (xi Int) (yi Int) (ri Int) (out Int)) Bool)
16   (B.Sem ((t B) (xi Int) (yi Int) (ri Int) (out Bool)) Bool))

17  ;; CHCs defining semantic relations
18  (;; Semantics of functions
19   (! (match t (...)) :input (x y) :output (ret))
20   ;; Semantics of statements
21   (! (match t
22      ...more S productions...
23      (($noop)         ;; Noop statement
24       (and (= xi xo) (= yi yo) (= ri ro)))
25      (($seq t1 t2)    ;; Sequential composition
26       (exists ((x1 Int) (y1 Int) (r1 Int))
27         (and (S.Sem t1 xi yi ri x1 y1 r1)
28              (S.Sem t2 x1 y1 r1 xo yo ro))))
29      (($while tb ts)  ;; While statement
30       (exists ((b Bool) (x1 Int) (y1 Int) (r1 Int))
31         (and (B.Sem tb xi yi ri b) ;; While-true
32              (= b true)
33              (S.Sem ts xi yi ri x1 y1 r1)
34              (S.Sem t x1 y1 r1 xo yo ro)))
35       (exists ((b Bool))
36         (and (B.Sem tb xi yi ri b) ;; While-false
37              (= b false)
38              (= xo xi) (= yo yi) (= ro ri))))))
39   :input (xi yi ri) :output (xo yo ro))
40   ;; Semantics of integer expressions
41   (! (match t (...)) :input (xi yi ri) :output (out))
42   ;; Semantics of Boolean expressions
43   (! (match t (...)) :input (xi yi ri) :output (out))))
```

Fig. 1. Definition of a programming language (i.e., a set of programs) in the SemGuS format. The syntax of terms is given in lines 1–8, and their semantics is given in lines 9–43. The `gray text` denotes parts that have been omitted for brevity.

performs a computation correctly, and (ii) has a cost that is less than a specific constant. The ability of SemGuS to describe multiple semantics enables reusable solving techniques and interoperability between solvers.

It should be noted that when one defines multiple semantics, the burden of showing that they are properly related (e.g., that an abstract semantics is related to the concrete one by a Galois connection [5]) is in the hands of the user. Doing so automatically is a research direction enabled by the SemGuS format.

`synth-fun` Command. SemGuS uses the same syntax as in SyGuS to declare what type of term we are interested in synthesizing (Fig. 2). Unlike SyGuS, a solution to a SemGuS problem is a term in the provided syntax, as opposed to a function in an SMT theory. For instance, the command `(synth-fun mul () F)` in Fig. 2 (line 2) asks for a term named `mul`, rooted at the non-terminal `F`. This command can optionally take a grammar (the second argument) to further restrict the search space, using the same format for grammars as in the SyGuS format [21]. For example, to synthesize programs that have the fewest number of while-loops [12], one might first solve the SemGuS problem discussed in this section and obtain a program with one loop, and then create a new SemGuS problem where the grammar is restricted to disallow loops. The two problems will share the same language definition despite having different grammars.

Specification. Specification constraints for SemGuS problems are stated using SMT expressions involving the root CHC for the term to be synthesized. The typical form for input/output examples is shown in Fig. 2 (lines 4–9). Note that in SemGuS, constraints are specified as relations and not functions (as in SyGuS). Relations allow modeling nondeterminism or nonterminating semantics—e.g., one can state that, for the specific input pair (5,3), the answer is a positive value if the program terminates: `(constraint (forall ((x Int)) (=> (F.Sem mul 5 3 x) (>= x 0))))`.

```
 1 ;; Function to synthesize
 2 (synth-fun mul () F)
 3 ;; Constraints for examples
 4 (constraint (F.Sem mul 0 0 0))
 5 (constraint (F.Sem mul 1 1 1))
 6 (constraint (F.Sem mul 2 2 4))
 7 (constraint (F.Sem mul 3 3 9))
 8 (constraint (F.Sem mul 5 3 15))
 9 (constraint (F.Sem mul 3 4 12))
10 ;; Perform synthesis
11 (check-synth)
```

Fig. 2. Constraints for a few example input/output pairs, used to synthesize a function `mul` that behaves like multiplication.

Synthesis Command. The `check-synth` command instructs the solver to solve the problem and produce an SMT term. The following term is a solution to the example presented in this section.

```
((define-fun mul () F ($function
        ($while ($< $0 $y)                ;; while (0<y)
            ($seq ($y<- ($- $y $1))       ;;     y <- y-1
                  ($r<- ($+ $r $x))))     ;;     r <- r+x
        $r)))                             ;; return r
```

Relationship Between SemGuS and SyGuS. Every SyGuS problem can be automatically converted to an equivalent SemGuS problem, and our parser implements this transformation. The only technical detail of interest is that SyGuS synthesizes *function SMT terms*, whereas SemGuS synthesizes terms in a term universe that is interpreted using a *relational semantics*. For example, if the predicate φ of the SyGuS specification contains invocations of the function g to be synthesized, e.g., $\varphi(g(i_1), \ldots, g(i_n))$, we can create the new SemGuS specification as $\exists o_1, \ldots, o_n.\ \varphi(o_1, \ldots, o_n) \wedge \mathrm{Sem}_G(g, i_1, o_1) \wedge \ldots \wedge \mathrm{Sem}_G(g, i_n, o_n)$.

Because SemGuS is more expressive than SyGuS, not every SemGuS problem can be converted to an equivalent SyGuS problem. In general, it is undecidable to check when such a translation is possible, because SemGuS is Turing complete. We have implemented a sound (but incomplete) translation of a limited fragment of statically detectable SemGuS problems into SyGuS. The fragment essentially captures when the SyGuS-to-SemGuS translation can be inverted.

3 A Baseline SemGuS Solver

In this section, we present KS2, a toolkit for researchers to build SemGuS solvers. KS2 implements techniques for (efficiently) verifying whether a candidate solution meets the specification (Sect. 3.1). KS2 also contains implementations of bottom-up and top-down enumerative synthesizers (Sect. 3.2). KS2 is written in Common Lisp, which makes it easy to compile code generated at synthesis-time for speeding up evaluation of candidate solutions. In addition, KS2 is implemented modularly, so new solvers and features can be easily added as plugins.

3.1 Verifying Candidate Solutions

When building synthesizers, one wants two types of verifiers: one that can quickly tell if a candidate solution is correct on a finite set of input examples E, and one that can (less quickly) tell if a solution is correct on all inputs, and thus satisfies a logical specification. When the latter verifier finds a violation of the specification, it will typically produce a new input example e that can be added to the set E to restart synthesis with a fresh set of examples. These two verifiers together form the basis of the counterexample-guided synthesis algorithm. For SemGuS, building either of these verifiers is generally undecidable as one may have to deal with an arbitrarily powerful programming language.

In this section, we present two sound (but incomplete) implementations of such verifiers. These implementations are not the only verifier implementations that can be built for SemGuS, but just two that were successful in meeting our needs. Building other verifier implementations based on other technologies, such as bounded model checkers, symbolic execution, and logic programming, is an interesting future research direction.

Building Executable Semantics from CHCs. To tell quickly whether a candidate program is correct on a given input, one needs to "run" the program on the input according to the semantics. To do so efficiently is nontrivial because the semantics of a candidate program is expressed declaratively using CHCs. KS2 first "operationalizes" the semantics given by CHCs into executable blocks, which are then compiled. In general, not all CHCs can be transformed into executable code (for example, non-deterministic CHCs that can map one input to different outputs); therefore, KS2 supports only a fragment of CHCs that is practically useful (all benchmarks discussed in Sect. 4 fall into this fragment).

We illustrate the compilation to native code using the following (recursive) CHC corresponding to the `While-true` case in Fig. 1:

$$\mathrm{Sem}_S(t(t_b, t_s), i, o) \impliedby \mathrm{Sem}_B(t_b, i, b) \wedge b \wedge \mathrm{Sem}_S(t_s, i, o') \wedge \mathrm{Sem}_S(t, o', o)$$

KS2 requires each position in each relation to be annotated as an input or output variable.[1] In the example, the first position of Sem_S and Sem_B is the term being executed, which is always assumed to be an input; the second position is the input on which the term should be executed, and the last position is the output. To operationalize the CHC, KS2 performs the following steps to identify an evaluation order: it analyzes each relation instance in the body of the CHC, and performs a dataflow analysis to determine an order in which the blocks can be executed. This step is done by building a dataflow graph and then performing a topological sort to identify an order in which each relation can be computed. In the given example, one possible order is to first evaluate $\mathrm{Sem}_B(t_b, i, b)$ because i is readily available, then determine whether b is true, then evaluate $\mathrm{Sem}_S(t_s, i, o')$, and finally evaluate $\mathrm{Sem}_S(t, o', o)$ (o' depends on one of the previous relations). Our implementation of this transformation has some basic requirements. First, no two relations can output the same variable—otherwise one cannot resolve which instance to use. Second, every input variable i' to a relation is either the output of another relation or appears in the first-order formula of the CHC in the form $i' = f(\cdot)$—i.e., the value of i' can be computed without having to call an SMT solver.

At this point, we generate code for each block. Child CHCs turn into function calls, guards into conditional statements, and value productions into assignments. This generated code is then compiled and turned into an executable function that implements the CHC's semantics. To execute a program on an input/output example, the top-level semantic function is called with the input state and the child's semantic functions, and the program returns the output state. This output state can be checked against the output example.

This implementation of an efficiently executable semantics is one of the main contributions in KS2. Identifying additional ways to compile the logical semantics into an efficiently-executable one is an interesting research direction that can benefit from techniques in compiler design and logic programming.

[1] Automatically inferring such annotations ("mode inference") is a classical analysis problem in logic programming [6, §10.2.2]. Automatically supplying annotations is an interesting research direction for SemGuS.

A Simple Incomplete Verifier for Logical Specifications. The declarative nature of SemGuS enables a simple way of building a verifier that can check a program against a specification and return a counterexample. As we argued, verification for SemGuS is undecidable, but the declarative nature of SemGuS allows us to build a simple, but incomplete, procedure for verifying some candidates in SemGuS solutions. Given a concrete term t (i.e., the program we are trying to verify), our verifier performs a pre-order traversal of t and emits a potentially recursive SMT function for each node that corresponds to the CHC (or CHCs) for that node. Because t is a concrete term, each child term in the CHC can also be replaced by the concrete function implementing it. For example, the program `($+ $x $1)` would be verified by emitting three SMT functions for `$+`, `$x`, and `$1`. The function for `$+` will call the ones for `$x` and `$1` to perform the evaluation. Operators like `$while` require recursive function calls. The specification can then be used to define constraints over the root node and verified with an SMT solver. In the case of recursive semantics, the SMT functions will potentially be mutually recursive, thus relying on undecidable theories for which current SMT solvers struggle in practice. This verifier, while incomplete, is "good enough" for many of our current benchmarks, and extending SMT solvers or our verifiers to better handle such cases is a challenging research question.

3.2 Baseline Enumerative Solvers

KS2 implements standard basic top-down and bottom-up enumeration algorithms as described in the literature [11]. Because we have a logical verifier that enables counterexample-guided inductive synthesis (CEGIS), our enumeration algorithms only check correctness on a set of examples.

For top-down enumeration, candidate programs are enumerated using a priority queue of potentially partial programs (i.e., with holes). At each iteration a program is extracted from the queue: if it has no holes, it is verified against the examples; otherwise, all the programs that can be obtained by expanding the leftmost hole with all possible child productions are added to the queue. The standard optimization for top-down enumerators is to check partial programs against the specification and prune them if possible. Existing optimizations are domain-specific and identifying ways to extend them to SemGuS problems is an open research question, which we hope this toolkit will help researchers work on.

For bottom-up enumeration, subterms of increasing size (or height) are enumerated and added to a program bank where they are grouped by size (or height). Enumeration of programs of a certain size or height happens lazily; they are verified and pushed into the bank of programs one at a time. A typical optimization used in a bottom-up enumeration is to use some form of equivalence-checking to deduplicate enumerated programs with the same behavior. One popular technique, observational equivalence, executes each enumerated program on the input example states and prunes a program if a previously enumerated program returns the same output state. However, because SemGuS supports imperative semantics, the possible input states for a sub-program are not necessarily the same as the top-level-program's input states (i.e., variable values change throughout the program execution), and thus there is not an easy way to perform an observational-equivalence check. The development of an appropriate pruning technique for a bottom-up SemGuS enumerator is an open research question, which we hope this toolkit will help researchers work on, for example by building on approaches such as equality saturation [25] and lifting interpretation to sets of programs [17].

3.3 Extensibility

KS2 can be extended by instantiating various interfaces with modules. For example, one might want to add a module that implements a technique for pruning enumerated programs with the bottom-up enumeration. To add this technique, the module would implement the `add-to-bank` interface, which is responsible for adding freshly enumerated programs to the bank of enumerated programs, and simply decline to add programs that the module can prune. In code, this implementation might look like:

```
(defmethod add-to-bank :around ((ext prune) bank prog metric)
  "Adds the program PROG to BANK unless it should be pruned"
  (unless (%should-prune prog) (call-next-method)))
```

where `prune` is the module class and `%should-prune` implements the predicate for whether or not a program should be pruned. At this time, among others, we have interfaces for adding solvers, adding verifiers, and inspecting and updating the SemGuS problem. We will continue to add more interfaces as the need arises; the most up-to-date documentation is available with KS2 and its supporting libraries.

Outside of KS2, the SemGuS Parser is available as a standalone tool for parsing SemGuS problems into JSON, as well as a .NET library for direct integration into solvers. We expect these parsing tools to lower the barrier to entry for building new SemGuS tooling.

4 Benchmarks and Performance of Baseline Solvers

We present an initial set of SemGuS benchmarks and evaluate the performance of our baseline solvers on such benchmarks.

Benchmarks. The ability of SemGuS to represent synthesis problems from disparate domains in the same solver-agnostic format is one of its key distinguishing features. We have created 431 SemGuS benchmarks, consisting of synthesis problems from a variety of domains.

Sample domains: 17 benchmarks of easy synthesis problems (10 for imperative programs with loops, 3 for SMT datatypes, and 4 integer-arithmetic benchmarks). These benchmarks are designed to help researchers build SemGuS solvers and are basic test of a solver's support of various features of the SemGuS format. They contain between 1 and 6 input/output examples each.

Table 1. Solved benchmarks by category.

Domain	Total	TopDown	BottomUp(H)	BottomUp(S)	Virtual Best
Sample Domains	17	14	11	13	15
Regular Expressions	72	52	8	45	54
Boolean	88	45	47	46	49
Bitvectors	100	38	27	36	43
Messy	154	0	0	0	0
Total	431	149	93	140	161

Regular expressions: 72 benchmarks for synthesizing regular expressions, which include problems from the original SemGuS paper [15], from the tool AlphaRegex [16], and CSV formatting problems. These benchmarks have between 2 and 244 input/output examples each. Benchmarks in this category may use two different semantics of regular expressions: one based on Boolean matrices and one based on SMT terms for the theory of regular expressions.

Boolean formulas: 88 benchmarks for synthesizing Boolean formulas, including DNF (32), CNF (33), and cube (23) formulas. Each benchmark has between 4 and 128 input/output examples.

Bitvectors: We provide 100 benchmarks over imperative loop-free bitvector programs [10]. In our adaptation of the existing benchmarks, we consider different bitvector semantics (e.g., one where bitvectors restart at 0 on overflow, and one where the values remain at INT_MIN or INT_MAX). The ability to customize programs semantics is a key feature of SemGuS. These benchmarks use logical specifications instead of input-output examples.

Messy: 154 benchmarks (15 bitvector, 18 imperative, 121 unrealizable SyGuS and imperative) from the original SemGuS paper.

We expect this set to be extended. New benchmarks may be submitted to the Semgus-Benchmarks GitHub repository via pull requests. All submissions are automatically checked for proper syntax and manually reviewed by maintainers for appropriateness before being included.

Performance of Baseline Solvers. Benchmark results for our top-down and bottom-up enumerators by height (H) and size (S) are shown Table 1 as a summary solved instances and Fig. 3 as a cactus plot illustrating the time taken to solve the benchmarks. All experiments are run on a cluster [4], with each node having an AMD EPYC 7763 64-Core Processor, of which we requested two cores and 12 GiB of RAM. We set a timeout value of 2000s and memory limit of 8 GiB. We run each experiment 5 times and report the median of these runs.

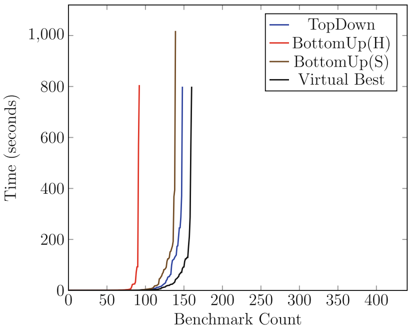


Fig. 3. Cactus plot of runtime. (Lower and to the right is better.)

For Sample Domains, the solvers performed similarly and cumulatively solved 15/17 benchmarks (Virtual Best). Top-down enumeration is a clear winner for Regular Expressions, with height-based bottom-up enumerator performing poorly because the solutions are typically narrow-but-tall. All solvers performed about equivalently on the Boolean benchmarks, although each solver solves a slightly different subset of the problems. For Bitvectors, the bottom-up height-based solver underperformed because these benchmarks have grammars with many productions per non-terminal, thus producing many programs at each height. However, size-based bottom-up enumeration could solve 5 problems that the top-down enumerator could not solve, and the top-down enumerator solved 7 that the bottom-up, size-based enumerator could not solve. Note that the Bitvector benchmarks have relational specifications and were solved with CEGIS, but for 19 benchmarks, the verifier failed to check a candidate program or generate counterexamples for at least one solver. For the 43 solved benchmarks, the verifier generated between 1 and 10 counterexamples (average 4.5), in less than 150 ms each (average 30 ms). The remaining 38 benchmarks generated up to 12 counterexamples before exceeding the timeout or memory limit. Our solver could not solve any Messy benchmarks: most are unrealizable (i.e., they have no solution) or use specifications that are hard to verify using KS2's SMT-based verifier. The Messy solver is particularly good at proving problems unrealizable, but it has not been ported to the SemGuS format and we cannot include it in our baseline.

In terms of enumeration throughput (enumerated programs per second), our solvers perform similarly, and they can enumerate up to 150,000 programs per second (average 33,000) for benchmarks for which verification is quick. The advantage of building and using executable semantics is obvious: if the logical verifier is instead called on each candidate, the throughput drops to at most 800 programs per second (average 175). On the benchmarks where solving with executable semantics and the logical verifier are both supported, the use of the executable semantics is on average 220 times faster than the logical verifier (geomean).

These results provide a baseline rate for future SemGuS solvers to be compared against; the advantage of simple enumerators is their raw speed.

5 Related Work

The syntax-guided-synthesis paradigm [1] has been successfully used in many applications, including invariant synthesis [8,18], and synthesis of rewrite rules and invertibility conditions [19,20]. Several efficient solvers are available for this format [2,13,24]. This effort has inspired several domain-specific extensions for domains that cannot be captured by standard SMT-LIB theories [21]. In contrast, this work develops a general framework for which these extensions can be expressed in a uniform way. Moreover, SemGuS allows one to define synthesis problems—e.g. for imperative programs—that cannot be captured in a natural way by an SMT theory. The syntax-guided-synthesis paradigm has been extended to signatures with oracles [14,22], or symbols whose semantics are given by user-provided binaries. In contrast, in SemGuS, the semantics of all symbols are fully expressed in the problem description.

Acknowledgements. The authors would like to thank Jinwoo Kim, for initial discussions about the SemGuS format; Wiley Corning, Rahul Krishnan, and Shaan Nagy, for code contributions to the SemGuS parser; Evan Geng, Jiangyi Liu, and Charlie Murphy for finding and reporting bugs; and, in addition to everyone previously listed, Kanghee Park, Anvay Grover, and all future contributors for providing SemGuS benchmarks.

Supported, in part, by a Microsoft Faculty Fellowship; a gift from Rajiv and Ritu Batra;and NSF under grants CCF-{1750965, 1918211, 2023222, 2211968, 2212558}. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design, pp. 1–8 (2013). https://doi.org/10.1109/FMCAD.2013.6679385
2. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) TACAS 2017, Part I. LNCS, vol. 10205, pp. 319–336. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_18
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
4. Center for High Throughput Computing: Center for high throughput computing (2006). https://doi.org/10.21231/GNT1-HW21, https://chtc.cs.wisc.edu/
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252. ACM (1977).https://doi.org/10.1145/512950.512973, https://doi.org/10.1145/512950.512973
6. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Log. Program. **13**(2&3), 103–179 (1992)
7. D'Antoni, L., Hu, Q., Kim, J., Reps, T.: Programmable program synthesis. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 84–109. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_4

8. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 259–277. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_14
9. Gulwani, S.: Synthesis from examples. In: WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings. vol. 10. Citeseer (2012)
10. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11, New York, NY, USA, pp. 62–73. Association for Computing Machinery (2011). https://doi.org/10.1145/1993498.1993506
11. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends® Program. Lang. **4**(1–2), 1–119 (2017)
12. Hu, Q., D'Antoni, L.: Syntax-guided synthesis with quantitative syntactic objectives. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 386–403. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_21
13. Huang, K., Qiu, X., Shen, P., Wang, Y.: Reconciling enumerative and deductive program synthesis. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15–20, 2020, pp. 1159–1174. ACM (2020). https://doi.org/10.1145/3385412.3386027, https://doi.org/10.1145/3385412.3386027
14. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010, pp. 215–224. ACM (2010). https://doi.org/10.1145/1806799.1806833
15. Kim, J., Hu, Q., D'Antoni, L., Reps, T.: Semantics-guided synthesis. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021)
16. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. SIGPLAN Not. **52**(3), 70–80 (2016). https://doi.org/10.1145/3093335.2993244
17. Li, X., Zhou, X., Dong, R., Zhang, Y., Wang, X.: Efficient bottom-up synthesis for programs with local variables. Proc. ACM Program. Lang. **8**(POPL) (2024). https://doi.org/10.1145/3632894
18. Miltner, A., Padhi, S., Millstein, T.D., Walker, D.: Data-driven inference of representation invariants. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15–20, 2020, pp. 1–15. ACM (2020). https://doi.org/10.1145/3385412.3385967
19. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part II. LNCS, vol. 10982, pp. 236–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_16
20. Nötzli, A., et al.: Syntax-guided rewrite rule enumeration for SMT solvers. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 279–297. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_20
21. Padhi, S., Polgreen, E., Raghothaman, M., Reynolds, A., Udupa, A.: The sygus language standard version 2.1. CoRR abs/2312.06001 (2023). https://doi.org/10.48550/ARXIV.2312.06001

22. Polgreen, E., Reynolds, A., Seshia, S.A.: Satisfiability and synthesis modulo oracles. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 263–284. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_13
23. Polozov, O., Gulwani, S.: FlashMeta: a framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 107–126 (2015)
24. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: cvc4sy: smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 74–83. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_5
25. 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) (2021). https://doi.org/10.1145/3434304

Relational Synthesis of Recursive Programs via Constraint Annotated Tree Automata

Anders Miltner[1](✉), Ziteng Wang[2], Swarat Chaudhuri[2], and Isil Dillig[2]

[1] Simon Fraser University, Burnaby, Canada
miltner@cs.sfu.ca
[2] University of Texas at Austin, Austin, USA
{ziteng,swarat,isil}@cs.utexas.edu

Abstract. In this paper, we present a new synthesis method based on the novel concept of a *constraint annotated tree automaton (CATA)*. A CATA is a variant of a finite tree automaton (FTA) where the acceptance of a term by the automaton is conditioned upon the logical satisfiability of a formula. In the context of program synthesis, CATAs allow the construction of a more precise version space than FTAs by ruling out programs that make inconsistent assumptions about the unknown semantics of functions under synthesis. We apply our proposed algorithm to synthesizing recursive (or mutually recursive) procedures from relational specifications and demonstrate that our method allows solving synthesis problems that are beyond the scope of existing approaches.

1 Introduction

Program synthesis, the task of automatically generating programs that meet a given specification, has found numerous applications, including both user-facing domains like data science [15,16,47,48] as well as software engineering tasks [21,32,36,37,40]. Program synthesizers can be classified among two dimensions, namely (1) whether they target a domain-specific or general programming language, and (2) what type of specification they require. Many synthesizers targeting end-users utilize domain-specific languages and only require informal specifications such as input-output examples or natural language [4,9,20,52]. In contrast, program synthesizers targeting developers tend to require formal specifications and need to handle a richer set of language features [33,39,41,49].

In the context of synthesizing general-purpose programs from logical specifications, two aspects have proven to be particularly challenging:

- **Recursion:** Problems that require synthesizing recursive, or mutually recursive, functions have proven to be particularly difficult to solve. Despite recent progress in this area [33,53], synthesizers that tackle recursive functions are not as effective as those that target domain-specific languages.
- **Relational specifications:** With the exception of one prior research effort [51], most synthesizers do not handle *relational specifications*. However, in practice, relational specifications are particularly relevant: for example,

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 41–63, 2024.
https://doi.org/10.1007/978-3-031-65633-0_3

parametrized unit tests [13,44] and *property-based tests* [11,18,28], which are becoming increasingly more popular, are, in essence, relational specifications.

While prior research has tried to tackle each of these problems in isolation, there is no prior work that has attempted to solve synthesis problems that involve *both* recursive procedures and relational specifications. In this paper, we ask the question, "Is it possible to synthesize recursive, or even mutually recursive, functions from relational specifications?" For example, given the specification:

$$\mathsf{even}(0) \land \forall x.\ (\mathsf{even}(x) \Leftrightarrow \neg\mathsf{odd}(x) \land \mathsf{even}(x) \Rightarrow \mathsf{odd}(x+1))$$

can we generate correct implementations of both `even` and `odd`? This task is quite difficult, as it requires simultaneously solving the challenges introduced by recursion and relational specifications. Intuitively, handling recursion is hard because the synthesizer does not know the semantics of terms that involve recursive calls to the function being synthesized. Similarly, relational specifications pose a significant challenge because such specifications do not constrain the input-output behavior of any *individual function*. For these reasons, the search space for the underlying synthesis problem becomes enormous, and standard techniques that facilitate search space pruning become insufficient.

Interestingly, some of the prior research efforts [33,51] on relational and recursive synthesis adopt roughly the same solution: they construct a finite tree automaton (FTA) whose language *over-approximates* the space of programs consistent with the specification. The key idea is to allow *non-deterministic* FTA transitions that encode uncertainty about the semantics of the function being synthesized. However, since the resulting version space is over-approximate, these techniques need to combine FTA construction with backtracking search.

Given the similarity between these two techniques that address two (apparently orthogonal) challenges, one might be tempted to ask: *"Can we use the same idea to solve synthesis problems that involve both relational specifications* and *that also require synthesizing recursive procedures?"* In principle, the answer to this question is "yes"; however, as we show experimentally, the resulting technique does not yield an effective solution in practice.

The main contribution of this paper is a more effective approach for synthesizing recursive programs from relational specifications. Our method is based on the novel concept of *constraint annotated tree automaton (CATA)*, a new type of FTA where non-deterministic transitions are constrained by logical formulae in a first-order theory. Similar to a standard FTA, a necessary condition for accepting a tree T is to find a run of the automaton on T that ends in an accepting state. However, because transitions of a CATA are only valid under certain conditions, a run of the automaton also induces an *acceptance constraint*, which must be logically satisfiable in order for that run to be valid. Intuitively, CATAs offer a more effective synthesis methodology because their acceptance condition allows us to build an *exact*, rather than *over-approximate*, version space with acceptable overhead. Furthermore, by leveraging an SMT solver to check the acceptance condition of the CATA, we can avoid the need for explicit backtracking search and can instead piggyback on all research results underlying modern SMT solvers.

In addition to proposing the concept of CATAs and showing how they can be used for synthesis, another key contribution of this paper is a *goal-directed* approach for CATA construction that exploits the specific problem instance at hand. In particular, a naive synthesis approach based on CATAs would require constructing multiple CATAs for different sub-terms in the specification and then taking their intersection. However, as is the case with any type of automaton, intersection is an expensive operation, so a synthesis algorithm that requires many intersections is unlikely to scale. Our method addresses this problem by proposing a more efficient algorithm that minimizes the number of automaton intersections required to create a precise version space.

We have implemented our proposed approach in a new tool called CONTATA and evaluate it on a suite of synthesis benchmarks involving recursion and specified by a relational specification. Our evaluation shows the advantages of our approach over prior techniques that first build a non-deterministic FTA and then perform backtracking search.

2 Motivating Example

In this section, we give an overview of the technique on an extended example where the goal is to automatically synthesize two functions:

```
evens : list a -> list a      odds : list a -> list a
```

Given a list, the `evens` function is expected to return all elements at even indices, and the `odds` function should return elements at odd indices. For example, we have `evens([3,8,2]) = [3,2]` and `odds([3,8,2]) = [8]`. The code for `evens` and `odds` is given below:

```
evens(x) = match x with        odds(x) = match x with
| [] -> []                     | [] -> []
| h:t -> h:odds(t)             | _:t -> evens(t)
```

Note that the `evens` and `odds` functions are mutually recursive: The `evens` function starts by extracting the head of the list, then prepends it to the result of calling `odds` on the tail. The `odds` function skips the first element in its input list, and merely returns the result of calling `evens` on the tail.

Specifying the Task. Since our goal is to automatically synthesize these functions, we first need a specification for this task. For the purposes of this example, suppose that the user provides the following specification:

$$\forall x. \forall xs.\ \mathsf{evens}(xs) = \mathsf{odds}(x : xs) \wedge \forall x. \forall y. \forall z. \mathsf{evens}([x, y, z]) = [x, z]$$

The first conjunct describes the *relationship* between evens and odds, namely, that the result of calling odds on $x : xs$ should be the same as calling on evens on xs. The second part of the specification provides a *symbolic* input-output example. Note that such relational specifications naturally arise in many contexts, including data structure specifications [36,37], parametrized unit tests [13], and 2-safety properties like commutativity [42].

Prior Work. Before describing our technique, we first briefly explain how prior work deals with relational specifications and recursion. First, let us assume that the universal quantifiers in the specification have been instantiated via a standard *counterexample-guided inductive synthesis (CEGIS)* loop. In particular, suppose we have the following *ground formula* during some iteration of CEGIS:

$$\mathsf{evens}(0:[1,2]) = \mathsf{odds}([1,2]) \wedge \mathsf{evens}([0,1,2]) = [0,2] \tag{1}$$

Given such a specification φ for counterexamples $I_1, \ldots, I_n$, a common theme behind prior work [33,49,50] is to construct a *version space* in the form of a *finite tree automaton (FTA)* that represents the space of all programs (up to a bounded depth) that are consistent with φ. Specifically, these techniques construct an automaton $\mathcal{A}_i$ for each counterexample I_i and then use FTA intersection to handle the set of all counterexamples. Finally, states that satisfy the specification φ are marked as final, so any tree accepted by the resulting FTA is a solution.

Constructing such an FTA is straightforward when the semantics of all expressions are known: Since the automaton states represent *constants* and the transitions correspond to operators/functions, we can simply add new transitions and nodes to the FTA using the operational semantics of the language. As an example, consider a DSL operator f that takes as input two integers x and y and produces $2x + y$, and let q_x be the automaton state representing constant x. Since the semantics of f are known, the FTA would contain the transition $f(q_1, q_2) \rightarrow q_4$ since $f(1, 2)$ is equal to 4.

Unfortunately, recursive procedures and relational specifications pose a significant challenge for FTA construction: When synthesizing a recursive function f, the implementation can recursively call f, but the semantics of f are not yet known, as f is currently under construction. Similarly, when dealing with relational specifications, the implementation of a function f could call another function g, but the semantics of g are also not yet known. To make matters worse, relational specifications constrain the *joint* behavior of multiple functions, so, when constructing the FTA for an *individual* function, we cannot even determine which FTA states should be marked as final.

Limitations of Prior Work. Existing techniques [33,51] deal with this challenge by adding *non-deterministic transitions* to the FTA. In particular, given FTA states representing values $c_1, \ldots, c_n$ and an n'ary function f that has yet to be synthesized, the idea is to add a transition of the form $f(c_1, \ldots, c_n) \rightarrow c$ as long as the formula $f(c_1, \ldots, c_n) = c$ is *consistent* with the specification. However, since there are many such output values c that are consistent with the specification, this introduces a high degree of non-determinism. Furthermore, relational specifications also introduce non-determinism with respect to final states, so the resulting FTA is very *over-approximate* — that is, the ground truth program is accepted by the FTA, but not every program accepted by the FTA satisfies the specification. As a result, existing techniques [33,51] combine FTA construction with backtracking search to look for a valid solution. However, if the synthesis task involves *both* recursion *and* relational specifications, the resulting FTA becomes *so* over-approximate that performing backtracking search over this space of programs is no longer feasible.

Insight Behind Our Approach. Our approach is motivated by the following observation about the shortcoming of prior techniques: *Many programs accepted by the over-approximate FTA make inconsistent assumptions about the unknown semantics of functions being synthesized.* For example, consider the following incorrect solution for the evens function for our running example:

```
evens(x) = match x with
| [] -> []
| h:t -> odds(t)++odds(t)
```

This program must be incorrect with respect to the specification (Eq. 1) even if we know *absolutely nothing* about the implementation of odds: The only way this program can return [0,2] on input list [0,1,2] is if the first call to odds on [1,2] returns [0] and the second call returns [2] on the same input list. But, assuming that odds is deterministic, this is clearly infeasible, as we cannot have odds([1,2]) = [0] and odds([1,2]) = [2] at the same time!

However, prior techniques construct a version space that includes this spurious program: Since the specification does not constrain the behavior of odds([1,2]) in any way, they would allow *any* transition from odds([1,2]) to any possible automaton state, including both [0] as well as [2]. Unfortunately, this leads to many spurious programs, including the "obviously wrong" implementation of evens from above.

In this paper, we show how to construct the version space in such a way that such *inconsistent programs* are never part of it. Our key idea is to qualify transitions in the FTA by *logical formulas* that indicate necessary conditions for a transition to be valid. We refer to such an FTA as a *Constraint Annotated Tree Automaton (CATA)* due to presence of constraints on its transitions. Then, a given tree will *only* be accepted by the CATA if there exists a run of the CATA that *both* ends in an accepting state *and* does *not* make inconsistent assumptions. However, because the evens implementation above makes inconsistent assumptions about the I/O behavior of odds, our approach can immediately rule it out.

3 Preliminaries

A *finite tree automaton* is a type of state machine that accepts trees rather than strings. More formally, FTAs are defined as follows:

Definition 1 (FTA). *A (bottom-up) finite tree automaton (FTA) over a finite alphabet Σ is a tuple $\mathcal{A} = (Q, Q_f, \Delta)$ where Q is a finite set of states, $Q_f \subseteq Q$*

is a set of final states, and Δ is a set of transitions (rewrite rules) of the form $f(q_1, \dots, q_n) \rightarrow q$ where $q, q_1, \dots, q_n \in Q$ and $f \in \Sigma$.

Each symbol in the alphabet Σ has an arity (rank), and terms of arity k are denoted Σ_k. Each ground term t can be represented in terms of its syntax tree (n, V, E) with root node n, vertices V, and edges E; hence, we use "tree" and "term" interchangeably. We say that a tree t is accepted by an FTA if we can rewrite t to some state $q \in Q_f$ using transitions Δ. The language of an FTA $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, includes all ground terms that $\mathcal{A}$ accepts.

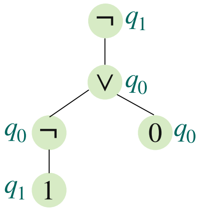


Fig. 1. Tree representing $\neg(\neg 1 \vee 0)$.

Example 1. Consider the FTA $\mathcal{A}$ with states $Q = \{q_0, q_1\}$, $\Sigma_0 = \{0, 1\}$, $\Sigma_1 = \{\neg\}$, $\Sigma_2 = \{\vee\}$, final states $Q_f = \{q_1\}$, and the transitions Δ:

$$\begin{array}{llll} 1 \rightarrow q_1 & 0 \rightarrow q_0 & \vee(q_0, q_0) \rightarrow q_0 & \vee(q_0, q_1) \rightarrow q_1 \\ \neg(q_0) \rightarrow q_1 & \neg(q_1) \rightarrow q_0 & \vee(q_1, q_0) \rightarrow q_0 & \vee(q_1, q_1) \rightarrow q_1 \end{array}$$

This FTA accepts propositional logic formulas that evaluate to *true*. For instance, Fig. 1 shows the tree for formula $\neg(\neg 1 \vee 0)$ where each sub-term is annotated with its state on the right. This formula is accepted by $\mathcal{A}$ because the rules in Δ "rewrite" the input to state q_1, which is a final state.

Definition 2 (Accepting run). *An* accepting run *of an FTA $\mathcal{A} = (Q, Q_f, \Delta)$ is a pair (t, L) where $t = (n_r, V, E)$ is a term that is accepted by $\mathcal{A}$ and L is a mapping from each node in V to an FTA state such that (1) $L(n_r) \in Q_f$; (2) If n has children $n_1, \dots, n_k$ such that $L(n) = q$ and $L(n_1) = q_1, \dots, L(n_k) = q_k$, then $\mathsf{Label}(n)(q_1, \dots, q_k) \rightarrow q$ is a transition in Δ.*

In other words, an accepting run labels each tree node with an automaton state.

Example 2. Let L be the mapping that assigns each node of the tree t in Fig. 1 to the state written next to it. Then, (t, L) is an accepting run for Example 1.

4 Constraint Annotated Tree Automata

In this section, we introduce the concept of *Constraint Annotated Tree Automata (CATA)*, which forms the basis of the synthesis algorithm described in the next section.

Definition 3 (CATA). *Let Σ be a finite alphabet and $\mathcal{T}$ be a decidable first-order theory ($\mathcal{T}$ may use symbols from Σ as well as additional symbols). A constraint annotated tree automaton (CATA) over Σ and $\mathcal{T}$ is a tuple $\mathcal{A}_{\mathcal{T}} = (Q, Q_f, \Delta)$ where:*

- *Q is a finite set of states*
- *Q_f is a mapping from states to their acceptance condition, which is a formula in theory $\mathcal{T}$*

- *$\Delta \subseteq \Sigma \times Q^* \times \mathcal{T} \times Q$ is a set of transitions of the form $\ell(q_1, \ldots, q_n) \rightarrow_\varphi q$ where $q, q_1, \ldots, q_n \in Q$ and $\ell \in \Sigma$ and $\varphi \in \mathcal{T}$.*

At a high level, a CATA differs from an FTA in two ways: First, the acceptance condition Q_f is a mapping from each state to a formula φ in theory $\mathcal{T}$. In other words, unlike the standard FTA where Q_f maps each state to a boolean constant, the CATA maps each state to a first-order formula under which that state is accepting. Second, the transitions in a CATA are qualified by formulas in a first-order theory $\mathcal{T}$. In particular, a transition $f(q_1, \ldots, q_n) \rightarrow_\varphi q$ can rewrite $f(q_1, \ldots, q_n)$ to q only if the transition condition φ is satisfied.

Next, we define a *run* of a CATA. Recall that an FTA run consists of a tree and mapping L from nodes of that tree to states in the automaton. For CATAs, we generalize this notion of a run by having two types of mappings: One maps each tree node to a state, and another maps each node to a formula. More formally, we have:

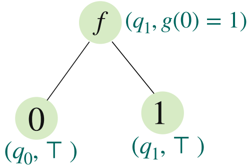


Fig. 2. CATA run on $f(0, 1)$

Definition 4 (CATA run). *Let $\mathcal{A}_\mathcal{T} = (Q, Q_f, \Delta)$ be a CATA over alphabet Σ and theory $\mathcal{T}$. A run of this CATA is a triple $r = (t, L_Q, L_\varphi)$ consisting of:*

1. *A tree $t = (n_r, V, E)$, where each $n \in V$ is labeled by an element of Σ*
2. *A function $L_Q : V \rightarrow Q$ mapping the nodes of t to the states of $\mathcal{A}_\mathcal{T}$*
3. *A function $L_\varphi : V \rightarrow \mathsf{Formulas}(\mathcal{T})$ mapping nodes of t to formulas over theory $\mathcal{T}$ such that if n has label f and children $n_1 \ldots n_k$ then there is a transition*

$$(f(L_Q(n_1), \ldots, L_Q(n_k)) \rightarrow_{L_\varphi(n)} L_Q(n)) \ \in \Delta$$

In other words, a CATA run not only labels tree nodes with states but also with the conditions under which the corresponding transition is legal. We say that a run (t, m_Q, m_Φ), *ends* at state q if $m_Q(t.\mathsf{Root}) = q$.

Example 3. Consider following CATA $\mathcal{A}_\mathcal{T}$ over the combined theory of uninterpreted functions and integers: $\mathcal{A}_\mathcal{T}$ has states $Q = \{q_0, q_1\}$, $\Sigma_0 = \{0, 1\}$, $\Sigma_2 = \{f\}$, final states $Q_f = \{q_0 \mapsto \bot, q_1 \mapsto g(0) < 1\}$, and the following transitions Δ:

$$1 \rightarrow_\top q_1 \quad 0 \rightarrow_\top q_0 \quad f(q_0, q_0) \rightarrow_{g(0)=0} q_0 \quad f(q_0, q_1) \rightarrow_{g(0)=1} q_1 \; f(q_1, \_) \rightarrow_\top q_1$$

Here, $\top, \bot$ denote true and false respectively. Figure 2 shows a run of this CATA, with L_Q, L_φ shown as a pair (q, φ) next to that node.

Definition 5 (Run Assumptions). *Given a run $r = (t, L_Q, L_\varphi)$, the* assumptions *of the run are defined as follows:*

$$\mathsf{Assumptions}(r) = \bigwedge_{n \in \mathsf{Nodes}(t)} L_\varphi(n)$$

For example, the assumptions for the run shown in Fig. 2 is just $g(0) = 1$.

Definition 6 (Accepting run). *Given a run $r = (t, L_Q, L_\varphi)$ of CATA $\mathcal{A}_\mathcal{T}$, the* acceptance condition *of the run is the conjunction of the assumptions of r and the formula corresponding to the root node, i.e.,*

$$\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, r) = \mathsf{Assumptions}(r) \wedge Q_f(L_Q(\mathsf{Root}(t)))$$

A run r is accepting *if there exists a model $\mathcal{M}$ such that $\mathcal{M} \models \mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, r)$. We refer to such a model $\mathcal{M}$ as a* witness *for run r.*

Example 4. The run from Example 3 is not accepting because the assumptions made by the run (namely, $g(0) = 1$) contradict the acceptance condition for node q_1, which is $g(0) < 1$.

Recall that an FTA accepts a tree t if there exists a corresponding accepting run for t. We generalize this notion to CATAs as follows:

Definition 7 (Accepted tree). *$\mathcal{A}_\mathcal{T}$ accepts tree t under witness $\mathcal{M}$, denoted $(t, \mathcal{M}) \models \mathcal{A}_\mathcal{T}$, if there is an accepting run $r = (t, L_Q, L_\varphi)$ of $\mathcal{A}_\mathcal{T}$ with witness $\mathcal{M}$.*

Example 5. The tree shown in Fig. 2 would be accepting (with the same corresponding run from Fig. 2) if we change $Q_f(q_1)$ to $g(0) \geq 1$.

Next, we define the language of a CATA. Recall that the language of an FTA is the set of all trees it accepts. However, since a CATA accepts a tree only under certain conditions, the language of a CATA consists of pairs of trees along with their witnesses. More formally, we have:

$$\mathcal{L}(\mathcal{A}_\mathcal{T}) = \{(t, \mathcal{M}) \mid (t, \mathcal{M}) \models \mathcal{A}_\mathcal{T} \}$$

As mentioned earlier, synthesis approaches based on tree automata rely on a *product* operation, $\mathcal{A}^1_\mathcal{T} \times \mathcal{A}^2_\mathcal{T}$, that produces a new automaton $\mathcal{A}_\mathcal{T}$ such that $\mathcal{L}(\mathcal{A}_\mathcal{T}) = \mathcal{L}(\mathcal{A}^1_\mathcal{T}) \cap \mathcal{L}(\mathcal{A}^2_\mathcal{T})$. This operation is defined as follows for CATAs:

Definition 8 (Intersection). *Let $\mathcal{A}^1_\mathcal{T} = (Q_1, Q_{f_1}, \Delta_1)$ and $\mathcal{A}^2_\mathcal{T} = (Q_2, Q_{f_2}, \Delta_2)$ be two CATAs over the same underlying theory $\mathcal{T}$ and alphabet Σ. Then, the* product CATA *$\mathcal{A}^1_\mathcal{T} \times \mathcal{A}^2_\mathcal{T}$ is defined as (Q, Q_f, Δ) where:*

- $Q = Q_1 \times Q_2$
- $Q_f((q_1, q_2)) = Q_{f_1}(q_1) \wedge Q_{f_2}(q_2)$
- *Δ contains the transition $\ell((q_{11}, q_{21}), \ldots, (q_{1n}, q_{2n})) \rightarrow_{\varphi_1 \wedge \varphi_2} (q_1, q_2)$ iff $\ell(q_{11}, \ldots, q_{1n}) \rightarrow_{\varphi_1} q_1 \in \Delta_1$ and $\ell(q_{21}, \ldots, q_{2n}) \rightarrow_{\varphi_2} q_2 \in \Delta_2$*

Example 6. Suppose $\mathcal{A}^1_\mathcal{T}$ contains the transition $q_1 \rightarrow_{g(0) \geq 1} q_2$ and $\mathcal{A}^2_\mathcal{T}$ contains the transition $q_3 \rightarrow_{g(0) \neq 1} q_4$. Then, assuming both CATAs are over the combined theory of integers and uninterpreted functions, the product CATA would contain the transition $(q_1, q_3) \rightarrow_{g(0) > 1} (q_2, q_4)$.

Theorem 1. *Let $\mathcal{A}^1_\mathcal{T} = (Q_1, Q_{f_1}, \Delta_1)$ and $\mathcal{A}^2_\mathcal{T} = (Q_2, Q_{f_2}, \Delta_2)$ be two CATAs over theory $\mathcal{T}$ and alphabet Σ. Then, $\mathcal{L}(\mathcal{A}^1_\mathcal{T} \times \mathcal{A}^2_\mathcal{T}) = \mathcal{L}(\mathcal{A}^1_\mathcal{T}) \cap \mathcal{L}(\mathcal{A}^2_\mathcal{T})$*

4.1 CATA Operations for Synthesis

We now define CATA operations that our synthesis algorithm relies on.

Definition 9 (Accepting Runs of Tree). *Given a CATA $\mathcal{A}_\mathcal{T}$ and tree t, the accepting runs for t, denoted* $\mathsf{Runs}(\mathcal{A}_\mathcal{T}, t)$ *are:*

$$\mathsf{Runs}(\mathcal{A}_\mathcal{T}, t) = \{r = (t, L_Q, L_\varphi) \mid \mathsf{SAT}(\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, r))\}$$

In other words, the accepting runs of $\mathcal{A}_\mathcal{T}$ on tree t are those runs whose acceptance conditions are logically satisfiable. We can similarly define accepting runs for a state as all accepting runs that end in that state:

Definition 10 (Accepting Runs of State). *Given a CATA $\mathcal{A}_\mathcal{T}$ and state q, the accepting runs for q, denoted* $\mathsf{Runs}(\mathcal{A}_\mathcal{T}, q)$*, are:*

$$\mathsf{Runs}(\mathcal{A}_\mathcal{T}, q) = \{r = (t, L_Q, L_\varphi) \mid r \in \mathsf{Runs}(\mathcal{A}_\mathcal{T}, t) \wedge L_Q(\mathsf{Root}(t)) = q\}$$

Given a state q or tree t, we often need to compute the acceptance condition for that tree/state, which we define as follows:

Definition 11 (Acceptance Condition). *Given a CATA $\mathcal{A}_\mathcal{T}$ and state or tree x, the acceptance condition of x, denoted* $\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, x)$ *is:*

$$\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, x) = \bigvee_{r \in \mathsf{Runs}(\mathcal{A}_\mathcal{T}, x)} \mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, r)$$

Example 7. Consider $\mathcal{A}_\mathcal{T}$ defined in Example 3, and let t_1 be the tree in Fig. 2. We have $\mathsf{Runs}(\mathcal{A}_\mathcal{T}, t_1) = \varnothing$, and $\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, q_1) = g(0) < 1$.

Finally, the acceptance condition for the CATA, $\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T})$, is the disjunction of acceptance conditions over all states, and a *minimum accepted tree*, denoted $\mathsf{MinTree}(\mathcal{A}_\mathcal{T})$ is a minimum size tree accepted by the CATA.

5 Synthesis Algorithm

In this section, we first define our synthesis problem more precisely (Sect. 5.1) and then present the basic synthesis technique (Sect. 5.2). However, since the basic algorithm ends up requiring too many CATA intersections, it does not lend itself to a practical implementation. In Sect. 5.3, we show how to construct the CATA in a goal-directed way to minimize the number of CATA intersections.

5.1 Problem Statement

Definition 12 (Relational spec). *Let $\mathcal{F} = \{f_1, \ldots, f_n\}$ be a set of function symbols. A* relational specification *over $\mathcal{F}$ is a formula of the form $\forall \overline{x}.\Phi(\overline{x})$ where Φ is a quantifier-free formula over some theory $\mathcal{T}$ and the only function symbols in Φ belong either in $\mathcal{F}$ or to the signature of $\mathcal{T}$.*

$$\begin{array}{lllllll}
P & ::= & \texttt{f}(x)\ \texttt{=}\ e\ \texttt{;}\ P & & & & \\
e & ::= & x & \mid\ e_1\ e_2 & \mid\ \texttt{()} & \mid\ \texttt{fst}\ e & \mid\ \texttt{snd}\ e \\
 & \mid & \texttt{(}e_1\texttt{,}\ e_2\texttt{)} & \mid\ \texttt{inl}\ e & \mid\ \texttt{inr}\ e & \mid\ \texttt{unl}\ e & \mid\ \texttt{unr}\ e \\
 & \mid & \multicolumn{5}{l}{\texttt{switch}\ e_3\ \texttt{on}\ \texttt{inl}\ \_ \rightarrow e_1\ \texttt{,}\ \texttt{inr}\ \_ \rightarrow e_2} \\
v & ::= & \texttt{()} & \mid\ \texttt{(}v_1\texttt{,}\ v_2\texttt{)} & \mid\ \texttt{inl}\ v & \mid\ \texttt{inr}\ v &
\end{array}$$

Fig. 3. A functional ML-like language. Programs are comprised of a list of mutually recursive function definitions.

$$\frac{P \vdash e \Downarrow v; \varphi_1 \qquad f(x) = e' \in P \qquad P \vdash e'[v/x] \Downarrow v'; \varphi_2}{f\ e \Downarrow v'; \varphi_1 \wedge \varphi_2 \wedge f(v) = v'}$$

$$\frac{}{P \vdash \texttt{()} \Downarrow \texttt{()}; \top} \qquad \frac{P \vdash e_1 \Downarrow v_1; \varphi_1 \qquad P \vdash e_2 \Downarrow v_2; \varphi_2}{P \vdash (e_1, e_2) \Downarrow (v_1, v_2); \varphi_1 \wedge \varphi_2} \qquad \frac{P \vdash e \Downarrow (v_1, v_2); \varphi}{P \vdash \texttt{fst}\ e \Downarrow v_1; \varphi}$$

$$\frac{P \vdash e \Downarrow (v_1, v_2); \varphi}{P \vdash \texttt{snd}\ e \Downarrow v_2; \varphi}$$

$$\frac{P \vdash e \Downarrow v; \varphi}{P \vdash \texttt{inl}\ e \Downarrow \texttt{inl}\ v; \varphi} \qquad \frac{P \vdash e \Downarrow v; \varphi}{P \vdash \texttt{inr}\ e \Downarrow \texttt{inr}\ v; \varphi} \qquad \frac{P \vdash e \Downarrow \texttt{inl}\ v}{P \vdash \texttt{unl}\ e \Downarrow v; \varphi}$$

$$\frac{P \vdash e \Downarrow \texttt{inr}\ v; \varphi}{P \vdash \texttt{unr}\ e \Downarrow v; \varphi} \qquad \frac{P \vdash e_3 \Downarrow \texttt{inl}\ v_3; \varphi_1 \qquad P \vdash e_1 \Downarrow v_1; \varphi_2}{P \vdash \texttt{switch}\ e_3\ \texttt{on}\ \texttt{inl}\ \_ \rightarrow e_1\ \texttt{inr}\ \_ \rightarrow e_2 \Downarrow v_1; \varphi_1 \wedge \varphi_2}$$

$$\frac{P \vdash e_3 \Downarrow \texttt{inr}\ v_3; \varphi_1 \qquad P \vdash e_2 \Downarrow v_2; \varphi_2}{P \vdash \texttt{switch}\ e_3\ \texttt{on}\ \texttt{inl}\ \_ \rightarrow e_1\ \texttt{inr}\ \_ \rightarrow e_2 \Downarrow v_2; \varphi_1 \wedge \varphi_2}$$

Fig. 4. Program Semantics. The symbols e range over expressions, v range over values, and φ range over formulas in the the theory of uninterpreted functions.

Relational specifications allow jointly constraining the behavior of multiple functions to be synthesized. For instance, examples of relational specifications include $\forall x. f(g(x)) = x$ (i.e., f and g are inverses) or $\forall x, y. f(x, y) = g(y, x)$.

In this paper, we consider the problem of synthesizing programs in an ML-like functional programming language with sums, products, and mutual recursion. Figure 3 shows the core subset of this programming language. A program in this language consists of one or more function definitions, and the body of each function is an expression e, which includes function applications, constructors (`inl`,`inr` for sums and (e_1, e_2) for products), destructors (`unl`,`unr` for sums, `fst`,`snd` for products) and `switch` statements for pattern matching. Figure 4 presents the semantics of this language using the notation $P \vdash e \Downarrow v; \varphi$, meaning that, under the function definitions given by P, expression e evaluates to value v and φ is a formula that tracks the results of procedure calls made by e.[1]

[1] These *instrumented semantics* for recording results of function calls will be useful for CATA construction in Sect. 5.

Given a program P in this language defining functions $\mathcal{F}'$ and a relational specification ψ over functions $\mathcal{F} \subseteq \mathcal{F}'$, we write $P \models \psi$ if the implementation of P satisfies specification ψ. Since the focus of this paper is not verification, we assume access to an oracle for checking $P \models \psi$. Given n programs $P_1, \ldots, P_n$ implementing different functions, we also use the notation $(P_1, \ldots, P_n) \models \psi$ to denote that these programs collectively satisfy specification ψ.

Definition 13 (Solution to synthesis problem). *Let ψ be a relational specification over functions $\mathcal{F} = \{f_1, \ldots, f_n\}$. A solution to this synthesis problem is a mapping from each $f_i \in \mathcal{F}$ to a program such P_i such that $(P_1, \ldots, P_n) \models \psi$.*

Since our top-level approach is based on counterexample-guided inductive synthesis (CEGIS) [2], it suffices to have a synthesis procedure that can only deal with *ground relational specifications*. In particular, a ground relational specification over $\mathcal{F}$ cannot contain any variables, either free or bound, besides those in $\mathcal{F}$. In the remainder of this section, we therefore only consider ground specifications and assume that quantifiers are handled using the standard CEGIS framework.

Assumptions. Our synthesis algorithm makes a few important, but realistic assumptions, that we rely on in the remainder of this section. First, we assume that there is a pre-defined partial order relation $\preceq$ between constants in the underlying language (e.g, $1 \prec 3, [1, 2] \prec [1, 2, 3]$ etc.). This partial ordering must be well founded and must not have infinite fan-out to ensure termination. Second, we assume that, when function f is called on some input x, other calls that f makes can only involve values satisfying $y \prec x$. This common assumption [1,24,33,35] is required to ensure that recursive calls are well-founded. Finally, to further simplify presentation, we assume that the language admits a finite number of constants; however, our implementation does not make this assumption (see Sect. 6).

5.2 Basic Synthesis Algorithm

In this section, we describe our CATA-based synthesis procedure. While this algorithm exposes how CATAs are used for synthesis, it does not lend itself to a practical implementation due to its eager nature. We first present the basic algorithm and then explain how to make it more goal-directed in the next section.

Our basic synthesis procedure is summarized in Algorithm 1 and takes two inputs, the set $\mathcal{F}$ of functions to synthesize and a ground relational specification over $\mathcal{F}$. The algorithm computes a solution for each $f \in \mathcal{F}$ in three steps:

- First, for each possible input c of $f \in \mathcal{F}$, the algorithm builds a CATA $\Pi(f, c)$ that encodes how different implementations of f can behave on input c (lines 2–5). In particular, for a possible output c', $\mathsf{AcceptCond}(\Pi(f, c), c')$ gives the conditions under which f can produce c' on input c.
- Next, in lines 6–11, the algorithm builds a CATA, $\Omega(f)$, encoding all possible input-output behaviors of different implementations of f. This is done

```
    input: Relational ground specification $\psi$ and set of functions $\mathcal{F}$ to synthesize
    output: Solution $\Psi$ mapping each function to its implementation
 1: procedure SYNTHESIZE($\psi, \mathcal{F}$)
    ▷ Create initial CATAs for each possible input for each function
 2:     $\Pi \leftarrow \varnothing$   ▷ Mapping from each function and constant to corresponding CATA
 3:     for each $c \in \mathcal{C}$ do
 4:         for each $f \in \mathcal{F}$ do
 5:             $\Pi(f, c) \leftarrow \text{CREATECATA}(f, c, \psi)$
    ▷ Obtain CATA per function and strengthen specification
 6:     $\phi \leftarrow \psi$                              ▷ Initialization for strengthened spec
 7:     $\Omega \leftarrow \varnothing$                        ▷ Mapping from each function to its CATA
 8:     for each $f \in \mathcal{F}$ do
 9:         $\mathcal{A} \leftarrow \Pi(f, c_1) \times \ldots \times \Pi(f, c_n)$
10:         $\phi \leftarrow \phi \wedge \mathsf{AcceptCond}(\mathcal{A})$
11:         $\Omega(f) \leftarrow \mathcal{A}$
    ▷ Synthesize implementation of each function
12:     $\Psi \leftarrow \varnothing$          ▷ Solution mapping from each function to its implementation
13:     for each $f \in \mathcal{F}$ do
14:         $\mathcal{A} \leftarrow \mathsf{StrengthenSpec}(\Omega(f), \phi)$          ▷ Update acceptance condition
15:         $P \leftarrow \mathsf{MinTree}(\mathcal{A})$
16:         $\phi \leftarrow \phi \wedge \mathsf{AcceptCond}(\mathcal{A}, P)$
17:         $\Psi(f) \leftarrow P$
18:     return $\Psi$
```

Algorithm 1: Basic synthesis procedure. $\mathcal{C}$ denotes the set of constants in the programming language, sorted to be consistent with partial order $\preceq$

by using the CATA product operation defined in Sect. 4. Lines 6–11 also strengthen the initial specification ψ to a stronger condition ϕ by taking into account the acceptance condition of the constructed CATAs.

- Finally, lines 12–17 of the algorithm use the per-function CATA $\Omega(f)$ and the strengthened specification ϕ to obtain a concrete implementation of f. To that end, the algorithm first strengthens the acceptance condition of each state by conjoining the global specification ϕ; it then obtains a minimum tree P accepted by the resulting automaton $\mathcal{A}$. This tree corresponds to the synthesized implementation for f, and the algorithm moves on to the next function after strengthening the global specification ϕ to be consistent with the acceptance condition for P.

The interesting aspect of this algorithm is that it is guaranteed to find a set of programs that collectively satisfy the relational specification without *any* need for backtracking search. Intuitively, there are three key reasons for this:

1. First, when building the CATA for each (f, c) pair, the CREATECATA procedure (formalized as inference rules in Fig. 5) generates constraints under which each transition is valid. In particular, consider the FUNCTION CALL rule in Fig. 5. When adding the transition $g(c) \rightarrow c'$, $\mathsf{AcceptCond}(\Pi(g, c), c')$

$$\textsc{Init}\quad \frac{}{q_{v_{in}} \in Q \qquad \mathtt{x} \to q_{v_{in}} \in \Delta} \qquad\qquad \textsc{Final}\quad \frac{q_v \in Q}{Q_f(q_v) = \psi \wedge f(v_{in}) = v}$$

$$\textsc{Unit}\quad \frac{}{q_{()} \in Q \qquad () \to q_{()} \in \Delta} \qquad\qquad \textsc{Pair}\quad \frac{q_{v_1} \in Q \qquad q_{v_2} \in Q}{q_{v_1,v_2} \in Q \qquad (\,\cdot\,,\,\cdot\,)(q_{v_1}, q_{v_2}) \to q_{v_1,v_2} \in \Delta}$$

$$\textsc{Fst}\quad \frac{q_{v_1,v_2} \in Q}{q_{v_1} \in Q \qquad \mathtt{fst}(q_{v_1,v_2}) \to q_{v_1} \in \Delta} \qquad\qquad \textsc{Inl}\quad \frac{q_v \in Q}{q_{\mathtt{inl}\ v} \in Q \qquad \mathtt{inl}(v) \to q_{\mathtt{inl}\ v} \in \Delta}$$

$$\textsc{Uneval}\quad \frac{}{\bot \in Q} \qquad \textsc{Uneval Prod}\quad \frac{\ell \in \Sigma}{\bot \in Q \qquad \ell(\bot, \ldots, \bot) \to \bot} \qquad \textsc{Switch Left}\quad \frac{q_{\mathtt{inl}\ v_3} \in Q \qquad q_{v_1} \in Q}{\mathtt{switch}(q_{\mathtt{inl}\ v_3}, q_{v_1}, \bot) \to q_{v_1} \in \Delta}$$

$$\textsc{Function Call}\quad \frac{q_v \in Q \qquad \mathcal{A}_{\mathcal{T}} = \text{CATA}(g, v, \psi) \qquad \begin{array}{c} v \prec v_{in} \\ q_{v'} \in \mathsf{States}(\mathcal{A}_{\mathcal{T}}) \end{array} \qquad \varphi = \mathsf{AcceptCond}(\mathcal{A}_{\mathcal{T}}, q_{v'})}{q_{v'} \in Q \qquad \mathtt{g}(q_v) \to_{\varphi} q_{v'} \in \Delta}$$

Fig. 5. Inference rules for $\textsc{CreateCATA}(f, v_{in}, \psi)$. CATA states correspond to constants in the language, and we write q_c to denote the state representing constant c. The state $\bot$ corresponds to the non-evaluated branch of a switch. The rules for Snd, Inr, and Switch Right are omitted for space reasons.

gives the exact conditions under which g will return c' on input c, and this is the case even when g is one of the functions being synthesized.

2. Second, the strengthened specification ϕ after lines 6–11 precisely encodes all possible *joint* behaviors of all functions to be synthesized. Thus, a model of ϕ corresponds to input-output behaviors of every $f \in \mathcal{F}$ that are both mutually consistent and that will also satisfy the relational specification. Conceptually, by sampling a model $\mathcal{M}$ of ϕ and plugging $\mathcal{M}$ into the transition and acceptance conditions of the CATAs, we can turn each CATA into an FTA and then obtain the solution by finding programs accepted by each FTA.
3. However, one problem with the above model-sampling approach is that it does not guarantee that the synthesized programs are small (e.g., the sampled model may only have very complex implementations). Thus, lines 12–17 of Algorithm 1 construct the model in a lazy way that guarantees minimality at each step. In particular, rather than obtaining a monolithic model of ϕ, the algorithm considers one function f at a time, strengthens its acceptance condition using ϕ, and then finds a minimum size accepting tree P for f. Since P induces certain assumptions on the other functions (or relies on certain assumptions being held), ϕ is gradually concretized (by strengthening it at line 16). Thus, the third step of the synthesis procedure can be viewed as incremental model construction for the formula ϕ obtained after step 2.

```
    input: Relational ground specification $\psi$ and set of functions $\mathcal{F}$ to synthesize
    output: Solution $\Psi$ mapping each function to its implementation
 1: procedure LazySynthesize($\psi, \mathcal{F}$)
    ▷ Initialization phase
 2:     $\Omega \leftarrow \{f \mapsto \mathcal{A}_{\mathcal{I}}^{\top} \mid f \in \mathcal{F}\}$          ▷ Mapping from functions to CATAs
 3:     $\Lambda \leftarrow \varnothing$    ▷ Mapping from each function to counterexamples that appear in $\psi$
    ▷ Iteratively refine CATAs until solution is found
 4:     while true do
            ▷ Initialization for this refinement iteration
 5:         $\phi \leftarrow \psi \wedge \bigwedge_{f \in \mathcal{F}} \mathsf{AcceptCond}(\Omega(f))$          ▷ Current global specification
 6:         $\Psi \leftarrow \varnothing$                   ▷ Mapping from functions to candidate solution
            ▷ Get candidate solution
 7:         for each $f \in \mathcal{F}$ do
 8:             $\mathcal{A} \leftarrow \mathsf{StrengthenSpec}(\Omega(f), \phi)$          ▷ Update acceptance condition
 9:             $\Psi(f) \leftarrow \mathsf{MinTree}(\mathcal{A})$
10:             $\phi \leftarrow \phi \wedge \mathsf{AcceptCond}(\mathcal{A}, \Psi(f))$
            ▷ Check if $\Psi$ is a valid solution
11:         $\theta \leftarrow \{\chi \mid (c', \chi) \in \mathsf{Eval}(\Psi(f), c), c \in \Lambda(f), f \in \mathcal{F}\}$
12:         if $\mathsf{SAT}(\psi \wedge \bigwedge_i \theta_i)$ then
13:             return $\Psi$                                    ▷ $\Psi$ is a valid solution
            ▷ Refinement phase
14:         $\gamma \leftarrow \mathsf{UnsatCore}(\psi \wedge \bigwedge_i \theta_i)$
15:         for each $f(c) \in \mathsf{Terms}(\gamma)$ do
16:             $\Omega(f) \leftarrow \Omega(f) \times \textsc{CreateCATA}(f, c, \psi)$
```

Algorithm 2: Lazy synthesis. $\mathcal{A}_{\mathcal{I}}^{\top}$ from line 2 denotes a CATA that accepts all terms, and Eval at line 11 refers to the instrumented semantics (Figure 4).

Theorem 2 (Soundness of synthesis). *If* Synthesize*$(\psi, \mathcal{F})$ returns Ψ such that $\Psi(f_i) = P_i$, then we have $(P_1, \ldots, P_n) \models \psi$, where $|\mathcal{F}| = n$.*

Theorem 3 (Completeness of synthesis). *Let ψ be a ground relational specification over functions $\mathcal{F}$. If there exists an implementation P_i for each $f_i \in \mathcal{F}$ such that $(P_1, \ldots, P_n) \models \psi$, then* Synthesize *will return a solution.*

5.3 Lazy Synthesis Algorithm

Despite exposing the core ideas underlying our approach, the synthesis algorithm described in Sect. 5.2 has two severe shortcomings that make it infeasible in practice: First, it considers all possible inputs, which may be very large or even infinite. Second, it eagerly performs CATA intersection, which is impractical due to the exponential blow-up in CATA size. To address these shortcomings, we now describe a *lazy* version of the previous synthesis algorithm that lends itself to a much more practical implementation.[2]

[2] We note that the eager algorithm as presented in Sect. 5.2 times out on all of our experimental benchmarks.

The lazy synthesis procedure is presented in Algorithm 2. As in the previous algorithm, the synthesis procedure maintains a mapping from each function $f \in \mathcal{F}$ to its corresponding CATA $\Omega(f)$. However, since $\Omega(f)$ is constructed lazily, $\mathsf{AcceptCond}(\Omega(f))$ *over-approximates* the possible input-output behaviors of f's implementations rather than characterizing them exactly. Thus, lines 4–16 of Algorithm 2 *iteratively refine* Ω as follows until a valid solution is found:

- It first computes the global specification ϕ (line 5) by conjoining all acceptance conditions of the current CATAs with the initial specification ψ.
- Next, it finds a solution Ψ consistent with each CATA and the global specification ϕ, exactly as done in Phase 3 of Algorithm 1 (lines 7–10).
- Then, it checks whether Ψ is a valid solution (lines 11–13). To do so, it executes the candidate implementations on all relevant inputs and tracks the observed input-output behaviors as a set of constraints θ (line 11). If the conjunction of all of these constraints and ψ is satisfiable, then Ψ is indeed a valid solution and is returned at line 13.
- Otherwise, the synthesis procedure obtains an unsat core γ of the resulting unsatisfiable constraint (line 14). Intuitively, if a term $f(c)$ appears in the unsat core, then the CATA for f does not adequately constrain the outputs of f on input c; hence, we must refine $\Omega(f)$ by constructing the CATA for f on this input. Thus, line 16 of Algorithm 2 lazily refines $\Omega(f)$ by considering inputs that appear in the unsat core rather than considering all inputs eagerly.

Our proposed lazy synthesis algorithm is also both sound and complete. The corresponding theorems and proofs are provided in the appendix in the full version of the paper.

Example 8. Consider the evens/odds example from §2. Initially, Ω maps both evens and odds to $\mathcal{A}_{\mathcal{I}}^{T}$, the automaton that accepts all terms for both evens and odds. Suppose, on line 9, Contata obtains $\mathsf{evens}(xs) = []$ and $\mathsf{odds}(xs) = []$ as the solution. Such a solution fails to pass the check on line 12 because it violates the specification that $\mathsf{evens}([x, y, z]) = [x, z]$.

Then, on line 14, Contata computes the unsat core to be $evens([0, 1, 2]) = []$. It now intersects a new CATA created from $evens([0, 1, 2])$ to the evens CATA, which constrains the output of evens.

In the beginning of the next iteration, Contata updates the global specification with the accept condition of the constrained automatas. It then pops another candidate program:

```
evens(l) = match l with
  | Nil -> Nil
  | Cons (h,t) -> odds(t)
```

Thus, $\mathsf{evens}(l)$ relies on $\mathsf{odds}([1, 2]) = [1, 2]$. But the odds automaton is unconstrained, and thus will simply return $[]$. So the unsat core will be $\mathsf{odds}([1, 2]) = []$, and so the automaton for $[1, 2]$ would then be intersected with the current odds automaton. This process will continue until eventually the algorithm is able to find a program that relies on valid assumptions.

Table 1. Statistics about the benchmark set

Benchmark type	Count	Avg. soln size	Example
Mutual recursion (MR)	7	31.0	Test if input is even or odd
Recursive comparators (RC)	7	64.3	Check equality of int-tuple list
Partial data structures (PDS)	12	33.6	Binary tree removal
Stack Overflow (SO)	4	45.5	Reverse a list twice

6 Implementation

We have implemented the proposed algorithm in a new tool called Contata, which is written in OCaml using Z3 for discharging satisfiability queries. In this section, we briefly discuss some implementation details and optimizations elided in the main technical section.

Incremental Search. To simplify technical presentation, earlier sections assume that we can build a CATA representing the space of *all* programs consistent with the specification. However, since this space can be very large (or even infinite), Contata builds CATAs of increasing size. In particular, Contata first builds a CATA of size k, increasing the CATA size to $k+1$ if the algorithm fails to find a solution within that search space.

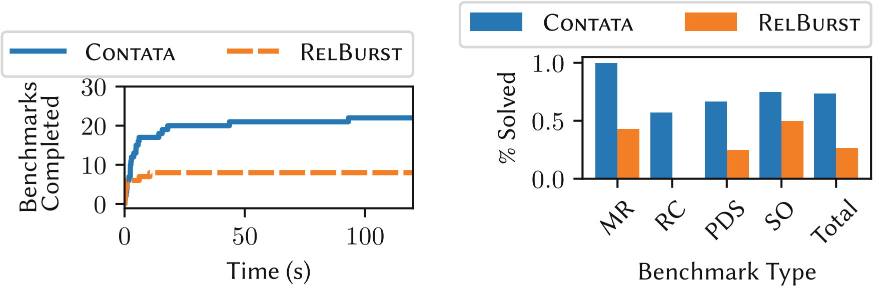
Optimizations. The implementation of Contata includes many standard type-directed synthesis optimizations. For example, to reduce the number of semantically equivalent programs, Contata only considers function implementations that are in eta-long beta-normal form. Additionally, whenever possible, Contata synthesizes generic functions with type parameters to further reduce the search space.

7 Evaluation

In this section, we evaluate Contata through experiments that aim to answer the following research questions:

RQ1. How does Contata compare against prior techniques?
RQ2. What benchmark features impact Contata's performance?

Benchmarks. To answer these questions, we collected a set of 30 benchmarks that exhibit two key characteristics that are relevant to our approach. First, all benchmarks involve recursion or mutual recursion; and, second, the task specification is relational in nature (i.e., relates two different functions to be synthesized or involves a k-safety property [42]). Because the relational specifications we found are often highly unconstrained, we augmented some relational specifications with between 1 and 3 additional input-output examples. The sources of these benchmarks include Stack Overflow posts, functional data structure verification benchmarks [34], and functional programming textbooks [23,29] (Table 1).



(a)

(b)

Fig. 6. (a) The number of benchmarks completed in a given amount of time. (b) The percentage of each class of benchmark solved.

Baseline Tool. To the best of our knowledge, there are no existing tools that can solve relational synthesis tasks involving recursion. Thus, to answer RQ1, we implemented another baseline, henceforth referred to as RelBurst, that combines Burst's approach [33] for dealing with recursion with the approach of Relish [51] for handling relational specifications. At a high level, RelBurst first builds FTAs of individual functions using angelic semantics for unknown functions; this introduces many non-deterministic transitions in the FTA. In the second step, RelBurst uses the relational synthesis technique from [51] to construct an automaton representing the specification. Finally, it uses the backtracking search algorithm of [33] to find a set of function implementations that jointly satisfy the relational specification.

Experimental Setup. All of our experiments are conducted on a machine with an Apple M1 Max CPU and 64 GB of physical memory, running the macOS 14.2.1 operating system. For each task, we set the timeout to 2 min. In addition to relational specifications for each benchmark, we supply a handful of input-output examples to eliminate the ambiguity of relational constraints.

Results. The results of our evaluation are presented in Fig. 6. The plot on the left shows the number of benchmarks solved as we vary the time limit, and the plot on the right compares the percentage of benchmarks solved by Contata against those solved by RelBurst for each class of benchmarks.

Overall, Contata can successfully complete the synthesis task for 22 out of the 30 benchmarks (73%), whereas the baseline completes only 8 (27%). Furthermore, for all completed benchmarks, Contata produces the desired ground truth solution. Additionally, as shown in Fig. 6(b), Contata consistently outperforms the baseline across all benchmark categories. Since the key difference between Contata and RelBurst is the use constraint annotated tree automata, these results support our claim that CATAs are useful for reducing backtracking search when synthesizing recursive programs from relational specifications.

Result for RQ1: Contata solves 2.8× as many benchmarks as a baseline that combines prior techniques for relational synthesis [51] and FTA-based synthesis of recursive procedures [33].

Failure Analysis. As shown in 6(b), Contata performs the best on the mutual recursion benchmarks and worst on the recursive comparators. The latter class of benchmarks are particularly difficult; some involve over a hundred AST nodes and multiple recursive calls. As expected, the synthesis algorithm is sensitive to the size of the target program, so the complexity of the ground truth program has a significant impact on running time. However, there are a few benchmarks in the Partial Data Structures category where the size of the synthesized code is relatively small (32 AST nodes) that Contata also times out on. Upon inspection, we noticed that this is due to the "loose" nature of the specification. In such cases, the language of the constructed CATA is quite large, making automaton operations like intersection very expensive. However, this situation can be averted by adding more input-output examples or augmenting the specification with additional constraints.

Result for RQ2: In addition to the complexity of the target program, Contata is sensitive to the precision of the specifications (i.e., performs better with more precise specifications).

8 Related Work

While there is a vast literature on program synthesis, this work is most closely related to techniques that address the synthesis of recursive procedures as well as those that handle relational specifications.

Synthesis of Recursive Procedures. Research on synthesizing recursive functional programs dates back to the 1970's [26,43] and has recently become a very active research area [17,19,24,27,31,33,35,39,53]. Many of these techniques perform type-directed top-down synthesis from input-output examples [17,19,31,35], whereas Synquid uses refinement types as specifications [39]. Among these approaches, the most related ones are Burst [33], Trio [30], Syrup [53], and SE^{2}GIS [14]. Our technique is directly inspired by Burst and aims to improve upon it by using CATAs to reduce the amount of backtracking search. Trio and Syrup combine deductive reasoning on input-output examples with bottom-up enumeration. In Trio, this is done by generating straight-line programs that are then "folded up" into a recursive program. In contrast, Syrup deduces candidate *recursion traces* to identify possible clusters of valid FTAs to intersect. Both of these approaches rely on input/output deduction and are therefore not easily extensible to the relational synthesis setting that we focus on in this paper.

SE^{2}GIS proves unrealizability of recursion skeletons during synthesis, whereas we use CATAs to rule out incorrect solutions by construction. Additionally, SE^{2}GIS requires a reference implementation and a user-provided recursion skeleton and doesn't consider mutual recursion or relational specifications.

Relational Verification and Synthesis. This work is also related to a long line of work on reasoning about relational properties [5,6,8,38,42,45,51]. Most techniques in this space address the verification problem and aim to prove a relational property, such as equivalence, between two programs [5,6,8,45]. Some techniques [3,42] in this space focus on k-safety properties, such as non-interference or associativity, where the goal is to prove that k different executions of the same function do not violate some desired relationship. On the synthesis side, most prior work handles specific classes of tasks, such as program inversion [22] or data type refactoring [10,36]. To the best of our knowledge, the only synthesis tool that targets a general class of relational properties is Relish [51]. This technique is also based on tree automata and composes FTAs for individual functions in a hierarchical manner by adding non-deterministic transitions between different functions (or different calls to the same function). However, this approach cannot handle recursive or mutually recursive procedures.

Tree Automata with Constraints. There have been many previous attempts to augment FTAs with constraints. In many of these efforts, e.g., data tree automata [7], finite-memory tree automata [25], and symbolic tree automata [46], the tree alphabet is potentially infinite, and transitions can check constraints over this alphabet. Other work considers finite tree alphabets but imposes global constraints such as the equality and disequality of subtrees [12]. In contrast to these FTA variants, transitions in our proposed CATA model can use symbols from outside the tree alphabet, and the CATA's models are not directly tied to labels of the input tree. To our knowledge, such an automaton model has not been considered in the literature.

9 Conclusion

In this paper, we introduced constraint annotated tree automata (CATA) and developed a program synthesis algorithm based on CATAs. Notably, our proposed algorithm can synthesize recursive and mutually-recursive functions from relational specifications. We also implemented this algorithm in a tool called Contata and showed experimentally that Contata outperforms prior approaches by avoiding backtracking search.

While our approach enables solving synthesis tasks that are out of scope for prior approaches, there remains significant future work in solving relational synthesis tasks involving recursion. In future work, we plan to explore the combination of CATAs with top-down synthesis and ML guidance.

References

1. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 934–950. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_67
2. Alur, R., et al.: Syntax-guided synthesis. IEEE (2013)
3. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. ACM SIGPLAN Notices **52**(6), 362–375 (2017)
4. Barnaby, C., Chen, Q., Samanta, R., Dillig, I.: Imageeye: batch image processing using program synthesis. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591248
5. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. ACM SIGPLAN Not. **39**(1), 14–25 (2004)
7. Björklund, H., Bojańczyk, M.: Bounded depth data trees. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 862–874. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_74
8. Chen, J., Wei, J., Feng, Y., Bastani, O., Dillig, I.: Relational verification using reinforcement learning. Proc. ACM Program. Lang. **3**(OOPSLA), 1–30 (2019)
9. Chen, Q., Wang, X., Ye, X., Durrett, G., Dillig, I.: Multi-modal synthesis of regular expressions. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 487–502 (2020)
10. Chen, Y., Wang, Y., Goyal, M., Dong, J., Feng, Y., Dillig, I.: Synthesis-powered optimization of smart contracts via data type refactoring. Proc. ACM Program. Lang. **6**(OOPSLA2), 560–588 (2022)
11. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. SIGPLAN Not. **35**(9), 268–279 (2000). https://doi.org/10.1145/357766.351266
12. Comon, H., et al.: Tree automata techniques and applications (2008)
13. de Halleux, J., Tillmann, N.: Parameterized unit testing with Pex. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 171–181. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_12
14. Farzan, A., Lette, D., Nicolet, V.: Recursion synthesis with unrealizability witnesses. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022, New York, NY, USA, pp. 244–259. Association for Computing Machinery (2022). https://doi.org/10.1145/3519939.3523726
15. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. SIGPLAN Not. **53**(4), 420–435 (2018). https://doi.org/10.1145/3296979.3192382
16. Feng, Y., Martins, R., Van Geffen, J., Dillig, I., Chaudhuri, S.: Component-based synthesis of table consolidation and transformation tasks from examples. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2017, New York, NY, USA, pp. 422–436. Association for Computing Machinery (2017). https://doi.org/10.1145/3062341.3062351

17. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. SIGPLAN Not. **50**(6), 229–239 (2015). https://doi.org/10.1145/2813885.2737977
18. Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Eng. Not. **22**(4), 74–80 (1997)
19. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. SIGPLAN Not. **51**(1), 802–815 (2016). https://doi.org/10.1145/2914770.2837629
20. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11, New York, NY, USA, pp. 317–330. Association for Computing Machinery (2011). https://doi.org/10.1145/1926385.1926423
21. Guo, Z., Cao, D., Tjong, D., Yang, J., Schlesinger, C., Polikarpova, N.: Type-directed program synthesis for restful apis. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022, New York, NY, USA, pp. 122–136. Association for Computing Machinery (2022). https://doi.org/10.1145/3519939.3523450
22. Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 376–389 (2017)
23. Hutton, G.: Programming in Haskell, 2nd edn. Cambridge University Press, Cambridge (2016)
24. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Cyclic program synthesis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021, New York, NY, USA, pp. 944–959. Association for Computing Machinery (2021). https://doi.org/10.1145/3453483.3454087
25. Kaminski, M., Tan, T.: Tree automata over infinite alphabets. In: Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, pp. 386–423 (2008)
26. Kitzelmann, E., Schmid, U., Olsson, R., Kaelbling, L.P.: Inductive synthesis of functional programs: an explanation based generalization approach. J. Mach. Learn. Res. **7**(2) (2006)
27. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '13, New York, NY, USA, pp. 407–426. Association for Computing Machinery (2013). https://doi.org/10.1145/2509136.2509555, https://doi.org/10.1145/2509136.2509555
28. Lampropoulos, L., Hicks, M., Pierce, B.C.: Coverage guided, property based testing. Proc. ACM Program. Lang. **3**(OOPSLA), 1–29 (2019)
29. Lampropoulos, L., Pierce, B.C.: QuickChick: Property-Based Testing in Coq. Software Foundations series, volume 4, Electronic textbook (2018)
30. Lee, W., Cho, H.: Inductive synthesis of structurally recursive functional programs from non-recursive expressions. Proc. ACM Program. Lang. **7**(POPL) (2023). https://doi.org/10.1145/3571263
31. Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. Proc. ACM Program. Lang. **4**(ICFP) (2020). https://doi.org/10.1145/3408991

32. Mariano, B., Chen, Y., Feng, Y., Durrett, G., Dillig, I.: Automated transpilation of imperative to functional code using neural-guided program synthesis. Proc. ACM Program. Lang. **6**(OOPSLA1) (2022). https://doi.org/10.1145/3527315
33. Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up synthesis of recursive functional programs using angelic execution. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498682
34. Miltner, A., Padhi, S., Millstein, T., Walker, D.: Data-driven inference of representation invariants. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2020, New York, NY, USA, pp. 1–15. Association for Computing Machinery (2020). https://doi.org/10.1145/3385412.3385967
35. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '15, New York, NY, USA, pp. 619–630. Association for Computing Machinery (2015). https://doi.org/10.1145/2737924.2738007
36. Pailoor, S., Wang, Y., Dillig, I.: Semantic code refactoring for abstract data types. Proc. ACM Program. Lang. **8**(POPL) (2024). https://doi.org/10.1145/3632870
37. Pailoor, S., Wang, Y., Wang, X., Dillig, I.: Synthesizing data structure refinements from integrity constraints. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021, New York, NY, USA, pp. 574–587. Association for Computing Machinery (2021). https://doi.org/10.1145/3453483.3454063
38. Pick, L., Fedyukovich, G., Gupta, A.: Exploiting synchrony and symmetry in relational verification. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part I. LNCS, vol. 10981, pp. 164–182. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_9
39. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '16, New York, NY, USA, pp. 522–538. Association for Computing Machinery (2016). https://doi.org/10.1145/2908080.2908093
40. Samak, M., Kim, D., Rinard, M.C.: Synthesizing replacement classes. Proc. ACM Program. Lang. **4**(POPL) (2019). https://doi.org/10.1145/3371120
41. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 404–415. ACM (2006). https://doi.org/10.1145/1168857.1168907
42. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. SIGPLAN Not. **51**(6), 57–69 (2016). https://doi.org/10.1145/2980983.2908092
43. Summers, P.D.: A methodology for lisp program construction from examples. J. ACM **24**(1), 161–175 (1977). https://doi.org/10.1145/321992.322002
44. Tillmann, N., Schulte, W.: Parameterized unit tests. ACM SIGSOFT Software Eng. Not. **30**(5), 253–262 (2005)
45. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 742–766. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_35
46. Veanes, M., Bjørner, N.: Symbolic tree automata. Inf. Process. Lett. **115**(3), 418–424 (2015)

47. Wang, C., Cheung, A., Bodik, R.: Interactive query synthesis from input-output examples. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1631–1634. SIGMOD '17, Association for Computing Machinery (2017). https://doi.org/10.1145/3035918.3058738, https://doi.org/10.1145/3035918.3058738
48. Wang, C., Feng, Y., Bodik, R., Dillig, I., Cheung, A., Ko, A.J.: Falx: synthesis-powered visualization authoring. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. CHI '21, New York, NY, USA. Association for Computing Machinery (2021). https://doi.org/10.1145/3411764.3445249
49. Wang, X., Dillig, I., Singh, R.: Program synthesis using abstraction refinement. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158151
50. Wang, X., Dillig, I., Singh, R.: Synthesis of data completion scripts using finite tree automata. Proc. ACM Program. Lang. **1**(OOPSLA) (2017). https://doi.org/10.1145/3133886
51. Wang, Y., Wang, X., Dillig, I.: Relational program synthesis. Proc. ACM Program. Lang. **2**(OOPSLA) (2018). https://doi.org/10.1145/3276525
52. Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: Sqlizer: query synthesis from natural language. Proc. ACM Program. Lang. **1**(OOPSLA), 1–26 (2017)
53. Yuan, Y., Radhakrishna, A., Samanta, R.: Trace-guided inductive synthesis of recursive functional programs. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591255

Information Flow Guided Synthesis with Unbounded Communication

Bernd Finkbeiner[1], Niklas Metzger[1](✉), and Yoram Moses[2]

[1] CISPA Helmholtz Center for Information Security, Saarland, Germany
{finkbeiner,niklas.metzger}@cispa.de

[2] The Andrew and Erna Viterbi Faculty of Electrical and Computer Engineering and the Taub Faculty of Computer Science, Technion, Haifa, Israel
moses@technion.ac.il

Abstract. Information flow guided synthesis is a compositional approach to the automated construction of distributed systems where the assumptions between the components are captured as information-flow requirements. Information-flow requirements are hyperproperties that ensure that if a component needs to act on certain information that is only available in other components, then this information will be passed to the component. We present a new method for the automatic construction of information flow assumptions from specifications given as temporal safety properties. The new method is the first approach to handle situations where the required amount of information is unbounded. For example, we can analyze communication protocols that transmit a stream of messages in a potentially infinite loop. We show that component implementations can then, in principle, be constructed from the information flow requirements using a synthesis tool for hyperproperties. We additionally present a more practical synthesis technique that constructs the components using efficient methods for standard synthesis from trace properties. We have implemented the technique in the prototype tool FlowSy, which outperforms previous approaches to distributed synthesis on several benchmarks.

1 Introduction

More than 65 years after its introduction by Alonzo Church [7], the synthesis of reactive systems, and especially the synthesis of *distributed* reactive systems, is still a most intriguing challenge. In the basic reactive synthesis problem, we translate a specification, given as a formula in a temporal logic, into an implementation that is guaranteed to satisfy the specification for every possible input from the environment. In the synthesis of *distributed systems* [32], we must find an implementation that consists of multiple components that communicate with

This work was funded by the German Israeli Foundation (GIF) Grant No. I-1513-407./2019, by DFG grant 389792660 as part of TRR 248 – CPEC, and by the ERC Grant HYPER (No. 101055412).

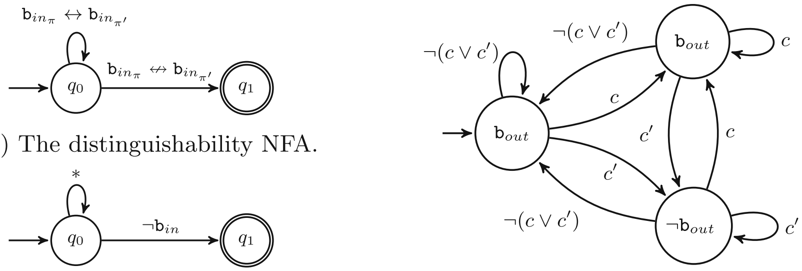
A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 64–86, 2024.
https://doi.org/10.1007/978-3-031-65633-0_4

each other via shared variables in a given architecture. While the basic synthesis problem is, by now, well-supported with algorithms and tools (cf. [5,23]), and despite a long history of theoretical advances [15,25,27,28,30,32], no practical methods are currently known for the synthesis of distributed systems.

A potentially game-changing idea is to synthesize the systems compositionally, one component at a time [4,6,14,18,24,26,33]. The key difficulty in automating compositional synthesis is to find assumptions on the behavior of each component that are sufficiently strong so that each component can guarantee the satisfaction of the specification based on the guarantees of the other components, and, at the same time, sufficiently weak, so that the assumptions can actually be realized. In our previous work on *information flow guided synthesis* [17], we identified situations in which certain components must act on information that these components cannot immediately observe, but must instead obtain from other components. Such situations are formalized as information-flow assumptions, which are hyperproperties that express that the component eventually receives this information. Once the information flow assumptions are known, the synthesis proceeds by constructing the components individually so that they satisfy the information-flow assumptions of the other components provided that their own information-flow assumptions are likewise taken care of.

Technically, the synthesis algorithm identifies a finite number of sets of infinite sequences of external inputs, so-called *information classes*, such that the component only needs to know the information class, but not the individual input trace. In the first step, the output behavior of the component is fixed based on an abstract input that communicates the information class to the component. This abstract implementation is called a *hyper implementation* because it leaves open how the information is encoded in the actual inputs of the component. Once all components have hyper implementations, the abstract input is then replaced by the actual input by inserting a monitor automaton that derives the information class from the input received by the component.

This approach has two major limitations. The first is that the information flow requirement only states that the information will *eventually* be transmitted. This is sufficient for liveness properties where the necessary action can be delayed until the information is received. For safety, however, such a delay may result in a violation of the specification. As a result, the information flow assumptions of [17] are insufficient for handling safety, and *the compositional synthesis approach is thus limited to liveness specifications.* The second limitation is due to the restriction to a *finite* number of information classes. As a result, the compositional synthesis approach is only successful if a solution exists that *acts on just a finite amount of information.* The two limitations severely reduce the applicability of the synthesis method. Most specifications contain a combination of safety and liveness properties (cf. [23]). While it is possible to effectively approximate liveness properties through *bounded* liveness properties (cf. [20]), which are safety properties, the converse is not true. Likewise, most distributed systems of interest are reactive in the sense that they maintain an ongoing interaction with the external environment. As a result, they do not conform to the



(a) The distinguishability NFA.

(b) An information class NFA.

(c) The safety hyper implementation.

Fig. 1. The prefix distinguishability of the sequence transmission protocol as NFA in (a). The NFA representing the information class for output $\mathtt{b}_{out}$ is shown in (b), where (c) is a hyper implementation of the receiver on information classes.

limitation that they only act on a finite amount of information. For example, a communication protocol receives a new piece of information in each message and is hence required to transmit an unbounded amount of information.

In this paper, we overcome both limitations with a new method for information flow guided synthesis that handles both safety properties and specifications of tasks that require the communication of an unbounded amount of information. In order to reason about safety, we consider *finite* prefixes of external inputs rather than infinite sequences. The key idea is to collect sets of finite sequences *of the same length* into information classes. Such an information class refers to a specific point in time (corresponding to the length of its traces) and identifies the information that is needed at this point in time to avoid a violation of the safety property. We then only require that the number of information classes is finite *at each point in time*, while the *total* number of information classes over the infinitely many prefixes of an execution may well be infinite. This allows us to handle situations where again and again some information must be transmitted in a potentially infinite loop.

2 Running Example: Sequence Transmission

Our running example is a distributed system that implements sequence transmission. The system consists of two components, the *transmitter* t and the *receiver* r. At every time step, the transmitter observes the current input bit $\mathtt{b}_{in}$ from the external environment, the transmitter can communicate via $\mathtt{c}_b$ with the receiver, and the receiver controls the output $\mathtt{b}_{out}$. To implement a sequence transmission protocol, the receiver must output the value of the input bit one time step after it is received by the transmitter. We can state this specification using the LTL formula $\square(\mathtt{b}_{in} \leftrightarrow \bigcirc \mathtt{b}_{out})$ for the receiver, and assume the transmitter specification to be *true*. In this example, compositional synthesis is only possible with assumptions about the communication between the components. We utilize an *information-flow assumption* for compositional synthesis specified in the HyperLTL formula $\forall\pi\forall\pi'.(\mathtt{c}_{b_\pi} \leftrightarrow \mathtt{c}_{b_{\pi'}})\mathcal{U}(\mathtt{b}_{in_\pi} \not\leftrightarrow \mathtt{b}_{in_{\pi'}} \wedge \mathtt{c}_{b_\pi} \not\leftrightarrow \mathtt{c}_{b_{\pi'}})$. The formula states that on any pair of traces π and π' of an implementation, the communication bit $\mathtt{c}_b$ on both traces must be equivalent until there is a difference on the input bit $\mathtt{b}_{in}$ as well as a difference on the communication bit $\mathtt{c}_b$. This implies that whenever the receiver must distinguish two input traces, it will observe a difference on its local inputs, namely $\mathtt{b}_{in}$. A nondeterministic finite automaton (NFA) accepting all finite traces that must be distinguished at the same time point is depicted in Fig. 1a. In the course of this paper, we show that, for safety properties, the distinguishability requirement yields an information-flow assumption specified over finite traces. Based on the assumption, we heuristically build information classes over finite traces, such that all finite traces in the same class do not need to be distinguished. Figure 1b shows an NFA for one of the two information classes. It accepts all finite traces that have $\neg\mathtt{b}_{in}$ in the last step. On all these traces, the output $\neg\mathtt{b}_{out}$ is correct. For this example, there is only one other information class, namely the finite traces with $\mathtt{b}_{in}$ in the last step. We use the information classes to synthesize a *hyper implementation* for the receiver, depicted in Fig. 1c. A hyper implementation receives the current information classes, which are c and c' on the transitions, as input, and outputs the local outputs of the component. Whenever c is the input, the correct output for all traces in c must be set by the receiver. Note that, in this example, c and c' cannot occur together as there is no common output for $\mathtt{b}_{in} \wedge \neg\mathtt{b}_{in}$. The hyper implementation is correct for all transmitter implementations. After synthesizing both hyper implementations, for the transmitter and the receiver, we compose and decompose them to obtain local implementations. Throughout this paper, we first define the prefix distinguishability and prefix information-flow assumption. We then build assume and guarantee specifications that, based on the information classes, guarantee the correctness of the hyper implementations, and finally, we show how to construct the local solutions to complete the synthesis procedure.

3 Preliminaries

Architectures. In this paper, we consider distributed architectures with two components: p and q. Such architectures are given as tuple $(I_p, I_q, O_p, O_q, O_e)$, where I_p, I_q, O_p, O_q, and O_e are all subsets of the set $\mathcal{V}$ of boolean variables. O_p and O_q are the sets of *output variables* of p and q. We denote by O_e the output variables of the uncontrollable external environment. We refer to O_e also as the *external inputs* of the system. O_p, O_q and O_e form a partition of $\mathcal{V}$. Finally, I_p and I_q are the *input variables* of components p and q, respectively. The inputs and outputs are disjoint, i.e., $I_p \cap O_p = \emptyset$ and $I_q \cap O_q = \emptyset$. Each of the inputs I_p and I_q of the components is either an output of the environment or an output of the other component, i.e., $I_p \subseteq O_q \cup O_e$ and $I_q \subseteq O_p \cup O_e$. For a set $V \subseteq \mathcal{V}$, every subset $V' \subseteq V$ defines a *valuation* of V, where the variables in V' have value *true* and the variables in $V \setminus V'$ have value *false*.

Implementations. For a set of atomic propositions AP divided into inputs I and outputs O, with $I \cap O = \emptyset$, a 2^O-labeled 2^I-transition system is a 4-tuple (T, t_0, τ, o), where T is a set of states, $t_0 \in T$ is an initial state, $\tau : T \times 2^I \rightarrow T$ is a transition function, and $o : T \rightarrow 2^O$ is a labeling function. An implementation of an architecture $(I_p, I_q, O_p, O_q, O_e)$ is a pair (T_p, T_q), consisting of T_p, a 2^{O_p}-labeled 2^{I_p} transition system T_p, and T_q, a 2^{O_q}-labeled 2^{I_q} transition system T_q. The *composition* $T = T_p||T_q$ of the two transition systems $(T^p, t_0^p, \tau^p, o^p)$ and $(T^q, t_0^q, \tau^q, o^q)$ is the $2^{O_p \cup O_q}$-labeled 2^{O_e}-transition system (T, t_0, τ, o), where $T = T^p \times T^q, t_0 = (t_0^p, t_0^q), \tau((t^p, t^q), x) = (\tau^p(t^p, (x \cup o^q(t^q)) \cap I_p), \tau^q(t^q, (x \cup o^p(t^p)) \cap I_q))$, $o(t^p, t^q) = o^p(t^p) \cup o^q(t^q)$, where $x \in 2^{O_e}$.

Specifications. The specifications are defined over the variables $\mathcal{V}$. For a set $V \subseteq \mathcal{V}$ of variables, a *trace* over V is an infinite sequence $x_0 x_1 x_2 \ldots \in (2^V)^\omega$ of valuations of V. A *specification* over $\mathcal{V}$ is a set $\varphi \subseteq (2^\mathcal{V})^\omega$ of traces over $\mathcal{V}$. Two traces over disjoint sets $V, V' \subset \mathcal{V}$ can be *combined* by forming the union of their valuations at each position, i.e., $x_0 x_1 x_2 \ldots \sqcup y_0 y_1 y_2 \ldots = (x_0 \cup y_0)(x_1 \cup y_1)(x_2 \cup y_2) \ldots$. Likewise, the *projection* of a trace onto a set of variables $V' \subseteq \mathcal{V}$ is formed by intersecting the valuations with V' at each position: $x_0 x_1 x_2 \ldots \downarrow_{V'} = (x_0 \cap V')(x_1 \cap V')(x_2 \cap V') \ldots$. For a trace π we use $\pi[n]$ to access the set on π at time step n, and $\pi[n \ldots m]$ for the interval of π from index n to m. Our specification language is linear-time temporal logic (LTL) [31] with the set $\mathcal{V}$ of variables serving as the atomic propositions. We use the usual Boolean operations, the temporal operators Next $\bigcirc$, Until $\mathcal{U}$, Globally $\square$, and Eventually $\diamondsuit$, and the semantic evaluation of (finite) traces π with $\pi \vDash \varphi$. LTL formulas can be represented by nondeterministic Büchi automata (NBAs) with an exponential blow-up. A finite trace $\pi \in (2^\mathcal{V})^*$ is a bad prefix of an LTL formula φ if $\pi \nvDash \varphi$ and $\pi \cdot \pi' \nvDash \varphi$ for all $\pi' \in (2^\mathcal{V})^\omega$. An LTL formula is a *safety* formula if every violation has a bad prefix. Specifications over architectures are conjunctions $\varphi_p \wedge \varphi_q$ of two LTL formulas, where φ_p is defined over $O_p \cup O_e$, i.e., φ_p relates outputs of the component p to the outputs of the environment, and φ_q is defined over $O_q \cup O_e$. We call these specifications the *local* specifications of the component. An initial run $T(i_0, i_1, \ldots) = t_0 t_1 \ldots \in T^\omega$ for an infinite sequence of inputs $i_0, i_1 \ldots \in 2^{O_e}$ is an infinite sequence of states produced by the transition function such that $t_i = \tau(t_{i-1}, i_{i-1})$ for all $i \in \mathbb{N}$ and t_0 is the initial state. The set of traces *Traces*(T) of an implementation $T = (T^p, T^q)$ is then defined as all $(o(t_0) \cup i_0)(o(t_1) \cup i_1) \ldots \in (2^\mathcal{V})^\omega$ where $T(i_o i_1 \ldots) = t_0 t_1 \ldots$ for some $i_o i_1 i_2 \ldots \in (2^{O_e})^\omega$. An implementation *satisfies* a specification φ if the traces of the implementation are contained in the specification, i.e., *Traces*$(T^p, T^q) \subseteq \varphi$. Given an architecture and a specification φ, the synthesis problem is to find an implementation $T = (T_p, T_q)$ that satisfies φ. We say that a specification φ is *realizable* in a given architecture if such an implementation exists, and *unrealizable* if not.

Automata. A non-deterministic automaton $\mathcal{A}$ is a tuple $(\Sigma, Q, q_o, \delta, F)$ where Σ is the input alphabet, Q is a set of states, q_o is the initial state, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, and F is a set of accepting states. For an input word $\sigma_0 \sigma_1 \ldots \sigma_k \in \Sigma^k$, a finite word automaton (NFA) $\mathcal{F}$ accepts a finite run

$q_0 q_1 \dots q_k \in Q^k$ where $q_i \in \delta(q_{i-1}, \sigma_{i-1})$, if $q_k \in F$. A Büchi automaton (NBA) $\mathcal{A}$ accepts all infinite runs $q_0 q_1 \dots \in Q^\omega$ that visit states in F infinitely often. An automaton is deterministic if the transition function δ is injective. The language of an automaton $\mathcal{A}$ is the set of its accepting runs, and is denoted by $\mathcal{L}(\mathcal{A})$.

Hyperproperties. Information-flow assumptions are hyperproperties. A *hyperproperty over* $\mathcal{V}$ is a set $H \subseteq 2^{(2^{\mathcal{V}})^\omega}$ of sets of traces over $\mathcal{V}$ [9]. An implementation (T_p, T_q) satisfies the hyperproperty H iff the set of its traces is an element of H, i.e., $\mathit{Traces}(T_p, T_q) \in H$. A convenient specification language for hyperproperties is the temporal logic HyperLTL [8], which extends LTL with trace quantification, i.e., $\forall \pi . \varphi$ and $\exists \pi . \varphi$. In HyperLTL, atomic propositions are indexed by a trace variables, which make expressing properties like "ψ *must hold on all traces*"possible, expressed by $\forall \pi .\ \psi$. Dually, one can express that "*there exists a trace on which* ψ *holds*", denoted by $\exists \pi .\ \psi$. Sometimes, a hyperproperty can be expressed as a binary relation on traces. A relation $R \subseteq (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega$ of pairs of traces defines the hyperproperty H, where a set T of traces is an element of H iff for all pairs $\pi, \pi' \in T$ of traces in T it holds that $(\pi, \pi') \in R$. We call a hyperproperty defined in this way a *2-hyperproperty.* In HyperLTL, 2-hyperproperties are expressed as formulas with two universal quantifiers and no existential quantifiers. A 2-hyperproperty can equivalently be represented as a set of infinite sequences over the product alphabet $\mathcal{V}^2$: we can represent a given 2-hyperproperty $R \subseteq \mathcal{V}^\omega \times \mathcal{V}^\omega$, by $R' = \{(\sigma_0, \sigma_0')(\sigma_1, \sigma_1') \dots \mid (\sigma_0 \sigma_1 \dots, \sigma_0' \sigma_1' \dots) \in R\}$. This representation is convenient for the use of automata to recognize 2-hyperproperties.

4 Prefix Information Flow

As argued in [17], identifying information flow between the components is crucial for distributed synthesis, because the specification may require a component's actions to depend on external inputs that are not directly observable by the component. To react to the external inputs correctly, at least the relevant information must be transferred to the component. The fundamental concept to identify when a component requires information transfer is captured by a *distinguishability* relation on sequences of environment outputs. We recall the definition of distinguishability for a component p from [17]:

Definition 1 (Trace distinguishability [17]**).** *Let* φ_p *be an LTL specification of* p*. The corresponding* trace distinguishability relation *is defined as*

$$\begin{aligned}\tau_p = \{(\pi_e, \pi_e') \in & (2^{O_e})^\omega \times (2^{O_e})^\omega \mid \\ & \forall \pi_p \in (2^{O_p})^\omega . \pi_e \sqcup \pi_p \nvDash \varphi_p \ \mathit{or}\ \pi_e' \sqcup \pi_p \nvDash \varphi_p\}\end{aligned}$$

The trace distinguishability relation is defined w.r.t. pairs of infinite traces, where each trace records all outputs of the environment, building up all the information that is presented to the system. Two traces are related iff there exists no infinite trace of p's outputs that satisfies the specification for both (environment) input traces. For example, the traces in the sequence transmission protocol are related by τ_r if they differ on $\mathtt{b}_{in}$ at least once. We now turn the distinguishability relation into an assumption for the component. On traces related by τ_p, the component must observe a difference in its *local* inputs, namely the set I_p. The relation itself only considers infinite traces over all variables that are *not outputs* of the single component, independent of the architecture. Therefore, the information-flow assumption (IFA) built from the distinguishability relation enforces that on all related (environment input) traces, there is a difference on the component's input:

Definition 2 (Trace information-flow assumption [17]). *Let τ_p be the trace distinguishability relation for p. The* information flow assumption $\mathcal{I}_p$ *is the 2-hyperproperty defined by the relation*

$$R_{\mathcal{I}_p} = \{(\pi, \pi') \in (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega \mid \ \textit{if } (\pi{\downarrow}_{O_e}, \pi'{\downarrow}_{O_e}) \in \tau_p \textit{ then } \pi{\downarrow}_{I_p} \neq \pi'{\downarrow}_{I_p}\}$$

The trace information-flow assumption is necessary for a component p; every implementation of the distributed system will satisfy the information-flow assumption from [17]. In its generality, this definition specifies that the values of the local inputs to p have to be different *at some time point*, without an explicit or implicit deadline. This is critical in two ways: On the one hand, liveness specifications, as in the example $\mathtt{b}_{in} \leftrightarrow \Diamond \mathtt{b}_{out}$, will never determine an explicit point in time where the information must be present. On the other hand, safety specifications always include a fixed deadline for the reaction of the component, which, if the information is not present, cannot be met. This deadline, however, is not accounted for in the information-flow assumption, and an algorithm cannot rely on availability of the information during synthesis.

In [17] we solve the liveness issue by introducing a time-bounded information-flow assumption. The time bound acts as a placeholder for the exact time point of information flow. The locally synthesized receiver must then be correct for all such possible time points. Because of the arbitrary deadline, the assumptions cannot suffice to find a solution for a safety specification of the receiver either; they are too weak. We solve this issue by restricting the attention to safety specifications. Consider, for example, the safety property $\varphi_r = \Box(\mathtt{b}_{in} \leftrightarrow \bigcirc \mathtt{b}_{out})$ of our running example. To satisfy this property, the receiver r must observe the value of $\mathtt{b}_{in}$ on its local inputs in exactly one time step, otherwise, it cannot react to $\mathtt{b}_{in}$ in time. With this observation, we can state a stronger distinguishability relation over pairs of *finite* traces.

Definition 3 (Prefix distinguishability). *Let φ_p be the safety specification for component p. The* prefix distinguishability relation *is defined as*

$$\begin{aligned}\rho_{\varphi_p} = \{(\pi, \pi') \in (2^{O_e})^m \times (2^{O_e})^m, m \in \mathbb{N} \mid \forall \pi_p \in (2^{O_p})^m.&\\ \pi \sqcup \pi_p \nvDash_m \varphi_p \textit{ or } \pi' \sqcup \pi_p \nvDash_m \varphi_p&\\ \textit{and } \forall n \in \mathbb{N}, n < m.\ \exists \pi'_p \in (2^{O_p})^n.&\\ \pi[0 \ldots n] \sqcup \pi'_p \vDash_n \varphi_p \textit{ and } \pi'[0 \ldots n] \sqcup \pi'_p \vDash_n \varphi_p\}&\end{aligned}$$

The first condition states that, for the two related input traces of length m, the specification is violated for all possible output sequences for p of the same length. The second condition enforces that m is the first position at which the trace pair must be distinguished, i.e., for all previous positions of the traces, there exists a common output sequence that satisfies the specification on both traces. Every violation of a safety specification has a *minimal bad prefix* [11], and hence every violation that originates in the indistinguishability of two traces is captured by Definition 3. For liveness specifications, no two traces are related by this definition: One can inductively reason that for every $(\pi, \pi') \in \tau_{\varphi_p}$ this pair of traces is not in ρ_{φ_p}, i.e., $(\pi, \pi') \notin \rho_{\varphi_p}$, since for every chosen m, one can find an output trace of p that violates the formula after time point m.

Prefix distinguishability is the core concept of our synthesis method. We now show that we can build an automaton that accepts a pair of finite environment output traces iff they are related. We say that an automaton $\mathcal{A}$ *recognizes* a relation R if $\mathcal{L}(\mathcal{A}) = R$.

Theorem 1. *For a component p with specification φ_p, there exists a non-deterministic finite automaton with a doubly exponential number of states in the length of φ_p that recognizes the prefix distinguishability relation ρ_{φ_p}.*

Proof. We construct a non-deterministic finite automaton (NFA) $\mathcal{F}$ that accepts precisely all pairs of traces over $(2^{O_e})^m \times (2^{O_e})^m$, where $m \in \mathbb{N}$, that are related by ρ_{φ_p}. Let φ'_p be the formula φ_p where all atomic propositions $a \in AP$ are renamed to a', and let $\mathcal{V}'$ be a set containing a copy v' of every variable $v \in \mathcal{V}$. We build the NBA $\mathcal{B} = \mathcal{A}_{\varphi_p} \times \mathcal{A}_{\varphi'_p}$, where $\mathcal{A}_{\varphi_p}$ and $\mathcal{A}_{\varphi'}$ are constructed with a standard LTL-to-NBA translation respectively, and the operator × builds the product of two NBAs. $\mathcal{B}$ now accepts all tuples of traces that each satisfy φ_p. Let $\mathcal{C}$ be the NBA that restricts the transition relation of $\mathcal{B}$ s.t. edges are only present if the output variables of p are equal $\bigwedge_{o \in O_p} o \leftrightarrow o'$ holds, enforcing that both traces agree on the output while satisfying the specification. We now existentially project to the set $O_e \cup O'_e$ to build $\mathcal{D}$, whose alphabet does not contain the component's outputs. To accept the pairs of traces that do not satisfy the formula, we negate $\mathcal{D}$, denoted by $\bar{\mathcal{D}}$. In the last step of the construction, we transform the NBA $\bar{\mathcal{D}}$ to an NFA $\mathcal{F}$ using the *emptiness per state* construction of [3]. This yields an NFA that accepts the prefix distinguishability relation. The size of the automaton is doubly exponential in the size of the formula. The first exponent stems from the LTL to NBA construction, and the second from negating the automaton $\mathcal{F}$. □

Similar to Definition 2, we now turn the safety distinguishability relation into an information-flow assumption that must be guaranteed by the component that observes the respective environment output. The assumptions include specific information-flow deadlines for pairs of traces at which the component must observe the information at the latest. The information-flow assumption, again, is a 2-hyperproperty enforcing that pairs of traces that are related by the prefix distinguishability relation have an observable difference for the component.

Definition 4 (Prefix information-flow assumptions). *Let ρ_{φ_p} be the prefix distinguishability relation for p. The corresponding* prefix information flow assumption *$\mathcal{P}_p$ is the 2-hyperproperty defined by the relation*

$$R_{\mathcal{P}_p} = \{(\pi, \pi') \in (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega \mid \textit{ if } \exists m \in \mathbb{N} \textit{ s.t. } (\pi[0 \ldots m], \pi'[0 \ldots m]) \in \rho_{\varphi_p} \\ \textit{then } \pi{\downarrow_{I_p}}[0 \ldots m-1] \neq \pi'{\downarrow_{I_p}}[0 \ldots m-1]\}$$

On all finite trace pairs in the prefix distinguishability relation ρ_{φ_p}, there must be a difference on I_p before the deadline m. Restricting the observable difference to happen before the *deadline* m is crucial for the receiving component. Whereas the prefix distinguishability relation relates finite traces, the prefix information-flow assumption is a hyperproperty over infinite traces. Unsurprisingly, every implementation of a distributed system satisfying safety LTL specifications satisfies the corresponding prefix information-flow assumption.

Lemma 1. *The prefix information-flow assumption is necessary for safety LTL specifications.*

Proof. Assume that there exists an implementation (T_p, T_q) satisfying the safety LTL specifications φ_p and φ_q but not $\mathcal{P}_p$ and $\mathcal{P}_q$. Since $\mathcal{P}_p$ is not satisfied, there exists a pair of traces π, π' such that $(\pi{\downarrow_{O_e}}[0 \ldots m], \pi'{\downarrow_{O_e}}[0 \ldots m]) \in \rho_{\varphi_p}$ and $\pi{\downarrow_{I_p}}[0 \ldots m+1] = \pi'{\downarrow_{I_p}}[0 \ldots m+1]$. The deterministic system must therefore choose the same output for the timestep $m+1$ since the inputs are the same. This contradicts the assumption: either $\pi[0 \ldots m+1]$ or $\pi'[0 \ldots m+1]$ is a minimal bad prefix since, otherwise, the traces would not be related by the prefix distinguishability relation. □

We are now ready to return to the sequence transmission example. The prefix distinguishability automaton for $\square(\mathtt{b}_{in} \leftrightarrow \bigcirc \mathtt{b}_{out})$ is depicted in Fig. 1a. The automaton accepts a 2-hyperproperty whose alphabet is a pair of valuations of $\mathtt{b}_{in}$. Note that the communication bit from t to r is not restricted by the prefix distinguishability. The automaton terminates whenever a sequence of inputs must be distinguished. For example, starting in the initial state, the input words $\mathtt{b}_{in}$ on π and $\neg\mathtt{b}_{in}$ on π' lead immediately to an accepting state; these finite traces need to be distinguished. However, if $\mathtt{b}_{in}$ is equivalent on both traces, the automaton stays in the initial non-accepting state. By abuse of notation, we use X_p for X_{φ_p}, e.g., ρ_p for ρ_{φ_p}, if φ_p is clear from context.

The automata for the prefix distinguishability and the prefix information-flow assumption can be very complex; even if two traces are different at point n, it can be decided at position $n + m$ if the difference of the inputs results in a necessary information flow, and the automaton might need to store the observed difference during all m intermediate steps. We evaluate the size of the prefix distinguishability automaton empirically in Sect. 7. With the prefix information-flow assumption, we could construct a hyperproperty synthesis problem similar to [17]. In practice, however, synthesis from hyperproperties is largely infeasible, because it hardly scales to more than a few system states [16]. In the following, we show that this problem can be avoided by reducing the compositional synthesis problem to the much more practical synthesis from trace properties.

5 Unbounded Communication in Distributed Systems

Computing the information flow between the components in a distributed system, as shown in Sect. 4, is the first step for compositional synthesis. In the second and more complex step, the synthesis procedure needs to guarantee (1) that the component that observes the information actually transmits the information, and (2) that the component requiring the information correctly assumes the reception. We construct an *assume specification*, which ensures that the component correctly assumes the information flow, and a *guarantee specification*, which enforces the correct transmission of information.

5.1 Receiving Information

A component cannot realize its specification only based on its local observations; it needs to assume that the required information is transmitted during execution. The prefix information-flow assumption is one class of necessary assumptions, i.e., every transmitter implementation must satisfy it, and the hyperproperty can be assumed without losing potential solutions. In many cases, this assumption is also sufficient; if the receiver assumes this exact information flow, the local synthesis problem is realizable. During synthesis, we do not know what actual information the component currently has. The synthesis procedure only has *partial information* of all environment outputs. Which information is actually transmitted at which time point is finally decided by the synthesis process of the *transmitter*. However, the receiver's implementation must be correct for every possible information in every step. We, therefore, collect all traces at a position that do not need to be distinguished by component p at time n, i.e., there exists a prefix of p's outputs that works on all traces.

Definition 5 (Prefix information class). *Let ρ_p be the prefix distinguishability relation for p. The information class of a trace π at position $n \in \mathbb{N}$ is the set of traces $[\pi]_p^n = (2^{O_e})^n \backslash \{\pi' \in (2^{O_e})^n \mid (\pi, \pi') \in \rho_p\}$*

We now construct a trace property that, given an information class c^n, enforces that the output by the component is correct for all traces in the information class c^n. This property specifies exactly one step of outputs, namely $n+1$. Since we consider safety LTL properties, it is sufficient to incrementally specify the outputs according to the satisfaction of the LTL formula.

Definition 6 (Information class specification). *Let φ_p be the LTL specification for component p, $n \in \mathbb{N}$, and let c_n be a prefix information class at position $n-1$. The information class specification $\mathbb{C}_p^n \subseteq (2^{\mathcal{V} \backslash O_q})^\omega$ is defined as*

$$\mathbb{C}_p^n = \{\pi_e \sqcup \pi_o \mid \pi_e \in (2^{\mathcal{V} \backslash O_p})^n, \pi_o \in (2^{O_p})^n \\ s.t.\ \forall \pi_e' \in c_{n-1}. \pi_e'[0 \ldots n] \sqcup \pi_o[0 \ldots n] \vDash_n \varphi_p\}.$$

The output traces in $\mathbb{C}_p^n$ need to be correct for every environment output trace that is in the information class. Here, if an environment output trace is not in the information class, we do not restrict any behavior. We now introduce a crucial assumption: That the number of information classes over all time steps is *bounded.* In general, this is not necessary: one can distinguish every trace from every other trace, such that the information classes increase in every time step. However, if the number of information classes is bounded, we present an effective heuristic for constructing them on the prefix distinguishability assumption in Sect. 5.1. Each information class c (which is now *not* parametric in the time point) is then a set of finite traces $(2^{O_e})^\star$, which is exactly the set of traces in each step that do not need to be distinguished by a component. Consider, for example, the sequence transmission specification $\varphi = \square(\mathtt{b}_{in} \leftrightarrow \bigcirc \mathtt{b}_{out})$. The information classes w.r.t. Definition 5 are all traces that are equal on the environment outputs up to time-point $n-1$. This builds infinitely many information flow classes. It is, however, possible to reduce the information classes to a finite representation. In our example, it is sufficient to check for the previous position of the traces: all finite traces that are equal at $n-1$ do not need to be distinguished. This yields two information classes, one for $\mathtt{b}_{in}$ at the previous step and one for $\neg\mathtt{b}_{in}$ at the previous step. The NFA accepting one of them is depicted in Fig. 1b. With the assumption that we are given a finite set of information classes as subsets of $(2^{O_e})^*$, we are able to build an *assume specification*, which assumes that information classes are received if necessary, and can react to any possible *consistent* sequence of information classes. The information classes $\mathcal{C}$ are now part of the alphabet for the input traces and we use c for refering to a specific information class and as an atomic proposition.

Definition 7 (Assume specification). *Let φ_p be the component specification and $\mathcal{C}$ be the finite set of information classes, where each $c \in \mathcal{C}$ is a subset of $(2^{O_e})^*$. The trace property $\mathbb{A} \subseteq (\mathcal{C} \cup 2^{O_p})^\omega$ is defined as*

$$\begin{aligned}\mathbb{A}_p^{\mathcal{C}} = \{\pi_{\mathcal{C}} \cup \pi_o \mid \pi_{\mathcal{C}} \in \mathcal{C}^\omega, \pi_o \in (2^{O_p})^\omega, \forall n \in \mathbb{N}. \forall c \in \pi_{\mathcal{C}}[n-1].\\ \forall \pi_e[0 \ldots n-1] \in c.\ \textit{if } \pi_c \textit{ is consistent, then } \pi_e \sqcup \pi_o[0 \ldots n] \vDash_n \varphi_p\},\end{aligned}$$

where a finite prefix $\pi_{\mathcal{C}} \in \mathcal{C}^n$ is consistent if it holds that for all $0 \leq m < n$, all finite traces in $\pi_e[0 \ldots m]$ have a prefix in $\pi_e[0 \ldots m-1]$.

The assume specification collects, for a sequence of information classes, all component outputs that are correct for all environment outputs in this information class. The consistency of input traces specifies the correct reveal of information classes. It cannot be the case that a trace that was distinguishable from the current trace in step $n-1$ is indistinguishable in n. Note that a correct transmitter will implement only consistent traces. Let's assume we are given the information classes $\mathcal{C} = \{c, c'\}$ for the sequence transmission problem, where $c = (\{\mathtt{b}_{in}\}, \{\neg\mathtt{b}_{in}\})^*\{\mathtt{b}_{in}\}$ and $c' = (\{\mathtt{b}_{in}\}, \{\neg\mathtt{b}_{in}\})^*\{\neg\mathtt{b}_{in}\}$. These classes suffice to implement the receiver: whenever the trace over $\mathcal{C}^n$ ends in c, the receiver has to respond with $\mathtt{b}_{out}$ and it should respond with $\neg\mathtt{b}_{out}$ whenever the trace ends in c'. Each information class c can be split into the information classes c_n by fixing the length of the traces to n.

Lemma 2. *Let $\mathcal{C}_p$ be the finite set of information classes for component p. Every implementation satisfying the assume specification $\mathbb{A}_p^{\mathcal{C}}$ also satisfies the information class specification $\mathbb{C}_p^n$ for all $n \in \mathbb{N}$ and $c \in \mathcal{C}_p$.*

This lemma follows directly from the definition of the assume specification: It collects all information class specifications for the given set of information classes. Note that correctness is only specified for the set of information classes, not the information flow assumption. If the information classes are not total, in the sense that all distinguished traces are in one of the classes, then the receiver is not correct for *all* implementations of the sender.

5.2 Transmitting Information

While a transmitter has to satisfy its local specification, it must also guarantee that the information flow that the receiver relies on is transmitted in time. In general, this is, again, a hyperproperty synthesis problem: The combination of the local specification of q and the prefix information-flow assumption of p is the 2-hyperproperty that the implementation of q needs to satisfy. However, we propose a framework for more involved (incomplete) trace property synthesis algorithms, potentially speeding up the transmitter synthesis significantly. In contrast to the receiver, the transmitter of information can choose the synthesis strategy; As long as the transmitter satisfies the information-flow assumption, the receiver will assume this implementation as feasible and can react to the information flow correctly during composition. We specify a class of trace properties s.t. each element specifies a subset of the implementations that satisfy a correct transmitter.

Definition 8 (Guarantee specification). *Let p and q be components and $\mathcal{I}\varphi_p$ be the IFA for φ_p. The set $\mathbb{G}_{\rho_p} \subseteq (2^{I_q \cup O_q})^\omega$ is a guarantee specification if all 2^{O_q}-labeled 2^{I_q}-transition systems that satisfy $\mathbb{G}$ also satisfy $\mathcal{I}_p$.*

The first crucial difference between the guarantee specification and the assume specification in Definition 7 is that the transmitter must guarantee a difference on the traces in ρ_p whereas the receiver can only assume to observe a difference whenever ρ_p relates two traces. Additionally, the guarantee specification can specify a subset of implementations of all possible transmitters. We show this difference in the following example: Consider our running example specification $\varphi = \square(\mathtt{b}_{in} \leftrightarrow \bigcirc \mathtt{b}_{out})$. One of the (infinitely) many guarantee specifications can be the set of traces specified by the LTL formula $\square(\mathtt{b}_{in} \leftrightarrow \neg \mathtt{c}_b)$, which enforces that every $\mathtt{b}_{in}$ is communicated to the receiver by setting $\mathtt{c}_b$ to *false*.

It remains to show that we can construct guarantee specifications for prefix information-flow assumptions effectively. We will highlight two useful guarantee specifications, one that is implemented in our prototype and one that utilizes the information classes. We begin with the *full-information specification.* It forces the transmitter to send, if possible, all information and therefore reduces the distributed synthesis problem to monolithic synthesis. This concept was already presented in [32] where it was called adequate connectivity and later extended by Gastin et al. [21].

Definition 9 (Full-information specification). *Let p and q be components, and $f : O_e \cap I_q \rightarrow O_q \cap I_p$ be a bijection. The* full-information specifciation *for q is the trace property*

$$\begin{aligned}\mathbb{F}_p = \{\pi_e \sqcup \pi_o \mid \pi_e \in (2^{O_e \cap I_q})^\omega, \pi_p \in (2^{O_q \cap I_p})^\omega, \forall v \in (O_e \cap I_q),\\ \textit{either}\ \forall n \in \mathbb{N}. v \in \pi_e[n]\ \textit{iff}\ f(v) \in \pi_o[n+1]\\ \textit{or}\ \forall n \in \mathbb{N}. v \in \pi_e[n]\ \textit{iff}\ f(v) \notin \pi_o[n+1]\}\end{aligned}$$

This specification forces the sender to assign exactly one value of a communication variable to every input variable. This choice must hold for every point in time and can not be changed, ensuring that every input combination is uniquely represented by the communication variables. The full-information specification is a guarantee specification for every possible information-flow assumption. Since *every* input bit is guaranteed to be transmitted, every different input trace can be distinguished, not only the ones required to be distinguished by the prefix distinguishability relation. The full-information specification is a sufficient condition for realizing the sender; if there is an implementation for satisfying $\mathbb{F}$, then this implementation is a correct sender. It is not a necessary specification, the sender might be able to encode the inputs to a smaller set of communication variables. The second guarantee specification is based on the information classes.

Definition 10 (Information Class Guarantee). *Let $\mathcal{C}'_p$ be the finite set of information classes of p projected to the inputs of q, s.t. the information classes $c \in \mathcal{C}'_p$ are subsets of $(2^{O_e \cap I_q})^*$. Let furthermore $f : \mathcal{C} \rightarrow 2^{O_q \cap I_p}$ be a bijection. The information class guarantee $\mathbb{I}^{\mathcal{C}}_q \subseteq (2^{(O_e \cap I_q) \cup (O_q \cap I_p))})^\omega$ is defined as*

$$\begin{aligned}\mathbb{I}^{\mathcal{C}}_q = \{\pi_e \cup \pi_o \mid \pi_e \in (2^{O_e \cap I_q})^\omega, \pi_o \in (2^{O_q \cap I_p})^\omega, \forall n \in \mathbb{N}, \forall c \in \mathcal{C}'_p\\ \textit{if}\ \pi_e[0 \ldots n] \in c\ \textit{then}\ f(c) \in \pi_o[n+1]\}.\end{aligned}$$

The specification tracks, for an environment output trace π_e, the current information class. Whenever the finite trace is in an information class c, the transmitter must set the combination of its outputs to the values as specified by the bijection f. The receiver p can therefore observe c by decoding the outputs of q on $O_q \cap I_p$. Similar to the assume specification, the correctness of the information class guarantee depends on the information classes:

Lemma 3. *If a set of information classes $\mathcal{C}$ is sufficient to synthesize φ_p then $\mathbb{I}^{\mathcal{C}}_q$ is a guarantee specification for φ_p.*

If providing the information classes at every step is not sufficient for synthesis, then either the specification is unrealizable or at least one information class falsely contains two traces that need to be distinguished. The assume and guarantee specifications in Sect. 5 build the foundation for synthesizing local components that satisfy the local specification and the information-flow assumption. In most distributed systems, however, components are not solely receivers nor transmitters, but both simultaneously. We now define local implementations that are correct w.r.t. information classes, called safety hyper implementations.

5.3 Safety Hyper Implementations

Hyper implementations were introduced in [17] specifying local implementations of a distributed system that are correct for all possible implementations of all other components. The hyper implementations observe all inputs of the environment but are forced to react to them only if necessary, without restricting the possible solution space of other components. For example, the implementation of the receiver r in the sequence transmission protocol is a 2^{O_r}-labeled 2^{I_r}-transition system, but any locally synthesized solution for r must react to inputs only observed by t. We use the *information classes* of Sect. 5 to specify and synthesize a different approach to hyper implementations. Recall that we assume a bounded number of information classes $\mathcal{C}$.

Definition 11 (Safety hyper implementation). *Let p and q be components, e be the environment, and $\mathcal{C}_p$ be a set of information classes. A safety hyper implementation $\mathcal{H}_p$ of p is a 2^{O_p}-labelled $\mathcal{C}_p \cup 2^{I_p}$-transition system.*

The safety hyper implementation branches over the information classes and the local inputs to p and reacts with local outputs. The safety hyper implementation of our running example is depicted in Fig. 1c. Compared to (non-safety) hyper implementations in [17], the safety hyper implementations do not contain a special input variable `t` that signalizes the reception of information. This deadline is explicitly present in the prefix distinguishability relation and can be computed on the automaton representing the prefix distinguishability relation. Since we consider safety properties, there always exists a pre-determined time frame between the environment input and the necessary reception of the information - a fact that we utilize heavily during hyper implementation construction. We now formalize when a safety hyper implementation is correct.

Definition 12 (Correctness of safety hyper implementation). *Let p, q, and e be the components of a distributed system and the environment, and φ_p, φ_q be the local specifications. A* safety hyper implementation *$\mathcal{H}_p$ is correct if it implements $\mathbb{A}_{\varphi_p}$ and some $\mathbb{G}_{\varphi_q}$.*

Correct hyper implementations of p are compatible with all correct implementations of q, i.e., all possible sequences of information provided by *some* transmitter, and implement *one* solution to the information-flow assumption of q. Since assume and guarantee specifications are trace properties, we can synthesize safety hyper implementations with trace property synthesis algorithms once the Büchi automata for the specifications are constructed.

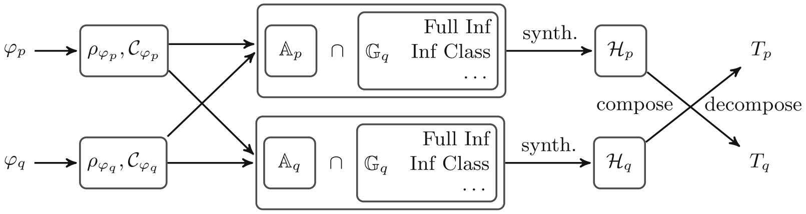


Fig. 2. The steps in the algorithm for compositional synthesis with prefix information flow assumptions.

6 Synthesis with Prefix Information Flow Assumptions

In this section, we present algorithms for generating assume and guarantee specifications, the synthesis of hyper implementations, and obtaining the solutions for each component. Combined, this builds our compositional synthesis approach with information-flow assumptions for distributed systems.

6.1 Automata for Assume and Guarantee Specifications

The first step in our synthesis approach is to construct the assume specification which builds on a finite set of information classes. According to Definition 5, there is, in theory, an unbounded number of information classes. Our Algorithm 1 therefore iteratively builds automata that accept, for each prefix length, one information class. Given the automaton for the prefix distinguishability relation over $\Sigma_\varphi \times \Sigma_{\varphi'}$, the function identicalAPs returns an automaton $\mathcal{A}_{id}$ that accepts exactly one input trace over the alphabet Σ_φ at each time step. This is achieved by choosing one explicit proposition combination for each edge in the automaton.

Algorithm 1: Information Classes

```
let informationClasses($\mathcal{A}_\rho$):=
    let $\mathcal{A}_c = \mathcal{A}_\rho$
    let $\mathcal{A}_{id}$ = identicalAPs($\mathcal{A}_\rho$)
    let result = $\emptyset$
    while $\mathcal{L}(\mathcal{A}_{id}) \neq \emptyset$ do
        result.add(allTraces($\mathcal{A}_{id} \cap \overline{\mathcal{A}_c}$))
        $\mathcal{A}_c = \mathcal{A}_\rho \cap \overline{\mathcal{A}_{id}}$
        $\mathcal{A}_{id}$ = identicalAPs($\mathcal{A}_c$)
    return result
```

On this automaton, the function call allTraces$(\mathcal{A}_{id} \cap \overline{\mathcal{A}_c})$ collects all traces that do not need to be distinguished from $\mathcal{A}_{id}$. These are the traces in the negation of the prefix distinguishability relation that are related to $\mathcal{A}_{id}$. The function allTraces can be computed by renaming the primed propositions on the edges of the automaton. This concludes the computation of the first information class. The algorithm continues by removing $\mathcal{A}_{id}$ from the prefix distinguishability automaton and computing the next information class until the current automaton for the prefix distinguishability relation is empty.

Algorithm 1 yields, if it terminates, n finite automata $\mathcal{F}_c$ where all traces in each $\mathcal{F}_c$ do not need to be distinguished. This implies that there exists a common output combination for each time-step that is correct for each trace in the automaton. We now show a construction for the assume specification in Definition 7.

Construction 1. *We first transform the n finite automata $\mathcal{F}_1, \ldots, \mathcal{F}_n$ for the information classes, as obtained as the result of Algorithm 1, to the respective information class specification (see Definition 6). For each automaton, we build the intersection of the goal automaton $\mathcal{A}_\varphi$ and $\mathcal{F}_i$. The resulting automaton accepts all traces in the information class with outputs as specified by φ. This yields an NBA $\mathcal{B}$ that only accepts a subset of all input traces. We lift it to an automaton for the information class specification by unionizing all input and output combinations that do not occur on $\mathcal{F}_i$, which is $\mathcal{A}_{true} \backslash \mathcal{B}$, where $\mathcal{A}_{true}$ is the automaton accepting all input and output combinations. After performing this construction for all n information class automata, the intersection of all of them accepts the assume specification.*

We use this automaton for the local synthesis of each component. The local specification is implicitly satisfied by the hyper implementation of the assume specification since it is encoded in the construction. We now show how to construct the full information specification in Definition 9.

Construction 2. *Let $I = I_q \cap O_e$ be the inputs observed by q and $O = O_q \cap I_p$. We assume that $|I| \leq |O|$ since we can only transmit all information if we have at least as many communication variables as environment output variables. Let $f : I \rightarrow O$ be a bijection that maps input variables to output variables. We construct the LTL formula $\varphi = \bigwedge_{i \in I} \square(i \leftrightarrow \bigcirc f(i)) \vee \square(i \leftrightarrow \bigcirc \neg f(i))$. This formula enforces that, for every $i \in I$, either the value of i is copied to $f(i)$ at every point in time, or the negation of i's value is copied to $f(i)$ at every point. The corresponding automaton whose language is a full-information specification is $\mathcal{A}_\varphi$, obtained by a standard LTL to NBA translation.*

Together with a guarantee specification, the hyper implementation satisfies its own local specification and the guarantee of the other component.

6.2 Compositional Synthesis

The last step in the compositional synthesis algorithm is the composition and decomposition of the hyper implementations. After this process, we obtain the local implementations of the components and therefore the implementation of the distributed system. During composition and decomposition, we need to replace the information class variables with the actual locally received input. The composition collects all environment and component outputs, as well as the information classes for both components. This includes unreachable states, namely combinations of information classes and environment outputs that are impossible (the finite environment output trace is not in the information class). We eliminate these states in Definition 14. The composition is defined as follows:

Definition 13 (Composition). *Let p, q be components and $\mathcal{H}_p = (T^p, t^p_o, \tau^p, o^p)$ and $\mathcal{H}_q = (T^q, t^q_0, \tau^q, o^q)$ be their safety hyper implementations. The composition $\mathcal{H} = \mathcal{H}_p || \mathcal{H}_q$ is a $2^{O_p \cup O_q}$-labeled $2^{O_e} \cup \mathcal{C}_p \cup \mathcal{C}_q$-transition system (T, t_p, τ, o), where $T = T^p \times T^q$, $t_0 = (t^p_0, t^q_0)$, $o((t^p, t^q)) = o^p(t^p) \cup o^q(t^p)$, and*

$$\tau((t^p, t^q), x) = \ \tau^p(t^p, (x \cup o^q(t^q)) \cap (I_p \cup \mathcal{C}_p)), \tau^q(t^q, (x \cup o^p(t^p)) \cap (I_q \cup \mathcal{C}_q))$$

The state space is the cross product of the hyper implementations and the labeling function is the union of the local hyper implementation's outputs. The transition function ensures that the global inputs over $2^{O_e} \cup \mathcal{C}_p \cup \mathcal{C}_q$ are separated into the inputs of the respective hyper implementations, namely $\mathcal{C}_p \cup I_p$ and $\mathcal{C}_q \cup I_q$. For every state in the cross-product, the composition branches for every environment output and information class to a local state of a component. Some of these states are unreachable. For our running example, the composition includes a transition with $\neg \mathsf{b}_{in}, c'$, even though the trace with $\neg \mathsf{b}_{in}$ in the last step cannot be in c'. We now filter states according to consistency. We consider $\mathcal{H}(x)$ as the hyper implementation $\mathcal{H}$ terminating in x.

Definition 14 (Filter). *Let $\mathcal{H} = (T, t_0, \tau, o)$ be the composition of the 2^{O_p}-labeled $2^{\mathcal{C}_p \cup I_p}$-transition system $\mathcal{H}_p$ and the 2^{O_q}-labeled $2^{\mathcal{C}_q \cup I_q}$-transition system $\mathcal{H}_q$. The consistent composition of $\mathcal{H}_p$ and $\mathcal{H}_q$ is the hyper implementation $\mathcal{H}' = (T', t'_0, \tau', o')$, with $T' = T$, $t'_0 = t_0$, $o' = o$, and*

$$\tau'((t^p, t^q), x) = \begin{cases} \tau((t^p, t^q), x) & \textit{if } \forall c \in x. \mathcal{H}(t^p, t^q) \subseteq \mathcal{L}(\mathcal{F}_c) \\ \emptyset & \textit{else} \end{cases}$$

A finite trace π of length n over $2^{O_e} \cup \mathcal{C}_p \cup \mathcal{C}_q$ is impossible to reach if c is in $\pi[n]$ but $\pi \downarrow_{O_e}$ is not in the information class represented by c. Computing if a state is unreachable includes language inclusion of the subsystem terminating in the state and the automaton of the information class. However, an algorithm that enforces consistency can monitor the current information class of a state during a forward traversal of the composed hyper implementations. In the next and final step, the decomposition then projects the composition to only the *observable* outputs of a component. For some input combinations, this yields a set of reachable states, of which we choose one for the decomposition. In essence, all these states are viable successors for the current input combination.

Definition 15 (Decomposition). *Let $\mathcal{H} = (T, t_p, \tau, o)$ be the consistent composition of the 2^{O_p}-labeled $2^{\mathcal{C}_p \cup I_p}$-transition system $\mathcal{H}_p$ and the 2^{O_q}-labeled $2^{\mathcal{C}_q \cup I_q}$-transition system $\mathcal{H}_q$. Furthermore, let min be a function returning the minimal element for a subset of T w.r.t. some total ordering over the states of T. The decomposition $\mathcal{H}|_p$ is a 2^{O_p}-labelled 2^{I_p}-transition system $(T^p, t^p_0, \tau^p, o^p)$ where $T^p = T$, $t^p_0 = t_0$, $o^p((t^p, t^q)) = o((t^p, t^q)) \cap O_p$, and*

$$\tau^p(t, x) = min\{t' \mid \exists y \in 2^{(O_e \cup \mathcal{C}_p) \setminus I_p}. t' = \tau(t, x \cup y)\}$$

The full compositional synthesis algorithm is shown in Fig. 2. Given the two local specifications, the first step is computing the prefix distinguishability NFAs.

Based on those, the assume specifications and guarantee specifications for both components are constructed and build the inputs to the local synthesis procedures. Note that the guarantee specification can be any strategy that implements the information flow assumption, e.g., any scheduling paradigm. After intersecting the two automata, the components must satisfy the assume and the guarantee specification together, which is achieved by trace property synthesis on the intersection of the automata. The problem is unrealizable if either the prefix information-flow assumption is not sufficient for synthesis (there could be necessary behavioral assumptions), or not all information can be communicated to the receiver. After composition, consistency, and decomposition, the algorithm terminates with two local implementations that, together, implement a correct distributed system:

Corollary 1. *Let p and q be components with local specifications φ_p and φ_q. The distributed system implementation returned by the algorithm depicted in Fig. 2 satisfies the local specifications.*

7 Experiments

We implemented the compositional synthesis algorithm described so far in our prototype called FlowSy. The implementation builds on the popular infinite word automaton manipulation tool spot [13] for translation, conversion, and emptiness checking of NBAs. FlowSy implements the support for the finite automata, the construction of prefix distinguishability in Construction 4, the construction of the information classes in Algorithm 1, and building automata for the assume specification in Construction 1 and full information specification Construction 2. The synthesis of the hyper implementations is performed by converting the Büchi automata to deterministic parity games and solving them with the solver oink [12]. We report on two research questions, (1) how do the prefix distinguishability automaton and the information classes scale w.r.t. formula size and information flow over time and (2) how does FlowSy compare to the existing bounded synthesis approach for distributed system HyperBosy presented in [16]. Note that, at the time of evaluation, HyperBosy was the only tool for distributed synthesis that we were able to compare against. A comparison with the existing information flow guided synthesis algorithm with bounded communication in [17] is infeasible since the supported languages of input specifications are disjoint. All experiments are run on a 2.8 GHz processor with 16 GB RAM, the timeout was 600 s, and the results are shown in Table 1.

Benchmarks. The benchmarks scale in 3 different dimensions: the number of independent variables, time-steps in between information reception and corresponding output, and combinatorics over input and output variables. The first one is independent communication of n input variables in *sequence transmission.* This parametric version of the running example has n conjuncted subformulas of the form $\square(i \leftrightarrow \bigcirc o)$. For the *delay* benchmark, the number of variables is constant, but the number of time steps between input and output is increased,

Table 1. This table summarizes the experimental results. The Benchmark and Parameter columns specify the current instance. The columns $|\varphi|$, $|\rho|$, and $|\mathcal{C}|$ give the size of the formula, the number of states in the prefix distinguishability automaton, and the number of information classes, respectively. The last two columns report the running time of FlowSy and BosyHyper in seconds.

Benchmark	Par.	$\lvert\varphi\rvert$	$\lvert\rho\rvert$	$\lvert\mathcal{C}\rvert$	**FlowSy**	**BosyHyper**
Delay	1	5	4	2	1.74	0.97
	2	6	8	2	**1.87**	TO
	3	7	16	2	**1.84**	TO
	4	8	32	2	**1.94**	TO
	5	9	64	2	**2.36**	TO
Sequence Transmission	1	5	4	2	1.83	**1.42**
	2	11	6	4	**5.28**	TO
	3	16	10	8	**36.81**	TO
Conjunctions	1	5	4	2	3.18	**0.92**
	2	9	4	4	**4.35**	91.80
	3	13	4	8	**9.20**	TO
	4	17	4	16	TO	TO
Disjunctions	1	5	4	2	**3.25**	6.26
	2	9	4	4	**5.63**	60.08
	3	13	4	8	**12.14**	TO
	4	17	4	16	TO	TO

i.e., the formulas have the form $\square(i \leftrightarrow \bigcirc^n o)$. The last two benchmarks build Boolean combinations over the inputs. The *conjunctions* benchmark enforces that the conjunctions over the inputs are mirrored in the outputs. *Disjunctions* is constructed in the same way but with disjunctions in between variables. Formulas are $\square(i_1 \wedge i_1 \wedge \ldots \leftrightarrow \bigcirc o_1 \wedge o_2 \wedge \ldots)$ and $\square(i_1 \vee i_1 \vee \ldots \leftrightarrow \bigcirc o_1 \vee o_2 \vee \ldots)$.

Scaling. FlowSy primarily scales in the number of computed information classes. Most interestingly, for benchmark *delay*, the number of information classes is constantly 2, even though the size of the prefix distinguishability automaton grows exponentially. Independent of the length of the current trace, the automaton for the information class checks that the current position is equal to the position n steps earlier. This can indeed be represented by two information classes. For synthesizing the conjunction and disjunction benchmarks, the situation is reversed. Even though the prefix distinguishability automaton is constant, the number of information classes grows exponentially in the parameter, collecting all possible combinations of input variables. For the sequence transmission benchmark, all reported values scale with the input parameter, which leads to an expected increase of the running time until the timeout at step 4 (not included in Table 1).

Comparison to BosyHyper. FlowSy clearly outscales BosyHyper. Most interestingly, the delay benchmark shows the almost constant running time for FlowSy. Since the number of information classes stays the same, the synthesis of the hyperproperties only scales for transmitting the information. BosyHyper must store all values for all $\bigcirc^n$ steps during synthesis, which immediately increases the search space to an infeasible size. For the benchmarks conjunction and disjunction, one can observe that, although the information classes scale exponentially, the running time of FlowSy is significantly faster than that of BosyHyper, which is already at 91 s for parameter 2. In summary, the compositionality of FlowSy is always beneficial for the synthesis process and it saves on the execution time dramatically when the complex communication in the distributed system can be reduced to a small number of information classes.

8 Related Work

Compositional synthesis for monolithic systems, i.e., architectures with one component and the environment, is a well-studied field in reactive synthesis, for example in [14,18,24,26] and most recently in [1]. In multi-component systems with partial observation, compositionality has the potential to improve algorithms significantly, for example in reactive controller synthesis [2,22]. Assume-guarantee synthesis adheres to the same synthesis paradigm as our approach: the local components infer assumptions over the other components to achieve the local goals [4,6]. The assumptions are *trace properties*, restricting the behavior of the components which often is not necessary. If the assumptions are not sufficient, i.e., too weak to locally guarantee the specification, the assumption can be iteratively refined [29]. Another approach is weakening the acceptance condition to dominance [10] or certificates that specify partial behavior of the components in an iterative fashion [19]. In our previous work on information flow guided synthesis [17], we have introduced the concept of compositional synthesis with information-flow assumptions. The work presented in the paper overcomes the two major limitations of this original approach, namely the limitation to liveness (or, more precisely, co-safety properties) and the limitation to specifications that can be realized by acting only on a a finite amount of information.

9 Conclusion

We have presented a new method for the compositional synthesis of distributed systems from temporal specifications. Our method is the first approach to handle situations where the required amount of information is unbounded. While the information-flow assumptions are hyperproperties, we have shown that standard efficient synthesis methods for trace properties can be utilized for the construction of the components. In future work, we plan to study the integration of the information-flow assumptions computed by our approach with the assumptions on the functional behavior of the components generated by techniques from behavioral assume-guarantee synthesis [4,6]. Such an integration will allow for the synthesis of systems where the components collaborate both on the distribution and on the processing of the distributed information.

References

1. Akshay, S., Basa, E., Chakraborty, S., Fried, D.: On dependent variables in reactive synthesis. In: Finkbeiner, B., Kovács, L. (eds.) ETAPS 2024, pp. 123–143. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-57246-3_8
2. Alur, R., Moarref, S., Topcu, U.: Compositional synthesis of reactive controllers for multi-agent systems. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 251–269. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_14
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. (2011). https://doi.org/10.1145/2000799.2000800
4. Bloem, R., Chatterjee, K., Jacobs, S., Könighofer, R.: Assume-guarantee synthesis for concurrent reactive programs with partial information. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 517–532. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_50
5. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Handbook of Model Checking, pp. 921–962. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_27
6. Chatterjee, K., Henzinger, T.A.: Assume-guarantee synthesis. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 261–275. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_21
7. Church, A.: Applications of recursive arithmetic to the problem of circuit synthesis. In: Summaries of the Summer Institute of Symbolic Logic, vol. 1, pp. 3–50. Cornell University, Ithaca, NY (1957)
8. 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
9. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur
10. Damm, W., Finkbeiner, B.: Automatic compositional synthesis of distributed systems. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 179–193. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_13
11. d'Amorim, Marcelo, Roşu, Grigore: Efficient Monitoring of -Languages. In: Etessami, Kousha, Rajamani, Sriram K.. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_36
12. Dijk, T.: Oink: an implementation and evaluation of modern parity game solvers. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 291–308. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_16
13. Duret-Lutz, A., et al.: From spot 2.0 to Spot 2.10: what's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II, pp. 174–187. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_9
14. Filiot, E., Jin, N., Raskin, J.-F.: Compositional algorithms for LTL synthesis. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 112–127. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_10
15. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: LICS (2005)
16. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesizing reactive systems from hyperproperties. In: Chockler, H., Weissenbacher, G. (eds.) CAV

2018. LNCS, vol. 10981, pp. 289–306. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_16

17. Finkbeiner, B., Metzger, N., Moses, Y.: Information flow guided synthesis. In: Shoham, S., Vizel, Y. (eds.) CAV 2022, Proceedings, Part II (2022). https://doi.org/10.1007/978-3-031-13188-2_25
18. Finkbeiner, B., Passing, N.: Dependency-based compositional synthesis. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 447–463. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_25
19. Finkbeiner, B., Passing, N.: Compositional synthesis of modular systems. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 303–319. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_20
20. Finkbeiner, B., Schewe, S.: Bounded synthesis. Int. J. Softw. Tools Technol. Transfer **15**(5–6), 519–539 (2013). https://doi.org/10.1007/s10009-012-0228-z
21. Gastin, P., Sznajder, N., Zeitoun, M.: Distributed synthesis for well-connected architectures. Formal Methods Syst. Des. **34**(3), 215–237 (2009)
22. Hecking-Harbusch, J., Metzger, N.O.: Efficient trace encodings of bounded synthesis for asynchronous distributed systems. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 369–386. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_22
23. Jacobs, S., et al.: The reactive synthesis competition (SYNTCOMP): 2018-2021. CoRR (2022). https://doi.org/10.48550/ARXIV.2206.00251
24. Kugler, H., Segall, I.: Compositional synthesis of reactive systems from live sequence chart specifications. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 77–91. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_9
25. Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: Logic in Computer Science (LICS) (2001)
26. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_6
27. Madhusudan, P., Thiagarajan, P.S.: Distributed controller synthesis for local specifications. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 396–407. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48224-5_33
28. Madhusudan, P., Thiagarajan, P.S.: A decidable class of asynchronous distributed controllers. In: Brim, L., Křetínský, M., Kučera, A., Jančar, P. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 145–160. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45694-5_11
29. Majumdar, R., Mallik, K., Schmuck, A., Zufferey, D.: Assume-guarantee distributed synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. (2020). https://doi.org/10.1109/TCAD.2020.3012641
30. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. TOPLAS **6**(1), 68–93 (1984)
31. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977 (1977). https://doi.org/10.1109/SFCS.1977.32
32. Pnueli, A., Rosner, R.: Distributed Reactive Systems Are Hard to Synthesize. In: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II. pp. 746–757. IEEE Computer Society (1990). https://doi.org/10.1109/FSCS.1990.89597

33. Schewe, S., Finkbeiner, B.: Semi-automatic distributed synthesis. Int. J. Found. Comput. Sci. **18**(1), 113–138 (2007)

Synthesis of Temporal Causality

Bernd Finkbeiner, Hadar Frenkel, Niklas Metzger,
and Julian Siber$^{(\boxtimes)}$

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{finkbeiner,hadar.frenkel,niklas.metzger,julian.siber}@cispa.de

Abstract. We present an automata-based algorithm to synthesize ω-regular causes for ω-regular effects on executions of a reactive system, such as counterexamples uncovered by a model checker. Our theory is a generalization of *temporal causality*, which has recently been proposed as a framework for drawing causal relationships between trace properties on a given trace. So far, algorithms exist only for verifying a single causal relationship and, as an extension, cause synthesis through enumeration, which is complete only for a small fragment of effect properties. This work presents the first complete cause-synthesis algorithm for the class of ω-regular effects. We show that in this case, causes are guaranteed to be ω-regular themselves and can be computed as, e.g., nondeterministic Büchi automata. We demonstrate the practical feasibility of this algorithm with a prototype tool and evaluate its performance for cause synthesis and cause checking.

Keywords: Actual causality · Cause synthesis · Reactive systems · Temporal logic · Büchi automata

1 Introduction

Causality is a key ingredient for explaining model-checking results [5,15,38,46] and a reasoning tool in several verification and synthesis algorithms [2,36,37]. These techniques have retrofitted causality definitions from philosophy [33,40] and artificial intelligence [31], which were not designed for reactive systems with infinite dynamics and often fall short in such ad-hoc applications. For instance, popular approaches for explaining model-checking results highlight the counterexample trace at events that constitute causes [7,18,32]. Yet, marking a (possibly infinite) set of events does not clearly describe the temporal behavior manifested by them since, e.g., two events can be individually responsible for the effect or only together. Similarly, the occurrence of events in the loop part of a trace can be relevant, e.g., only once or infinitely often.

To address such reoccurring problems arising with causal reasoning in reactive systems, Coenen et al. have recently proposed *temporal causality* for drawing causal relationships between temporal properties on a given trace of a system [19]. Causal properties can then be described symbolically with logics or automata, which give a concise description of the possibly infinite causal behavior, and are, moreover, amenable to verification algorithms.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 87–111, 2024.
https://doi.org/10.1007/978-3-031-65633-0_5

1.1 Temporal Causality

At its core, temporal causality uses counterfactual reasoning to infer a causal relationship: A property is a cause for some effect property on a given trace, where both properties hold, if on all closest traces that do not satisfy the cause, the effect is not satisfied either. Additionally, the cause property has to be semantically minimal. Hence, it is a form of *actual causation* [30], which describes the concrete causal behavior in the given, *actual* observation (the trace), and not all of the system behavior that may cause the effect (which loosely corresponds to the concept of *general causation*).[1]

To illustrate, consider the system depicted in Fig. 1, where x and y are inputs and e is an output. We are interested in what input behavior causes the effect $\diamondsuit e$ on the trace $\pi = (\{x, e\})^\omega$ – we skip the output label of the first position. Our first guess may be $y \vee \diamondsuit x$, which characterizes all system traces that satisfy $\diamondsuit e$. However, this is too general to describe the causal behavior on π. After all, the left disjunct y is not even satisfied by π. Let us see which condition fails. The counterfactual criterion holds: The closest system traces that do not satisfy $y \vee \diamondsuit x$ also do not satisfy the effect, as these are exactly the traces that go directly to the lower state labeled with the empty set and loop there infinitely. However, minimality is not satisfied, as the property $\diamondsuit x$ implies $y \vee \diamondsuit x$ (i.e., is semantically smaller) and also satisfies the counterfactual criterion: the closest trace that does not satisfy it is $(\{\})^\omega$. In particular, the existence of, e.g., trace $\{y, e\}(\{\})^\omega$ that also does not satisfy the cause $\diamondsuit x$, but still satisfies the effect $\diamondsuit e$, is irrelevant, as $(\{\})^\omega$ is closer to π than the trace $\{y, e\}(\{\})^\omega$. It is worth pointing out that we only measure distance over inputs. Picking a property that is *too* small fails the counterfactual criterion: If we picked $\square x$, which implies $\diamondsuit x$, there would be, e.g., the closest trace $\{\}(\{x, e\})^\omega$ that still satisfies the effect.

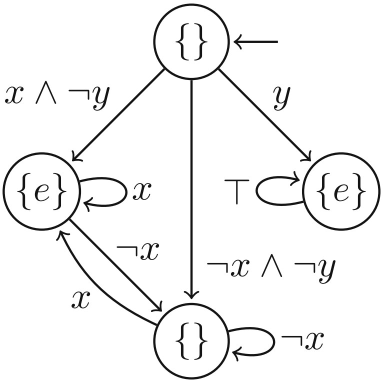


Fig. 1. Example system.

In their original work [19], Coenen et al. showed that the requirements for a valid causal relationship can be encoded as a hyperproperty [17], such that checking whether a given ω-regular property is indeed the cause for a given ω-regular effect on a trace can be decided via model checking. This has recently been implemented in a sketch-based algorithm for enumerating causes [11], which is complete for effects containing $\bigcirc$ as the only temporal operator. That approach, of course, covers only a tiny fragment of the original theory. How to compute the cause for an arbitrary ω-regular effect has remained an open question.

[1] Actual and general causality are also called token and type causality in the literature.

1.2 Contributions and Structure

As it turns out, the intricate balance between the counterfactual criterion and minimality of temporal causality gives rise to an intuitive order-theoretic characterization of causes: The complement of the cause is the upward closure of the negated effect property in the partial order defined by the similarity relation (measuring distance from the actual trace). We illustrate the intuition behind this characterization in Sect. 3.1, and formally develop it in Sect. 5.1.

The consequence of our characterization is that if we can compute the upward closure of the negated effect $\overline{\mathsf{E}}$ and the complement of the result, then we can compute the cause for E on π. We show that if E is an ω-regular property, π in a lasso shape, and the similarity relation is also defined by a (relational) ω-regular property, such an upward closure can be constructed as a nondeterministic Büchi automaton, which means that the cause (i.e., the complement of the automaton) again is an ω-regular property. This approach forms the core of our cause synthesis algorithm, which we describe in Sect. 5.

The complexity of our algorithm significantly scales in the size of the description of the similarity relation, which is problematic due to the complex and large similarity relations of previous work. Coenen et al. [19] observed that with the original counterfactual criterion, these similarity relations need to satisfy the assumption that there is a non-empty set of closest traces for any actual trace and candidate cause, otherwise the counterfactual condition can be vacuously true. We tie this restriction to the *limit assumption* first introduced by Lewis [41] and study similarity relations through this lens. Concrete similarity relations that have been proposed so far [11,19] satisfy the limit assumption by adding additional criteria, but these increase the size of the formula describing the similarity relation significantly. In Sect. 4, we show that we can instead modify the counterfactual condition of the causality definition to allow similarity relations that do not satisfy the limit assumption, using Lewis' semantics for counterfactuals [41], as extended to non-total similarity relations by Finkbeiner and Siber [23]. Crucially, this modification retains the original semantics of Coenen et al. for similarity relations that satisfy the limit assumption as long as the actual trace is deterministic. Hence, it generalizes our closure-based characterization and the corresponding algorithm to significantly simpler similarity relations. All proofs can be found in the full version of this paper [22].

In Sect. 6, we show through experiments with our prototype tool CORP that our modified counterfactual criterion leads to significantly faster computations in practice. We further compare our cause synthesis algorithm with the incomplete sketching approach of the tool CATS [11]. Last, we extend our approach to cause checking through cause synthesis with an additional equivalence check, which we compare with the checker implemented in CATS.

Contributions. In summary, we make the following contributions:

- We extend the theory of temporal causality to similarity relations that do not satisfy the limit assumption.

- We prove an order-theoretic characterization of causes as downward closed sets of the similarity relation.
- Based on this characterization, we develop the first complete method for ω-regular cause synthesis.
- We present and evaluate a prototype implementation of our approach.

2 Preliminaries

We start by recalling preliminaries regarding our system model. Then, we provide background on automata and logics for describing temporal properties.

Systems and Traces. We model *systems* as nondeterministic finite state machines $\mathcal{T} = (S, s_0, AP, \delta, l)$ where S is a finite set of *states*, $s_0 \in S$ is the *initial state*, $AP = I \uplus O$ is the set of *atomic propositions* consisting of *inputs* I and *outputs* O, $\delta : S \times 2^I \to 2^S$ is the *transition function* determining a set of successor states for a given state and input, and $l : S \to 2^O$ is the *labeling function* mapping each state to a set of outputs. A *trace* of $\mathcal{T}$ is an infinite sequence $\pi = \pi[0]\pi[1]\ldots \in (2^{AP})^\omega$, with $\pi[i] = A \cup l(s_{i+1})$ for some $A \subseteq I$ and $s_{i+1} \in \delta(s_i, A)$ for all $i \geq 0$, i.e., we skip the label of the initial state in the first position. *traces*$(\mathcal{T})$ is the set of all traces of $\mathcal{T}$. For two subsets of atomic propositions $V, W \subseteq AP$, let $V|_W = V \cap W$, $\pi|_W = \pi_0|_W \pi_1|_W \ldots$ and $\pi =_V \pi'$ iff $\pi|_V = \pi'|_V$ for traces π, π'. A trace π_0 is *deterministic* in $\mathcal{T}$ iff for all $\pi_1 \in$ *traces*$(\mathcal{T}) : \pi_0 =_I \pi_1 \to \pi_0 = \pi_1$. A trace π is *lasso-shaped*, if there exist $i, j = i+1, k \in \mathbb{N}$ such that $\pi = \pi_0 \ldots \pi_i \cdot (\pi_j \ldots \pi_k)^\omega$, we then define $|\pi| = k-1$.

Büchi Automata. A *nondeterministic Büchi automaton* (NBA) [13] is a tuple $\mathcal{A} = (Q, \Sigma, Q^0, F, \Delta)$, where Q denotes a finite set of *states*, Σ is a finite *alphabet*, $Q^0 \subseteq Q$ is a set of *initial states*, $F \subseteq Q$ is the set of *accepting states*, and $\Delta : Q \times \Sigma \to 2^Q$ is the *transition function* that maps a state and a letter to a set of possible successor states. The *size* of an NBA $|\mathcal{A}|$ is the number of its states $|Q|$. A *run* of $\mathcal{A}$ on an infinite *word* $w = w_1 w_2 \cdots \in \Sigma^\omega$ is an infinite sequence $r = q_0 q_1 \cdots \in Q^\omega$ with $q_0 \in Q^0$ and $q_{i+1} \in \Delta(q_i, w_i)$ for all $i \in \mathbb{N}$. A run r of the NBA is *accepting* if there exist infinitely many $i \in \mathbb{N}$ such that $q_i \in F$. The *language* $\mathcal{L}(\mathcal{A})$ is the set of all words that have an accepting run. We say that some trace property $\mathsf{P} \subseteq (2^A)^\omega$ is *ω-regular*, if there is an NBA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathsf{P}$. A trace π *satisfies* any $\mathsf{P} \subseteq (2^A)^\omega$, denoted by $\pi \vDash \mathsf{P}$, iff $\pi|_A \in \mathsf{P}$.

Linear-Time Temporal Logic. We use *Linear-time Temporal Logic* (LTL) [44] to succinctly specify a fragment of ω-regular properties throughout the paper. LTL formulas are built using the following grammar, where $a \in AP$:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi \;.$$

The semantics of LTL are given by the following satisfaction relation, which recurses over the positions i of the trace π.

$$\begin{array}{lll}
\pi, i \vDash a & \text{iff} & a \in \pi[i] \\
\pi, i \vDash \neg\varphi & \text{iff} & \pi, i \nvDash \varphi \\
\pi, i \vDash \varphi \wedge \psi & \text{iff} & \pi, i \vDash \varphi \text{ and } \pi, i \vDash \psi \\
\pi, i \vDash \bigcirc\varphi & \text{iff} & \pi, i+1 \vDash \varphi \\
\pi, i \vDash \varphi \mathcal{U} \psi & \text{iff} & \exists j \geq i \text{ such that } \pi, j \vDash \psi \text{ and } \forall i \leq k < j.\, \pi, k \vDash \varphi
\end{array}$$

A trace π *satisfies* a formula φ, denoted by $\pi \vDash \varphi$ iff the formula holds at the first position: $\pi, 0 \vDash \varphi$. The *language* $\mathcal{L}(\varphi)$ is the set of all traces that satisfy a formula φ. We also consider the usual derived Boolean connectives: $\vee$, $\rightarrow$, $\leftrightarrow$; and temporal operators: $\varphi \mathcal{R} \psi \equiv \neg(\neg\varphi \mathcal{U} \neg\psi)$, $\Diamond\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$, $\Box\varphi \equiv \mathit{false}\,\mathcal{R}\,\varphi$.

Relational Properties. Relational properties, or, *hyperproperties* [17], allow us to relate multiple system executions, and reason about their interaction. Counterfactual reasoning often is a hyperproperty, and in particular, temporal causality as defined by Coenen et al. was formally shown to be a hyperproperty [19]. Many logics to express temporal hyperproperties have been suggested in recent years (e.g., [6,8,10,28]), the most prominent one being HyperLTL [16]. In this paper, we do not use a hyperlogic to express temporal causality, but we use the related notion of *zipped traces* (e.g., [9]) for defining similarity relations. A *zipped trace* of three traces $\pi_{0,1,2}$ is defined as $\mathit{zip}(\pi_0, \pi_1, \pi_2)[i] = \{(a, t_k) \mid a \in \pi_k[i]\}$, i.e., we construct the zipped trace from disjoint unions of the positions of the three traces, where atomic propositions from the traces $\pi_{0,1,2}$ are distinguished through pairing them with the trace variables $t_{0,1,2}$.

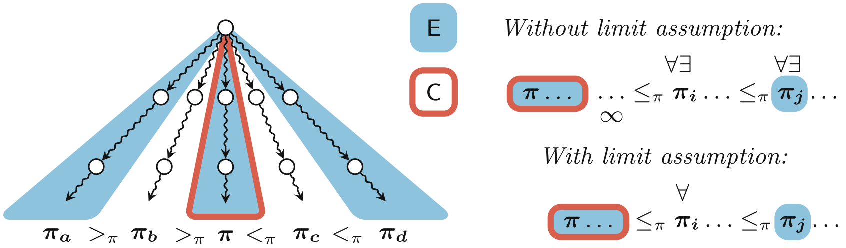
3 Overview: The Topology of Causality

Our main results on cause synthesis heavily rely on a characterization of causes as certain downward closed sets of system traces that are ordered by a similarity relation. We illustrate the main intuition behind this characterization in Sect. 3.1. Then, in Sect. 3.2, we outline how we extend this result to more general similarity relations than originally considered by Coenen et al. [19].

3.1 Actual Causes as Downward Closed Sets of Traces

Our central theorem states that the temporal cause for an effect E on some actual trace π is the largest subset of E that is downward closed[2] in the preordered set of system traces $(\mathit{traces}(\mathcal{T}), \leq_\pi)$, where $\leq_\pi$ is a (comparative) similarity relation that orders traces based on their similarity to π. Figure 2a illustrates this abstractly. Arrows together with nodes represent system executions, whose

[2] $X \subseteq \mathit{traces}(\mathcal{T})$ is downward (upward) closed in $(\mathit{traces}(\mathcal{T}), \leq_\pi)$ if for all $\pi_x \in X$ and $\pi_t \in \mathit{traces}(\mathcal{T})$, $\pi_t \leq_\pi \pi_x$ ($\pi_x \leq_\pi \pi_t$) implies $\pi_t \in X$.


E
C
Without limit assumption:
∀∃
∀∃
$\pi \dots$ $\dots \leq_\pi \pi_i \dots \leq_\pi \pi_j \dots$
∞
With limit assumption:
∀
$\pi \dots$ $\leq_\pi \pi_i \dots \leq_\pi \pi_j \dots$
$\pi_a >_\pi \pi_b >_\pi \pi <_\pi \pi_c <_\pi \pi_d$

(a) The cause C as a subset of the effect E. **(b)** Possible situations at the limit of C.

Fig. 2. Two highlighted aspects of the cause C in the preordered set $(\mathit{traces}(\mathcal{T}), \leq_\pi)$. Figure 2a illustrates that the cause is the largest downward-closed subset of the effect E. The quantifiers in Fig. 2b show which traces outside of the cause are required to avoid the effect in our formalization (*without limit assumption*) and in Coenen et al.'s definition [19] (*with limit assumption*). (Color figure online)

traces form $\mathit{traces}(\mathcal{T})$ and are ordered by the irreflexive reduction $<_\pi$ of the similarity relation. The set of system traces is, in general, infinite, such that there may be infinitely many other traces which are omitted from the illustration for sake of clarity. However, note that similarity relations must be designed such that all traces are further away from the actual trace π than itself, i.e., π is a minimum of $\leq_\pi$. The set of traces that satisfy the effect is depicted by the area that is colored in light blue. The actual trace π is an element of this set, as this is the trace on which the cause for a given effect is analyzed.

Coenen et al.'s temporal causality is counterfactual in nature, and now requires that the *closest* traces outside of the cause C, which in Fig. 2a is marked by the red border, do not satisfy the effect. In the illustration, this is reflected by π_b and π_c not satisfying the effect, i.e., not being in a light blue area. At the same time, Coenen et al. require the cause to be the smallest set that satisfies this, which means that only traces that satisfy the effect are included: Otherwise, the upward closure[3] of traces that do not satisfy the effect could be removed. Hence, in Fig. 2a the area inside the red border is light blue.

In this paper, we show that the balance between these criteria defines causes that are the largest subsets of E that are downward closed in the preordered set $(\mathit{traces}(\mathcal{T}), \leq_\pi)$. We also propose an algorithm that constructs these causes for effects that are ω-regular properties and traces that are in a lasso-shape. Our algorithm first constructs a nondeterministic Büchi automaton for the complement of the cause C. This complement is the upward closure of the negated effect $\overline{\mathsf{E}}$, which means it includes all traces for which there exists an at-least-as close trace that does not satisfy the effect. Since $\leq_\pi$ is reflexive, this naturally includes all traces in $\overline{\mathsf{E}}$, such as π_b and π_c in Fig. 2a. It also includes all traces that are further away than a trace in $\overline{\mathsf{E}}$, such as π_a and π_d.

[3] The upward closure of a set X is the smallest upward closed set containing X.

In the end, these mechanisms make temporal causality a form of *actual* causality that describes a local generalization of the behavior that causes the effect on the actual trace. In the introductory example from Fig. 1 with the actual trace $\pi = \{x, e\}^\omega$, traces in, e.g., $\mathcal{L}(y \wedge \neg \Diamond x)$ are all further away from π than the trace $\{\}^\omega$, which is in $\overline{\mathsf{E}} = \mathcal{L}(\Box \neg e)$. Hence, $\mathcal{L}(y \wedge \neg \Diamond x)$ is included in the upward closure of $\overline{\mathsf{E}}$, and none of its elements is included in the cause.

3.2 Causality Without the Limit Assumption

With our approach based on set closure, we can solve a central issue of temporal causality: Since the preordered set $(\mathit{traces}(\mathcal{T}), \leq_\pi)$ is infinite, there only exist traces in $\overline{\mathsf{C}}$ that are the *closest* with respect to the actual trace π if $(\overline{\mathsf{C}}, \leq_\pi)$ is well-founded. If this is the case for all possible pairs of actual trace π and cause candidate C, we say the similarity relation satisfies the *limit assumption*, after Lewis [41], who formalized it for counterfactual modal logic. Since Coenen et al.'s definition [19] requires that all *closest* traces avoid the effect, it is restricted to similarity relations that satisfy this assumption. Their counterfactual condition is illustrated in the lower part of Fig. 2b. If the limit assumption holds, any descending chain $\pi_j \geq_\pi \pi_{j-1} \geq_\pi \dots$ stabilizes at some π_i, for which Coenen et al. require $\pi_i \in \overline{\mathsf{E}}$.

If the limit assumption does not hold (upper part of Fig. 2b), there may be infinite chains $\pi_j \geq_\pi \pi_{j-1} \geq \dots$ for which a closest π_i does not exist. In these instances, Coenen et al.'s criterion would be vacuously true. This is particularly problematic as the canonical similarity relation $\leq_\pi^{subset}$ does not satisfy the limit assumption. This metric orders two traces as $\pi_j \leq_\pi^{subset} \pi_k$ if the changes between π_j and π are a subset of the changes between π_k and π. This may lead, for example, to the infinite chain $\{\}^\omega \geq_\pi^{subset} \{x\}\{\}^\omega \geq_\pi^{subset} \dots$ in the preordered set $(\mathcal{L}(\Diamond\Box\neg x), \leq_\pi^{subset})$, where $\pi = \{x\}^\omega$. Coenen et al. add additional constraints on top of $\leq_\pi^{subset}$ to ensure that it satisfies the limit assumption. These, however, make cause checking more expensive, as observed by Beutner et al. [11], who therefore combine $\leq_\pi^{subset}$ with a vacuity check. While this is computationally better, this check simply fails in instances as outlined above, and so certain causes cannot be checked by this method [11].

In this work, we solve this conundrum by modifying the definition of temporal causality to accommodate similarity relations that satisfy the limit assumption. We change the central counterfactual condition from a universal quantification over the closest traces in $\overline{\mathsf{C}}$ to an $\forall\exists$-quantification over all traces $\pi_j \in \overline{\mathsf{C}}$. For each such trace π_j, we require the existence of a closer trace $\pi_i \leq_\pi \pi_j$ that does not satisfy the effect. This is depicted in the upper part of Fig. 2b. Naturally, this quantification mirrors exactly the characterization of cause-complements via upward closed sets (cf. Sect. 5.1). On the theoretical side, we show that if the similarity relation satisfies the limit assumption and a minor assumption on nondeterminism is met, our definition is equivalent to Coenen et al.'s original definition (Sect. 4.3). On the practical side, we confirm experimentally that our approach leads to significant improvements through the accommodation of simpler similarity relations that do not satisfy the limit assumption (Sect. 6).

4 Generalized Temporal Causality

In this section, we generalize the definition of temporal causality to accommodate similarity relations that do not satisfy the limit assumption. We first recall similarity relations and formalize the limit assumption (Sect. 4.1). Then we present our updated definition of temporal causality (Sect. 4.2). Last, we prove that it retains the original semantics in the special case considered by Coenen et al. with a minor additional assumption on nondeterminism (Sect. 4.3).

4.1 Similarity Relations and the Limit Assumption

A *comparative similarity relation* $\leq_\pi \subseteq (2^I)^\omega \times (2^I)^\omega$ is a partial order that orders traces by their *comparative* distance from the given actual trace π, i.e., it gives no quantitative but a relative measurement of distance: $\pi_0 \leq_\pi \pi_1$ means π_0 is *at-least-as close* to π as π_1. We measure distance over the set of inputs I, i.e., for two traces $\pi_{0,1} \in (2^{AP})^\omega$ we are only interested in $\pi_0|_I \leq_\pi \pi_1|_I$.

If I is clear from the context, we write $\pi_0 \leq_\pi \pi_1$. We require the actual trace to be closer to itself than any other trace, i.e., $\pi \leq_\pi \pi'$ for all $\pi' \in (2^{AP})^\omega$. The ternary relation $\leq$, where $(\pi_0, \pi_1, \pi_2) \in \leq$ iff $\pi_1 \leq_{\pi_0} \pi_2$, encodes the comparative similarity relations of all possible actual traces π_0.

Example 1. To illustrate our formalism for similarity relations, consider the following *subset-based* similarity relation $\leq^{subset}$ defined via the zipped trace $\mathit{zip}(\pi_0, \pi_1, \pi_2) \in (2^{AP \times \{t_0, t_1, t_2\}})^\omega$. To ease comprehension, for some $a \in AP$ we write a_{actual} for (a, t_0), a_{close} for (a, t_1), and a_{far} for (a, t_2) to explicitly identify, e.g., propositions on the actual trace, in a given formula. We then have $\pi_{close} \leq^{subset}_{\pi_{actual}} \pi_{close}$ iff

$$\mathit{zip}(\pi_{actual}, \pi_{close}, \pi_{far}) \vDash \square \bigwedge_{i \in I} \big((i_{actual} \not\leftrightarrow i_{close}) \rightarrow (i_{actual} \not\leftrightarrow i_{far})\big) \ .$$

For the three traces $\pi_{actual}, \pi_{close}, \pi_{far}$ this requirement states that the changes between π_{actual} and π_{close} are a subset of the changes between π_{actual} and π_{far}, where we define the changes between two traces π_0, π_1 as $\mathit{changes}(\pi_0, \pi_1) = \{(a, i) \mid \pi_0[i] \neq_{\{a\}} \pi_1[i]\}$. For example, let $\pi = \{x\}(\{\})^\omega, \pi_0 = \{\}(\{\})^\omega, \pi_1 = \{\}\{y\}(\{\})^\omega$ and $I = \{x, y\}$. Then, $\pi_0 \leq_\pi \pi_1$, since $\mathit{changes}(\pi, \pi_0) = \{(x, 0)\} \subseteq \{(x, 0), (y, 1)\} = \mathit{changes}(\pi, \pi_1)$. The trace $\pi_2 = \{x\}(\{y\})^\omega$, however, is incomparable to π_0 and π_1, as $\mathit{changes}(\pi, \pi_2) = \{(y, j) \mid j \geq 1\}$ is not in any subset relationship with the respective sets for π_0, π_1.

The similarity relations considered in previous works [11,19] are all fundamentally based on $\leq^{subset}$ as defined in Example 1, with added conditions to avoid infinite chains of closer traces. This is directly tied to the *limit assumption* first studied by Lewis in his seminal work on counterfactual modal logic [41]. In our setting, this assumption can be formalized as follows.

Definition 1 (Limit Assumption). *A similarity relation* $\leq\, \subseteq (2^I)^\omega \times (2^I)^\omega \times (2^I)^\omega$ *satisfies the* limit assumption, *if for all traces* $\pi \in (2^{I \cup O})^\omega$ *and all possible causes* $\mathsf{C} \subseteq (2^I)^\omega$*, we have that* $(\overline{\mathsf{C}}, <_\pi)$ *is well-founded, i.e., there is no infinite descending chain* $\pi_0 >_\pi \pi_1 >_\pi \dots$ *with* $\pi_i \in \overline{\mathsf{C}}$.

This requirement means that there always exist *closest* counterfactual traces that do not satisfy the cause no matter which actual trace we pick (except if all traces satisfy the cause). These closest traces would be ideal candidates for causal analysis, but unfortunately, they do not always exist, in particular not for the similarity relation $\leq^{subset}$, as stated in Proposition 1. Note that all proofs can be found in the full version of this paper [22].

Proposition 1. $\leq^{subset}$ *does not satisfy the limit assumption.*

Since the original definition of Coenen et al. [19] quantifies universally over closest traces, it can be vacuously satisfied if the similarity relation does not satisfy the limit assumption. Previous works have therefore added additional constraints. For instance, Beutner et al. [11] propose $\leq^{full}$, which *additionally* to the constraints of $\leq^{subset}$ (cf. Example 1) requires the following:

$$zip(\pi_{actual}, \pi_{close}, \pi_{far}) \vDash \bigwedge_{i \in I} \big(\Box\Diamond(i_{actual} \not\leftrightarrow i_{close}) \rightarrow \Box(i_{close} \leftrightarrow i_{far})\big) \ .$$

This encodes that whenever π_{close} differs differs from π_{actual} on some input at infinitely many locations, then π_{far} agrees with π_{close} on this input. Hence, on any chain in $<_\pi^{full}$, infinite changes on some $i \in I$ eventually get converted into finite ones, which ensures finiteness of the chain since there are only finitely many atomic propositions. We confirm that this results in $\leq^{full}$ satisfying the limit assumption.

Proposition 2. $\leq^{full}$ *satisfies the limit assumption.*

While satisfying the limit assumption is, in principle, useful, in the case of $\leq^{full}$ this comes at a significant cost: Its logical description contains a large conjunction over the inputs, each containing an implication between temporal formulas. Hence, any algorithmic approach to cause synthesis (and checking) that uses $\leq^{full}$ will scale poorly in the size of I. This motivates us to develop a modified definition of temporal causality that can directly work with the smaller, canonical similarity relation $\leq^{subset}$, while retaining most of the original semantics of Coenen et al. for similarity relations that satisfy the limit assumption, such as $\leq^{full}$.

4.2 A General Definition of Temporal Causality

We now develop our generalized definition of temporal causality for similarity relations that do not satisfy the limit assumption.

The idea behind our generalization stems from counterfactual modal logic as formalized by Lewis [41]. Lewis' semantics a priori only work for total similarity relations, making them unsuitable for our setting. However, they were recently extended to non-total similarity relations by Finkbeiner and Siber [23]. We apply these semantics to our concrete problem to obtain a well-defined notion of causality for similarity relations that do not satisfy the limit assumption. In Sect. 4.3, we show that our definition retains the original semantics proposed by Coenen et al. for similarity relations that satisfy the limit assumption.

Definition 2 (Temporal Causality). *Let $\mathcal{T}$ be a system, $\pi \in traces(\mathcal{T})$ a trace, $\leq_\pi$ a similarity relation, and $\mathsf{E} \subseteq (2^{AP})^\omega$ an effect property. We say that $\mathsf{C} \subseteq (2^I)^\omega$ is a* cause *of E on π in $\mathcal{T}$ if the following conditions hold.*

SAT: *For all $\pi_0 \in traces(\mathcal{T})$ such that $\pi_0 =_I \pi$ we have $\pi_0|_I \in \mathsf{C}$ and $\pi_0 \in \mathsf{E}$.*
CF: *For all $\pi_0 \in \overline{\mathsf{C}}$ there is an at-least-as close trace $\pi_1 \in \overline{\mathsf{C}}$, i.e., with $\pi_1 \leq_\pi \pi_0$, such that there is a $\pi_2 \in traces(\mathcal{T})$ with $\pi_1 =_I \pi_2$ and $\pi_2 \in \overline{\mathsf{E}}$.*
MIN: *There is no $\mathsf{C}' \subset \mathsf{C}$ such that C' satisfies* **SAT** *and* **CF***.*

The main idea of the counterfactual criterion CF is that for every trace π_0 that does not satisfy the cause, there exists a closer trace π_2 that does not satisfy the cause *and* the effect. The additional quantification over π_1 is a technicality included because the cause $\mathsf{C} \subseteq (2^I)^\omega$ consists of input sequences while $\pi_2 \in$ *traces*$(\mathcal{T})$ is a full system trace. It also closely mirrors the structure of Coenen et al.'s PC2 criterion (cf. Definition 3) which it neatly generalizes to similarity relations that do not satisfy the limit assumption: If the assumption holds, then a π_2 is, in particular, required for the *closest* traces π_0 in $\overline{\mathsf{C}}$, for which π_2 can only be instantiated by themselves. Hence, the closest traces are required to not satisfy the effect (we develop this comparison more formally in Sect. 4.3). If the limit assumption does not hold and there exists an infinite chain of ever-closer traces $\pi_0 \in \overline{\mathsf{C}}$, the condition requires that for all these π_0 there is a closer π_2 that avoids the effect, even in infinity: No matter how far we descend on this chain, we are always guaranteed that we can descend further towards a closer counterfactual trace that does not satisfy the effect.

Example 2. To illustrate these conditions with a concrete example, consider the system from Fig. 1, the trace $\pi = \{x, e\}^\omega$, the effect $\mathsf{E} = \mathcal{L}(\Box\Diamond e)$, and the cause $\mathsf{C} = \mathcal{L}(\Box\Diamond x)$, with similarity relation $\leq^{subset}$. It is easy to that SAT is satisfied, as the system is deterministic and $\pi|_I = \{x\}^\omega \in \mathsf{C}$ and $\pi \in \mathsf{E}$. There is, as discussed in Sect. 3.2, an infinite chain in $(\mathcal{L}(\Diamond\Box\neg x), \leq_\pi^{subset})$ and, hence, no closest trace. We require for all $\pi_0 \in \mathcal{L}(\Diamond\Box\neg x) = \overline{\mathsf{C}}$ a $\pi_1 \in \overline{\mathsf{C}}$ with $\pi_1 \in \mathcal{L}(\Diamond\Box\neg e) = \overline{\mathsf{E}}$ and a $\pi_2 =_I \pi_1$ such that $\pi_2 \in \overline{\mathsf{E}}$. In this case, we can pick π_1 as π_0 and π_2 as the corresponding system trace, hence CF is satisfied. To see that MIN is satisfied, consider any strict subset $\mathsf{C}' \subset \mathsf{C}$. Hence, there is some $\pi' \in \overline{\mathsf{C}'}$ such that $\pi' \vDash \Box\Diamond x$. Then, all system traces π_2 with $\pi_2 \leq_\pi \pi'$ satisfy $\pi_2 \vDash \Box\Diamond x$ by the definition of $\leq_\pi^{subset}$, and in this system this also means $\pi_2 \vDash \Box\Diamond e$. Hence, C satisfies MIN because no strict subset satisfies CF.

Remark 1. Note that Definition 2 is not restricted to similarity relations that can be expressed via zipped traces and LTL formulas as used in the previous examples, but instead applies to any comparative similarity relation as defined at the start of this section.

4.3 Proving Generalization

This section is dedicated to proving that our generalization (Definition 2) is conservative, i.e., agrees with Coenen et al.'s original definition whenever the underlying similarity relation satisfies the limit assumption and the actual trace is deterministic. First, we recall Coenen et al.'s definition.

Definition 3 (Coenen et al. [19]). *Let $\mathcal{T}$ be a system, $\pi \in traces(\mathcal{T})$ a trace, $\leq_\pi$ a similarity relation, and $\mathsf{E} \subseteq (2^O)^\omega$ an effect property. $\mathsf{C} \subseteq (2^I)^\omega$ is a cause of E on π in $\mathcal{T}$ if the following three conditions hold.*

PC1: $\pi|_I \in \mathsf{C}$ *and* $\pi \in \mathsf{E}$.

PC2: *For all closest counterfactual traces $\pi_0 \in \overline{\mathsf{C}}$, i.e., traces for which there are no closer traces $\pi_1 \in \overline{\mathsf{C}}$ with $\pi_1 <_\pi \pi_0$, there exists a $\pi_2 \in traces(\mathcal{T})$ such that $\pi_0 =_I \pi_2$ and $\pi_2 \in \overline{\mathsf{E}}$.*

PC3: *There is no $\mathsf{C}' \subset \mathsf{C}$ such that C' satisfies* **PC1** *and* **PC2**.

Unlike in our updated definition, PC1 only works if the actual trace π is deterministic. If the π is nondeterministic, the effect can be avoided with no modifications at all to π (which is minimal), hence the cause should be empty. PC1 does not reflect this and allows to build a cause that includes $\pi|_I$ (and possibly more), wrongfully implying that a modification of the sequence is required to avoid the effect. PC2 may be vacuously satisfied if the similarity relation does not satisfy the limit assumption, as outlined in Sect. 3.2.

Remark 2. Note that Coenen et al. consider traces $\pi \in traces(\mathcal{C}_\pi^\mathcal{T})$ of the *counterfactual automaton* $\mathcal{C}_\pi^\mathcal{T}$ for PC2. This automaton models *contingencies*, which allow to partially reset outputs back to as they were on the actual trace π, and to change the system state accordingly. For PC2 in Definition 3, this means that the closest counterfactual traces π_2 do not have to avoid the effect themselves, but together with some contingency. This mechanism, inspired by Halpern's modified version of actual causality [29], was extended by Coenen et al. [18,19] to lasso-shaped traces and finite state machines to sometimes obtain more accurate causes. However, to guarantee meaningful results, the original system has to have unique output labels. Beutner et al.'s implementation [11] therefore allows to toggle the usage of contingencies. Similarly, our generalization works both with contingencies and without. For the latter case, one simply supplants $\mathcal{T}$ with $\mathcal{C}_\pi^\mathcal{T}$ in both definitions. Our cause synthesis algorithm can also handle contingencies, and our implementation allows to toggle them as a feature. Our theoretical contribution is independent of this detail.

We now proceed to show the equivalence between our definition (Definition 2) and Coenen et al.'s definition (Definition 3) in case the limit assumption is fulfilled and the actual trace is deterministic. We start with proving the equivalence of the counterfactual conditions CF and PC2, which holds regardless of nondeterminism on the actual trace.

Lemma 1. *Let $\mathcal{T}$ be a system, $\pi \in traces(\mathcal{T})$ a trace, $\mathsf{C} \subseteq (2^I)^\omega$ a cause property, and $\mathsf{E} \subseteq (2^{AP})^\omega$ an effect property. Let $\leq$ be a similarity relation that satisfies the limit assumption. Then we have that* PC2 *is satisfied iff* CF *is satisfied.*

With Lemma 1 at hand, we only need to address the differences between PC1 and SAT. It is easy to see that their equivalence fails when behavior on the actual trace π is nondeterministic, i.e., when there is another trace that is input-equivalent to π but does not satisfy the effect. In such a case, PC1 is satisfied but SAT is not. Hence, our definition is equivalent to Coenen et al.'s definition only in deterministic systems, as we deliberately diverge in the case of nondeterminism on the actual trace. Notably, Lemma 1 holds for both deterministic and nondeterministic systems, and determinism is only relevant on the actual trace. The restriction to output-only effects $\mathsf{E} \subseteq (2^O)^\omega$ is inherited from Coenen et al.'s definition, but technically not necessary.

Theorem 1. *Let $\leq$ be a similarity relation that satisfies the limit assumption. Then $\mathsf{C} \subseteq (2^I)^\omega$ is a cause for $\mathsf{E} \subseteq (2^O)^\omega$ on a trace π that is deterministic in $\mathcal{T}$ according to our definition (Definition 2) if and only if it is a cause according to Coenen et al.'s definition (Definition 3).*

5 Cause Synthesis

In this section, we develop our algorithm for synthesizing causes. In Sect. 5.1 we formalize the characterization of a cause as the complement of the upper closure of the negated effect, which we have discussed intuitively in Sect. 3.1. In Sect. 5.2 we provide an algorithm for cause synthesis in the ω-regular setting, when the effect is given as a nondeterministic Büchi automaton and the actual trace is in a lasso shape.

5.1 Proving Our Characterization

For this section, we fix a system $\mathcal{T}$, an actual trace $\pi \in traces(\mathcal{T})$, a similarity relation $\leq$, and an effect E. We now show that, if it exists, the cause for E on π is the complement of the upward closure of $\overline{\mathsf{E}}$ in $(traces(\mathcal{T}), \leq_\pi)$. Formally, we construct a set D that is a cause for E on π via its complement:

$$\overline{\mathsf{D}} = \{\, \rho \in (2^I)^\omega \mid \exists \sigma \in traces(\mathcal{T}).\ \sigma \leq_\pi \rho \land \sigma \in \overline{\mathsf{E}} \,\}\ ,\ \text{hence}$$
$$\mathsf{D} = \{\, \rho \in (2^I)^\omega \mid \forall \sigma \in traces(\mathcal{T}).\ \sigma \leq_\pi \rho \rightarrow \sigma \in \mathsf{E} \,\}\ .$$

The set D directly corresponds to the (unique) cause if there exists one, and is empty if there is none. We establish this in a series of lemmas.

Lemma 2. *If the set* D *is non-empty, it is a cause for* E *on* π *in* $\mathcal{T}$*.*

Lemma 2 shows that D satisfies Definition 2 assuming it is non-empty. The assumption is only required for SAT, as this criterion requires that π and all input-equivalent traces are in the cause. CF follows from the definition of D, and for MIN we can show that any strict subset of D does not satisfy CF.

Lemma 3. *Iff the set* D *is empty, there exists no cause that satisfies SAT.*

Lemma 3 serves two purposes. First, it helps us argue for the completeness of our construction. Second, it shows that the only reason why there may be no cause is due to a nondeterministic actual trace. To fully argue completeness, we show that causes are unique, and hence D is the only relevant cause in all cases.

Lemma 4. *Causes are (semantically) unique: There can be no two sets* $\mathsf{C} \neq \mathsf{C}'$ *that are both causes for some effect property* E *on a trace* π *in some system* $\mathcal{T}$*.*

Remark 3. This does *not* mean that there can only exist a single causal event, such as "a at position 0" or "b at position 1", in a given scenario. Instead, Lemma 4 states that the semantics of the symbolic description of the causal behavior in a given scenario is unique. It is precisely the idea of temporal causality to encompass multiple single events in a single symbolic description, e.g., through a conjunction such as $a \wedge \bigcirc b$.

5.2 Cause-Synthesis Algorithm for ω-Regular Effects

In Sect. 5.1, we have established a direct characterization of causes as downward closed sets, independent of any concrete descriptions of cause, effect, and trace. In this section, we develop an automata-based algorithm for synthesizing causes of ω-regular effects given, e.g., by a nondeterministic Büchi automaton (NBA), on lasso-shaped traces. We assume that the relation $\leq\ \subseteq (2^I)^\omega \times (2^I)^\omega \times (2^I)^\omega$ is definable by a relational ω-regular property $\mathsf{P}_{\leq} \subseteq (2^{I \times \{t_0, t_1, t_2\}})^\omega$, such that $(\pi_0, \pi_1, \pi_2) \in\ \leq$ iff the zipped trace $\mathit{zip}(\pi_0, \pi_1, \pi_2)$ satisfies $\mathsf{P}_{\leq}$. Note that this applies to all concrete similarity relations introduced in Sect. 4. We show that under these assumptions, the set D from Sect. 5.1 can be constructed as an NBA. First, we construct an NBA for $\overline{\mathsf{D}}$ and subsequently complement it. This is necessary because we start out from an NBA representation for the effect, and assume the similarity relation to be given by an NBA as well. Since the NBAs acceptance condition is existential, we need the additional complementations to express the universal quantification over the closer traces σ appearing in the definition of D.

The main technical difficulty that remains is to ensure that the conditions on the three traces π_{actual}, π_{close} and π_{far}, as they appear in the alphabet of a similarity relation, are applied consistently, and that the quantification over σ in D, which corresponds to π_{close}, is resolved at the correct step, as the automaton should range over the inputs I and not, e.g., over $I \times \{t_0, t_1, t_2\}$ as used by the similarity relation.

Similarity Relation. Our starting point is the NBA for the similarity relation defined by the ω-regular property $\mathsf{P}_{\leq}$: $\mathcal{A}^I_{\leq} = (Q_{\leq}, 2^{I\times\{t_0,t_1,t_2\}}, Q^0_{\leq}, F_{\leq}, \Delta^I_{\leq})$. The automaton $\mathcal{A}^I_{\leq}$ only reasons about inputs and uses tuples with the trace variables t_0, t_1 and t_2 to encode whether the input appears on the actual, closer or farther trace, respectively. We lift the automaton to the full set of atomic propositions as the automaton $\mathcal{A}_{\leq} = (Q_{\leq}, 2^{(I\times\{t_0,t_1,t_2\})\cup(O\times\{t_0,t_1\})}, Q^0_{\leq}, F_{\leq}, \Delta_{\leq})$. The transition relation is defined as follows, for a letter w: $q_2 \in \Delta_{\leq}(q_1, w)$ iff $q_2 \in \Delta^I_{\leq}\big(q_1, w \setminus (O \times \{t_0, t_1\})\big)$. Hence, $\mathcal{A}_{\leq}$ specifies the same relation between the inputs of the three traces as $\mathcal{A}^I_{\leq}$, but allows arbitrary output behavior. Its alphabet does not contain outputs for π_2, as these traces eventually form the elements of the cause, which only ranges over the inputs.

Effect. Next, we modify the NBA $\mathcal{A}^*_{\mathsf{E}} = (Q_{\mathsf{E}}, 2^{AP}, q_{\mathsf{E}}, F_{\mathsf{E}}, \Delta^*_{\mathsf{E}})$ for the ω-regular effect E such that it refers to the closer trace t_1 and ranges over the same alphabet as $\mathcal{A}_{\leq}$. We obtain $\mathcal{A}_{\mathsf{E}} = (Q_{\mathsf{E}}, 2^{(I\times\{t_0,t_1,t_2\})\cup(O\times\{t_0,t_1\})}, q_{\mathsf{E}}, F_{\mathsf{E}}, \Delta_{\mathsf{E}})$ with:

$$q_2 \in \Delta_{\mathsf{E}}\big(q_1, (w \times \{t_1\}) \cup X \cup Y\big) \text{ iff}$$
$$q_2 \in \Delta^*_{\mathsf{E}}(q_1, w) \wedge X \subseteq (AP \times \{t_0\}) \wedge Y \subseteq (I \times \{t_2\}) \ .$$

Hence, $\mathcal{A}_{\mathsf{E}}$ restricts π_1 to be in E by restricting it to the transition relation of $\mathcal{A}^*_{\mathsf{E}}$, while allowing an arbitrary trace π_0 and arbitrary input sequence in π_2.

Intersection. For the conjunction that defines the set $\overline{\mathsf{D}}$, we intersect $\mathcal{A}_{\leq}$ with the complement of $\mathcal{A}_{\mathsf{E}}$ to obtain $\mathcal{A}_{\cap} = (Q_{\cap}, 2^{(I\times\{t_0,t_1,t_2\})\cup(O\times\{t_0,t_1\})}, Q^0_{\cap}, F_{\cap}, \Delta_{\cap})$ such that: $\mathcal{A}_{\cap} = \mathcal{A}_{\leq} \cap \overline{\mathcal{A}_{\mathsf{E}}}$.

System Product. As the next step, we construct the product of the automaton $\mathcal{A}_{\cap}$ with the system $\mathcal{T} = (S, s_0, AP, \delta, l)$, ensuring that the atomic propositions t_1 are picked from a valid system trace. When building the product, we project away explicit atomic propositions paired with t_1, as the traces of the desired set D are only the traces paired with t_2. The resulting automaton is $\mathcal{A}_{\times} = (S \times Q_{\cap}, 2^{(I\times\{t_0,t_2\})\cup(O\times\{t_0\})}, \{s_0\} \times Q^0_{\cap}, S \times F_{\cap}, \Delta_{\times})$, where

$$\Delta_{\times}\big((s_i, q_i), w\big) = \big\{(s_{i+1}, q_{i+1}) \mid \exists A \subseteq I.\ s_{i+1} \in \Delta_{\cap}(s_i, A)$$
$$\wedge\, q_{i+1} \in \Delta_{\cap}(q_i, (l(s_{i+1}) \cup A) \times \{t_1\})\big\} \ .$$

Cause Automaton. To obtain the final result, we first complement the automaton from the previous step to obtain $\overline{\mathcal{A}_{\times}} = (Q_{\times}, 2^{(I\times\{t_0,t_2\})\cup(O\times\{t_0\})}, Q^0_{\times}, F_{\times}, \Delta_{\times})$, and then build the product with the trace. At the same step we project away atomic propositions paired with t_0, and remove the trace variable t_2 to obtain the alphabet 2^I for the cause. For the lasso-shaped trace $\pi = \pi_0 \ldots \pi_{j-1} \cdot (\pi_j \ldots \pi_k)^\omega$ we define the set of positions as $\Pi = \{\pi_0, \ldots, \pi_k\}$ and a successor function $succ : \Pi \mapsto \Pi$ as $succ(\pi_r) = \pi_{r+1}$ for $r < k$, and $succ(\pi_k) = \pi_j$. The cause automaton is then $\mathcal{A}_{\mathsf{D}} = (\Pi \times Q_{\times}, 2^I, \{\pi_0\} \times Q^0_{\times}, \Pi \times F_{\times}, \Delta_{\mathsf{D}})$, where

$$\Delta_{\mathsf{D}}\big((\pi_i, q_i), w\big) = \big\{(succ(\pi_i), q_{i+1}) \mid\ q_{i+1} \in \Delta_{\times}\big(q_i, (\pi_i \times \{t_0\}) \cup (w \times \{t_2\})\big)\big\} \ .$$

From the lemmas established in Sect. 5.1, we conclude that there is a cause iff $\mathcal{A}_{\mathsf{D}}$ is non-empty, and then the cause is uniquely determined by its language.

Corollary 1. *The language of* $\mathcal{A}_{\mathsf{D}}$ *is empty iff there is no cause* C *for* E *on* π *in* $\mathcal{T}$*, and if* $\mathcal{L}(\mathcal{A}_{\mathsf{D}})$ *is non-empty, then it is the unique cause for* E *on* π *in* $\mathcal{T}$*.*

We can also state an upper bound on the size of $\mathcal{A}_{\mathsf{D}}$, which is dominated by the potentially exponential growth from NBA complementation [47].

Proposition 3. *If the effect* E *and the similarity relation* $\leq$ *are given as NBAs* $\mathcal{A}_{\mathsf{E}}$ *and* $\mathcal{A}_{\leq}$*, respectively, then the size of* $\mathcal{A}_{\mathsf{D}}$ *is in* $|\pi| \cdot 2^{(2^{\mathcal{O}(|\mathcal{A}_{\mathsf{E}}|)} \cdot |\mathcal{A}_{\leq}| \cdot |\mathcal{T}|)}$*.*

Note that the doubly-exponential upper bound in the description of E persists independent of whether it is given as an NBA or LTL formula. In the latter case, we simply translate the negated formula, which again leads to an exponential blow-up. In theory, the description does make a difference for $\leq$: If it is represented as a formula, we first need to translate it with a potentially exponential increase in size, hence it would move up one exponent in the bound. In practice, the canonical similarity relation $\leq^{subset}$ can always be represented by a 1-state NBA, such that its contribution to the bound is less relevant.

While the stated upper bound may seem daunting, it mirrors the (tight) bounds of related problems, such as LTL synthesis [45]. In the following section, we show that, not only can our approach solve many cause-synthesis problems in practice, it also significantly improves upon previous methods for cause checking.

6 Implementation and Evaluation

In this section, we evaluate a prototype tool implementing our cause-synthesis approach, called CORP - *Causes for Omega-Regular Properties.*[4] Our prototype is written in Python and uses Spot [21] for automata operations and manipulation. The prototype allows for both cause synthesis and cause checking, where in the latter case the correct cause is first synthesized and than checked for equivalence against the cause candidate. This allows for a direct comparison with the cause checking tool CATS [11] in Sect. 6.2. Before, we report on our experiments on cause synthesis, where we compare our method with the incomplete, sketch-based approach of CATS. All experiments were carried out on a machine equipped with a 2.8 GHz Intel Xeon processor and 64 GB of memory, running Ubuntu 22.04.

6.1 Cause Synthesis

We conducted three different experiments that highlight how the similarity relations, effect size and system size contribute to the performance of our algorithm.

[4] Our prototype is on GitHub: https://github.com/reactive-systems/corp. Our full evaluation can be reproduced with the artifact on Zenodo: https://doi.org/10.5281/zenodo.10946309.

Table 1. Cause synthesis for arbiters. $|\mathcal{T}|$ is the number of system states. In all instances, π is the (unique) trace where all n clients send a request at every position, which has length $|\pi| = n$. φ_E is the effect. We report the time taken to synthesize the causal NBA with the metrics $\leq^{full}$ and $\leq^{subset}$ in seconds and the respective NBA sizes $|\mathcal{A}_\mathsf{C}^{full}|$ and $|\mathcal{A}_\mathsf{C}^{subset}|$, and provide an LTL description φ_C of the NBA language (guessed manually). TO denotes the timeout of 60 seconds.

Instance	$\|\mathcal{T}\|$	φ_E	$t(\leq^{full})$	$t(\leq^{subset})$	$\|\mathcal{A}_\mathsf{C}^{full}\|$	$\|\mathcal{A}_\mathsf{C}^{subset}\|$	φ_C
Spurious 1	1	$\Diamond g_0$	0.11	0.11	1	1	*true*
Spurious 2	2	$\Diamond g_0$	0.11	0.11	1	1	*true*
Spurious 3	3	$\Diamond g_0$	0.21	0.11	1	1	*true*
Spurious 4	4	$\Diamond g_0$	TO	0.11	-	1	*true*
Unfair 2	2	$\Box\neg g_0$	0.11	0.11	1	1	$\Box r_{prio}$
Unfair 3	4	$\Box\neg g_0$	0.16	0.11	1	1	$\Box r_{prio}$
Unfair 4	6	$\Box\neg g_0$	TO	0.11	-	1	$\Box r_{prio}$
Full 1	1	$\Diamond g_0$	0.11	0.11	2	2	$\Diamond r_0$
		$\Box\Diamond g_0$	0.11	0.11	2	2	$\Box\Diamond r_0$
Full 2	4	$\Diamond g_0$	0.11	0.11	2	2	$\Diamond r_0$
		$\Box\Diamond g_0$	0.11	0.11	12	4	$\Box\Diamond r_0$
Full 3	11	$\Diamond g_0$	0.16	0.11	2	2	$\Diamond r_0$
		$\Box\Diamond g_0$	2.04	0.16	215	24	$\Box\Diamond r_0$
Full 4	46	$\Diamond g_0$	TO	0.11	-	2	$\Diamond r_0$
		$\Box\Diamond g_0$	TO	33.22	-	214	$\Box\Diamond r_0$

Arbiters. We computed causes on traces of resource arbiters to compare the performance of our algorithm under different similarity relations, whose logical description scales in the number of system inputs. An arbiter instance is parameterized by a number of clients n, each with its own input. This let us easily scale the size of the similarity relation's description. For some number n of clients (indexed by k) that request access to a shared resource with a request r_k, an arbiter grants mutually exclusive access to the resource with a grant g_k. We considered different arbiter strategies, and for each we synthesize causes as NBAs $\mathcal{A}_\mathsf{C}^{full}$ and $\mathcal{A}_\mathsf{C}^{subset}$ with the similarity relations $\leq^{full}$ and $\leq^{subset}$, respectively. The results of these instances are depicted in Table 1. Spurious arbiters simply give out grants to all clients in a round-robin manner, regardless of previous requests. Unfair arbiters prioritize one client with request r_{prio} over the others, while full arbiters are fully functional arbiters that only give out grants that were requested beforehand. In all instances, we computed causes on the (unique) trace π where all clients send requests continuously, i.e., $\pi|_I = \{r_0, \ldots, r_n\}^\omega$. Consequently, on this trace both the spurious and the full arbiter send grants to all clients, while the unfair arbiter only gives grants to the prioritized client. These varying strategies are reflected in the synthesized cause-effect pairs. In the spurious arbiters, the language of the synthesized cause NBA for the effect $\Diamond g_0$ is *true*, which reflects that the effect appears on *all* system traces. In the unfair arbiters, the cause for no grant being given to client 0 is that the prioritized arbiter sends requests permanently, i.e., the causal NBA has the language $\square r_{prio}$. In the full arbiters, $\diamond g_0$ is caused, as expected, by $\diamond r_0$ and $\square\diamond g_0$ is caused by $\square\diamond r_0$. From a performance standpoint, the arbiter instances show us that accommodating the canonical similarity relation $\leq^{subset}$, as we did through our generalization of temporal causality in Sect. 4, leads to significant improvements in practice: In all instances, synthesizing causes with $\leq^{subset}$ was faster than with $\leq^{full}$, and the resulting causal NBAs were smaller as well. This is mostly because of the number of inputs involved: The other parameters stay comparably small when going from the spurious 1-arbiter to the spurious 4-arbiter, but the latter times out when using $\leq^{full}$. When the systems get larger and the effects more complex, e.g., in the instance of the full 4-arbiter with the effect $\square\diamond g_0$, the automata produced can become bigger even with $\leq^{subset}$. However, the language of the produced automata has a small representation, i.e., $\square\diamond r_0$, such that we see potential for improvement through automata minimization techniques.

Neural Synthesis. For more diverse effects, we considered mispredicted circuits from a neural synthesis model [48]. Given some specification (in this case, generated by Spot's `randltl`) the neural model predicts an implementation as an AIGER [12] circuit, which is in the end model-checked against the specification. Since neural synthesis is not sound, this check fails occasionally and returns a counterexample, which may be used for further repair [20]. We used our tool CORP to compute the cause for the violation of the specification on such a counterexample. In Fig. 3 we report the time of computing causes with respect to size of the syntax tree of the effect formula, and the system size. The timeout was set to 100 s. The size of the points in the scatter plot corresponds to the length of the counterexample and the color to the system size. From the plot we can deduce that a large effect does not mean a long runtime of our tool per se. However, a combination of large effects, bigger systems, and longer counterexamples usually means that the tool takes longer. The sizes of the synthesized causes are diverse and range from 2 to 60 states.

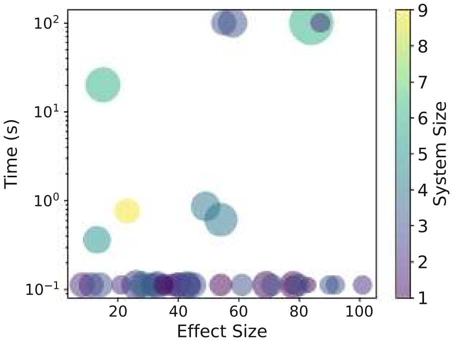
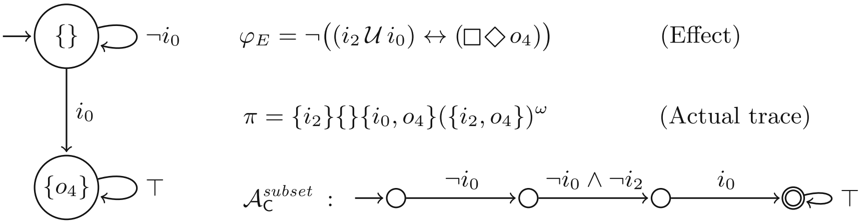


Fig. 3. Computing causes for neural synthesis mispredictions with CORP. Size of a point represents the length of the counterexample (between 2 and 16).

Example 3. We discuss an illustrative example of cause synthesis with a small benchmark from the neural synthesis datatset. All relevant inputs and outputs of


$\{\}$
$\neg i_0$
i_0
$\{o_4\}$
$\top$
$\varphi_E = \neg\big((i_2 \mathcal{U} i_0) \leftrightarrow (\Box\Diamond o_4)\big)$
(Effect)
$\pi = \{i_2\}\{\}\{i_0, o_4\}(\{i_2, o_4\})^\omega$
(Actual trace)
$\mathcal{A}_{\mathsf{C}}^{subset}$:
$\neg i_0$
$\neg i_0 \wedge \neg i_2$
i_0
$\top$

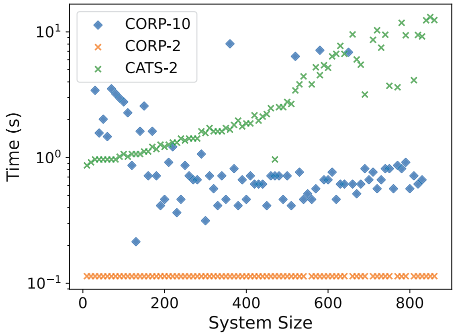
(a) Faulty system. **(b)** Effect φ_{E}, actual trace π and cause automaton $\mathcal{A}_{\mathsf{C}}^{subset}$.

Fig. 4. A system predicted wrongly by a neural synthesis model (Fig. 4a) for the specification $\neg\varphi_{\mathsf{E}}$, i.e., the negation of the effect. The effect is shown together with the actual trace π, i.e., a counterexample obtained from model checking, and the computed cause automaton $\mathcal{A}_{\mathsf{C}}^{subset}$ in Fig. 4b.

our cause synthesis algorithm are depicted in Fig. 4. First, we have the system (cf. Fig. 4a), which is a wrongly predicted circuit of the neural synthesis model. This model tried to come up with a solution for the specification $(i_2 \mathcal{U} i_0) \leftrightarrow (\Box\Diamond o_4)$, i.e., o_4 appears infinitely often if and only if input i_2 is enabled until input i_0 is enabled. The predicted system does not satisfy this specification, because there are cases where $\Box\Diamond o_4$ holds without the inputs meeting the required condition. Hence, model checking the specification returns a counterexample π that violates the formula, which means the negated specification can be seen as an effect φ_{E} that is present on the counterexample π (cf. Fig. 4b). Our algorithm then computes the cause for this effect, i.e., for the violation of the specification, on the counterexample π, as a nondeterministic Büchi automaton. The computed automaton $\mathcal{A}_{\mathsf{C}}^{subset}$ is depicted at the bottom of Fig. 4b. It is language-equivalent to the LTL formula $\neg i_0 \wedge \bigcirc(\neg i_0 \wedge \neg i_2 \wedge \bigcirc i_0)$, which basically states that the effect is caused by a conjunction of four inputs spread out over the first three steps. Indeed, it is easy to see that modifying any of these four inputs results in a trace that satisfies the specification: For instance, setting i_0 at the first position results in the trace that immediately enters the state labeled with o_4 and loops there forever such that the left part of the equivalence is satisfied, while removing i_0 from the third position results in looping in the initial state such that the right part of the equivalence is not satisfied anymore.

Comparison with Cause Sketching. CATS, the tool of Beutner et al. [11], allows to enumerate non-temporal formulas in holes of a provided cause sketch until a cause is found. If the effect contains $\bigcirc$ as the only temporal operator and a cause exists, there is a sketch that is guaranteed to encompass the cause. This provides us with a baseline with which we can compare our cause-synthesis algorithm. We constructed random benchmarks that fall into CATS' complete fragment using Spot's `randaut` function to generate systems with 10 up to 1000 states, obtaining traces of length 2 and then inserting a new atomic proposition e at the last position of the trace and in the system. The effect then is defined as the occurrence of e at exactly this position. We chose such small traces and effects because CATS timed out already on slightly larger instances. We conducted additional experiments using just our tool CORP with traces (and effects) of size 10. Figure 5a shows the time taken by CATS and CORP to synthesize causes. The influence of the system size on the runtime of CORP in this setting is negligible, which we believe is due to the efficient automata operations performed by Spot. The hyperproperty encoding of CATS does not seem as amenable to similar optimizations.

(a) Cause synthesis.



(b) Cause checking.

Instance	CATS	CORP
Spurious Arbiter	0.72	0.11
Simple Arbiter	0.97	0.11
Arbiter	22.84	0.11
Instance 6, odd	1.81	0.11
Instance 8	0.92	0.11
TP Right	0.87	0.11

Fig. 5. Direct comparisons between our tool CORP and the tool CATS [11]. Figure 5a shows the time CATS needs to synthesize a cause in its complete fragment with trace and effect of size 2, and the time taken by CORP for sizes 2 and 10. Figure 5b shows the time taken to check single causal relationships. These problems are taken from Beutner et al. [11] (where “Instances” are “Examples”).

6.2 Cause Checking

It is straightforward to use our cause synthesis algorithm to also check causes through an equivalence check between the synthesized causal NBA and the candidate formula (or automaton). This allows a direct performance comparison with the cause checking tool CATS of Beutner et al. [11], which we conducted on the publicly available benchmarks of their paper. In these cause-checking benchmarks, a cause candidate is given in addition to the system, actual trace and effect. The time CATS and our tool CORP took in each instance to check whether the given candidate is a cause is depicted in Fig. 5b. Somewhat surprisingly, our cause checker based on cause synthesis performs significantly better on all benchmarks. This shows that our characterization of causes as complements of the upward closure of the negated effect (cf. Section 5.1) is more efficient than encoding the cause-checking instances into a hyperlogic, as done by CATS.

7 Related Work

The study of causality and its applications in formal methods has gained great interest in recent years [3]. In a finite setting, Ibrahim et al. use SAT solvers and linear programming to check [35] and infer [34] actual causes. Our definition of actual causality for reactive systems extends the definitions of Coenen et al. [19] to cases in which the limit assumption does not hold. While Coenen et al. study the theory of actual causality [29] in reactive systems, they do not provide a way to *generate* causes and explanations. In terms of cause *synthesis*, the most related work is by Beutner et al. [11], which checks causality and generates causes based on sketching. Unlike ours, their tool is only applicable for the small fragment of LTL containing only $\bigcirc$ operators, while we are able to generate temporal causes for all ω-regular specifications.

In a series of works, Leue et al. study symbolic description of counterfactual causes in *Event Order Logic* [14,38,39]. However, this logic can only reason about the ordering of events, and not their absolute timing, as we can do with ω-regular properties (e.g., specifying that the input at the second position is the cause).

Gössler and Métayer [24] define causality for component-based systems, and Gössler and Stefani [25] study causality based on counterfactual builders. Their formalisms differ from ours, which is based on Coenen et al. [19], and none of the works considers cause *synthesis*.

Most other works related to cause synthesis concern generating explanations for effects observed on finite traces [5,26,27,49], or effects restricted to safety properties [43]. In the context of cause synthesis over infinite traces for effects given as temporal specifications, existing works are limited to causes given as sets of events (i.e., atomic propositions and times points) [7,18,32] or take a state-centric view to, e.g., measure the responsibility of a state for an observed effect property [1,4,42].

8 Conclusion

This paper presents the first complete algorithm to compute temporal causes for arbitrary ω-regular properties. It is based on a new, generalized version of temporal causality that solves a central dilemma of previous definitions by loosening the assumptions on similarity relations. From a philosophical perspective, this is an immense step forward since it is the first definition that accommodates the canonical similarity relation used in previous literature. Our experimental results show that our generalization also leads to significant improvements from a practical perspective. These mainly stem from characterizing causes based on set-closure properties, which may be an interesting approach for counterfactual causality in other formalisms. Besides, our work opens up exciting research directions on generating explanations from temporal causes, i.e., as formulas or annotations in highlighted counterexamples.

Acknowledgements. We thank Matthias Cosler and Frederik Schmitt for providing us with the neural synthesis benchmarks. This work was partially supported by the DFG in project 389792660 (Center for Perspicuous Systems, TRR 248) and by the ERC Grant HYPER (No. 101055412).

References

1. Baier, C., van den Bossche, R., Klüppelholz, S., Lehmann, J., Piribauer, J.: Backward responsibility in transition systems using general power indices. In: Wooldridge, M.J., Dy, J.G., Natarajan, S. (eds.) Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, 20-27 February 2024, Vancouver, Canada, pp. 20320–20327. AAAI Press (2024). https://doi.org/10.1609/AAAI.V38I18.30013
2. Baier, C., Coenen, N., Finkbeiner, B., Funke, F., Jantsch, S., Siber, J.: Causality-based game solving. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 894–917. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_42
3. Baier, C., Dubslaff, C., Funke, F., Jantsch, S., Majumdar, R., Piribauer, J., Ziemek, R.: From Verification to Causality-Based Explications. In: Bansal, N., Merelli, E., Worrell, J. (eds.) 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 198, pp. 1:1–1:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://drops.dagstuhl.de/opus/volltexte/2021/14070
4. Baier, C., Dubslaff, C., Funke, F., Jantsch, S., Piribauer, J., Ziemek, R.: Operational causality - necessarily sufficient and sufficiently necessary. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 13560, pp. 27–45. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15629-8_2
5. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: Aiken, A., Morrisett, G. (eds.) Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, 15-17 January 2003, pp. 97–105. ACM (2003). https://doi.org/10.1145/604131.604140
6. 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
7. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_11
8. Beutner, R., Finkbeiner, B.: HyperATL*: A logic for hyperproperties in multi-agent systems. Log. Methods Comput. Sci. **19**, 13:1–13:44 (2023)
9. Beutner, R., Finkbeiner, B.: Model checking omega-regular hyperproperties with autohyperq. In: Piskac, R., Voronkov, A. (eds.) LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and

Reasoning, Manizales, Colombia, 4-9th June 2023. EPiC Series in Computing, vol. 94, pp. 23–35. EasyChair (2023). https://doi.org/10.29007/1XJT

10. Beutner, R., Finkbeiner, B., Frenkel, H., Metzger, N.: Second-order hyperproperties. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, 17-22 July 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 309–332. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37703-7_15
11. Beutner, R., Finkbeiner, B., Frenkel, H., Siber, J.: Checking and sketching causes on temporal sequences. In: André, É., Sun, J. (eds.) Automated Technology for Verification and Analysis - 21st International Symposium, ATVA 2023, Singapore, 24-27 October 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14216, pp. 314–327. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-45332-8_18
12. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. Rep. 07/1, Inst. f. Form. Model. u. Verifikation, Johannes Kepler University (2007)
13. Buechi, J.R.: On a decision method in restricted second-order arithmetic. In: International Congress on Logic, Methodology, and Philosophy of Science (1962)
14. Caltais, G., Guetlein, S.L., Leue, S.: Causality for general LTL-definable properties. In: Workshop on Formal Reasoning About Causation, Responsibility, and Explanations in Science and Technology, CREST 2018. EPTCS, vol. 286 (2018). https://doi.org/10.4204/EPTCS.286.1
15. Chaki, S., Groce, A., Strichman, O.: Explaining abstract counterexamples. In: Taylor, R.N., Dwyer, M.B. (eds.) Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004, pp. 73–82. ACM (2004). https://doi.org/10.1145/1029894.1029908
16. 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
17. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
18. Coenen, N., Dachselt, R., Finkbeiner, B., Frenkel, H., Hahn, C., Horak, T., Metzger, N., Siber, J.: Explaining hyperproperty violations. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. LNCS, vol. 13371, pp. 407–429. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_20
19. Coenen, N., Finkbeiner, B., Frenkel, H., Hahn, C., Metzger, N., Siber, J.: Temporal causality in reactive systems. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, 25-28 October 2022, Proceedings. LNCS, vol. 13505, pp. 208–224. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19992-9_13
20. Cosler, M., Schmitt, F., Hahn, C., Finkbeiner, B.: Iterative circuit repair against formal specifications. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, 1-5 May 2023. OpenReview.net (2023). https://openreview.net/pdf?id=SEcSahl0Ql
21. Duret-Lutz, A., et al.: From spot 2.0 to spot 2.10: What's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, 7-10 August 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer, Cham (2022. https://doi.org/10.1007/978-3-031-13188-2_9

22. Finkbeiner, B., Frenkel, H., Metzger, N., Siber, J.: Synthesis of temporal causality. CoRR (2024). https://doi.org/10.48550/ARXIV.2405.10912, https://arxiv.org/abs/2405.10912, full version
23. Finkbeiner, B., Siber, J.: Counterfactuals modulo temporal logics. In: Piskac, R., Voronkov, A. (eds.) LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023. EPiC Series in Computing, vol. 94, pp. 181–204. EasyChair (2023). https://doi.org/10.29007/QTW7
24. Gössler, G., Le Métayer, D.: A general trace-based framework of logical causality. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 157–173. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07602-7_11
25. Gössler, G., Stefani, J.: Causality analysis and fault ascription in component-based systems. Theor. Comput. Sci. **837** (2020). https://doi.org/10.1016/j.tcs.2020.06.010
26. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. **8**(3) (2006). https://doi.org/10.1007/s10009-005-0202-0
27. Groce, A., Kroening, D., Lerda, F.: Understanding counterexamples with explain. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 453–456. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_35
28. 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
29. Halpern, J.Y.: A modification of the Halpern-pearl definition of causality. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, 25-31 July 2015, pp. 3022–3033. AAAI Press (2015). http://ijcai.org/Abstract/15/427
30. Halpern, J.Y.: Actual Causality. MIT Press (2016)
31. Halpern, J.Y., Pearl, J.: Causes and explanations: a structural-model approach: Part 1: Causes. In: Breese, J.S., Koller, D. (eds.) UAI 2001: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, 2-5 August 2001, pp. 194–202. Morgan Kaufmann (2001)
32. Horak, T., et al.: Visual analysis of hyperproperties for understanding model checking results. IEEE Trans. Vis. Comput. Graph. **28**(1) (2022) https://doi.org/10.1109/TVCG.2021.3114866
33. Hume, D.: An Enquiry Concerning Human Understanding. London (1748)
34. Ibrahim, A., Pretschner, A.: From checking to inference: actual causality computations as optimization problems. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 343–359. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_19
35. Ibrahim, A., Rehwald, S., Pretschner, A.: Efficiently checking actual causality with sat solving. Eng. Secure Dependable Softw. Syst. **53**, 241–255 (2019)
36. Kupriyanov, A., Finkbeiner, B.: Causality-based verification of multi-threaded programs. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 257–272. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_19
37. Kupriyanov, A., Finkbeiner, B.: Causal termination of multi-threaded programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 814–830. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_54

38. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 248–267. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_16
39. Leitner-Fischer, F., Leue, S.: SpinCause: a tool for causality checking. In: International Symposium on Model Checking of Software, SPIN 2014. ACM (2014). https://doi.org/10.1145/2632362.2632371
40. Lewis, D.K.: Causation. J. Philos. **70**(17), 556–567 (1973). https://doi.org/10.2307/2025310
41. Lewis, D.K.: Counterfactuals. Blackwell, Cambridge, MA, USA (1973)
42. Mascle, C., Baier, C., Funke, F., Jantsch, S., Kiefer, S.: Responsibility and verification: importance value in temporal logics. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021, pp. 1–14. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470597
43. Parreaux, J., Piribauer, J., Baier, C.: Counterfactual causality for reachability and safety based on distance functions. In: Achilleos, A., Monica, D.D. (eds.) Proceedings of the Fourteenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2023, Udine, Italy, 18-20 September 2023. EPTCS, vol. 390, pp. 132–149 (2023). https://doi.org/10.4204/EPTCS.390.9
44. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
45. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, 11-13 January 1989, pp. 179–190. ACM Press (1989). https://doi.org/10.1145/75277.75293
46. Sallinger, S., Weissenbacher, G., Zuleger, F.: A formalization of heisenbugs and their causes. In: Ferreira, C., Willemse, T.A.C. (eds.) Software Engineering and Formal Methods - 21st International Conference, SEFM 2023, Eindhoven, The Netherlands, 6-10 November 2023, Proceedings. Lecture Notes in Computer Science, vol. 14323, pp. 282–300. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-47115-5_16
47. Schewe, S.: Büchi complementation made tight. In: Albers, S., Marion, J. (eds.) 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, 26-28 February 2009, Freiburg, Germany, Proceedings. LIPIcs, vol. 3, pp. 661–672. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2009). https://doi.org/10.4230/LIPICS.STACS.2009.1854
48. Schmitt, F., Hahn, C., Rabe, M.N., Finkbeiner, B.: Neural circuit synthesis from specification patterns. In: Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, 6-14 December 2021, virtual, pp. 15408–15420 (2021). https://proceedings.neurips.cc/paper/2021/hash/8230bea7d54bcdf99cdfe85cb07313d5-Abstract.html
49. Wang, C., Yang, Z., Ivančić, F., Gupta, A.: Whodunit? Causal analysis for counterexamples. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 82–95. Springer, Heidelberg (2006). https://doi.org/10.1007/11901914_9

Dynamic Programming for Symbolic Boolean Realizability and Synthesis

Yi Lin, Lucas Martinelli Tabajara, and Moshe Y. Vardi(✉)

Rice University, Houston, TX 77005, USA
yl182@rice.edu, vardi@cs.rice.edu

Abstract. Inspired by recent progress in dynamic programming approaches for weighted model counting, we investigate a dynamic-programming approach in the context of boolean realizability and synthesis, which takes a conjunctive-normal-form boolean formula over input and output variables, and aims at synthesizing witness functions for the output variables in terms of the inputs. We show how graded project-join trees, obtained via tree decomposition, can be used to compute a BDD representing the realizability set for the input formulas in a bottom-up order. We then show how the intermediate BDDs generated during realizability checking phase can be applied to synthesizing the witness functions in a top-down manner. An experimental evaluation of a solver – DPSynth – based on these ideas demonstrates that our approach for Boolean realizabilty and synthesis has superior time and space performance over a heuristics-based approach using same symbolic representations. We discuss the advantage on scalability of the new approach, and also investigate our findings on the performance of the DP framework.

Keywords: Boolean synthesis · Binary decision diagram · Dynamic programming · Bucket elimination

1 Introduction

The *Boolean-Synthesis Problem* [19] – a fundamental problem in computer-aided design – is the problem of taking in a declarative boolean relation between two sets of boolean variables – input and output – and generating boolean functions, called *witness functions*, that yield values to the output variables with respect to the input variables so as to satisfy the boolean relation. As a fundamental problem in computer-aided design, there are many applications of boolean synthesis in circuit design. For example, based on a circuit's desired behavior, we can use boolean synthesis to automatically construct the missing components of the circuit [3]. In addition to being used in circuit design, boolean synthesis has recently found applications also in temporal synthesis [30,31], where the goal is

A full version of this paper, with appendices included is available on arXiv at [21].
L. Martinelli Tabajara—Currently at Runtime Verification, Inc.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 112–134, 2024.
https://doi.org/10.1007/978-3-031-65633-0_6

to construct a sequential circuit that responds to environment inputs in a way that is guaranteed to satisfy given temporal-logic specification.

Many approaches have been investigated for boolean synthesis, such as knowledge compilation [1], QBF solving [24], and machine learning [17,18]. Here we build on previous work on boolean synthesis using Binary Decision Diagrams (BDDs) [28]. The main advantage of the decision-diagram approach is that it provides not just the synthesized witness functions, but also the realizability set of the specification, which is the set of inputs with respect to which the specification is realizable. In *modular circuit design* [19], where a system is composed of multiple modules that are independently constructed, it is imperative to confirm the realizability set of a module, as it has to match the output set of prior modules. Similarly, in the context of temporal synthesis [30,31], the winning set is constructed iteratively by taking the union of realizability sets, where for each realizability set we construct a witness function. The winning strategy is then constructed by stitching together all these witness functions.

Motivated by applications that require the computation of both realizability sets and witness functions, we describe here a decision-diagram approach that can also handle *partially realizable* specifications, where the realizability set is neither empty nor necessarily universal. (Indeed, in our benchmark suit, about 30% of the benchmarks are partially realizable.) Our tool, *DPSynth* computes the realizability sets and witness functions with respects to these sets. While several recent synthesis tools do provide witnesses for partially-realizable specifications, cf. [1,17,18], not all of them directly output the realizability set, requiring it, instead, to be computed from the witnesses and the original formula.

The boolean-synthesis problem starts with a boolean formula $\varphi(X, Y)$ over sets X, Y of *input* and *output* variables. The goal is to construct boolean formulas, called *witness functions* (sometimes called *Skolem functions*) [1,18] – for the *output variables* expressed in terms of the *input variables*. The BDD-based approach to this problem [15] constructs the BDD $B_\varphi(X, Y)$ for φ, and then quantifies existentially over the Y variables to obtain a BDD over the X variables that captures the *realizability set* – the set of assignments τ_X in 2^X for which an assignment τ_Y in 2^Y exists where $B_\varphi(\tau_X, \tau_y) = 1$. The witness functions can then be constructed by iterating over the intermediate steps of the realizability computation [15].

A challenge of this BDD-based approach is that it is often infeasible to construct the BDD B_φ. *Factored Boolean Synthesis* [28] assumes that the formula φ is given in conjunctive normal form (CNF), where the individual clauses are called *factors*. Rather than constructing the monolithic BDD B_φ, this approach constructs a BDD for each factor, and then applies conjunction in a lazy way and existential quantification in an eager way, using various heuristics to order conjoining and quantifying. As shown in [28], the factored approach, *Factored RSynth*, is more scalable than the monolithic approach, RSynth. Thus, we use here Factored RSynth as the baseline for comparison in this paper.

By *dynamic programming* we refer to the approach that simplifies a complicated problem by breaking it down into simpler sub-problems in a recursive manner [5]. Unlike search-based approaches to Boolean reasoning, cf. [22], which directly manipulate truth assignments, we use data structure based on decision diagrams [7] that represents sets of truth assignments. This approach is referred to as *symbolic*, going back to [11].

The *symbolic dynamic-programming* approach we propose here is inspired by progress in *weighted model counting*, which is the problem of counting the number of satisfying (weighted) assignments of boolean formulas. Dudek, Phan, and Vardi proposed in [14], an approach based on *Algebraic Decision Diagrams*, which are the quantitative variants of BDDs [4]. Dudek et al. pointed out that a monolithic approach is not likely to be scalable, and proposed a factored approach, ADDMC, analogous to the approach in [28], in which conjunction is done lazily and quantification eagerly. In follow-up work [12], they proposed a more systematic way to order the quantification and conjunction operations, based on dynamic programming over *project-join trees*; the resulting tool, DPMC, was shown to scale better than ADDMC. In further follow-on work, they proposed *graded project-join trees* for *projected model counting*, where the input formula has two sets of variables – quantified variables and counting variables [13]. Graded project-join tree offers a recursive decomposition of the Boolean-Synthesis Problem into smaller tasks of projections and join.

We show here how symbolic dynamic programming over graded project-join trees can be applied to boolean synthesis. In our approach, realizability checking is done using a BDD-based *bottom-up* execution, analogous to the handling of counting quantifiers in [13]. This enables us to compute the realizability set and check its nonemptiness. We then present a novel algorithm for synthesizing the witness functions using *top-down* execution on graded project-join trees. (In contrast, [13] both types of quantifiers are handled bottom-up). We demonstrate the advantage of our bottom-up-top-down approach by developing a tool, *DPSynth*. For a fair evaluation, we compare its performance to Factored RSynth, which can also handle partially realizable specifications.

The main contributions of this work are as follows. First, we show how to adapt the framework of projected counting to Boolean synthesis. In projected counting there are two types of existential quantifiers – additive and disjunctive [13], while Boolean synthesis combine universal and existential quantifiers. Second, projected counting requires only a bottom-up pass over the graded project-join trees, while here we introduce an additional, top-down pass over the tree to perform the major part of boolean synthesis, which is, witness construction.

The organization of the paper is as follows. After preliminaries in Sect. 2, we show in Sect. 3 how to use graded project-join trees for boolean realizability checking, and show that DPSynth generally scales better than Factored RSynth. Then, in Sect. 4, we show how to extend this approach from realizability checking to witness-function construction. Experimental evaluation shows that DPSynth generally scales better than Factored RSynth also for witness-function construction. We offer concluding remarks in Sect. 6.

2 Preliminary Definitions

2.1 Boolean Formula and Synthesis Concepts

A *boolean formula* $\varphi(X)$, over a set X of variables, represents a *boolean function* $f : 2^X \rightarrow \mathbb{B}$, which selects subsets of 2^X. A truth assignment τ *satisfies* φ iff $\varphi(\tau) = 1$. A boolean formula in *conjunctive normal form (CNF)* is a conjunction of *clauses*, where a clause is a disjunction of *literals* (a boolean variable or its negation). When φ is in CNF, we abuse notation and also use φ to denote its own set of clauses. Given a CNF formula $\varphi(X, Y)$ over *input* and *output* variable X and Y, the *realizability set* of φ, denoted $R_\varphi(X) \subseteq 2^X$, is the set of assignments $\sigma \in 2^X$ for which there exists an assignment $\tau \in 2^Y$ such that $\varphi(\sigma \cup \tau) = 1$. When φ is clear from context, we simply denote the realizability set by R.

Definition 1 (Realizability). *Let $\varphi(X, Y)$ be a CNF formula with X and Y as input and output variables. We say that φ is* fully realizable *if $R = 2^X$. We say that φ is* partially realizable *if $R \neq \varnothing$. Finally, we say that φ is* nullary realizable *if $R = \varnothing$.*

Given the condition that the formula is at least partially realizable, the boolean synthesis problem asks for a set of *witnesses* for the output variables generated on top of given input values, such that the formula is satisfied. This sub-problem of constructing witnesses, is usually referred to as synthesis.

The motivation behind synthesizing partially-realizable specifications is that there are cases where a specification is not fully realizable, but it is still useful to synthesize a function that works for all inputs in the realizability set. An example is the factorization benchmark family, as discussed in the introduction of [3], which takes an integer and aims to factor it into two integers that are both not equal to 1. If the integer is prime, then there is no valid factorization for this particular input, but it would still be valuable to have a solution that works for all composite numbers. This is why our attention to partially-realizable cases is a contribution of the paper, while related works tend to focus mostly on fully-realizable instances.

Definition 2 (Witnesses in Boolean Synthesis Problem). *Let $\varphi(X, Y)$ denote a fully or partially realizable boolean formula with input variables in $X = \{x_1, \ldots, x_m\}$ and output variables $Y = \{y_1, \ldots, y_n\}$, and let $R_\varphi(X) \neq \varnothing$ be its realizability set. A sequence $g_1(X), \ldots, g_n(X)$ of boolean functions is a sequence of* witness functions *for the Y variables in $\varphi(X, Y)$ if for every assignment $x \in R_\varphi(X)$, we have that $\varphi[X \mapsto x][y_1 \mapsto g_1(x)] \ldots [y_n \mapsto g_n(x)]$ holds.*

Definition 3 (Synthesis). *Given a partially or fully realizable CNF formula $\varphi(X, Y)$ with input and output variables X and Y, the* synthesis problem *asks to algorithmically construct a set of* witness functions *for the Y variables in terms of the X variables.*

2.2 Dynamic Programming Concepts - Project-Join Trees

A *binary decision diagrams (BDD)* [7] is directed acyclic graphs with two terminals labeled by 0 and 1. A BDD provides a canonical representation of boolean functions (and, by extension, boolean formulas). A (reduced, ordered) BDD is constructed from a binary decision tree, using a uniform variable order, of a boolean function by merging identical sub-trees and suppressing redundant nodes (nodes where both children are the same). Each path from the root of the BDD to the 1-terminal represents a satisfying assignment of the boolean function it represents. Based on the combination of BDDs and project-join trees, which is to be defined below, our dynamic-programming algorithm solves the boolean synthesis problem.

Definition 4 (Project-Join Tree of a CNF formula). *A* project-join tree *[12] for a CNF formula $\varphi(X)$ is defined as a tuple $\mathcal{T} = (T, r, \gamma, \pi)$, where*

1. *T is a tree with a set $\mathcal{V}(T)$ of vertices, a set $\mathcal{L}(T) \subseteq \mathcal{V}(T)$ of leaves, and a root $r \in \mathcal{V}(T)$*
2. *$\gamma : \mathcal{L}(T) \to \varphi$ is a bijection that maps the leaves of T to the clauses of φ*
3. *$\pi : \mathcal{V}(T) \setminus \mathcal{L}(T) \to 2^X$ is a function which labels internal nodes with variable sets, where the labels $\{\pi(n) \mid n \in \mathcal{V}(T) \setminus \mathcal{L}(T)\}$ form a partition of X, and*
4. *If a clause $c \in \varphi$ contains a variable x that belongs to the label $\pi(n)$ of an internal node $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$, then the associated leaf node $\gamma^{-1}(c)$ must descends from n.*

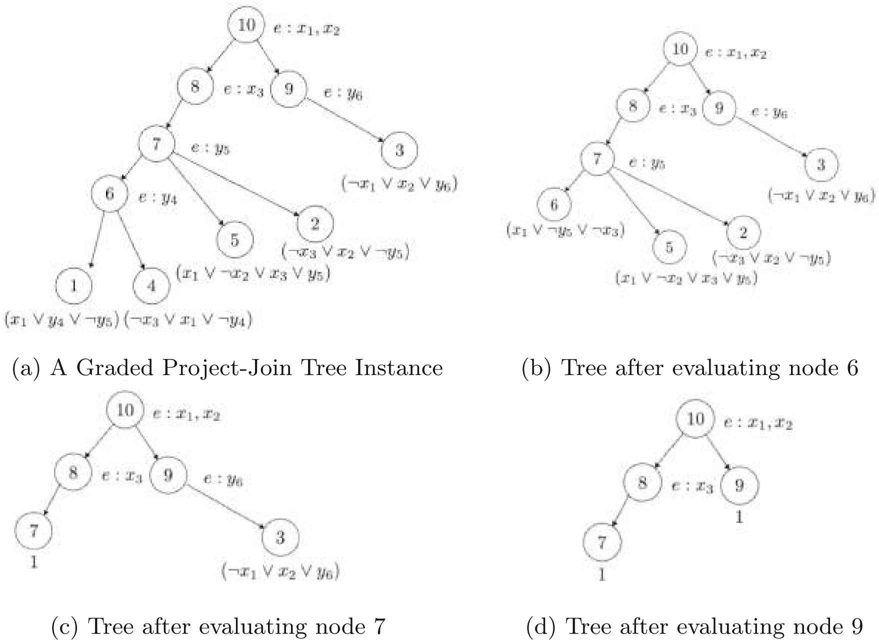
A *Graded Project-Join Tree* [13] is a generalization of project-join trees.

Definition 5 (Graded Project-Join Tree of a CNF formula [13]). *A project-join tree $\mathcal{T} = (T, r, \gamma, \pi)$ of a CNF formula $\varphi(X, Y)$ over variables $X \cup Y$, where $X \cap Y = \varnothing$, is* (X, Y)-graded *if there exist* grades $\mathcal{I}_X, \mathcal{I}_Y \subseteq \mathcal{V}(T)$ *that partition the internal nodes $\mathcal{V}(T) \setminus \mathcal{L}(T)$, such that:*

1. *For a node, its grade is always consistent with its labels. i.e., If $n_X \in \mathcal{I}_X$ then $\pi(n_X) \subseteq X$, and if $n_Y \in \mathcal{I}_Y$ then $\pi(n_Y) \subseteq Y$.*
2. *If $n_X \in \mathcal{I}_X$ and $n_Y \in \mathcal{I}_Y$, then n_X is not a descendant of n_Y in T.*

Intuitively, in a graded project-join tree, the nodes are partitioned according to a partition (X, Y) of the variables, with nodes in the X block always appearing higher than nodes in the Y block. As we shall see, this is useful for formulas with two different types of quantifiers.

Figure 1 shows an example Graded Project-Join Tree for a CNF formula, along with intermediate trees produced in the course of the execution of our synthesis algorithm. In the upcoming sections we will refer back to this example to illustrate the individual steps of the algorithm related to each intermediate tree.


10
e : x_1, x_2
8
e : x_3
9
e : y_6
7
e : y_5
3
$(\neg x_1 \vee x_2 \vee y_6)$
6
e : y_4
2
$(\neg x_3 \vee x_2 \vee \neg y_5)$
5
$(x_1 \vee \neg x_2 \vee x_3 \vee y_5)$
1
4
$(x_1 \vee y_4 \vee \neg y_5)$
$(\neg x_3 \vee x_1 \vee \neg y_4)$
$(x_1 \vee \neg y_5 \vee \neg x_3)$
1

(a) A Graded Project-Join Tree Instance

(b) Tree after evaluating node 6

(c) Tree after evaluating node 7

(d) Tree after evaluating node 9

Fig. 1. Original and intermediate trees generated by our algorithms for the CNF example $(x_1 \vee y_4 \vee \neg y_5) \wedge (\neg x_3 \vee x_2 \vee \neg y_5) \wedge (\neg x_1 \vee x_2 \vee y_6) \wedge (\neg x_3 \vee x_1 \vee \neg y_4) \wedge (x_1 \vee \neg x_2 \vee x_3 \vee y_5)$. The label for each internal node is denoted by e.

3 Realizability Checking[1]

Our overall approach has three phases: (1) *planning* – constructing graded project-join trees, (2) *realizability checking*, and (3) *witness-function synthesis*. The focus in this section is on realizability checking. We construct the realizability set $R_\varphi(X)$ for an input formula $\varphi(X, Y)$, and then use it to check for full and partial realizability, as described in Definition 1.

For the planning phase, we use the planner described in [13], which is based on *tree decomposition* [25]. Computing minimal tree decomposition is known to be an NP-hard problem [6], so planning may incur high computational overhead. Nevertheless, tree-decomposition tools are getting better and better. The planner uses an *anytime* tree-decomposition tool, cf. [16], which outputs tree decompositions of progressively lower width. Deciding when to quit planning and start executing is done heuristically. In contrast, Factored RSynth applies a *fixed* set of fast heuristics, which incurs relatively small overhead. We discuss the planning overhead further below.

[1] Proofs for All Lemmas and Theorems Can Be Found in the Appendix A.

3.1 Theoretical Basis and Valuations in Trees

The realizability set $R_\varphi(X)$ can be interpreted as a constraint over the X variables stating the condition that there exists an assignment to the Y variables that satisfies φ. In other words, $R_\varphi(X) \equiv (\exists y_1) \ldots (\exists y_n)\varphi$, for $Y = \{y_1, \ldots, y_n\}$.

Therefore, we can construct R from $\varphi = \bigwedge_j \varphi_j$ by existentially quantifying all Y variables. As observed in [28], however, a clause that does not contain y_i can be moved outside the existential quantifier $(\exists y_i)$. In other words, $(\exists y_i) \bigwedge_j \varphi_j \equiv \bigwedge_{y_i \notin AP(\varphi_j)} \varphi_j \wedge (\exists y_i) \bigwedge_{y_i \in AP(\varphi_j)} \varphi_j$, where AP stands for atomic propositions in a formula. This allows us to perform *early quantification* to compute R more efficiently, since applying quantification on the smaller formula $\bigwedge_{y_i \in AP(\varphi_j)} (\varphi_j)$ is likely less expensive computationally than doing so on the full formula $\varphi = \bigwedge_j (\varphi_j)$.

Inspired by a similar observation in [13], we use the insight that a graded project-join tree can be employed to guide early quantification. Consider the following definition, which allows us to interpret a node n in a project-join tree as a boolean formula:

Definition 6 (BDD-Valuations of Nodes in Project Join Tree). *Let n be a node in a graded project-join tree $\mathcal{T} = (T, r, \gamma, \pi)$ with a partition of internal nodes into two grades $\mathcal{I}_X$ and $\mathcal{I}_Y$, as defined in Sect. 2. Let the set of children nodes of n be denoted by $C(n)$. Let $[\![\alpha]\!]$ denote the BDD encoding a boolean expression α, and $\exists_Z f$ denote the existential projection on f with respect to variables in Z. We now define a pair of mutually related valuation concepts for nodes in a project-join tree, interpreting their BDD representations.*

The post-valuation *of n is defined as*

$$\mathtt{BV}_{\mathtt{post}}(\mathcal{T}, n) = \begin{cases} [\![\gamma(n)]\!], & \text{if } n \text{ is a leaf node} \\ \exists_{\pi(n)} \mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, n), & \text{if } n \text{ is an internal node.} \end{cases}$$

The pre-valuation *of n is defined as*

$$\mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, n) = \begin{cases} [\![\gamma(n)]\!], & \text{if } n \text{ is a leaf node} \\ \bigwedge_{n' \in C(n)} \mathtt{BV}_{\mathtt{post}}(\mathcal{T}, n'), & \text{if } n \text{ is an internal node.} \end{cases}$$

Intuitively, we evaluate an internal node n by first taking the conjunction of post-valuations of its children and then existentially quantifying the variables $\pi(n)$ in its label. The former step generates a pre-valuation, while the latter produces a post-valuation. We can safely perform quantification of the variables in the label of n after conjoining, because every internal node n must satisfy the property that all clauses containing variables in $\pi(n)$ are descendants of n, by the definition of a project-join tree.

Furthermore, recall that, by the definition of a (X, Y)-graded project-join tree, all nodes in $\mathcal{I}_Y$ occur below all nodes in $\mathcal{I}_X$. This allows us to turn Definition 6 into a procedure for efficiently computing realizability of the CNF formula φ using a graded project-join tree as a guide:

1. First, apply Definition 6 to the $\mathcal{I}_Y$ nodes of the tree, producing a project-join tree for $R_\varphi(X) \equiv (\exists y_1) \dots (\exists y_n)\varphi$.
2. Inspect this tree for full realizability (see below).
3. Apply Definition 6 again in order to check partial realizability (see below).

For the CNF with the graded project-join tree in Fig. 1a, the pre- and post-valuations for nodes $1, 4, 5, 2, 3$ are equivalent to the BDDs representing their original clauses. $\texttt{BV}_{\texttt{pre}}(\mathcal{T}, 6) = [\![(x_1 \lor y_4 \lor \neg y_5) \land (\neg x_3 \lor x_1 \lor \neg y_4)]\!], \texttt{BV}_{\texttt{post}}(\mathcal{T}, 6) = (\exists y_4)\texttt{BV}_{\texttt{pre}}(\mathcal{T}, 6) = (x_1 \lor \neg y_5 \lor \neg x_3), \texttt{BV}_{\texttt{pre}}(\mathcal{T}, 7) = (x_1 \lor \neg x_2 \lor x_3 \lor y_5) \land (\neg x_3 \lor x_2 \lor \neg y_5) \land (x_1 \lor \neg y_5 \lor \neg x_3), \texttt{BV}_{\texttt{post}}(\mathcal{T}, 7) = (\exists y_5)\texttt{BV}_{\texttt{pre}}(\mathcal{T}, 7) = [\![1]\!], \texttt{BV}_{\texttt{pre}}(\mathcal{T}, 9) = (\neg x_1 \lor x_2 \lor y_6), and\texttt{BV}_{\texttt{post}}(\mathcal{T}, 9) = (\exists y_6)\texttt{BV}_{\texttt{pre}}(\mathcal{T}, 9) = [\![1]\!]$.

Notations. We denote the set of children nodes and the set of descendants of n by $C(n)$ and $D(n)$. Let $[\![\cdot]\!]$ denote the BDD encoding of the enclosed expression, and use $\texttt{projn}(B, Z)$ to denote a series of existential projections on BDD B with respect to variables in set Z.

Algorithm 1: $GenericValuation(\mathcal{T}, n)$

```
Input: : an ((X, Y)-graded) project-join tree $\mathcal{T} = (T, r, \gamma, \pi)$ of $\varphi$, and a
         particular node $n \in \mathcal{V}(\mathcal{T})$
Output: early determination of nullary realizability, otherwise outputs
         $\texttt{BV}_{\texttt{post}}(\mathcal{T}, n)$ and $\texttt{BV}_{\texttt{pre}}(\mathcal{T}, n)$
if $n \in \mathcal{L}(T)$
    $\alpha \leftarrow [\![\gamma(n)]\!]$, pre-BV$(\mathcal{T}, n) \leftarrow \alpha$, post-BV$(\mathcal{T}, n) \leftarrow \alpha$// if $n$ is a leaf
else
    post-BV$(\mathcal{T}, n) \leftarrow [\![1]\!]$, pre-BV$(\mathcal{T}, n) \leftarrow [\![1]\!]$
    for $n' \in C(n)$ do
        if post-BV$(\mathcal{T}, n') == [\![0]\!]$
            pre-BV$(\mathcal{T}, n) \leftarrow [\![0]\!]$, post-BV$(\mathcal{T}, n) \leftarrow [\![0]\!]$
            return nullary realizable, no further synthesis needed
        end if
        else
            pre-BV$(\mathcal{T}, n) \leftarrow$ pre-BV$(\mathcal{T}, n) \land$ post-BV$(\mathcal{T}, n')$
        end if
        if pre-BV$(\mathcal{T}, n) == [\![0]\!]$
            post-BV$(\mathcal{T}, n) \leftarrow [\![0]\!]$
            return nullary realizable, no further synthesis needed
        end if
    end for
    post-BV$(\mathcal{T}, n) \leftarrow$ Projn(pre-BV$(\mathcal{T}, n), \pi(n)$)
end if
return pre-BV$(\mathcal{T}, n)$, post-BV$(\mathcal{T}, n)$
```

Algorithm 1 presents a procedure to compute the pre and post-valuations of Definition 6. In this algorithm pre-valuations are intermediate BDDs used to compute the post-valuations, but their values are also used in Sect. 4 for witness-function synthesis. Post-valuations, meanwhile, are used in Sect. 3.2 to determine if a given instance has a fully, partially, or unrealizable domain.

Note that children nodes are always visited before parent nodes, per Algorithm 2. This guarantees that line 6 in `GenericValuation` is always viable. We now assert the correctness of the Algorithm 1 by the following theorem.

Theorem 1. *If a graded project-join tree $\mathcal{T}$ and a particular node n are given, then (i) $\mathtt{BV}_{\mathtt{post}}(\mathcal{T}, n)$ and (ii) $\mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, n)$ returned by Algorithm 1 are as defined in Definition 6.*

Theorem 2. *Given a graded project-join tree $\mathcal{T}$ of a CNF formula φ, let $X_{\mathtt{leaves}}(\mathcal{T})$ denote the set of highest level nodes $n \in \mathcal{T}$ such that $\pi(n) \subseteq Y_\varphi$. Then the realizability set $R_\varphi(X)$ can be represented by the conjunction $\bigwedge_{n \in X_{\mathtt{leaves}}(\mathcal{T})} \mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, n)$ of BDDs returned by Algorithm 1.*

The pair of post and pre valuations defined in this section offer support for both realizability checking and synthesis, respectively. Section 3.2 applies $\mathtt{BV}_{\mathtt{post}}$ for realizability checking, and we show in Sect. 4 how $\mathtt{BV}_{\mathtt{pre}}$ is used in witnesses construction.

3.2 Determining Nullary, Partial and Full Realizability

Using the post-valuations of nodes computed in Algorithm 1 as the result of projecting variables on the conjunction of children nodes, we can construct the realizability set and determine if it is full, partial or empty. In practice, we represent the realizability set as a conjunction of BDDs, where an input is in the set if it satisfies all BDDs in the conjunction.

Notation Let $\mathtt{XLeaves}(\mathcal{T})$ denote the set of Y internal nodes whose parents are not in $\mathcal{I}_Y$. This set is easily obtainable by means such as graph-search algorithms.

We start by applying the pair of valuations computed in Algorithm 1 to the first layer of $\mathcal{I}_Y$ nodes in the graded tree; that is, all internal nodes in $\mathcal{I}_Y$ whose parent is not in $\mathcal{I}_Y$. By replacing these nodes with leaves labeled by their post-valuation, we obtain a project-join tree $\mathcal{T}_X$ for the formula $(\exists y_1) \ldots (\exists y_n)\varphi$, which, as explained in Sect. 3.1, corresponds to the realizability set. This procedure is implemented in Algorithm 2.

In the case of the example formula in Fig. 1, once we obtain the pre- and post-valuations from Algorithm 1, $B_{pureX} = [\![1]\!]$ and $b = 1$ from the post-valuations $\mathtt{BV}_{\mathtt{post}}(\mathcal{T}, 7)$ (Fig. 1c) and $\mathtt{BV}_{\mathtt{post}}(\mathcal{T}, 9)$ (Fig. 1d) on `XLeaves`. Hence we get full realizability.

Note that the formula φ is fully realizable if and only if the conjunction of the leaves in $\mathcal{T}_X$ is 1, which holds true if and only if all leaves are 1. Therefore, at the end of Algorithm 2 we are also able to answer whether φ is fully realizable. Note also that if any of the leaves is 0, then their conjunction is 0, meaning that φ is nullary realizable. Therefore, in some cases it might be possible to detect nullary realizability in this step. If the formula is not fully realizable and nullary realizability is not detected, then $\mathcal{T}_X$ is passed to the next step to test partial realizability.

Algorithm 2: *LowValuation*$(\mathcal{T})$

Explanation: This algorithm computes pre and post-valuations of nodes in the leaves and in the Y partition, simultaneously in the bottom-up manner checks if the realizability set is tautology (fully realizable) or an *obvious* negation (not realizable). If neither applies, it passes the new tree to Algorithm 3.

Input: $\mathcal{T} = (T, r, \gamma, \pi)$: an (X, Y)-graded project-join tree with internal nodes partitioned into $\mathcal{I}_X$, $\mathcal{I}_Y$.

Output: Returns *full* or *nullary realizability* if can determine by the end of this algorithm. Otherwise, a project-join tree $\mathcal{T}_X$ of $(\exists_{y \in Y} y)\varphi$ is passed to Algorithm 3 for further determining between *partial* and *nullary realizability*.

```
$B_{pureX}$ ←conjunction of clauses without $y$ variables
if $B_{pureX} == [\![0]\!]$ return nullary realizable
$\mathcal{T}_X \leftarrow \mathcal{T}$, $b \leftarrow 1$
if $\mathcal{I}_Y$ is empty return fully realizable
for $n \in \mathcal{I}_Y \cup \mathcal{L}(T)$ do
    GenericValuation($\mathcal{T}, n$) // in bottom-up order from leaves to root
    if $\mathtt{BV}_{\mathtt{pre}}(\mathcal{T}_X, n) == [\![0]\!]$ return nullary realizable
end for
for $n \in$ XLeaves($\mathcal{T}$) do
    if $\mathtt{BV}_{\mathtt{post}}(\mathcal{T}_X, n) == [\![0]\!]$ return nullary realizable
    else if $\mathtt{BV}_{\mathtt{post}}(\mathcal{T}_X, n) \neq [\![1]\!]$
        $b \leftarrow 0$
    end if
    $\mathcal{V}(T_X) \leftarrow \mathcal{V}(T_X) \setminus D(n)$// remove the descendants of $n$ from $\mathcal{T}_X$
    $\mathcal{L}(T_X) \leftarrow \mathcal{L}(T_X) \cup \{n\}$// add $n$ to leaves of $\mathcal{T}_X$
end for
if $b == 1$ and $B_{pureX} == [\![1]\!]$ return fully realizable
return HighValuation($\mathcal{T}_X$)
```

Theorem 3. *Given a graded project-join tree $\mathcal{T}$ for a CNF formula φ, Algorithm 2 returns full realizability if and only if the formula $(\forall X)(\exists Y)\varphi(X, Y)$ is true. And the algorithm returns nullary realizability, if and only if the formula $(\exists X)(\exists Y)\varphi(X, Y)$ is false.*

When the formula is not fully realizable, we proceed to pass the tree returned by Algorithm 2 to Algorithm 3, which then distinguishes partial realizability from nullary realizability. It does this by simply using Algorithm 1 to compute the post-valuation of the root of $\mathcal{T}_X$.

In the case of the example in Fig. 1, for this formula, we do not need to check Algorithm 3 because by Algorithm 2 full realizability is returned.

```
Algorithm 3: $HighValuation(\mathcal{T}_X)$
Input: $\mathcal{T}_X = (T_X, r_X, \gamma_X, \pi_X)$ is the generated tree from Algorithm 2,
       which is the project-join tree generated by projecting out all $Y$
       variables from the $(X, Y)$-graded tree for CNF $\varphi$.
Output: whether or not $\varphi$ is partially realizable
for $n \in \mathcal{I}_X$ // for all nodes in $I_X$ partition
do
    GenericValuation($\mathcal{T}, n$) // in bottom-up order leaves to the root
end for
if $\texttt{BV}_{\texttt{post}}(\mathcal{T}_X, r_X) == [\![0]\!]$ return nullary realizable
else return partially realizable
```

Since this corresponds to taking the conjunction of the leaves (which are themselves the post-valuation of the first layer of Y nodes) and existentially quantifying the X variables, the result is 1 if φ is partially realizable, and 0 if not. Built on top of Theorem 2, we formulate the correctness of the algorithm:

Theorem 4. *Given a CNF formula φ and its graded project-join tree $\mathcal{T}$, if we generate $\mathcal{T}_X$ by Algorithm 2, then Algorithm 3 returns the correct status of realizability of φ.*

4 Synthesis of Witness Functions

We now move to the third phase of our boolean-synthesis approach, where we construct boolean expressions for the output variables, which are the witness functions. More precisely, when the realizability set $R_\varphi(X)$, as defined in Sect. 3, is nonempty, we proceed to witness construction. We now formally define the concept of witnesses, in the context where the boolean synthesis problem is given as a CNF specification and reduced ordered BDDs are used for boolean-function representations.

The following lemma describes the *self-substitution method* for witness construction.

Lemma 1. [15] *Let X be a set of input variables and y a single output variable in a Boolean CNF formula $\varphi(X, y)$ with $R_\varphi(X) \neq \varnothing$. Then $g(X) = \varphi(X, y)[y \mapsto 1]$ is a witness for y in $\varphi(X, y)$.*

We now show how to extend this method to multiple output variables, building towards an approach using graded project-join trees.

4.1 Monolithic Approach

The synthesis procedure here builds on the condition that partial realizability is known, provided by the algorithms in Section. 3. While solvers constructed by related works, as discussed in Sect. 1, apply only to fully realizable formulas, we show here that synthesis can also be performed to obtain witness functions in the case of partial realizability.

As a stepping stone towards graded-tree-based synthesis, we first explain how witness functions are constructed in the monolithic case. We review the basic framework in [15], where the realizability set $R_\varphi(X)$ is obtained through iterative quantification on y_n to y_1, while witnesses are obtained via iterative substitution on y_1 to y_n.

Denote the BDD encoding the original CNF formula $\varphi(X,Y)$ to be $B_\varphi(X,Y)$. Then, a series of intermediate BDDs can be defined on the way of obtaining the realizability sets $R_\varphi(X)$:

$$\begin{aligned}
&B_n(X, y_1, \ldots, y_n) \equiv B_\varphi,\\
&\ldots\\
&B_i(X, y_1, \ldots, y_i) \equiv (\exists y_{i+1}) B_{i+1},\\
&\ldots\\
&B_0(X) \equiv (\exists y_1) B_1
\end{aligned}$$

Finally, the realizability set $R_\varphi(X)$ is $B_0(X)$.

Note that existential quantification proceeds here *inside-out*, since larger-indexed output variables are quantified before smaller-indexed output variables. Witness construction, on the other hand, proceeds *outside-in*: witnesses are constructed in the reverse direction starting from the smallest-indexed output variable y_1. The witnesses for variables $y_1, \ldots, y_{i-1}$ are substituted into B_i, from which we then construct the next witness g_i. Using the witness given by Lemma 1, we have:

g_1 is computed from: $B'_1(X, y_1) = B_1(X, y_1)$, via $g_1 = B'_1[y_1 \mapsto 1]$;

...

g_i is computed from: $B'_i(X, y_i) = B_i(X, y_1, \ldots, y_i)[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}]$, via $g_i = B'_i(X, y_i)[y_i \mapsto 1]$.

The following lemma is based on [15].

Lemma 2. *If $R_0(X) \neq \varnothing$, then the g_i's above are witness functions for Y in $\varphi(X,Y)$.*

As the witnesses above are computed using the self-substitution method from Lemma 1, each formula can have potentially many different witnesses. The correctness of the procedure does not depend on which witness is used. RSynth [15] applies the `SolveEqn` function of the CUDD package to compute a block of witnesses at once by essentially the procedure above. The procedure produces several witnesses for each variable, from which we chose one witnesses (by setting a parameter to 1).

The above monolithic synthesis framework was generalized in [28] to the case where the formula φ is given as a conjunction of *factors*. A factor can be a formula, for example, a single clause or a conjunction of clauses (called a "cluster" in that work). In that case, the principle of early quantification mentioned in Sect. 3.1 can be applied. See Appendix B for details. As shown in [28], the tool, *Factored RSynth*, generally outperforms RSynth.

In the following section, we describe a factored approached to witness construction using graded project-join trees.

4.2 Synthesis Using Graded Project-Join Trees

As we saw above, in the monolithic setting we compute the realizability set by quantifying the Y variables inside out, and then computing the witness function for these variables outside in. In the graded project-tree framework, we saw that the realizability set is computing by quantifying the Y variables bottom-up.

Our framework works for both partial and fully realizable cases. We first compute realizability sets going *bottom up* the graded project-join tree for φ as in Sect. 3. We now show that the witness functions for the Y variables can then be constructed by iterated substitution *top-down*. In other words, we first compute the witnesses for the variables in the labels of internal nodes at higher levels, and then propagate those down toward the leaves. We compute witnesses from the pre-valuation $\texttt{BV}_{\texttt{pre}}$ of a node in the tree, computed as in Algorithm 1. Note that, in contrast to the bottom-up realizability and top-down synthesis described here, in projected model counting, where graded project-join trees were introduced [13], the trees are processed fully in a bottom-up fashion.

The following lemma is crucial to our approach.

Lemma 3. *Let m and n be two different internal nodes of a graded project-join tree $\mathcal{T}$ such that n is not a descendant of m. Then no variable in $\pi(m)$ appears in $\texttt{BV}_{\texttt{pre}}(\mathcal{T}, n)$.*

We can derive from Lemma 3 the essential relation among witness functions for different variables: since the witness for a variable $y_i \in \pi(n)$ is computed from $\texttt{BV}_{\texttt{pre}}(\mathcal{T}, n)$, this witness can only depend on the witnesses of output variables $y_j \in \pi(m)$ such that n is a descendant of m.

Based on these insights, we present in Algorithm 4 our dynamic-programming synthesis algorithm for producing the witnesses $g_y(X)$ for each output variable y, represented by a BDD $W_y(X)$. The algorithm starts at the top-most layer of Y nodes, those whose parents are X nodes, represented by the set called `XLeaves` (line 1). For each node n in this set (line 3), we compute the set of witnesses $\{W_y \mid y \in \pi(n)\}$ from $\texttt{BV}_{\texttt{pre}}(\mathcal{T}, n)$ (line 9) using the monolithic procedure from Sect. 4.1 (represented by the CUDD function $\texttt{SolveEqn}(\pi(n), \texttt{BV}_{\texttt{pre}}(\mathcal{T}, n))$, mentioned in that section). Note that, since the tree is graded, these nodes do not descend from any Y nodes. Therefore, by Lemma 3, these witnesses depend only on the X variables.

As we compute the witnesses for a node, we add all of its (non-leaf) children to the set of nodes to visit (line 6), representing the next layer of the tree. After all nodes in the current set have been processed, we repeat the process with the new layer (line 3).

```
Algorithm 4: $DPSynth(\mathcal{T})$
    Notation: $C(n)$: set of the children nodes of $n$
    Notation: $D(n)$: set of the descendant nodes of $n$
    Notation: $W_y$: BDD representing the witness of variable $y$
    Notation: $[\![\alpha]\!]$: the BDD representation of boolean expression $\alpha$
    Notation: $\texttt{XLeaves}(\mathcal{T})$ as in Algorithm 2
    Input: $\mathcal{T} = (T, r, \gamma, \pi)$: the original $(X, Y)$-graded project-join tree of
           CNF $\varphi$
    Input: BDDs $\texttt{BV}_{\texttt{pre}}(m)$ for all $m \in \mathcal{I}_Y$
    Output: a series of BDDs $W_y, \forall y \in Y$, encoding the witness functions for
            output variables
 1  $\texttt{synthNodes} \leftarrow \texttt{XLeaves}(\mathcal{T})$
 2  while $\texttt{synthNodes} \neq \{\}$ do
        // from top-down order from the root to leaves
 3      for all $n \in \texttt{synthNodes}$ do
 4          for $n' \in C(n)$ do
 5              if $n'$ is not a leaf
 6                  add $n'$ to $\texttt{synthNodes}$
 7              end if
 8          end for
            // monolithic algorithm presented in Section 4.1
 9          $\{W_y \mid y \in \pi(n)\} \leftarrow \texttt{SolveEqn}(\pi(n), \texttt{BV}_{\texttt{pre}}(\mathcal{T}, n))$
10          for $n'' \in D(n)$ do
11              for all $y \in \pi(n)$ do
                    // substitute new synthesized $y$ by their witnesses $W_y$ in
                       the BDD representing the pre-valuation of descendants
                       $n''$
12                  $\texttt{BV}_{\texttt{pre}}(\mathcal{T}, n'') \leftarrow \texttt{BV}_{\texttt{pre}}(\mathcal{T}, n'')[y \mapsto W_y]$
13              end for
14          end for
15      end for
16  end while
17  return $W_y, \forall y \in Y$
```

This continues until the set of children is empty (line 2). Note that, since for each node n we apply the monolithic synthesis procedure only to the variables in $\pi(n)$, the witnesses can be dependent on the Y variables of its ancestors. Therefore, we finish the algorithm by iterating over the new synthesized witnesses W_y and substituting each into the pre-valuation of descendant nodes that are computed later (lines 10-14). Note that this overapproximates the set of dependencies on $y \in \pi(n)$ of the witnesses for $y' \in \pi(n'')$ for $n'' \in D(n)$, but since $W_{y'}[y \mapsto W_y] \equiv W_{y'}$ when y does not appear in $W_{y'}$, the result is still correct. At the end of this procedure, all W_y will be dependent only in the input variables X.

Continuing with the running example for the problem in Fig. 1, once full realizability is detected, we apply Algorithm 4 and construct the witnesses in top-down manner. First, we get the witness $g_6 = 1$ for y_6 by $\mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, 9) = [\![(\neg x_1 \vee x_2 \vee y_6)]\!]$. Then we construct the witness $g_5 = (x_1 \wedge x_2) \vee \neg x_3$ for y_5 by substituting $y_5 = 1$ in $\mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, 7) = [\![(x_1 \vee \neg x_2 \vee x_3 \vee y_5) \wedge (\neg x_3 \vee x_2 \vee \neg y_5) \wedge (x_1 \vee \neg y_5 \vee \neg x_3)]\!]$. After that, we go to node 6 and get $g_4 = x_1 \vee \neg x_3$ from $\mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, 6) = [\![(x_1 \vee y_4 \vee \neg y_5) \wedge (\neg x_3 \vee x_1 \vee \neg y_4)]\!]$. By the algorithm, we would need to substitute g_5 into $\mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, 6)$ before computing g_4, but since for this specific CNF g_4 is not actually dependent on g_5, this does not change the witness. The witnesses are correct by $\varphi[y_4 \mapsto g_4][y_5 \mapsto g_5][y_6 \mapsto g_6] = 1$.

The following theorem proves the correctness of witnesses constructed. First, it is easy to see by an inductive argument that a witness is synthesized for all output variables: all Y nodes that do not descend from other Y nodes are included in the set processed in the first iteration, and if a node is processed in one iteration, all of its children are included in the set for the next iteration. Since the tree is graded, we have processed all of the Y nodes, and since every output variable y is in the label of some node, a witness W_y is computed for every y. Then, the following theorem states the correctness of the witnesses constructed:

Theorem 5. *Algorithm 4 returns a set of BDDs encoding the witness functions for output variables that satisfy the given CNF φ, assuming that $\mathtt{SolveEqn}(\pi(n), \mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, n))$ returns correct witnesses for the variables in $\pi(n)$ in $\mathtt{BV}_{\mathtt{pre}}(\mathcal{T}, n)$.*

5 Experimental Evaluation

5.1 Realizability-Checking Phase

Methodology: To examine our dynamic-programming graded-project-join-tree-based approach for boolean realizability, we developed a software tool, *DPSynth*, which implements the theoretical framework described above. We choose to compare *DPSynth* to *Factored RSynth*, which as explained in the introduction is the closest existing tool, also being based on decision diagrams and outputting the realizability set along with the witnesses. Prior work [28] already demonstrated that RSynth practically never outperforms Factored RSynth, while Factored RSynth typically outperforms RSynth, so we compare in this paper to Factored RSynth. (See Sect. 5.4 for an additional comparison to a non-decision-diagram tool.) We measured the time and space performance for determining realizability on a set of mature benchmarks, described below. The experiments aim at answering the following research questions:

- Does our DP-based solution improve execution time for realizability checking? Does the overhead of *planning* time investment in DPSynth get paid off?
- How does the relative weight of planning overhead vary between small and large input instances?
- How does DPSynth improve the scalability of realizability checking?

(Tree width is one of the critical parameters impacting the performance of the graded project-jointree-based approach in running time. We discuss this issue further in separate a subsection.)

We selected 318 benchmarks that are neither too easy (taking less than 1ms) not too hard (such that the whole benchmark family is not solvable by either solver) from the data set of forall-exists Π_2^P CNFs from the QBFEVAL [23] datasets from 2016 to 2019, without any more selection criteria, as the editions of QBFEVAL in 2020 and 2022 do not include additional 2QBF instances or tracks other than those included by 2019. Families of benchmarks that our experiments run on include *reduction-finding query, mutexP, qshifter, ranking functions, sorting networks, tree and fix-point detection* families, and also two additional scalable families, consisting of parametric integer factorization and subtraction benchmarks from [2,3].

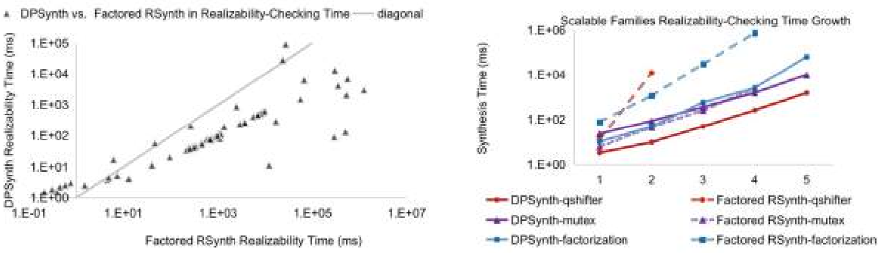
Among the full benchmark suite, at least 33% of the 285 instances where realizability can be checked by either DPSynth or factored RSynth are partially realizable. In the benchmarks for which both tools are able to synthesize a complete group of witnesses, 28% are partially realizabile. (No benchmark in the suite is identified as nullary realizable). We conclude that partial realizability is a significant issue in Boolean synthesis. We now present the experimental results analyzed according to the research questions above.

For each benchmark instance, we run the FlowCutter-based planner until we obtain the first tree decomposition, and we declare timeout if no tree decomposition is generated within ten minutes, in which case, the instance is marked unsolved. Otherwise, we take the first tree generated, and proceed to the execution phase of DPSynth, which includes BDD compilation, realizability checking, and synthesis of witnesses. The maximal time limit for each instance is set to be two hours for the execution phase. We measure planning time, execution time, and end-to-end time.

Our implementation is based on the CUDD Library [27] with BDD operations, and the FlowCutter tree-decomposition tool [20]. In our implementation, BDD variables are in MCS order [28]. This ordering is based on the primal graph of the input clause set, and was also used by Factored RSynth.

We ran the experiments on Rice University NOTS cluster, which assigns the jobs simultaneously to a mix of HPE SL230s, HPE XL170r, and Dell PowerEdge C6420 nodes, each of which has 16–40 cores with 32–192 GB RAM that runs at 2.1–2.60 GHz. Each solver-benchmark combination ran on a single core.

Experimental Results: Our goal in this paper was to compare the performance of the fast, CSP-based, formula-partitioning techniques of Factored RSynth [28], to the heavier-duty formula-partitioning techniques based on tree decomposition described above. For a fair comparison, the charts do not include data points for those instances that timed out for one of the solvers. (In total, DPSynth was able to solve 126 instances end-to-end, and RSynth 111.) Thus, our experiments compare *DPSynth* to Factored RSynth. Our conjecture is that such heavier-duty techniques pay off for larger formulas, but not necessarily for smaller formulas.



(a) (b)

Fig. 2. (a) Time Comparison. (b) Scalable Families Time Growth

Figure 2a shows overall running-time comparison between DPSynth and Factored RSynth for realizability checking. A clear pattern that emerges is there is a difference in relative performance between very small problems (solvable within 1 millisecond) and larger problems. DPSynth underperforms Factored RSynth on small instances, but outperforms on larger instances and the difference increases exponentially as input size grows. We conclude that planning overhead dominates on small input instances, but that effect fades off as instances get larger and the planning pays off. In memory usage comparison[2] – using peak node count as a measurement – DPSynth uses less memory than Factored RSynth. Here the graded project-join-tree approach is advantageous, and planning incurs no overhead.

To evaluate scalability, we take the scalable benchmark families mentioned previously, and compare the logarithmic-scale slope in their running time as sizes of benchmarks increase. As indicated in Figure. 2b, DPSynth scales exponentially better than Factored RSynth, as the planning overhead fades in significance as instances grow. We see a steeper slope on the exponential scale in DPSynth trends over factored RSynth. Some data points for larger benchmarks in the families (horizontal coordinates $3, 4, 5$ on chart) are missing because factored RSynth is not able to finish solving these instances in realizability-checking phase within the time limit.

5.2 Synthesis

The experiments on synthesis of witnesses answer the following research questions:

- How does DPSynth compare in execution time to factored RSynth?
- Does the influence of *planning* investment reduces as problem gets large?
- What do we see in growth of tree widths and synthesis execution time?

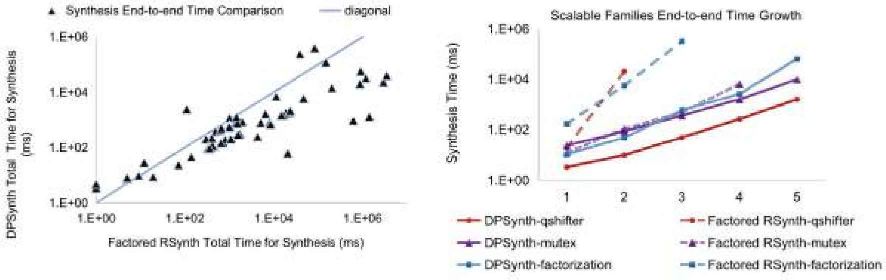
Using the same set of benchmarks and setting under the methodology as in Sect. 5.1, we applied our synthesis procedure to both fully realizable and

[2] The charts for space consumption for both phases are in the Appendix C.

partially realizable benchmarks. This broadens the scope of Boolean synthesis beyond that of fully realizable benchmarks, which is the scope of earlier work, as discussed above. Regarding end-to-end synthesis (planning, realizability checking, and synthesis, combined) DPSynth outperforms Factored RSynth in running time as illustrated in Fig. 3a. As with realizability checking, planning overhead dominates for small instances, as discussed in Sect. 3, but DPSynth solves large benchmarks faster and shows significant advantage as problem size increases, once the planning overhead fades in significance. In memory usage there is consistent relative performance of DPSynth vs. Factored RSynth.

We again selected three scalable benchmark families, scaled based on a numerical parameter. As shown in Fig. 3b, DPSynth scales exponentially as benchmark size increases. Similarly to the case in realizability checking, the missing data points on larger benchmarks in the scalable families presented by the dashed lines are caused by the timeouts by factored RSynth.

While DPSynth shows performance advantage over Factored RSynth with respect to our benchmark suite, one cannot conclude that DPSynth always dominates Factored RSynth. This is because DPSynth involves computationally nontrivial planning phase, and it is not possible to say definitively that the planning overhead always pays off.



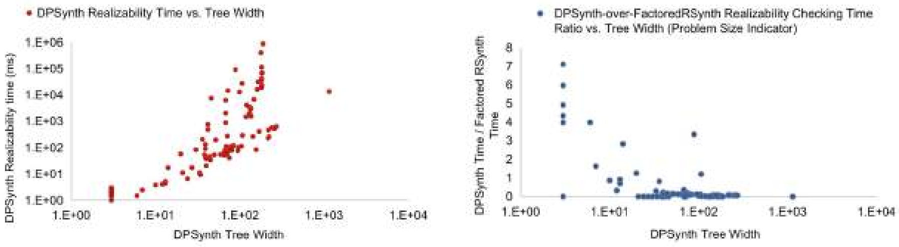
(a) (b)

Fig. 3. (a) Synthesis Time: DPSynth vs. Factored RSynth. (b) Scalable Families Comparison

We can conclude, however, that DPSynth is an important addition to the portfolio of algorithms for boolean synthesis.

5.3 Tree Widths and Realizability

Graded project-join trees enable the computation of the realizability set in a way that minimizes the set of support of intermediate ADDs (Algebraic Decision Diagrams), saving time and memory. But computing these trees is a heavy computational task. In this section, we study the impact of tree width on realizability checking.



(a) (b)

Fig. 4. (a) DPSynth Realizability Time as Widths Increases. (b) DPSynth/Factored RSynth Realizability Time Ratio as Widths Increases

Fig. 4a presents the range of realizability time run by DPSynth along increasing tree widths. As we see, increases in *tree widths* implies an increase in running time for realizability-checking. Figure 4b, depicts the ratio of realizability time, computed by that of DPSynth over Factored RSynth, with respect to tree width, for problem instances that can be solved by both solvers. As treewidth increases, the over-performance of DPSynth over Factored RSynth increases, as planning overhead decreases in significance for higher-treewidth instances.

Synthesis execution time has similar relation with tree widths.

5.4 Comparison with Non-BDD-Based Synthesis

To complement out evaluation, we include additional experiments to verify whether DPSynth is competitive with non-BDD-based synthesis solvers, as motivated in Sect. 1. We compare with Manthan [17], a leading tool that is not based on decision diagrams, and find that DPSynth performs favorably.

We present a general picture by the following table, which shows the overall strength of the dynamic-programming decision-diagram approach, by measuring the time and space usage of experiments on the dataset selected from QBFEVAL'16 to QBFEVAL'19. DPSynth and Manthan each is able to solve some benchmarks that the other does not solve.

In order to compare the running time, we compare against Manthan using the DPSynth *end-to-end synthesis* time, which is the sum of tree-decomposition time, compilation, realizability-checking time, and synthesis time.

As the overall picture of synthesis solving, DPSynth shows a better time performance on most instances. DPSynth and Manthan each has strength on some instances, but the number of benchmarks that only DPSynth solves is larger than those solved only by Manthan. On those that are solved by both, DPSynth takes less time in most of them. See Appendix D for more illustrative data on specific fully and partially-realizable benchmarks.

	Number of benchmarks
solved by Manthan	79
solved by DPSynth	102
solved by both Manthan and DPSynth	70
solved by Manthan but not by DPSynth	9
solved by DPSynth but not by Manthan	32

6 Concluding Remarks

To summarize the contribution in this work, we propose a novel symbolic dynamic programming approach for realizability checking and witness construction in boolean synthesis, based on graded project-join trees. The algorithm we propose here combine a bottom-up realizability-checking phase with a top-down synthesis phase. We demonstrated experimentally that our approach, implemented in the DPSynth tool, is powerful and more scalable than the approach based on CSP heuristics (Factored RSynth). Another crucial contribution of this work is the inclusion of partial realizability checking, which applies to 30% of the total number of benchmarks. As we explain in introduction, this consideration is motivated by the need in modular circuit design and temporal synthesis to locate the scope of realizable inputs in iterative constructions.

There are many directions for future work. Variable ordering is a critical issue in decision-diagram algorithms, and should be explored further in the context of our approach. In particular, dynamic variable ordering should be investigated [26]. Also, in our work here we used high-level API of the BDD package CUDD, but it is possible that performance gains can be obtained by using also low-level BDD-manipulating APIs. The question of how to provide certificates of unrealizability also needs to be explored, c.f., [8,9].

As mentioned earlier, quantifier elimination is a fundamental algorithmic component in temporal synthesis [31]. Tabajara and Vardi explored a factored approach for temporal synthesis [29]. It would be worthwhile to explore also the graded project-tree approach for boolean synthesis in the context of factored temporal synthesis. Finally, quantifier elimination is also a fundamental operation in symbolic model checking [11] and partitioning techniques have been explored in that context [10]. Therefore, exploring the applicability of our dynamic-programming approach in that setting is also a promising research direction.

Acknowledgements. This work was supported in part by NSF grants IIS-1527668, CCF-1704883, IIS-1830549, CNS-2016656, DoD MURI grant N00014-20-1-2787, an award from the Maryland Procurement Office, the Big-Data Private-Cloud Research Cyberinfrastructure MRI-award funded by NSF under grant CNS-1338099 and by Rice University's Center for Research Computing (CRC).

References

1. Akshay, S., Arora, J., Chakraborty, S., Krishna, S., Raghunathan, D., Shah, S.: Knowledge compilation for Boolean functional synthesis. In: 2019 Formal Methods in Computer Aided Design (FMCAD), pp. 161–169. IEEE (2019)
2. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What's hard about Boolean functional synthesis? In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part I. LNCS, vol. 10981, pp. 251–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_14
3. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: Boolean functional synthesis: hardness and practical algorithms. Formal Methods Syst. Des. **57**(1), 53–86 (2021)
4. Bahar, R.I., et al.: Algebric decision diagrams and their applications. Formal Meth. Syst. Des. **10**(2), 171–206 (1997)
5. Bellman, R.: Dynamic programming. Science **153**(3731), 34–37 (1966)
6. Bodlaender, H.L., et al.: Treewidth is np-complete on cubic graphs (and related results). arXiv preprint arXiv:2301.10031 (2023)
7. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE TC **100**(8), 677–691 (1986)
8. Bryant, R.E., Heule, M.J.: Dual proof generation for quantified Boolean formulas with a BDD-based solver. In: CADE, pp. 433–449 (2021)
9. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: TACAS 2021. LNCS, vol. 12651, pp. 76–93. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_5
10. Burch, J.R., Clarke, E.M., Long, D.E.: Symbolic model checking with partitioned transistion relations. In: Proceeding of the IFIP 10.5 International Conference on Very Large Scale Integration. IFIP Transactions, vol. A-1, pp. 49–58. North-Holland (1991)
11. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 1020 states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
12. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: DPMC: weighted model counting by dynamic programming on project-join trees. In: Simonis, H. (ed.) CP 2020. LNCS, vol. 12333, pp. 211–230. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_13
13. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: ProCount: weighted projected model counting with graded project-join trees. In: Li, C.-M., Manyà, F. (eds.) SAT 2021. LNCS, vol. 12831, pp. 152–170. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_11
14. Dudek, J.M., Phan, V.H., Vardi, M.Y.: ADDMC: weighted model counting with algebraic decision diagrams. In: AAAI, vol. 34, pp. 1468–1476 (2020)
15. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based Boolean functional synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 402–421. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_22
16. Gogate, V., Dechter, R.: A complete anytime algorithm for treewidth. arXiv preprint arXiv:1207.4109 (2012)
17. Golia, P., Roy, S., Meel, K.S.: Manthan: a data-driven approach for Boolean function synthesis. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 611–633. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_31
18. Golia, P., Slivovsky, F., Roy, S., Meel, K.S.: Engineering an efficient Boolean functional synthesis engine. In: 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1–9. IEEE (2021)

19. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms. Springer, Cham (2007)
20. Hamann, M., Strasser, B.: Graph bisection with pareto-optimization. In: Goodrich, M.T., Mitzenmacher, M. (eds.) Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016, pp. 90–102. SIAM (2016)
21. Lin, Y., Tabajara, L.M., Vardi, M.Y.: Dynamic programming for symbolic Boolean realizability and synthesis (2024). arXiv:2405.07975
22. 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)
23. Narizzano, M., Pulina, L., Tacchella, A.: The QBFEVAL web portal. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 494–497. Springer, Heidelberg (2006). https://doi.org/10.1007/11853886_45
24. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 375–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_23
25. Robertson, N., Seymour, P.D.: Graph minors. x. obstructions to tree-decomposition. J. Comb. Theory Ser. B **52**(2), 153–190 (1991)
26. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD), pp. 42–47. IEEE (1993)
27. Somenzi, F.: CUDD: CU Decision Diagram Package Release 3.0.0. University of Colorado at Boulder (2015)
28. Tabajara, L.M., Vardi, M.Y.: Factored Boolean functional synthesis. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October, 2017, pp. 124–131. IEEE (2017)
29. Tabajara, L.M., Vardi, M.Y.: Partitioning techniques in LTLF synthesis. In: International Joint Conference on Artificial Intelligence (2019)
30. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: A symbolic approach to safety LTL synthesis. In: HVC 2017. LNCS, vol. 10629, pp. 147–162. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70389-3_10
31. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLF synthesis. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, pp. 1362–1369. ijcai.org (2017)

Localized Attractor Computations for Infinite-State Games

Anne-Kathrin Schmuck[1] ⓡ Philippe Heim[2](✉) ⓡ Rayna Dimitrova[2] ⓡ
Satya Prakash Nayak[1]

[1] Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
{akschmuck,sanayak}@mpi-sws.org
[2] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{philippe.heim,dimitrova}@cispa.de

Abstract. Infinite-state games are a commonly used model for the synthesis of reactive systems with unbounded data domains. Symbolic methods for solving such games need to be able to construct intricate arguments to establish the existence of winning strategies. Often, large problem instances require prohibitively complex arguments. Therefore, techniques that identify smaller and simpler sub-problems and exploit the respective results for the given game-solving task are highly desirable.

In this paper, we propose the first such technique for infinite-state games. The main idea is to enhance symbolic game-solving with the results of localized attractor computations performed in sub-games. The crux of our approach lies in identifying useful sub-games by computing permissive winning strategy templates in finite abstractions of the infinite-state game. The experimental evaluation of our method demonstrates that it outperforms existing techniques and is applicable to infinite-state games beyond the state of the art.

1 Introduction

Games on graphs provide an effective way to formalize the automatic synthesis of *correct-by-design* software in cyber-physical systems. The prime examples are algorithms that synthesize *control software* to ensure high-level logical specifications in response to external environmental behavior. These systems typically operate over unbounded data domains. For instance, in smart-home applications [35], they need to regulate real-valued quantities like room temperature and lighting in response to natural conditions, day-time, or energy costs. Also, unbounded data domains are valuable for over-approximating large countable numbers of products in a smart manufacturing line [20]. The tight integration of many specialized machines makes their efficient control challenging. Similar control synthesis problems occur in robotic warehouse systems [18], underwater robots for oil-pipe inspections [25], and electric smart-grid regulation [29].

Authors are ordered randomly, denoted by ⓡ. The publicly verifiable record of the randomization is available at www.aeaweb.org.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 135–158, 2024.
https://doi.org/10.1007/978-3-031-65633-0_7

Algorithmically, the outlined synthesis problems can be formalized via *infinite-state games* that model the ongoing interaction between the system (with its to-be-designed control software) and its environment over their *infinite* data domains. Due to their practical relevance and their challenging complexity, there has been an increasing interest in *automated techniques* for solving infinite-state games to obtain correct-by-design control implementations. As the game-solving problem is in general undecidable in the presence of infinite data domains, this problem is substantially more challenging than its finite-state counterpart.

Within the literature[1], there are two prominent directions to attack this problem. One comprises *abstraction-based approaches*, where either the overall synthesis problem (e.g. [23,38]) or the specification (e.g. [8,14,27]) are abstracted, resulting in a finite-state game, to which classical techniques apply. The other one are *constraint-based techniques* [9,10,32, 33], that work directly on a symbolic representation of the infinite-state game. Due to the undecidability of the overall synthesis problem, both categories are inherently constrained. While abstraction-based approaches are limited by the abstraction domain they employ, constraint-based techniques typically diverge due to non-terminating fixpoint computations.

To address these limitations, a recent constraint-based technique called *attractor acceleration* [21] employs *ranking arguments* to improve the convergence of symbolic game-solving algorithms. While this technique has shown superior performance over the state-of-the art, the utilized ranking arguments become complex, and thus difficult to find, as the size of the games increases. This makes the approach from [21] infeasible in such cases, often resulting in divergence in larger and more complex games.

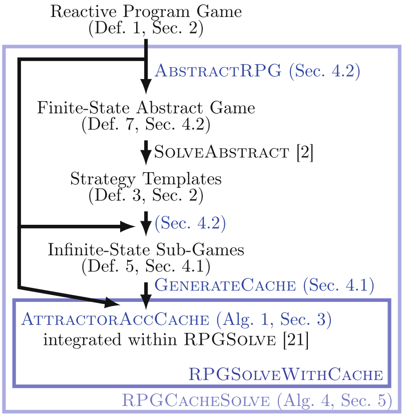


Fig. 1. Schematic paper outline; contributions highlighted in blue. (Color figure online)

In this paper, we propose an approach to overcoming the above limitation and thus extending the applicability of synthesis over infinite state games towards realistic applications. The key idea is to utilize efficient abstraction-based pre-computations that *localize* attractor computations to *small and useful sub-games*. In that way, acceleration can be applied locally to small sub-games, and the results utilized by the procedure for solving the global game. This often avoids the computationally inefficient attractor acceleration over the complete game. To *guide* the identification of useful sub-games, our approach computes *strategy templates* [2] – a concise representation of a possibly infinite number of winning strategies – in finite abstractions of the infinite-state game. Figure 1 shows an overview of our method which also serves as an outline of the paper.

[1] See Sect. 7 for a detailed discussion of related work.

Our experimental evaluation demonstrates the superior performance of our approach compared to the state of the art. Existing tools fail on almost all benchmarks, while our implementation terminates within minutes.

To build up more intuition, we illustrate the main idea of our approach with the following example, which will also serve as our running example.

Example 1. Figure 2 shows a reactive program game for a sample-collecting robot. The robot moves along tracks, and its position is determined by the integer program variable *pos*. The robot remains in location *base* until prompted by the environment to collect *inpReq* many samples. It cannot return to *base* until the required samples are collected, as enforced by the variable *done*. From the right position, it can enter the *mine*, where it must stay and collect samples from two sites, *a* and *b*. However, it has to choose the correct site in each iteration, as they might not have samples all the time (if both do not have samples, it can get one sample itself). Once enough samples are collected, the robot can return to *base*. The requirement on the robot's strategy is to be at *base* infinitely often.

Attractor acceleration [21] uses ranking arguments to establish that by iterating some strategy an unbounded number of times through some location, a player in the game can enforce reaching a set of target states. In this example, to reach $samp \geqslant req$ in location *mine* (the target) the robot can iteratively increase the value of *samp* by choosing the right updates (the iterated strategy). This works, since if *samp* is increased repeatedly, eventually $samp \geqslant req$ will hold (the ranking argument). Establishing the existence of the iterated strategy (i.e. the robot can increment *samp*) is a game-solving problem, since the behavior of the robot is influenced by the environment. This game-solving problem potentially considers the whole game, since the iterated strategy is not known a priori. In addition, identifying locations where acceleration can be applied and finding the right ranking arguments is challenging. This impacts the scalability and applicability of acceleration, making it infeasible for large games.

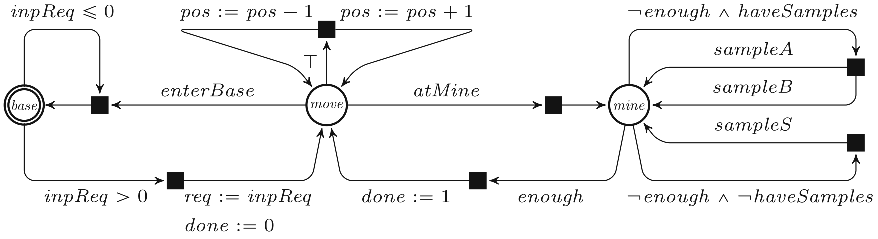


Fig. 2. A reactive program game for a sample-collecting robot with locations *base*, *move*, *mine*, integer-type program variables *pos*, *done*, *req*, *samp* and input variable *inpReq*. We use the following abbreviations: $enterBase \mathrel{\widehat{=}} (pos = 12 \wedge done = 1)$, $atMine \mathrel{\widehat{=}} (pos = 23)$, $haveSamples \mathrel{\widehat{=}} (a > 0 \vee b > 0)$, $enough \mathrel{\widehat{=}} samp \geqslant req$, $sampleA \mathrel{\widehat{=}} (samp := samp + a)$, $sampleB \mathrel{\widehat{=}} (samp := samp + b)$, and $sampleS \mathrel{\widehat{=}} (samp := samp + 1)$. In each round of the game, the environment chooses a value for the input *inpReq*. Based on guards over program variables and inputs, the game transitions to a black square. The system then chooses one of the possible updates to the program variables, thus determining the next location.

Consequently, our method aims to identify *small and useful sub-games* and *cache the results obtained by solving these sub-games.* In Example 1, a useful sub-game would be the game restricted to the *mine* location with the target state *samp* $\geqslant$ *req*. Applying the acceleration technique to this sub-game, provides the ranking argument described earlier. These cached results are then utilized to enhance the symbolic game-solving procedure for the entire game.

To identify these small and useful sub-games, we use *permissive strategy templates* [2] in finite-state abstracted games. They describe a potentially infinite set of winning strategies using local conditions on the transitions of the game. These local conditions (in the abstract game) provide guidance about local behavior in the solution of the infinite-state game without solving it. This local behavior (e.g. incrementing *samp* in *mine*) induces our sub-games.

2 Preliminaries

Sequences and First-Order Logic. For a set V, V^* and V^ω denote the sets of finite, respectively infinite, sequences of elements of V, and let $V^\infty = V^* \cup V^\omega$. For $\pi \in V^\infty$, we denote with $|\pi| \in \mathbb{N} \cup \{\infty\}$ the length of π, and define $\mathit{dom}(\pi) := \{0, \ldots, |\pi| - 1\}$. For $\pi = v_0 v_1 \ldots \in V^\infty$ and $i, j \in \mathit{dom}(\pi)$ with $i < j$, we define $\pi[i] := v_i$ and $\pi[i, j] := v_i \ldots v_j$. $\mathit{last}(\pi)$ is the last element of a finite sequence π.

Let $\mathcal{V}$ be the set of all values of arbitrary types, *Vars* be the set of all variables, $\mathcal{F}$ be the set of all functions, and Σ_F be the set of all function symbols. Let $\mathcal{T}_F$ be the set of all function terms defined by the grammar $\mathcal{T}_F \ni \tau_f ::= x \mid f(\tau_f^1, \ldots \tau_f^n)$ for $f \in \Sigma_F$ and $x \in \mathit{Vars}$. A function $\nu : \mathit{Vars} \to \mathcal{V}$ is called an *assignment.* The set of all assignments over variables $X \subseteq \mathit{Vars}$ is denoted as $\mathit{Assignments}(X)$. We denote the combination of two assignments ν', ν'' over disjoint sets of variables by $\nu' \uplus \nu''$. A function $\mathcal{I} : \Sigma_F \to \mathcal{F}$ is called an *interpretation.* The set of all interpretations is denoted as $\mathit{Interpretations}(\Sigma_F)$. The evaluation of function terms $\chi_{\nu,\mathcal{I}} : \mathcal{T}_F \to \mathcal{V}$ is defined by $\chi_{\nu,\mathcal{I}}(x) := \nu(x)$ for $x \in \mathit{Vars}$, $\chi_{\nu,\mathcal{I}}(f(\tau_0, \ldots \tau_n)) := \mathcal{I}(f)(\chi_{\nu,\mathcal{I}}(\tau_0), \ldots \chi_{\nu,\mathcal{I}}(\tau_n))$ for $f \in \Sigma_F$ and $\tau_0, \ldots \tau_n \in \mathcal{T}_F$. We denote the set of all first-order formulas as *FOL* and by *QF* the set of all quantifier-free formulas in *FOL*. Let φ be a formula and $X = \{x_1, \ldots, x_n\} \subseteq \mathit{Vars}$ be a set of variables. We write $\varphi(X)$ to denote that the free variables of φ are a subset of X. We also denote with $\mathit{FOL}(X)$ and $\mathit{QF}(X)$ the set of formulas (respectively quantifier-free formulas) whose free variables belong to X. For a quantifier $Q \in \{\exists, \forall\}$, we write $QX.\varphi$ as a shortcut for $Qx_1. \ldots Qx_n.\varphi$. We denote with $\models$: $\mathit{Assignments}(\mathit{Vars}) \times \mathit{Interpretations}(\Sigma_F) \times \mathit{FOL}$ the entailment of first-order logic formulas. A *first-order theory* $T \subseteq \mathit{Interpretations}(\Sigma_F)$ with $T \neq \emptyset$ restricts the possible interpretations of function and predicate symbols. Given a theory T, for a formula $\varphi(X)$ and assignment $\nu \in \mathit{Assignments}(X)$ we define that $\nu \models_T \varphi$ if and only if $\nu, \mathcal{I} \models \varphi$ for all $\mathcal{I} \in T$.

For exposition on first-order logic and first-order theories, see c.f. [7].

Two-Player Graph Games. A *game graph* is a tuple $G = (V, V_{Env}, V_{Sys}, \rho)$ where $V = V_{Env} \uplus V_{Sys}$ are the vertices, partitioned between the environment player (*player Env*) and the system player (*player Sys*), and $\rho \subseteq (V_{Env} \times V_{Sys}) \cup (V_{Sys} \times V_{Env})$ is the *transition relation.* A *play* in G is a sequence $\pi \in V^\infty$ where $(\pi[i], \pi[i+1]) \in \rho$ for all $i \in \mathit{dom}(\pi)$, and if π is finite then $\mathit{last}(\pi)$ is a dead-end.

For $p = \mathit{Sys}$ (or Env) we define $1-p := \mathit{Env}$ (respectively Sys). A *strategy for player p* is a partial function $\sigma : V^*V_p \to V$ where $\sigma(\pi \cdot v) = v'$ implies $(v, v') \in \rho$ and σ is defined for all $\pi \cdot v \in V^*V_p$ unless v is a dead-end. $\mathit{Strat}_p(G)$ denotes the set of all strategies for player p in G. A play π is *consistent with σ for player p* if $\pi[i+1] = \sigma(\pi[0,i])$ for every $i \in \mathit{dom}(\pi)$ where $\pi[i] \in V_p$. $\mathit{Plays}_G(v, \sigma)$ is the set of all plays in G starting in v and consistent with strategy σ.

An *objective* in G is a set $\Omega \subseteq V^\infty$. A *two-player turn-based game* is a pair (G, Ω), where G is a game graph and Ω is an objective for player *Sys*. A sequence $\pi \in V^\infty$ is *winning for player Sys* if and only if $\pi \in \Omega$, and is *winning for player Env* otherwise. The definitions of different types of common objectives can be found in [34]. The *winning region $W_p(G, \Omega)$ of player p in (G, Ω)* is the set of all vertices v from which player p has a strategy σ such that every play in $\mathit{Plays}_G(v, \sigma)$ is winning for player p. A strategy σ of player p is *winning* if for every $v \in W_p(G, \Omega)$, every play in $\mathit{Plays}_G(v, \sigma)$ is winning for player p.

Acceleration-Based Solving of Infinite-State Games. We represent infinite-state games using the same formalism as [21], called reactive program games. Intuitively, reactive program games describe symbolically, using *FOL* formulas and terms, the possible interactions between the system player and the environment player in two-player games over infinite data domains.

Definition 1 (Reactive Program Game Structure [21]). *A* reactive program game structure *is a tuple $\mathcal{G} = (T, \mathbb{I}, \mathbb{X}, L, \mathit{Inv}, \delta)$ with the following components. T is a first-order theory.* $\mathbb{I} \subseteq \mathit{Vars}$ *is a finite set of* input variables. $\mathbb{X} \subseteq \mathit{Vars}$ *is a finite set of* program variables *where* $\mathbb{I} \cap \mathbb{X} = \varnothing$. *$L$ is a finite set of* game locations. *$\mathit{Inv} : L \to \mathit{FOL}(\mathbb{X})$ maps each location to a* location invariant. $\delta \subseteq L \times \mathit{QF}(\mathbb{X} \cup \mathbb{I}) \times (\mathbb{X} \to \mathcal{T}_F) \times L$ *is a finite* symbolic transition relation *where*

(1) for every $l \in L$ the set of outgoing transition guards $\mathit{Guards}(l) := \{g \mid \exists u, l'.\ (l, g, u, l') \in \delta\}$ *is such that $\bigvee_{g \in \mathit{Guards}(l)} g \equiv_T \top$, and for all $g_1, g_2 \in \mathit{Guards}(l)$ with $g_1 \neq g_2$ it holds that $g_1 \wedge g_2 \equiv_T \bot$,*

(2) for all l, g, u, l_1, l_2, if $(l, g, u, l_1) \in \delta$ and $(l, g, u, l_2) \in \delta$, then $l_1 = l_2$, and

(3) for every $l \in L$ and $\mathbf{x} \in \mathit{Assignments}(\mathbb{X})$ such that $\mathbf{x} \models_T \mathit{Inv}(l)$, and $\mathbf{i} \in \mathit{Assignments}(\mathbb{I})$, there exist a transition $(l, g, u, l') \in \delta$ such that $\mathbf{x} \uplus \mathbf{i} \models_T g$ and $\mathbf{x}' \models_T \mathit{Inv}(l')$ where $\mathbf{x}'(x) = \chi_{\mathbf{x} \uplus \mathbf{i}, \mathcal{I}}(u(x))$ for all $x \in \mathbb{X}$ and $\mathcal{I} \in T$, and

(4) for every $(l, g, u, l') \in \delta$, $f \in \Sigma_F(u)$, $\mathcal{I}_1, \mathcal{I}_2 \in T$ it holds that $\mathcal{I}_1(f) = \mathcal{I}_2(f)$.

The requirements on δ imply for each $l \in L$ that: (1) the guards in $\mathit{Guards}(l)$ partition the set $\mathit{Assignments}(\mathbb{X} \cup \mathbb{I})$, (2) each pair of $g \in \mathit{Guards}(l)$ and update u can label at most one outgoing transition from l, (3) if there is an assignment satisfying the invariant at l, then for every input assignment there is a possible transition, and (4) the theory T determines the meaning of functions in updates uniquely. Given locations $l, l' \in L$, we define $\mathit{Labels}(l, l') := \{(g, u) \mid (l, g, u, l') \in \delta\}$ as the set of *labels on transitions from l to l'*. We define as RPGS the set of all reactive program game structures. The semantics of the reactive program game structure $\mathcal{G}$ is a (possibly infinite) game graph defined as follows.

Definition 2 (Semantics of Reactive Program Game Structures). *Let $\mathcal{G} = (T, \mathbb{I}, \mathbb{X}, L, \mathit{Inv}, \delta)$ be a reactive program game structure. The semantics of $\mathcal{G}$ is the game graph $[\![\mathcal{G}]\!] = (\mathcal{S}, \mathcal{S}_{\mathit{Env}}, \mathcal{S}_{\mathit{Sys}}, \rho)$ where $\mathcal{S} := \mathcal{S}_{\mathit{Env}} \uplus \mathcal{S}_{\mathit{Sys}}$ and*

- $\mathcal{S}_{\mathit{Env}} := \{(l, \mathbf{x}) \in L \times \mathit{Assignments}(\mathbb{X}) \mid \mathbf{x} \models_T \mathit{Inv}(l)\}$*;*
- $\mathcal{S}_{\mathit{Sys}} := \mathcal{S}_{\mathit{Env}} \times \mathit{Assignments}(\mathbb{I})$*;*
- $\rho \subseteq (\mathcal{S}_{\mathit{Env}} \times \mathcal{S}_{\mathit{Sys}}) \cup (\mathcal{S}_{\mathit{Sys}} \times \mathcal{S}_{\mathit{Env}})$ *is the smallest relation such that*
  - $(s, (s, \mathbf{i})) \in \rho$ *for every* $s \in \mathcal{S}_{\mathit{Env}}$ *and* $\mathbf{i} \in \mathit{Assignments}(\mathbb{I})$*,*
  - $(((l, \mathbf{x}), \mathbf{i}), (l', \mathbf{x}')) \in \rho$ *iff* $\mathbf{x}' \models_T \mathit{Inv}(l')$ *and there exists* $(g, u) \in \mathit{Labels}(l, l')$ *such that* $\mathbf{x} \uplus \mathbf{i} \models_T g$*,* $\mathbf{x}'(x) = \chi_{\mathbf{x} \uplus \mathbf{i}, \mathcal{I}}(u(x))$ *for every* $x \in \mathbb{X}$ *and* $\mathcal{I} \in T$*.*

Note that this semantics differs from the original one in [21] where the semantic game structure is not split into environment and system states. We do that in order to consistently use the notion of a game graph. Both semantics are equivalent. We refer to the vertices of $[\![G]\!]$ as *states*. We define the function $\mathit{loc} : \mathcal{S} \to L$ where $\mathit{loc}(s) := l$ for any $s = (l, \mathbf{x}) \in \mathcal{S}_{\mathit{Env}}$ and any $s = ((l, \mathbf{x}), \mathbf{i}) \in \mathcal{S}_{\mathit{Sys}}$. By abusing notation, we extend the function loc to sequences of states, defining $\mathit{loc} : \mathcal{S}^{\infty} \to L^{\infty}$ where $\mathit{loc}(\pi) = l_0 l_1 l_2 \ldots$ iff $\mathit{loc}(\pi[i]) = l_i$ for all $i \in \mathit{dom}(\pi)$. For simplicity of the notation, we write $W_p(\mathcal{G}, \Omega)$ instead of $W_p([\![\mathcal{G}]\!], \Omega)$. We represent and manipulate possibly infinite sets of states symbolically, using formulas in $\mathit{FOL}(\mathbb{X})$ to describe sets of assignments to the variables in $\mathbb{X}$. Our *symbolic domain* $\mathcal{D} := L \to \mathit{FOL}(\mathbb{X})$ is the set of functions mapping locations to formulas in $\mathit{FOL}(\mathbb{X})$. An element $d \in \mathcal{D}$ represents the states $[\![d]\!] := \{((l, \mathbf{x}) \in \mathcal{S} \mid \mathbf{x} \models_T d(l)\}$. With $\{l_1 \mapsto \varphi_1, \ldots, l_n \mapsto \varphi_n\}$ we denote $d \in \mathcal{D}$ s.t. $d(l_i) = \varphi_i$ and $d(l) = \bot$ for $l \notin \{l_1, \ldots, l_n\}$. For brevity, we sometimes refer to elements of $\mathcal{D}$ as sets of states.

Note that the elements of the symbolic domain $\mathcal{D}$ represent subsets of $\mathcal{S}_{\mathit{Env}}$, i.e., sets of environment states. Environment states are pairs of location and valuation of the program variables. The system states, on the other hand, correspond to intermediate configurations that additionally store the current input from the environment. This input is not stored further on (unless assigned to program variables). Thus, we restrict the symbolic domain to environment states.

Solving Reactive Program Games. We consider *objectives defined over the locations* of a reactive program game structure $\mathcal{G}$. That is, we require that if $\pi', \pi'' \in \mathcal{S}^{\infty}$ are such that $\mathit{loc}(\pi') = \mathit{loc}(\pi'')$, then $\pi' \in \Omega$ iff $\pi'' \in \Omega$. We consider the problem of solving reactive program games. Given $\mathcal{G}$ and an objective Ω for Player Sys defined over the locations of $\mathcal{G}$, we want to compute $W_{\mathit{Sys}}([\![\mathcal{G}]\!], \Omega)$.

Attractor Computation and Acceleration. A core building block of many algorithms for solving two-player games is the computation of *attractors*. Intuitively, an attractor is the set of states from which a given player p can enforce reaching a given set of target states no matter what the other player does. Formally, for a reactive program game structure $\mathcal{G}$, and $R \subseteq \mathcal{S}$ the *player-p attractor for R* is

$$Attr_{[\![\mathcal{G}]\!],p}(R) := \{s \in \mathcal{S} \mid \exists \sigma \in Strat_p([\![\mathcal{G}]\!]).\forall \pi \in Plays_{[\![\mathcal{G}]\!]}(s, \sigma).\exists n \in \mathbb{N}.\ \pi[n] \in R\}.$$

In this work, we are concerned with the symbolic computation of attractors in reactive program games. Attractors in reactive program games are computed using the so-called *enforceable predecessor operator* over the symbolic domain $\mathcal{D}$. For $d \in \mathcal{D}$, $CPre_{\mathcal{G},p}(d) \in \mathcal{D}$ represents the states from which player p can enforce reaching $[\![d]\!]$ in one step in $\mathcal{G}$ (i.e. one move by each player). More precisely,

$$\begin{aligned}[\![CPre_{\mathcal{G},Sys}(d)]\!] &= \{s \in \mathcal{S}_{Env} \mid \forall s'.\ ((s, s') \in \rho) \rightarrow \exists s''.\ (s', s'') \in \rho \wedge s'' \in [\![d]\!]\}, \text{ and}\\ [\![CPre_{\mathcal{G},Env}(d)]\!] &= \{s \in \mathcal{S}_{Env} \mid \exists s'.\ ((s, s') \in \rho) \wedge \forall s''.\ ((s', s'') \in \rho) \rightarrow s'' \in [\![d]\!]\}.\end{aligned}$$

The player-p attractor for $[\![d]\!]$ can be computed as a fixpoint of the enforceable predecessor operator:

$$Attr_{[\![\mathcal{G}]\!],p}([\![d]\!]) \cap \mathcal{S}_{Env} = [\![\mu X.\ d \vee CPre_{\mathcal{G},p}(X)]\!],$$

where μ denotes the least fixpoint. Note that since $\mathcal{S}$ is infinite, an iterative computation of the attractor is not guaranteed to terminate.

In Example 1, consider the computation of the player-*Sys* attractor for $[\![d]\!]$ where $d = \{base \mapsto \top, move \mapsto \top, mine \mapsto \bot\}$. Applying $CPre_{\mathcal{G},Sys}(d)$ will produce $\{base \mapsto \top, move \mapsto \top, mine \mapsto samp \geqslant req\}$ as in one step player-*Sys* can enforce reaching *move* if $samp \geqslant req$ in *mine*. Since in *mine* the system player can enforce to increment *samp* by at least one, a second iteration of $CPre_{\mathcal{G},Sys}(\cdot)$ gives $\{\dots, mine \mapsto samp \geqslant req - 1\}$, a third $\{\dots, mine \mapsto samp \geqslant req - 2\}$, and so on. Thus, a naive iterative fixpoint computation does no terminate here. To avoid this non-termination, [21] introduced *attractor acceleration*. It will compute that, as explained in Sect. 1, the fixpoint is indeed $\{\dots, mine \mapsto \top\}$.

Permissive Strategy Templates. The main objective of this work is to identify small and useful sub-games, for which the results can enhance the symbolic game-solving process. To achieve this, we use a technique called *permissive strategy templates* [2], designed for finite game graphs. These templates can represent (potentially infinite) sets of winning strategies through local edge conditions. This motivates our construction of sub-games based on templates in Sect. 4.2.

These strategy templates are structured using three local edge conditions: *safety*, *co-live*, and *live-group* templates. Formally, given a game (G, Ω) with $G = (V, V_{Env}, V_{Sys}, \rho)$ and $E_p = \rho \cap (V_p \times V_{p-1})$, a *strategy template for player p* is a tuple $(U, D, \mathcal{H})$ consisting of a set of *unsafe* edges $U \subseteq E_p$, a set of *co-live* edges $D \subseteq E_p$, and a set of live-groups $\mathcal{H} \subseteq 2^{E_p}$. A strategy template $(U, D, \mathcal{H})$ represents the set of plays $\Psi = \Psi_U \cap \Psi_D \cap \Psi_{\mathcal{H}} \subseteq \mathit{Plays}(G)$, where

$$\Psi_U := \{\pi \mid \forall i.\ (\pi[i], \pi[i+1]) \notin U\}, \quad \Psi_D := \{\pi \mid \exists k.\ \forall i > k.\ (\pi[i], \pi[i+1]) \notin D\},$$

$$\Psi_{\mathcal{H}} := \bigcap_{H \in \mathcal{H}} \{\pi \mid (\forall i.\ \exists j > i.\ \pi[j] \in \textsc{src}(H)) \to (\forall i.\ \exists j > i.\ (\pi[j], \pi[j+1]) \in H)\},$$

where $\textsc{src}(H)$ contains the sources $\{u \mid (u, v) \in H\}$ of the edges in H. A strategy σ for player p *satisfies* a strategy template Ψ if it is winning in the game (G, Ψ) for player p. Intuitively, σ satisfies a strategy template if every play π consistent with σ for player p is contained in Ψ, that is, (i) π never uses the unsafe edges in U (i.e., $\pi \in \Psi_U$), (ii) π stops using the co-live edges in D eventually (i.e., $\pi \in \Psi_D$), and (iii) for every live-group $H \in \mathcal{H}$, if ρ visits $\textsc{src}(H)$ infinitely often, then it also uses the edges in H infinitely often (i.e., $\pi \in \Psi_{\mathcal{H}}$). Strategy templates can be used as a concise representation of winning strategies as formalized next.

Definition 3 (Winning Strategy Template [2]). *A strategy template Ψ for player p is* winning *if every strategy satisfying Ψ is winning for p in (G, Ω).*

We note that the algorithms for computing winning strategy templates in safety, Büchi, co-Büchi, and parity games, presented in [2], exhibit the same worst-case computation time as standard methods for solving such (finite-state) games.

3 Attractor Computation with Caching

As outlined in Sect. 1, the core of our method consists of the pre-computation of attractor sets for local sub-games and the utilization of the results in the attractor computations performed when solving the complete reactive program game. We call the pre-computed results *attractor cache*. We use the cache during attractor computations to *directly add* to the computed attractor sets of states from which, *based on the pre-computed information*, the respective player can enforce reaching the current attractor subset. In that way, if the local attractor computation requires acceleration, we can avoid performing the acceleration during the attractor computation for the overall game. This section presents the formal definition of an attractor cache and shows how it is used.

Intuitively, an attractor cache is a finite set of tuples or *cache entries* of the form $(\mathcal{G}, p, \mathit{src}, \mathit{targ}, \mathbb{X}_{ind})$. $\mathcal{G}$ is a reactive program game structure and p the player the cache entry applies to. The sets of states $\mathit{src}, \mathit{targ} \in \mathcal{D}$ are related via enforceable reachability: player p can enforce reaching $[\![\mathit{targ}]\!]$ from $[\![\mathit{src}]\!]$ in $\mathcal{G}$. $\mathbb{X}_{ind}$ are the so-called *independent variables* – the enforcement relation must hold independently of and preserve the values of $\mathbb{X}_{ind}$. Independent variables are useful when a cache entry only concerns a part of the game structure where these variables are irrelevant. This allows the utilization of the cache entry under different conditions on those variables. We formalize this intuition in the next definition.

Definition 4 (Attractor Cache). *A finite set* $C \subseteq \mathsf{RPGS} \times \{Sys, Env\} \times \mathcal{D} \times \mathcal{D} \times 2^{\mathbb{X}}$ *is called an* attractor cache *if and only if for all* $(\mathcal{G}, p, src, targ, \mathbb{X}_{ind}) \in C$ *and all* $\varphi \in FOL(\mathbb{X}_{ind})$ *it holds that* $[\![src \wedge \lambda l.\ \varphi]\!] \subseteq Attr_{[\![\mathcal{G}]\!],p}([\![targ \wedge \lambda l.\ \varphi]\!])$.

We use the *lambda abstraction* $\lambda l.\ \varphi$ to denote the anonymous function that maps each location in L to the formula φ.

Example 2. Recall the game from Example 1. From every state with location *mine*, player *Sys* can enforce eventually reaching $samp \geqslant req$ by choosing at every step the update that increases variable *samp*. As this argument only concerns location *mine*, the program variables *done* and *pos* are independent. Since it is not updated, *req* is also independent (we prove this in the next section). Hence, $C_{ex} = \{(\mathcal{G}_{ex}, Sys, src, targ, \mathbb{X}_{ind})\}$ where $\mathcal{G}_{ex}$ is from Fig. 2, $src = \{mine \mapsto \top\}$, $targ = \{mine \mapsto samp \geqslant req\}$, and $\mathbb{X}_{ind} = \{done, pos, req\}$ is an attractor cache.

Algorithm 1: Attractor computation using an attractor cache.

```
function AttractorAccCache( $\mathcal{G}$, $p \in \{Sys, Env\}$, $d \in \mathcal{D}$, $C$: cache)
    $a^0 := \lambda l.\ \bot$; $a^1 := d$
    for $n = 1, 2, \ldots$ do
        if $a^n \equiv_T a^{n-1}$ then return $a^n$
        foreach $(\mathcal{G}', p', src, targ, \mathbb{X}_{ind}) \in C$ with $\mathcal{G}' = \mathcal{G}$ and $p' = p$ do
            $\varphi :=$ StrengthenTarget$(targ, \mathbb{X}_{ind}, a^n)$
            $a^n := a^n \vee (src \wedge (\lambda l.\ \varphi))$
        $a^n := a^n \vee$ Accelerate$(\mathcal{G}, p, l, a^n)$ /* Accelerate(...) is the result of
            applying attractor acceleration as in [21]                    */
        $a^{n+1} := a^n \vee CPre_{\mathcal{G},p}(a^n)$
```

Algorithm 1 shows how we use an attractor cache to enhance accelerated attractor computations. AttractorAccCache extends the procedure AttractorAcc for accelerated symbolic attractor computation presented in [21]. AttractorAccCache takes a cache as an additional argument and at each iteration of the attractor computation checks if some cache entry is applicable. For each such cache entry, if $[\![targ]\!]$ is a subset of $[\![a^n]\!]$, we can add *src* to a^n since we know that *targ* is enforceable from *src*. However, a^n may constrain the values of $\mathbb{X}_{ind}$ making this subset check fail unnecessarily. Therefore, StrengthenTarget computes a formula $\varphi \in FOL(\mathbb{X}_{ind})$ such that *targ* strengthened with φ is a subset of a^n. Intuitively, φ describes the values of the independent variables that remain unchanged in the cached attractor. Note that φ always exists as we could pick $\bot$, which we have to do if *targ* is truly *not* a subset of a^n.

The next lemma formalizes this intuition and the correctness of AttractorAccCache under the above condition on StrengthenTarget. Note that since the cache is used in the context of attractor computation, the objective Ω of the reactive program game is not relevant here.

Lemma 1 (Correctness of Cache Utilization). *Let $\mathcal{G}$ be a reactive program game structure, $p \in \{Sys, Env\}$, $d \in \mathcal{D}$ and C be an attractor cache. Furthermore, suppose that for every $targ \in \mathcal{D}$, $a \in \mathcal{D}$ and every $\mathbb{X}_{ind} \subseteq \mathbb{X}$ it holds that if* $\textsc{StrengthenTarget}(targ, \mathbb{X}_{ind}, a) = \varphi$*, then $\varphi \in FOL(\mathbb{X}_{ind})$ and $[\![targ \wedge \lambda l.\ \varphi]\!] \subseteq [\![a]\!]$. Then, if the procedure* $\textsc{AttractorAccCache}(\mathcal{G}, p, d, C)$ *terminates returning $attr \in \mathcal{D}$, then it holds that $[\![attr]\!] = Attr_{[\![\mathcal{G}]\!],p}([\![d]\!]) \cap \mathcal{S}_{Env}$.*

We realize $\textsc{StrengthenTarget}(targ, \mathbb{X}_{ind}, a)$ such that it returns the formula $\bigwedge_{l \in L} \big(\forall(\mathbb{X}\backslash\mathbb{X}_{ind}).\ targ(l) \rightarrow a(l)\big)$ which satisfies the condition in Lemma 1.

Example 3. Recall the game from Example 1 and the cache C_{ex} from Example 2. Suppose that we are computing the attractor for player Sys to $d = \{base \mapsto \top\}$, i.e. $\textsc{AttractorAccCache}(\mathcal{G}_{ex}, Sys, d, C_{ex})$ without acceleration, i.e., $\mathsf{Accelerate}$ returns $\perp$ in line 11 in Algorithm 1. Initially, $a^1 = \{base \mapsto \top\}$. After one iteration of applying $Cpre$, we get $a^2 = \{base \mapsto \top, move \mapsto pos = 12 \wedge done = 1\}$. Then we get $a^3 = \{\ldots, mine \mapsto pos = 12 \wedge samp \geqslant req\}$. In the only entry of C_{ex}, the target set $targ = \{mine \mapsto samp \geqslant req\}$ contains more states in $mine$ (i.e., all possible positions of the robot) then a^3 (which asserts $pos = 12$). However, $\textsc{StrengthenTarget}(targ, \mathbb{X}_{ind}, a^3)$ as implemented above, will return the strengthening $pos = 12$ (after simplifying the formula), which makes the cache entry with $targ$ applicable. Since $src = \{mine \mapsto \top\}$, we update a^3 to $\{\ldots, mine \mapsto pos = 12\}$ in line 10 of the algorithm.

4 Abstract Template-Based Cache Generation

Section 3 defined attractor caches and showed their utilization for attractor computations via Algorithm 1. We motivated this approach by the observation that there often exist *small local sub-games* that entail essential attractors, and precomputing these attractors within the sub-games, caching them and then using them via Algorithm 1 is more efficient then only applying acceleration over the entire game (as in [21]). To formalize this workflow, Sect. 4.1 explains the generation of cache entries from sub-game structures of the given reactive program game, and Sect. 4.2 discusses the identification of helpful sub-game structures via permissive strategy templates in finite-state abstractions of the given game.

4.1 Generating Attractor Caches from Sub-Games

Within this subsection, we consider a sub-game structure $\mathcal{G}'$ which is induced by a subset of locations $L_{sub} \subseteq L$ of the original reactive program game structure $\mathcal{G}$, as formalized next. Intuitively, we remove all locations from $\mathcal{G}$ not in L_{sub} and redirect their incoming transitions to a new sink location sink_{sub}.

Definition 5 (Induced Sub-Game Structure). *Let $\mathcal{G} = (T, \mathbb{I}, \mathbb{X}, L, Inv, \delta)$ be a reactive program game structure and let $L_{sub} \subseteq L$ be a set of locations. The*

sub-game structure induced by L_{sub} *is the reactive program game structure* $\mathsf{SubGame}(\mathcal{G}, L_{sub}) := (T, \mathbb{I}, \mathbb{X}', L', Inv', \delta')$ *where* $L' := L_{sub} \cup \{\mathsf{sink}_{sub}\}$, $\mathbb{X}' := \{x \in \mathbb{X} \mid x$ *appears in transitions from or invariants of* L_{sub} *in* $\mathcal{G}'\}$, $Inv'(l) := Inv(l)$ *for all* $l \in L' \setminus \{\mathsf{sink}_{sub}\}$ *and* $Inv'(\mathsf{sink}_{sub}) := \top$, *and* $\delta' := \{(l, g, u, l') \in \delta \mid l, l' \in L'\} \cup \{(\mathsf{sink}_{sub}, \top, \lambda x.\ x, \mathsf{sink}_{sub})\} \cup$
$\{(l, g, \lambda x.x, \mathsf{sink}_{sub}) \mid \exists l' \in L.\ (l, g, u, l') \in \delta \wedge l \in L' \wedge l' \notin L'\}$.

Recall that $\mathcal{D} = L \to FOL(\mathbb{X})$. Let $\mathcal{D}' := L' \to FOL(\mathbb{X}')$ be the symbolic domain for a sub-game structure with locations L'. As $\mathbb{X}' \subseteq \mathbb{X}$, $FOL(\mathbb{X}') \subseteq FOL(\mathbb{X})$ which allows us to extend each element of $\mathcal{D}'$ to an element of $\mathcal{D}$ that agrees on L'. Formally, we define $\mathsf{extend}_L : \mathcal{D}' \to \mathcal{D}$ such that for $d' \in \mathcal{D}'$ and $l \in L$ we have $\mathsf{extend}_L(d')(l) := \texttt{if } l \in L' \texttt{ then } d'(l) \texttt{ else } \bot$.

The computation of an attractor cache from an induced sub-game is detailed in Algorithm 2. Given a reactive program game structure $\mathcal{G}$, a player p, and a subset of locations L_{sub}, Algorithm 2 first computes the induced sub-game (line 2). The quantifier elimination ([7, Ch. 7]) QElim in line 3 projects the given $d \in \mathcal{D}$ to an element d' of the symbolic domain $\mathcal{D}'$ of the sub-game structure. Then, in line 4, we perform the accelerated attractor computation from [21] with target set d' to obtain the set of states a from which player p can *enforce* reaching d' in $\mathcal{G}'$. The independent variables are those variables in $\mathbb{X}$ that are not updated in any of the transitions in $\mathcal{G}'$. Formally, we define those as $\mathsf{IndependentVars}(\mathcal{G}, \mathcal{G}') := \{x \in \mathbb{X} \mid \forall (l, g, u, l') \in \delta'.\ u(x) = x\}$. In order to output an attractor cache for the original game $\mathcal{G}$, we extend the computed source and target sets a and d' via the previously defined function extend_L (line 6). Intuitively, the attractor computed over a sub-game $\mathcal{G}'$ is also an attractor for the overall game $\mathcal{G}$ as sub-games are only restricted by location (not by variables). Hence, player p can also enforce reaching the target set in the original game $\mathcal{G}$, if he can do so in $\mathcal{G}'$. This is formalized by the next lemma.

Lemma 2. *Let* $\mathcal{G} = (T, \mathbb{I}, \mathbb{X}, L, Inv, \delta)$ *be a reactive program game structure, and let* $\mathcal{G}' = (T, \mathbb{I}, \mathbb{X}', L', Inv', \delta')$ *be an induced sub-game structure with sink location* sink_{sub} *constructed as above. Let* $src', targ' \in \mathcal{D}'$ *be such that* $targ'(\mathsf{sink}_{sub}) = \bot$ *and* $[\![src']\!] \subseteq Attr_{[\![\mathcal{G}']\!],p}([\![targ']\!])$ *for some player* $p \in \{Sys, Env\}$. *Furthermore, let* $\mathbb{Y} \subseteq \mathsf{IndependentVars}(\mathcal{G}, \mathcal{G}')$. *Then, for every* $\varphi \in FOL(\mathbb{Y})$ *it holds that*

$$[\![\mathsf{extend}_L(src') \wedge \lambda l.\varphi]\!] \subseteq Attr_{[\![\mathcal{G}]\!],p}([\![\mathsf{extend}_L(targ') \wedge \lambda l.\varphi]\!]).$$

This results in the following correctness statement.

Lemma 3. $\textsc{SubgameCache}(\mathcal{G}, p, L_{sub}, d)$ *returns an attractor cache over* $\mathcal{G}$.

Example 4. Consider the reactive program game structure $\mathcal{G}_{ex}$ from Example 1. We apply $\textsc{SubgameCache}(\mathcal{G}_{ex}, Sys, \{mine\}, d)$ with $d = \{mine \mapsto samp \geqslant req \wedge pos = 12 \wedge done \neq 1\}$. First, we construct the induced sub-game structure in Figrue 3. Quantifier elimination produces the target set $d' = \{mine \mapsto samp \geqslant req\}$. If we compute the attractor in this sub-game to set d', we get $\{mine \mapsto \top\}$.

Algorithm 2: Cache generation based on an induced sub-game.

```
function SubgameCache($\mathcal{G}$, $p$, $L_{sub}$, $d \in \mathcal{D}$)
    $\mathcal{G}' = (T, \mathbb{I}, \mathbb{X}', L', Inv', \delta') := \mathsf{SubGame}(\mathcal{G}, L_{sub})$
    $d' := \lambda l.$ if $l \in L_{sub}$ then $\mathsf{QElim}(\exists(\mathbb{X} \setminus \mathbb{X}').d(l))$ else $\bot$
    $a :=$ AttractorAcc($\mathcal{G}', p, d'$) /* attractor computation from [21] */
    $\mathbb{X}_{ind} := \mathsf{IndependentVars}(\mathcal{G}, \mathcal{G}')$
    return $\{(\mathcal{G}, p, \mathsf{extend}_L(a), \mathsf{extend}_L(d'), \mathbb{X}_{ind})\}$
```

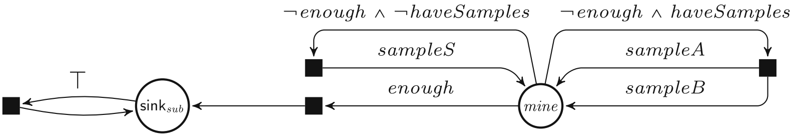

$\neg enough \wedge \neg haveSamples$
$\neg enough \wedge haveSamples$
$sampleS$
$sampleA$
$\top$
$enough$
$sampleB$
sink_{sub}
$mine$

Fig. 3. Induced sub-game structure $\mathsf{SubGame}(\mathcal{G}_{ex}, \{mine\})$ of the reactive program game structure $\mathcal{G}_{ex}$ from Fig. 2, with the same abbreviations as in Fig. 2.

Note that since the number of steps needed to reach d' depends on the initial value of *samp* and is hence unbounded, a technique like acceleration [21] is necessary to compute this attractor. As in this sub-game structure only the variable *samp* is updated, the independent variables are $\mathbb{X}_{ind} = \{done, pos, req\}$. With this we get the cache entry from Example 2.

4.2 Constructing Sub-games from Abstract Strategy Templates

The procedure from the previous subsection yields attractor caches regardless of how the sub-games are chosen. In this section we describe our approach to identifying "useful" sub-game structures. These sub-game structures are induced by so-called *helpful edges* determined by permissive strategy templates. Since the game graph described by a reactive program game structure is in general infinite, we first construct finite abstract games in which we compute permissive strategy templates for the two players. We start by describing the abstract games.

Finite Abstractions of Reactive Program Games. Here we describe the construction of a game graph $\widehat{G} = (V, V_{Env}, V_{Sys}, \widehat{\rho})$ from a reactive program game structure $\mathcal{G} = (T, \mathbb{I}, \mathbb{X}, L, Inv, \delta)$ with semantics $[\![\mathcal{G}]\!] = (\mathcal{S}, \mathcal{S}_{Env}, \mathcal{S}_{Sys}, \rho)$. While $[\![\mathcal{G}]\!]$ is also a game graph, its vertex set is typically infinite. The game graph $\widehat{G}$, which is an abstraction of $[\![\mathcal{G}]\!]$, has a finite vertex set instead.

We construct the game graph $\widehat{G}$ from $\mathcal{G}$ by performing abstraction with respect to a given abstract domain. The abstract domain consists of two finite sets of quantifier-free first-order formulas which are used to define the vertex sets of the game graph $\widehat{G}$. The conditions that we impose in the definition of abstraction domain given below ensure that it can partition the state space of $\mathcal{G}$.

Definition 6 (Game Abstraction Domain). *A* game abstraction domain *for a reactive program game structure $\mathcal{G} = (T, \mathbb{I}, \mathbb{X}, L, Inv, \delta)$ is a pair of finite sets of quantifier-free first-order formulas $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}}) \in QF(\mathbb{X}) \times QF(\mathbb{X} \cup \mathbb{I})$ such that for $\mathcal{P} = \mathcal{P}_{\mathbb{X}}$ (resp. $\mathcal{P} = \mathcal{P}_{\mathbb{X}\cup\mathbb{I}}$) and $V = \mathbb{X}$ (resp. $V = \mathbb{X} \cup \mathbb{I}$), $\mathcal{P}$ partitions $Assignments(V)$, i.e. $Assignments(V) = \bigcup_{\varphi\in\mathcal{P}}\{\mathbf{v} \mid \mathbf{v} \models_T \varphi\}$ and for every $\varphi_1, \varphi_2 \in \mathcal{P}$ with $\varphi_1 \wedge \varphi_2$ satisfiable it holds that $\varphi_1 = \varphi_2$.*

The abstraction domain we use consists of all conjunctions of atomic predicates (and their negations) that appear in the guards of the reactive program game structure $\mathcal{G}$. Let GA be the set of atomic formulas appearing in the guards of $\mathcal{G}$. We use the abstraction domain $\mathsf{AbstractDomain}(\mathcal{G}) := (\mathcal{P}_{\mathbb{X}}^{GA}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}}^{GA})$ where

$$\begin{aligned}\mathcal{P}_{\mathbb{X}}^{GA} &:= \{\textstyle\bigwedge_{\varphi\in J}\varphi \wedge \bigwedge_{\varphi\notin J}\neg\varphi \mid J \subseteq GA \cap FOL(\mathbb{X})\},\\ \mathcal{P}_{\mathbb{X}\cup\mathbb{I}}^{GA} &:= \{\textstyle\bigwedge_{\varphi\in J}\varphi \wedge \bigwedge_{\varphi\notin J}\neg\varphi \mid J \subseteq GA \cap (FOL(\mathbb{X}\cup\mathbb{I}) \setminus FOL(\mathbb{X}))\}.\end{aligned}$$

Example 5. In the game structure $\mathcal{G}_{ex}$ from Example 1, we get for $\mathcal{P}_{\mathbb{X}}^{GA}$ all combinations of $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$, where $\varphi_1 \in \{req < samp, req \geqslant samp\}$, $\varphi_2 \in \{pos = 12, pos = 23, pos \neq 12 \wedge pos \neq 23\}$, and $\varphi_3 \in \{task = 1, task \neq 1\}$. For $\mathcal{P}_{\mathbb{X}\cup\mathbb{I}}^{GA}$ we get all combinations of $\psi_1 \wedge \psi_2 \wedge \psi_3$, where $\psi_1 \in \{a \leqslant 0, a > 0\}$, $\psi_2 \in \{b \leqslant 0, b > 0\}$, and $\psi_3 \in \{inpReq \leqslant 0, inpReq > 0\}$.

We choose this abstraction domain as a baseline since the predicates appearing in the guards are natural delimiters in the program variable state space. However, the abstraction we define now is independent of this specific domain.

Given a game abstraction domain $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}})$, we construct two abstract game graphs, $\widehat{G}^{\uparrow}$ and $\widehat{G}^{\downarrow}$. They have the same sets of vertices but differ in the transition relations. The transition relation in $\widehat{G}^{\uparrow}$ overapproximates the transitions originating from states of player *Sys* and underapproximates the transitions from states of player *Env*. In $\widehat{G}^{\downarrow}$ the approximation of the two players is reversed.

Definition 7 (Abstract Game Graphs). *Let $\mathcal{G} = (T, \mathbb{I}, \mathbb{X}, L, Inv, \delta)$ be a reactive program game structure and $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}})$ be an abstraction domain. The game graphs $\widehat{G}^{\circ} = (V, V_{Env}, V_{Sys}, \widehat{\rho}^{\circ})$ with $\circ \in \{\uparrow, \downarrow\}$ are the $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}})$-induced abstractions of $\mathcal{G}$ if $V := V_{Sys} \cup V_{Env}$, $V_{Env} := L \times \mathcal{P}_{\mathbb{X}}$, and $V_{Sys} := L \times \mathcal{P}_{\mathbb{X}} \times \mathcal{P}_{\mathbb{X}\cup\mathbb{I}}$; and $\widehat{\rho}^{\circ} \subseteq (V_{Env} \times V_{Sys}) \cup (V_{Sys} \times V_{Env})$ is the smallest relation such that*

- *$((l, \varphi), (l, \varphi, \varphi_I)) \in \widehat{\rho}^{\circ} \cap (V_{Env} \times V_{Sys})$ iff the following formula is valid*

  $$move_{\mathbb{X}}^{\bullet}(Inv(l) \wedge \varphi(\mathbb{X}), \exists\mathbb{I}.\ \varphi_I(\mathbb{X}, \mathbb{I}))$$

  for $\bullet =\downarrow$ if $\circ =\uparrow$ and $\bullet =\uparrow$ if $\circ =\downarrow$,
- *$((l, \varphi, \varphi_I), (l', \varphi')) \in \widehat{\rho}^{\circ} \cap (V_{Sys} \times V_{Env})$ iff the following formula is valid*

  $$move_{\mathbb{X}\cup\mathbb{I}}^{\circ}(Inv(l) \wedge \varphi(\mathbb{X}) \wedge \varphi_I(\mathbb{X}, \mathbb{I}), \exists(g, u) \in Labels(l, l').\ trans(g, u, l', \varphi'))$$

  for $trans(g, u, l', \varphi') := g(\mathbb{X}, \mathbb{I}) \wedge \big(\varphi' \wedge Inv(l')\big)(u(\mathbb{X}, \mathbb{I}))$, *where* $move^{\uparrow}_{V}(\varphi, \varphi') := \exists V.\varphi(V) \wedge \varphi'(V)$ *and* $move^{\downarrow}_{V}(\varphi, \varphi') := \forall V.\varphi(V) \rightarrow \varphi'(V)$.

Definition 7 provides us with a procedure AbstractRPG for constructing the pair of abstractions $(\widehat{G}^{\uparrow}, \widehat{G}^{\downarrow}) := \textsc{AbstractRPG}(\mathcal{G}, (\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}}))$.

We refer to the vertices in the abstract game graphs as abstract states. By slightly overloading notation, we define the projection from abstract states $v \in V$ to the respective location by $loc : V \to L$ s.t. $loc((l, \varphi)) = l$ and $loc((l, \varphi, \varphi_I)) = l$. This definition naturally extends to sequences of abstract states $\pi \in V^{\infty}$ s.t. $loc(\widehat{\pi})[i] = loc(\widehat{\pi}[i])$ for all $i \in dom(\pi)$, and to sets of vertices: $loc : 2^V \to 2^L$. Given $\mathcal{G}$ with semantics $[\![\mathcal{G}]\!] = (\mathcal{S}, \mathcal{S}_{Env}, \mathcal{S}_{Sys}, \rho)$ and an abstract game graph $\widehat{G}^{\circ} = (V, V_{Env}, V_{Sys}, \widehat{\rho}^{\circ})$, we define the following functions between their respective state spaces. The *concretization function* $\gamma : V \to 2^{\mathcal{S}}$ is defined s.t.

$$\begin{aligned} \gamma((l, \varphi)) &:= \{(l, \mathbf{x}) \in \mathcal{S}_{Env} \mid \mathbf{x} \models_T \varphi\} \text{ and} \\ \gamma((l, \varphi, \varphi_I)) &:= \{((l, \mathbf{x}), \mathbf{i}) \in \mathcal{S}_{Sys} \mid \mathbf{x} \uplus \mathbf{i} \models_T \varphi \wedge \varphi_I\}. \end{aligned}$$

The *abstraction function* $\alpha : \mathcal{S} \to 2^V$ is defined s.t. $v \in \alpha(s)$ iff $s \in \gamma(v)$. We extend both function from states to (finite or infinite) state sequences $\pi \in \mathcal{S}^{\infty}$ and $\widehat{\pi} \in V^{\infty}$ s.t.

$$\begin{aligned} \gamma(\widehat{\pi}) &:= \{\pi \in \mathcal{S}^{\infty} \mid |\pi| = |\widehat{\pi}| \wedge \forall i \in dom(\pi).\ \pi[i] \in \gamma(\widehat{\pi}[i])\}, \text{ and} \\ \alpha(\pi) &:= \{\widehat{\pi} \in V^{\infty} \mid |\pi| = |\widehat{\pi}| \wedge \forall i \in dom(\pi).\ \widehat{\pi}[i] \in \alpha(\pi[i])\}. \end{aligned}$$

Both functions naturally extend to *sets* of states or infinite sequences of states by letting $\gamma(A) := \bigcup_{a \in A} \gamma(a)$ for $A \subseteq V$ and $A \subseteq V^{\omega}$ and $\alpha(C) := \bigcup_{c \in C} \alpha(c)$ for $C \subseteq \mathcal{S}$ and $C \subseteq \mathcal{S}^{\omega}$. Note that it follows from the partitioning conditions imposed on $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}})$ in Definition 6 that α is a total function and always maps states and state sequences to a singleton set. We abuse notation and write $\alpha(s) = v$ (resp. $\alpha(\pi) = \widehat{\pi}$) instead of $\alpha(s) = \{v\}$ (resp. $\alpha(\pi) = \{\widehat{\pi}\}$).

Let $\Omega \subseteq \mathcal{S}^{\infty}$ be an objective for the semantic game $[\![\mathcal{G}]\!]$. With the relational functions $\langle \alpha, \gamma \rangle$ defined before, Ω naturally induces an abstract objective $\widehat{\Omega} := \alpha(\Omega) \subseteq V^{\infty}$ over the abstract state space V.

Recall that we consider winning conditions $\Omega \subseteq \mathcal{S}^{\infty}$ for $[\![\mathcal{G}]\!]$ defined over the set L of locations of $\mathcal{G}$. As α preserves the location part of the states, $\widehat{\pi} \in \widehat{\Omega}$ iff $\gamma(\widehat{\pi}) \subseteq \Omega$. That is, a sequence of abstract states is winning according to $\widehat{\Omega}$ iff all the corresponding concrete state sequences are winning according to Ω.

The next lemma states the correctness property that the abstraction satisfies. More concretely, $\widehat{G}^{\uparrow}$ overapproximates the winning region of player *Sys* in the concrete game, and $\widehat{G}^{\downarrow}$ underapproximates it.

Lemma 4 (Correctness of the Abstraction). *Given a reactive program game structure* $\mathcal{G}$ *with semantics* $[\![\mathcal{G}]\!] = (\mathcal{S}, \mathcal{S}_{Env}, \mathcal{S}_{Sys}, \rho)$ *and location-based objective* Ω*, let* $\widehat{G}^{\circ} = (V, V_{Env}, V_{Sys}, \widehat{\rho}^{\circ})$ *with* $\circ \in \{\uparrow, \downarrow\}$ *be its* $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}})$*-induced abstractions with relational functions* $\langle \alpha, \gamma \rangle$*. Then it holds that (1)* $W_{Sys}([\![\mathcal{G}]\!], \Omega) \subseteq \gamma(W_{Sys}(\widehat{G}^{\uparrow}, \widehat{\Omega}))$*, and (2)* $\gamma(W_{Sys}(\widehat{G}^{\downarrow}, \widehat{\Omega})) \subseteq W_{Sys}([\![\mathcal{G}]\!], \Omega)$*.*

By duality, $\widehat{G}^{\downarrow}$ results in an overapproximation of the winning region of player *Env* in the concrete game. Given an abstraction $\widehat{G}^{\circ}$, we denote with $\mathit{OverapproxP}(\widehat{G}^{\circ})$ the player whose winning region is overapproximated in $\widehat{G}^{\circ}$: $\mathit{OverapproxP}(\widehat{G}^{\circ}) := \mathit{Sys}$ if $\circ = \uparrow$ and $\mathit{OverapproxP}(\widehat{G}^{\circ}) := \mathit{Env}$ if $\circ = \downarrow$.

Abstract Strategy Templates and Their Induced Sub-games. We now describe how we use a permissive strategy template for a player p in an abstract game to identify sub-game structures of the given reactive program game from which to generate attractor caches for player p.

We determine the sub-game structures and local target sets based on so-called *helpful edges* for player p in the abstract game where p is over-approximated. A helpful edge is a live-edge or an alternative choice to a co-live edge of a permissive strategy template. Intuitively, a helpful edge is an edge that player p might have to take eventually in order to win the abstract game. As our chosen abstraction domain is based on the guards, a helpful edge often corresponds to the change of conditions necessary to enable a guard in the reactive program game. Since reaching this change might require an unbounded number of steps, our method attempts a local attractor computation and potentially acceleration. Identifying helpful edges based on permissive strategy templates rather than on winning strategies has the following advantages. First, templates reflect multiple abstract winning strategies for player p, capturing multiple possibilities to make progress towards the objective. Moreover, they describe local conditions, facilitating the localization our method aims for. Helpful edges are defined as follows.

Definition 8 (Helpful Edge). *Given a strategy template $(U, D, \mathcal{H})$ for player p in a game (G, Ω) with $G = (V, V_{Env}, V_{Sys}, \rho)$, we call an edge $e \in \rho$* helpful *for player p* w.r.t. *the template $(U, D, \mathcal{H})$ if and only if the following holds: There exists a live-group $H \in \mathcal{H}$ such that $e \in H$, or $e \notin U \cup D$ and there exists a co-live edge $(v_s, v_t) \in D$ with $v_s = \textsc{src}(e)$. We define $\mathsf{Helpful}_{G,p}(U, D, \mathcal{H})$ to be the set of helpful edges for player p in G w.r.t. $(U, D, \mathcal{H})$.*

For each helpful edge, we define *pre-* and *post-sets* which are the abstract environment states before and after that edge. This is formalized as follows.

Definition 9 (Pre- and Post-Sets). *Let $\widehat{G}^{\circ} = (V, V_{Env}, V_{Sys}, \widehat{\rho}^{\circ})$ for some $\circ \in \{\uparrow, \downarrow\}$ be a $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X} \cup \mathbb{I}})$-induced abstraction of $\mathcal{G}$, let $p_{over} := \mathit{OverapproxP}(\widehat{G}^{\circ})$, and $e = (v_s, v_t) \in \mathsf{Helpful}_{\widehat{G}^{\circ}, p_{over}}(U, D, \mathcal{H})$ for some template $(U, D, \mathcal{H})$. If $p_{over} = \mathit{Env}$, we have $e \in V_{Env} \times V_{Sys}$ and define $\mathsf{Pre}(e, p_{over}) := \{v_s\}$ and $\mathsf{Post}(e, p_{over}) := \{v \in V \mid (v_t, v) \in \widehat{\rho}^{\circ}\}$. If $p_{over} = \mathit{Sys}$ we have that $e \in V_{Sys} \times V_{Env}$ and define $\mathsf{Pre}(e, p_{over}) := \{v \in V \mid (v, v_s) \in \widehat{\rho}^{\circ}\}$ and $\mathsf{Post}(e, p_{over}) := \{v_t\}$. Note that in both cases it holds that $\mathsf{Pre}(e, p_{over}), \mathsf{Post}(e, p_{over}) \subseteq V_{Env} \subseteq L \times \mathcal{P}_{\mathbb{X}}$.*

As a helpful edge represents potential "progress" for player p, we consider the question of whether player p has a strategy in the concrete game to reach the *post-set* from the *pre-set.* This motivates the construction of sub-game structures induced by the locations connecting those two sets in the reactive program game.

Algorithm 3: Generation of a cache based on a strategy template.

```
function GenerateCache($\mathcal{G}$, $\widehat{G}^{\circ}$, $p_{over}$, $(U, D, \mathcal{H})$, $b \in \mathbb{N}$)
    $SubgameLocs := \varnothing$, $PostSet := \varnothing$,
    foreach $e \in \mathsf{Helpful}_{\widehat{G}^{\circ}, p_{over}}(U, D, \mathcal{H})$ do
        $L_S := loc(\mathsf{Pre}(e, p_{over}))$; $L_T := loc(\mathsf{Post}(e, p_{over}))$
        $L_{sub} := \{l \mid \exists w \in \mathsf{SimplePaths}(\mathcal{G}, L_S, L_T).\ |w| \leqslant b \wedge \exists i.\ w[i] = l\}$
        $SubgameLocs := SubgameLocs \cup \{L_{sub}\}$
        $PostSet := PostSet \cup \{(L_{sub}, \mathsf{Post}(e, p_{over}))\}$
    $C := \varnothing$
    foreach $L_{sub} \in SubgameLocs$ do
        $TargetSet :=$ ConstructTargets($L_{sub}, PostSet$) /* see Eq. (1)    */
        foreach $targ \in TargetSet$ do
            $C := C \cup$ SubgameCache($\mathcal{G}, p_{over}, L_{sub}, targ$)
    return $C$
```

Procedure GenerateCache in Algorithm 3 formalizes this idea. It takes an abstract game and a strategy template for the over-approximated player p_{over} in this game. For each helpful edge e, it constructs the sub-game structure induced by the set of locations that lie on a simple path in the location graph from the locations of the *pre-set* to the *post-set* of e. The optional parameter b allows for heuristically tuning the locality of the sub-games by bounding the paths' length.

For each sub-game structure, the target sets for the local attractor computations are determined by the post-sets of the helpful edges that induced this sub-game structure (it might be more than one). They are computed by

$$\textsc{ConstructTargets}(L_{sub}, PostSet) = T_1 \cup T_2 \cup T_3 \tag{1}$$

where the sets T_1, T_2 and T_3 of elements of $\mathcal{D}$ are defined as follows.

- $T_1 := \{d \in \mathcal{D} \mid \exists P.\ (L_{sub}, P) \in PostSet \wedge \forall l \in L.\ d(l) = \bigvee_{(l,\varphi) \in P} \varphi\}$ consists of targets that are determined by a single post-set.
- $T_2 := \{d_{\cup}\}$, where for every $l \in L$, $d_{\cup}(l) = \bigvee_{P \text{ s.t. } (L_{sub}, P) \in PostSet} \bigvee_{(l,\varphi) \in P} \varphi$ is the singleton containing the union of the targets of all post-sets.
- $T_3 := \{d_{\top}\}$, where for $l \in L$, $d_{\top}(l) = \exists P, \varphi.(L_{sub}, P) \in PostSet \wedge (\varphi, l) \in P$ contains the target that is $\top$ iff the location appears in some post-set.

Once the targets are constructed, GenerateCache uses SubgameCache from Algorithm 2 to compute the attractor caches for those targets and respective sub-game structures. By Lemma 3, SubgameCache returns attractor caches. As attractor caches are closed under set union, we conclude the following.

Corollary 1. *The set C returned by* GenerateCache *is an attractor cache.*

Example 6. The abstractions of $\mathcal{G}_{ex}$ from Example 1 and respective templates are too large to depict. One helpful edge for *Sys* is $e = ((mine, \varphi, \varphi_I), (mine, \varphi'))$

Algorithm 4: Game solving with abstract template-based caching.

```
function RPGCacheSolve($\mathcal{G} = (T, \mathbb{I}, \mathbb{X}, L, Inv, \delta)$, $\Omega$, $b \in \mathbb{N}$)
    $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}}) :=$ AbstractDomain($\mathcal{G}$)
    $(\widehat{G}^{\uparrow}, \widehat{G}^{\downarrow}) :=$ AbstractRPG($\mathcal{G}, (\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}})$) /* see Definition 7 */
    $C := \varnothing$
    foreach $(p, \circ) \in \{(Sys, \uparrow), (Env, \downarrow)\}$ do
        $(U, D, \mathcal{H}) :=$ SolveAbstract($\widehat{G}^{\circ}, \Omega$)
        $C := C \cup$ GenerateCache($\mathcal{G}, \widehat{G}^{\circ}, p, (U, D, \mathcal{H}), b$)
    return RPGSolveWithCache($\mathcal{G}, C$) /* solves $\mathcal{G}$ using
        AttractorAccCache in Algorithm 1 for attractor computation */
```

with $\varphi = samp < req \wedge pos = 12 \wedge done \neq 1$, $\varphi' = samp \geqslant req \wedge pos = 12 \wedge done \neq 1$, and $\varphi_I = a > 0 \wedge b \leqslant 0 \wedge inpReq \leqslant 0$. This edge e is in a live group where the other edges are similar with different φ_I. They correspond to the situation where the value of *samp* finally becomes greater or equal to *req*. For e, $\mathsf{Pre}(e, Sys) = \{(mine, \varphi)\}$ and $\mathsf{Post}(e, Sys) = \{(mine, \varphi')\}$ result in $L_{sub} = \{mine\}$ and the target $\{mine \mapsto \varphi'\}$. With this, we generate a cache as in Example 4.

5 Game Solving with Abstract Template-Based Caching

This section summarizes our approach for reactive progam game solving via Algorithm 4, which combines the procedures introduced in Sect. 3 and Sect. 4 as schematically illustrated in Fig. 1 of Sect. 1. Algorithm 4 starts by computing the abstract domain and both abstractions. For each abstract game, SolveAbstract computes a strategy template [2]. Then, GenerateCache is invoked to construct the respective attractor cache. RPGSolveWithCache solves reactive program games in direct analogy to RPGSolve from [21], but instead of using AttractorAcc, it uses the new algorithm AttractorAccCache which utilizes the attractor cache C. The overall correctness of RPGCacheSolve follows from Lemma 1, Corollary 1, and the correctness of [21].

Theorem 1 (Correctness). *Given a reactive program game structure $\mathcal{G}$ and a location-based objective Ω, for any $b \in \mathbb{N}$, if* RPGCacheSolve *terminates, then it returns $W_{Sys}([\![\mathcal{G}]\!], \Omega)$.*

Remark 1. In addition to using the strategy templates from the abstract games for caching, we can make use of the winning regions in the abstract games, which are computed together with the templates. Thanks to Lemma 4, we know that outside of its winning region in the abstract game the over-approximated player loses for sure. Thus, we can *prune* parts of the reactive program game that correspond to the abstract states where the over-approximated player loses. As our experiments show that the main performance advantage is gained by caching rather than pruning, we give the formal details for pruning in the extended version [34].

Discussion. The procedure RPGCacheSolve depends on the choice of game abstraction domain $(\mathcal{P}_{\mathbb{X}}, \mathcal{P}_{\mathbb{X}\cup\mathbb{I}})$ and on the construction of the local games performed in GenerateCache. The abstraction based on guards is natural, as it is obtained from the predicates appearing in the game. Acceleration [21] is often needed to establish that some guards can eventually be enabled. Therefore, we choose an abstraction domain that represents precisely the guards in the game.

Helpful edges capture transitions that a player might need to take, hence the game solving procedure has to establish that the player can eventually enable their guards. This might require acceleration, and hence motivates our use of helpful edges to construct the local games. Investigating alternatives to these design choices and their further refinement is a subject of future work.

6 Experimental Evaluation

We implemented Algorithm 4 for solving reactive program games in a prototype tool[2] rpg-STeLA (Strategy Template-based Localized Acceleration). Our implementation is based on the open-source reactive program game solver rpgsolve from [21]. Specifically, we use rpgsolve for the AttractorAcc and RPGSolveWithCache methods to compute attractors via acceleration and to solve reactive program games utilizing the precomputed cache, respectively. We realize SolveAbstract by using PeSTel [2], which computes strategy templates in finite games. We do not use the bound b in Algorithm 4.

We compare our tool rpg-STeLA to the solver rpgsolve and the μCLP solver MuVal [36]. Those are the only available techniques that can handle unbounded strategy loops, as stated in [21]. Other tools from [5,8–10,27,28,31–33] cannot handle those, are outperformed by rpgsolve, or are not available. For MuVal, we encoded the games into μCLP as outlined in [36] and done in [21].

Benchmarks. We performed the evaluation on three newly introduced sets of benchmarks (described in detail in [34]). They all have unbounded variable ranges, contain unbounded strategy loops, and have Büchi winning conditions. On the literature benchmarks from [6,21,27,31,39] rpgsolve performs well as [21] shows. Hence, we did not use them as local attractor caches are unnecessary, and they are smaller than our new ones. Our new benchmark categories are:
(1) Complex Global Strategy (Scheduler and Item Processing). These benchmarks consist of a scheduler and an item processing unit. The core feature of these benchmarks is that the system needs to perform tasks that require complex global strategic decisions and local strategic decisions requiring acceleration.
(2) Parametric Benchmarks (Chains). These benchmarks each consist of two parametric chains of local sub-tasks requiring acceleration and local strategic reasoning and more lightweight global strategic reasoning. The number of variables scales differently in both chains, showcasing differences in scalability.
(3) Simple Global Strategy (Robot and Smart Home). These benchmarks represent different tasks for a robot and a smart home. The robot moves along tracks

[2] Available at https://doi.org/10.5281/zenodo.10939871.

Table 1. Evaluation Results. ST is the variable domain type (additional to $\mathbb{B}$). $|L|$, $|\mathbb{X}|$, $|\mathbb{I}|$ are the number of respective game elements. We show the wall-clock running time in seconds for our prototype rpg-STeLA in three settings (one with normal caching, one with additional pruning, one that only prunes), rpgsolve, and MuVal (with clause exchange). TO means timeout after 30 min, MO means out of memory (8GB). We highlight in bold the fastest solving runtime result. The evaluation was performed on a computer equipped with an Intel(R) Core(TM) i5-10600T CPU @ 2.40 GHz.

Name	ST	$\|L\|$	$\|\mathbb{X}\|$	$\|\mathbb{I}\|$	rpg-STeLA			rpgsolve	MuVal
					normal	pruning	prune-only		
scheduler	$\mathbb{Z}$	7	3	3	110.3	73.43	202.23	99.57	**52.66**
item processing	$\mathbb{Z}$	7	4	2	**473.85**	479.34	TO	TO	TO
chain 4	$\mathbb{Z}$	7	6	1	**128.02**	128.48	TO	TO	TO
chain 5	$\mathbb{Z}$	8	7	1	**410.90**	413.75	TO	TO	TO
chain 6	$\mathbb{Z}$	9	8	1	**1464.86**	1470.13	TO	TO	TO
chain 7	$\mathbb{Z}$	10	9	1	TO	TO	TO	TO	TO
chain simple 5	$\mathbb{Z}$	8	3	1	**27.54**	29.10	1364.91	1362.38	TO
chain simple 10	$\mathbb{Z}$	13	3	1	**76.41**	80.01	TO	TO	TO
chain simple 20	$\mathbb{Z}$	23	3	1	**236.74**	244.53	TO	TO	TO
chain simple 30	$\mathbb{Z}$	33	3	1	**485.73**	503.89	TO	TO	TO
chain simple 40	$\mathbb{Z}$	43	3	1	**813.05**	826.67	TO	TO	TO
chain simple 50	$\mathbb{Z}$	53	3	1	**1212.90**	1240.36	TO	TO	TO
chain simple 60	$\mathbb{Z}$	63	3	1	**1704.02**	1718.39	TO	TO	TO
chain simple 70	$\mathbb{Z}$	73	3	1	TO	TO	TO	TO	TO
robot running (Example 1)	$\mathbb{Z}$	3	4	3	**470.69**	471.59	TO	TO	TO
robot repair	$\mathbb{Z}$	6	4	2	TO	91.66	**51.40**	TO	TO
robot analyze samples	$\mathbb{Z}$	6	3	1	**104.02**	113.06	684.67	632.39	TO
robot collect samples v1	$\mathbb{Z}$	4	3	1	**22.89**	26.89	TO	TO	TO
robot collect samples v2	$\mathbb{Z}$	3	4	1	**478.33**	483.50	TO	TO	TO
robot collect samples v3	$\mathbb{Z}$	4	3	3	**60.55**	65.76	TO	TO	TO
robot deliver products 1	$\mathbb{Z}$	6	5	1	**95.08**	101.75	TO	TO	TO
robot deliver products 2	$\mathbb{Z}$	7	6	2	**724.20**	741.01	TO	TO	TO
robot deliver products 3	$\mathbb{Z}$	7	6	3	**1116.31**	1133.57	TO	TO	TO
robot deliver products 4	$\mathbb{Z}$	7	6	4	**1580.03**	1615.72	TO	TO	TO
robot deliver products 5	$\mathbb{Z}$	7	6	5	TO	TO	TO	TO	TO
smart home day not empty	$\mathbb{R}$	5	5	2	**84.17**	100.99	TO	TO	TO
smart home day warm	$\mathbb{R}$	6	5	3	**162.06**	187.82	TO	TO	TO
smart home day cold	$\mathbb{R}$	6	5	3	**162.06**	193.81	TO	TO	TO
smart home day warm or cold	$\mathbb{R}$	6	5	4	**320.27**	380.88	TO	TO	TO
smart home day empty	$\mathbb{R}$	5	5	2	TO	TO	TO	TO	MO
smart home night sleeping	$\mathbb{R}$	6	5	2	**80.69**	99.38	TO	TO	MO
smart home night empty	$\mathbb{R}$	6	5	2	TO	TO	TO	TO	TO
smart home nightmode	$\mathbb{R}$	6	6	3	TO	TO	TO	TO	MO

(with one-dimensional discrete position) and must perform tasks like collecting several products. The smart home must, e.g., maintain temperature levels and adjust blinds depending on whether the house is empty or on the current time of day. These benchmarks need acceleration and local strategic reasoning, but their global reasoning is usually simpler and more deterministic.

Analysis. The experimental results in Table 1 demonstrate that local attractor pre-computation and caching have a significant impact on solving complex games. This is evidenced by the performance of rpg-STeLA that is superior to the other two tools. We further see that pruning (without caching) is not sufficient, which underscores the need to use more elaborate local strategic information in the form of an attractor cache. This necessitates the computation of strategy templates, and simply solving an abstract game is insufficient. However, as pruning does not cause significant overhead, it offers an additional optimization.

7 Related Work

A body of methods for solving infinite-state games and synthesizing reactive systems operating over unbounded data domains exists. Abstraction-based approaches reduce the synthesis problem to the finite-state case. Those include abstraction of two-player games [15,19,23,37,38], which extends ideas from verification, such as abstract interpretation and counterexample-guided abstraction refinement, to games. The temporal logic LTL has recently been extended with data properties, resulting in TSL [14] and its extension with logical theories [13]. Synthesis techniques for those [8,14,27] are based on propositional abstraction of the temporal specification and iterative refinement by introducing assumptions. The synthesis task's main burden in abstraction-based methods falls on the finite-state synthesis procedure. In contrast, we use abstraction not as the core solving mechanism but as a means to derive helpful sub-games. Another class of techniques reason directly over the infinite-state space. Several constraint-based approaches [9,10,24] have been proposed for specific types of objectives. [32,33] lift fixpoint-based methods for finite-state game solving to a symbolic representation of infinite state sets. However, a naive iterative fixpoint computation can be successful on a relatively limited class of games. Recently, [21] proposed a technique that addresses this limitation by accelerating symbolic attractor computations. However, as we demonstrate, their approach has limited scalability when the size of the game structure grows. Our method mitigates this by identifying small helpful sub-games and composing their solutions to solve the game.

There are many approaches for compositional synthesis from LTL specifications [11,12,16]. To the best of our knowledge, no techniques for decomposing infinite-state games exist prior to our work.

In verification, acceleration [3,4,17] and loop summarization [26] are applied to the loops in *given program* and can thus be easily combined with subsequent analysis. In contrast, in the setting of games, acceleration relies on *establishing the existence of a strategy* which needs more guidance.

Permissive strategy templates were introduced in [1] and used in [2] to represent sets of winning strategies for the system player in two-player games. They were used to synthesize hybrid controllers for non-linear dynamical systems [30]. Similar to our work, [30] uses templates over abstractions to localize the compuation of continuous feedback controllers. While this inspired the solution methodology for infinite-state systems developed in this paper, the abstraction methodology and the semantics of the underlying system and its controllers are very different in [30]. Our work is the first which uses permissive strategy templates as a guide for localizing the computation of fixpoints in infinite-state games.

8 Conclusion

We presented a method that extends the applicability of synthesis over infinite-state games towards realistic applications. The key idea is to reduce the game solving problem to smaller and simpler sub-problems by utilizing winning strategy templates computed in finite abstractions of the infinite-state game. The resulting sub-problems are solved using a symbolic method based on attractor acceleration. Thus, in our approach abstraction and symbolic game solving work in concert, using strategy templates as the interface between them. This opens up multiple avenues for future work, such as exploring different abstraction techniques, as well as developing data-flow analysis techniques for reactive program games that can be employed in the context of symbolic game-solving procedures.

Data Availability Statement. The software generated during and/or analysed during the current study is available in the Zenodo repository [22]. A full version of this paper including proofs is available through arXiv [34].

References

1. Anand, A., Mallik, K., Nayak, S.P., Schmuck, A.K.: Computing adequately permissive assumptions for synthesis. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 211–228. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_15
2. Anand, A., Nayak, S.P., Schmuck, A.: Synthesizing permissive winning strategy templates for parity games. In: Enea, C., Lal, A. (eds.) CAV 2023, Part I. LNCS, vol. 13964, pp. 436–458. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37706-8_22
3. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: fast acceleration of symbolic transition systems. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 118–121. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_12
4. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005). https://doi.org/10.1007/11562948_35
5. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20–21, 2014, pp. 221–234. ACM (2014). https://doi.org/10.1145/2535838.2535860
6. Bodlaender, M.H.L., Hurkens, C.A.J., Kusters, V.J.J., Staals, F., Woeginger, G.J., Zantema, H.: Cinderella versus the wicked stepmother. In: Baeten, J.C.M., Ball, T., de Boer, F.S. (eds.) TCS 2012. LNCS, vol. 7604, pp. 57–71. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33475-7_5
7. Bradley, A.R., Manna, Z.: The Calculus of Computation - Decision Procedures with Applications to Verification. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74113-8

8. Choi, W., Finkbeiner, B., Piskac, R., Santolucito, M.: Can reactive synthesis and syntax-guided synthesis be friends? In: Jhala, R., Dillig, I. (eds.) PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, 13–17 June, 2022, pp. 229–243. ACM (2022). https://doi.org/10.1145/3519939.3523429
9. Faella, M., Parlato, G.: Reachability games modulo theories with a bounded safety player. In: Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence. AAAI'23/IAAI'23/EAAI'23. AAAI Press (2023). https://doi.org/10.1609/aaai.v37i5.25779
10. Farzan, A., Kincaid, Z.: Strategy synthesis for linear arithmetic games. Proc. ACM Program. Lang. **2**(POPL), 61:1-61:30 (2018). https://doi.org/10.1145/3158149
11. Filiot, E., Jin, N., Raskin, J.: Antichains and compositional algorithms for LTL synthesis. Formal Methods Syst. Des. **39**(3), 261–296 (2011). https://doi.org/10.1007/S10703-011-0115-3
12. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis. Innov. Syst. Softw. Eng. **19**(4), 339–357 (2023). https://doi.org/10.1007/S11334-022-00462-6
13. Finkbeiner, B., Heim, P., Passing, N.: Temporal stream logic modulo theories. In: FoSSaCS 2022. LNCS, vol. 13242, pp. 325–346. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99253-8_17
14. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal stream logic: synthesis beyond the bools. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 609–629. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_35
15. Finkbeiner, B., Mallik, K., Passing, N., Schledjewski, M., Schmuck, A.: BOCoSy: small but powerful symbolic output-feedback control. In: Bartocci, E., Putot, S. (eds.) HSCC '22: 25th ACM International Conference on Hybrid Systems: Computation and Control, Milan, Italy, May 4–6, 2022, pp. 24:1–24:11. ACM (2022). https://doi.org/10.1145/3501710.3519535
16. Finkbeiner, B., Passing, N.: Dependency-based compositional synthesis. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 447–463. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_25
17. Finkel, A., Leroux, J.: How to compose presburger-accelerations: applications to broadcast protocols. In: Agrawal, M., Seth, A. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36206-1_14
18. Girija, P., Mareena, J., Fenny, J., Swapna, K., Kaewkhiaolueang, K.: Amazon robotic service (ARS) (2021)
19. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: When not losing is better than winning: Abstraction and refinement for the full mu-calculus. Inf. Comput. **205**(8), 1130–1148 (2007). https://doi.org/10.1016/j.ic.2006.10.009
20. Gueye, S.M.K., Delaval, G., Rutten, E., Diguet, J.P.: Discrete and logico-numerical control for dynamic partial reconfigurable FPGA-based embedded systems: a case study. In: 2018 IEEE Conference on Control Technology and Applications (CCTA), pp. 1480–1487. IEEE (2018)
21. Heim, P., Dimitrova, R.: Solving infinite-state games via acceleration. Proc. ACM Program. Lang. **8**(POPL) (2024). https://doi.org/10.1145/3632899

22. Heim, P., Nayak, S.P., Dimitrova, R., Schmuck, A.K.: Artifact of "Localized Attractor Computations for Infinite-State Games" (2024). https://doi.org/10.5281/zenodo.10939871
23. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 886–902. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_69
24. Katis, A., et al.: Validity-guided synthesis of reactive systems from assume-guarantee contracts. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 176–193. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_10
25. Kelasidi, E., Liljebäck, P., Pettersen, K.Y., Gravdahl, J.T.: Innovation in underwater robots: biologically inspired swimming snake robots. IEEE Robotics Autom. Mag. **23**(1), 44–62 (2016). https://doi.org/10.1109/MRA.2015.2506121
26. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using state and transition invariants. Formal Methods Syst. Des. **42**(3), 221–261 (2013). https://doi.org/10.1007/s10703-012-0176-y
27. Maderbacher, B., Bloem, R.: Reactive synthesis modulo theories using abstraction refinement. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022, pp. 315–324. IEEE (2022). https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_38
28. Markgraf, O., Hong, C.-D., Lin, A.W., Najib, M., Neider, D.: Parameterized synthesis with safety properties. In: Oliveira, B.C.S. (ed.) APLAS 2020. LNCS, vol. 12470, pp. 273–292. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64437-6_14
29. Masselot, M., Patil, S., Zhabelova, G., Vyatkin, V.: Towards a formal model of protection functions for power distribution networks. In: IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society, pp. 5302–5309. IEEE (2016)
30. Nayak, S.P., Egidio, L.N., Della Rossa, M., Schmuck, A.K., Jungers, R.M.: Context-triggered abstraction-based control design. IEEE Open J. Control Syst. **2**, 277–296 (2023). https://doi.org/10.1109/OJCSYS.2023.3305835
31. Neider, D., Topcu, U.: An automaton learning approach to solving safety games over infinite graphs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 204–221. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_12
32. Samuel, S., D'Souza, D., Komondoor, R.: Gensys: a scalable fixed-point engine for maximal controller synthesis over infinite state spaces. In: Spinellis, D., Gousios, G., Chechik, M., Penta, M.D. (eds.) ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23–28, 2021, pp. 1585–1589. ACM (2021). https://doi.org/10.1145/3468264.3473126
33. Samuel, S., D'Souza, D., Komondoor, R.: Symbolic fixpoint algorithms for logical LTL games. In: 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11–15, 2023, pp. 698–709. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00212
34. Schmuck, A.K., Heim, P., Dimitrova, R., Nayak, S.P.: Localized attractor computations for infinite-state games (full version) (2024). https://doi.org/10.48550/ARXIV.2405.09281

35. Sylla, A.N., Louvel, M., Rutten, E., Delaval, G.: Modular and hierarchical discrete control for applications and middleware deployment in IoT and smart buildings. In: 2018 IEEE Conference on Control Technology and Applications (CCTA), pp. 1472–1479. IEEE (2018)
36. Unno, H., Satake, Y., Terauchi, T., Koskinen, E.: Program verification via predicate constraint satisfiability modulo theories. CoRR abs/2007.03656 (2020). https://arxiv.org/abs/2007.03656
37. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. Int. J. Softw. Tools Technol. Transf. **15**(5–6), 413–431 (2013). https://doi.org/10.1007/S10009-012-0232-3
38. Walker, A., Ryzhyk, L.: Predicate abstraction for reactive synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21–24, 2014. pp. 219–226. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987617
39. Woeginger: Combinatorics problem c5 (2009)

Learning

Bisimulation Learning

Alessandro Abate[1], Mirco Giacobbe[2], and Yannik Schnitzer[1](✉)

[1] University of Oxford, Oxford, UK
{alessandro.abate,yannik.schnitzer}@cs.ox.ac.uk
[2] University of Birmingham, Birmingham, UK
m.giacobbe@bham.ac.uk

Abstract. We introduce a data-driven approach to computing finite bisimulations for state transition systems with very large, possibly infinite state space. Our novel technique computes stutter-insensitive bisimulations of deterministic systems, which we characterize as the problem of learning a state classifier together with a ranking function for each class. Our procedure learns a candidate state classifier and candidate ranking functions from a finite dataset of sample states; then, it checks whether these generalise to the entire state space using satisfiability modulo theory solving. Upon the affirmative answer, the procedure concludes that the classifier constitutes a valid stutter-insensitive bisimulation of the system. Upon a negative answer, the solver produces a counterexample state for which the classifier violates the claim, adds it to the dataset, and repeats learning and checking in a counterexample-guided inductive synthesis loop until a valid bisimulation is found. We demonstrate on a range of benchmarks from reactive verification and software model checking that our method yields faster verification results than alternative state-of-the-art tools in practice. Our method produces succinct abstractions that enable an effective verification of linear temporal logic without next operator, and are interpretable for system diagnostics.

Keywords: Data-driven verification · Stutter-insensitive bisimulation · Reactive verification · Software model checking · Abstraction

1 Introduction

Abstraction of state transition systems is the process for which a system under analysis—the concrete system—is reduced to another system—the abstract system—that is simpler to analyze and preserves certain temporal properties of the former [20,25,38,48]. It is a fundamental approach to state space reduction in the verification of finite-state systems and an essential element for the verification of infinite-state systems. Bisimulations are the abstractions that preserve linear and branching behaviour with respect to propositional observations, for which the model checking question for both linear- and branching-time logics have the same answer on the abstract and the concrete system [14,29].

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 161–183, 2024.
https://doi.org/10.1007/978-3-031-65633-0_8

Computing a bisimulation amounts to computing an equivalence relation on the state space that is stable with respect to a notion of state change, and preserves propositional observations. An equivalence relation defines a partition of the concrete state space and induces an abstract system where every abstract state corresponds to an equivalence class. The problem of computing bisimulations over an explicit representation of the state graph has been widely studied in the past [5,32], since Hopcroft's graph minimisation algorithm and the Paige-Tarjan algorithm for iterative partition refinement [31,46]. Partition refinement was improved with on-the-fly partition refinement of the reachable state space as well as parallelisation [21,35–37]. Yet, explicit-state algorithms fall short on systems with very large or infinite state space, for which one must resort to procedures that represent regions of state space symbolically [10].

Partition refinement relies on computing exact pre- and post-images through the transition function of the system [10,26]. This entails quantifier elimination, which is computationally costly. Counterexample-guided abstraction refinement (CEGAR) provides an approach to avoid pre- and post-image computation; it computes *simulations* of state transition systems incrementally, from infeasibility proofs of spurious counterexamples [17,30]. The resulting abstract system is tight enough to verify a specific property of interest, but cannot generally provide concrete counterexamples when a property is false and, for this purpose, methods based on CEGAR are usually coupled with bounded model checking [8]. Similarly, methods for temporal logic verification based on proof rules (i.e., certificates) provide sufficient conditions to verify whether a property holds but do not provide a counterexample when this is false [2,18,27,42,52]. By contrast, bisimulations provide a tight abstraction where abstract counterexamples correspond to concrete counterexamples and, as such, these are directly interpretable for system debugging and diagnostics.

We present a data-driven approach to computing finite bisimulations from sample states and transitions of the system, which skips partition refinement entirely. We adapt the notion of *well-founded bisimulations*, where the condition of stability of the equivalence relation with respect to stuttering is characterised as the existence of ranking functions over well-founded sets [43]. While originally introduced solely as a proof rule, we leverage well-founded bisimulations for the first time to directly compute finite bisimulations. We instantiate well-founded bisimulations with ranking functions that, for every state transition to a different state in the abstract system, map states to natural numbers that decrease strictly as the system stutters. This characterises *stutter-insensitive bisimulations* for deterministic transition systems and also applies to strong bisimulations, which is the special case of our method where ranking functions are constant.

Stutter-insensitive bisimulations are stable bisimulations with respect to observation change in the system, and is closed with respect to all state transitions between these changes. A system stutters when it changes concrete state without changing observation [33], and stutter-insensitive bisimulations abstract stuttering away. In contrast to strong bisimulations, stutter-insensitive bisimulations result in much more succinct abstractions, while being sufficiently strong to preserve the validity of any linear temporal logic specification without next operator. While our approach also applies to strong bisimulations, we generalise our method to stutter-insensitive bisimulations, because they more effectively yield finite abstractions on infinite-state systems in practice.

We build on the observation that a finite partition can be characterized as a state classifier mapping the (possibly infinite) state space into a finite set of classes. This reduces the problem of computing a stutter-insensitive bisimulation to training a classifier and a ranking function for each class [22,45,51]. For the partition classifier, we employ a binary decision tree (BDT) with parametric linear predicates at each decision node, and we associate each leaf node with a parametric linear ranking function. This structure forms our template.

Our approach is underpinned by a learner and a verifier interacting with each other, both using a satisfiability modulo theory (SMT) solver. The learner proposes a candidate bisimulation by computing parameters of the classifier and ranking function templates to satisfy conditions over sampled transitions. The verifier then checks if these conditions hold over the *entire state space*. If affirmed, the classifier induces a stutter-insensitive bisimulation. If not, the verifier provides a counterexample, a state where stutter-insensitive bisimulation conditions are violated. This counterexample is fed back to the learner, which updates the classifier and ranking functions. The process repeats in a counterexample-guided inductive synthesis (CEGIS) loop until the verifier confirms the bisimulation's validity [50]. If the template cannot fit the finite set of samples, for instance, due to an insufficient number of partitions, our procedure automatically enlarges the BDT with an additional layer and resumes the CEGIS loop.

We demonstrate the experimental efficacy of our approach on numerical programs and reactive software systems with integer state spaces. We consider benchmarks from reactive verification and software model checking, in particular discrete-time synchronisation protocols and conditional termination analysis problems. We benchmark the former set against the nuXmv model checker for reactive verification and the latter against the Ultimate and the CPAChecker tools for software verification [7,16,28]. The results are two-fold. For the reactive verification benchmarks, our approach has faster verification times than nuXmv on systems with long stuttering intervals. For the conditional termination benchmarks, our approach is able to generate exact preconditions for which the program terminates, unlike the baselines that return negative answers when the program does not terminate for at least one input. In summary, we demonstrate that, on these problems, our approach yields both faster and more informative results than the alternative state-of-the-art tools.

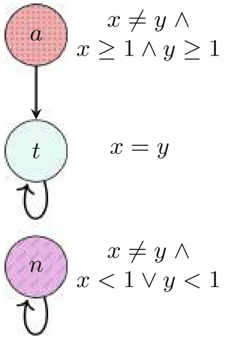
We summarise our contributions in the following three points: (1) we introduce the first data-driven approach to construct bisimulations, as an alternative approach to partition refinement; (2) we implement the theory of well-founded bisimulations which we synthesise in a CEGIS loop, as a means to compute stutter-insensitive bisimulations; (3) we demonstrate the efficacy of our novel approach on reactive verification and software model checking benchmarks. Our approach is fully automatic and requires no user input beyond the system itself. It produces succinct abstractions of infinite-state systems, which effectively enables their LTL (without next) verification using finite-state model checkers.

2 Illustrative Example

We motivate our procedure with an example from software model checking. Consider the code snippet in Fig. 1a. The program takes two arbitrary integers as input and subtracts the smaller from the larger until the two values coincide. We ask the question of whether the program terminates for every initial condition, which is not straightforward to answer for this example. Given two positive inputs, the program runs the Euclidean algorithm for the greatest common divisor and terminates once it is found. However, for any two unequal non-positive inputs, this implementation will never exit the loop and run forever.

```
int x = *, y = *;
while (x != y) {
    if (x > y)
        x = x - y;
    else
        y = y - x;
}
```

(a) Concrete program

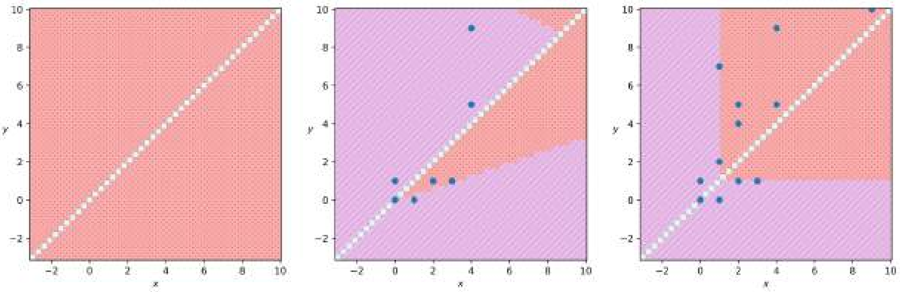


(b) Abstract program

Fig. 1. Learned stutter-insensitive bisimulation of the Euclidean algorithm.

Our procedure solves the termination problem by iteratively learning parameters for a given state classifier template, such that its induced partition of the state space satisfies the stutter-insensitive bisimulation conditions over a finite set of sample transitions of the program. We ensure this by simultaneously computing parameters for given ranking function templates, which, together with the partition induced by the classifier, satisfy the equivalent conditions of a well-founded bisimulation. We leverage an SMT solver to check for counterexamples, i.e., states that are not equivalent to other states with the same class assigned by the classifier. These counterexample states are passed back to the learning procedure to update the classifier and the ranking function parameters until the SMT solver cannot generate a counterexample anymore and, thus, certifies that the learned classifier generalises to the entire infinite state space and induces a valid stutter-insensitive bisimulation.

Figure 2 illustrates the iterative update of the classifier with respect to the sampled program behaviour, given an initial partitioning of the state space into the class of *terminated* states violating the loop condition `x != y` and the disjoint class of *not terminated* states. Upon termination the learned classifier correctly separates the states into those for which both variables are positive and which will eventually reach a terminated state after stuttering for a finite number of steps and the states that infinitely stutter in the class of not-terminated states.



(a) Initial partition (b) Intermediate partition (c) Final partition

Fig. 2. Iterative process of bisimulation learning. Starting from the initial label-preserving partition (a), our procedure generates counterexamples (blue dots) until it attains a valid stutter-insensitive bisimulation (c). (Color figure online)

In addition to the stutter-insensitive bisimulation, our procedure generates the corresponding abstract system by computing the behavior of the abstract states (i.e., the classes of the partition) alongside the classifier. Figure 1b shows the synthesized abstract system for the Euclidean algorithm, where each abstract state corresponds to an infinite subset of the concrete state space. The stutter-insensitive bisimulation ensures that the termination question has the same answer for all concrete states within the same class. A key advantage of our approach over methods providing a single counterexample is that it produces interpretable representations of the abstract system, aiding in system diagnostics. Specifically, our approach yields interpretable classifiers as binary decision trees. Figure 1b shows the abstract system and the automatically generated predicates defining the partition. Even for high-dimensional state spaces and complex partitions, this approach provides accessible means to interpret and diagnose the system for potential faults and undesired behavior [3,13].

3 Stutter-Insensitive Bisimulations of Deterministic Transition Systems

We introduce the fundamental concepts underpinning our approach.

Definition 1 (Transition Systems). *A transition system $\mathcal{M}$ consists of*

- *a state space S,*

- *an initial region $I \subseteq S$, and*
- *a non-blocking transition function $T\colon S \to (2^S \setminus \emptyset)$.*

We say that $\mathcal{M}$ is deterministic if $|T(s)| = 1$ for all $s \in S$. It is labelled when it additionally comprises

- *a set of atomic propositions AP (the observables), and*
- *a labelling (or observation) function $\langle\!\langle \cdot \rangle\!\rangle\colon S \to 2^{AP}$.*

A trajectory of $\mathcal{M}$ is any sequence of states $\tau = s_0, s_1, s_2, \ldots$ such that $s_{i+1} \in T(s_i)$ for all consecutive s_i, s_{i+1} in τ. We say that τ is initialised if $s_0 \in I$.

Definition 2 (Partitions). *A partition on $\mathcal{M}$ is an equivalence relation $\simeq \subseteq S \times S$ on S, which defines the quotient space $S/_{\simeq}$ (i.e., the set of equivalence classes of $\simeq$) of pairwise-disjoint regions of S whose union is S.*

Since we are interested in a notion of state equivalence insensitive to behaviour that does not change the observation of a state, the concept of divergence will be essential to distinguish between states that progress while not changing observation and those that do not progress at all [4,53].

Definition 3 (Divergence Sensitivity). *Let $\simeq$ be a partition on $\mathcal{M}$. A state $s \in S$ is $\simeq$-divergent if there exists an infinite trajectory $s_0, s_1, \ldots$ such that $s_0 = s$ and $s_i \simeq s$ for all $i > 0$. Partition $\simeq$ is divergence-sensitive when $s \simeq t$ and s is $\simeq$-divergent implies that t is $\simeq$-divergent.*

A partition of the state space induces a reduced transition system — the corresponding abstract system or *quotient*.

Definition 4 (Quotient). *The quotient of $\mathcal{M}$ under the partition $\simeq$ is the transition system $\mathcal{M}/_{\simeq}$ with*

- *state space $S/_{\simeq}$,*
- *initial region $I/_{\simeq}$ where $R \in I/_{\simeq}$ iff $R \cap I \neq \emptyset$, and*
- *transition function $T/_{\simeq}$ where*
  1. *$R \neq Q \in T/_{\simeq}(R)$ iff $T(s) \in Q$ for some $s \in R$,*
  2. *$R \in T/_{\simeq}(R)$ iff some $s \in R$ is $\simeq$-divergent.*

The quotient is the aggregation of equivalent states and their behaviours. The specifications preserved by the quotient, i.e., the statements that carry over from the abstract to the concrete system, depend on the properties of the underlying partition [46]. The most important property to preserve sensible specifications is that equivalent states must have equal observations.

Definition 5 (Label-preserving Partitions). *A partition $\simeq$ on a labelled transition system is label-preserving when $s \simeq t$ implies $\langle\!\langle s \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle$. The quotient $\mathcal{M}/_{\simeq}$ of a labelled transition system $\mathcal{M}$ under a label-preserving partition $\simeq$ is labelled with the extended labelling function $\langle\!\langle \cdot \rangle\!\rangle\colon S \cup S/_{\simeq} \to 2^{AP}$ where, for every region $R \in S/_{\simeq}$, $\langle\!\langle R \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle$ for any representative $s \in R$.*

A standard notion of state equivalence on labelled transition systems is *bisimilarity* [39]. Bisimilarity preserves both linear- and branching-time behaviour by co-inductively requiring that every pair of related states can match each others' transitions with equivalent transitions. However, this stability with respect to stepwise behaviour often results in large quotients, thus limiting its suitability to facilitate reasoning over the system [46]. Therefore, we focus on *stutter-insensitive bisimulations* [14]. By abstracting from stepwise behaviour that does not change the observation of a state, stutter-insensitive bisimulations yield smaller quotients while preserving important specifications, as we will see in the following section.

Definition 6 (Stutter-insensitive Bisimulation). *A label-preserving partition $\simeq$ is a stutter-insensitive bisimulation if, for all states $s, s', t \in S$ such that $s \simeq t$ and $s \not\simeq s' \in T(s)$, there exists a finite trajectory $t_0, t_1, \ldots, t_k$ such that $t_0 = t$, $t_i \simeq s$ for all $i = 1, \ldots k-1$, and $t_k = t'$ for some $t' \simeq s'$.*

Figure 3 illustrates the stability condition of stutter-insensitive bisimulations. This condition requires that for related states, transitions to unrelated states can be matched by finite trajectories that pass through the same equivalence class.

$$\begin{tikzcd}
s \arrow[r] \arrow[d, phantom, "\rotatebox{90}{$\simeq$}"] & s' & \text{with } s \not\simeq s' & s \arrow[d, phantom, "\rotatebox{90}{$\simeq$}"] & s \arrow[d, phantom, "\rotatebox{90}{$\simeq$}"] & \ldots & s \arrow[r] \arrow[d, phantom, "\rotatebox{90}{$\simeq$}"] & s' \arrow[d, phantom, "\rotatebox{90}{$\simeq$}"] \\
t & & \text{can be completed to} & t \arrow[r] & t_1 \arrow[r] & \ldots \arrow[r] & t_{k-1} \arrow[r] & t'
\end{tikzcd}$$

Fig. 3. Trajectory-based representation of the stutter-insensitive stability condition.

Lemma 1. *Every stutter-insensitive bisimulation on any deterministic labelled transition system admits a deterministic quotient.*

Proof. Let $\mathcal{M}$ be a deterministic transition system and $\simeq$ be a stutter-insensitive bisimulation on $\mathcal{M}$. Assume $M/_{\simeq}$ is nondeterministic, this implies that there exists pairwise distinct $R, Q, V \in S/_{\simeq}$ such that $\{Q, V\} \subseteq T/_{\simeq}(R)$. It follows that there exist $s, t \in R$ with $T(s) \in Q$ and $T(t) \in V$. Since $s, t \in R$ it holds that $s \simeq t$ and as $\mathcal{M}$ is deterministic and $Q \neq V$, $\simeq$ cannot satisfy Def. 6. □

3.1 Model Checking

We introduce Linear Temporal Logic without *next*-operator (LTL$_{\setminus\bigcirc}$) as a formal specification language for the temporal behaviour of a system and its states [4, 49]. LTL$_{\setminus\bigcirc}$ formulas are constructed according to the following grammar:

$$\varphi ::= \text{true} \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi\, U\, \varphi$$

The model checking problem for $\mathrm{LTL}_{\setminus\bigcirc}$ is to decide whether transition system $\mathcal{M}$ satisfies a given $\mathrm{LTL}_{\setminus\bigcirc}$ formula φ, where the satisfaction relation $\models$ for trajectories of $\mathcal{M}$ is defined as

$$\begin{aligned}
&\tau, i \models \text{true} && \\
&\tau, i \models p && \text{iff } p \in \langle\!\langle s_i \rangle\!\rangle \text{ where } \tau = s_0, s_1, s_2, \ldots \\
&\tau, i \models \varphi_1 \wedge \varphi_2 && \text{iff } \tau, i \models \varphi_1 \text{ and } \tau, i \models \varphi_2 \\
&\tau, i \models \neg\varphi && \text{iff } \tau, i \not\models \varphi \\
&\tau, i \models \varphi_1 \, U \, \varphi_2 && \text{iff for some finite } k \geq i,\ \tau, k \models \varphi_2 \text{ and} \\
& && \qquad \tau, j \models \varphi_1 \text{ for all } j = i, \ldots, k-1
\end{aligned}$$

and is lifted to the entire transition system by requiring that every initialised trajectory satisfies φ:

$$\mathcal{M} \models \varphi \text{ iff } \tau, 0 \models \varphi \text{ for all infinite initialised trajectories } \tau \text{ of } \mathcal{M}.$$

We also introduce the derived operators *"eventually"* $\Diamond$ and *"globally"* $\Box$. The formula $\Diamond\varphi := \text{true}\, U \,\varphi$ states that φ must be true in some state on the trajectory. The formula $\Box\varphi := \neg(\Diamond\neg\varphi)$ requires that φ holds true in all states of the trajectory. We do not include the *"next"* operator $\bigcirc$ from full LTL since we are interested in stutter-insensitive bisimulations, which do not preserve a system's stepwise behavior as expressed by the *next*-operator. It is a well-known fact that divergence-sensitive stutter-insensitive bisimulations preserve specifications expressable in $\mathrm{LTL}_{\setminus\bigcirc}$ [4]. Divergence-sensitivity is crucial to properly treat stutter-trajectories, i.e., trajectories that forever stutter inside the same equivalence class [44]. However, for deterministic transition systems, each state has only one outgoing trajectory that either eventually leaves its equivalence class or stutters indefinitely. Therefore, any stutter-insensitive bisimulation on a deterministic system must be divergence-sensitive, as stated in Lemma 2.

Lemma 2. *Every stutter-insensitive bisimulation on any deterministic labelled transition system is divergence-sensitive.*

Proof. Let $\mathcal{M}$ be a deterministic transition system and $\simeq$ be a stutter-insensitive bisimulation on $\mathcal{M}$. Let $s \simeq t$ and assume s $\simeq$-divergent but t not $\simeq$-divergent. As t not $\simeq$-divergent, there exists a finite trajectory $\tau = t, t_1, \ldots, t_n, t'$ with $t \simeq t_i, \forall i \leq n$ and $t \not\simeq t'$, for some $n \geq 0$. This implies that there exists a state $u \simeq s$ with $s \not\simeq t' \in T(u)$. However, since $\mathcal{M}$ is deterministic and s is $\simeq$-divergent, the unique trajectory $\tau = s, s_1, \ldots$ initalised in s satisfies $s \simeq s_i, \forall i \geq 0$, which is a contradiction. □

Theorem 1. *Let $\mathcal{M}$ be a deterministic labelled transition system. If $\simeq$ is a stutter-insensitive bisimulation on $\mathcal{M}$, then $\mathcal{M} \models \varphi$ if and only if $\mathcal{M}/_{\simeq} \models \varphi$ for any $LTL_{\setminus\bigcirc}$ formula φ.*

Proof. Any divergence-sensitive stutter-insensitive bisimulation $\simeq$ on any (possibly non-deterministic) transition system $\mathcal{M}$ implies, for every $\mathrm{LTL}_{\setminus\bigcirc}$ formula

φ, the model checking problems $\mathcal{M} \models \varphi$ and $\mathcal{M}/_{\simeq} \models \varphi$ have the same answer. Lemma 2 establishes that, since $\mathcal{M}$ is deterministic and $\simeq$ is stutter-insensitive on $\mathcal{M}$, then $\simeq$ is also divergence-sensisitve. Therefore, the statement follows. □

Remark 1. Theorem 1 in general does not hold for nondeterministic transition systems, as can be seen by the counterexample in Fig. 4.

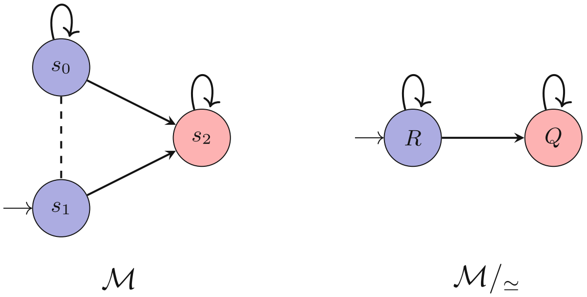


Fig. 4. A stutter-insensitive but not divergence-sensitive bisimulation $\simeq$ is indicated by the dashed lines. It holds that $\mathcal{M} \models \Diamond(\text{red})$ but $\mathcal{M}/_{\simeq} \not\models \Diamond(\text{red})$.

Remark 2. It may seem counterintuitive to the reader to relate stutter-insensitive bisimulation and $\text{LTL}_{\setminus\bigcirc}$-equivalence, as it is usually associated with the more expressive $\text{CTL}^*_{\setminus\bigcirc}$ [4]. However, recall that we focus on deterministic transition systems for which both logics coincide in expressivity.

4 Counterexample-Guided Bisimulation Learning

This section introduces our main contributions. We present our adaption of Namjoshi's well-founded bisimulations to deterministic transition systems [43] and describe a counterexample guided learning algorithm for simultaneous computation of a stutter-insensitive bisimulation and its corresponding quotient from a finite sampling of the state space. Well-founded bisimulation implements the stability conditions of stutter-insensitive bisimulation (see Def. 6) in the form of ranking functions that map states to a well-founded set, ensuring that a finite trajectory matches every transition of equivalent states. We present an adaption to deterministic systems that only requires the ranking functions to map single states of certain classes to a well-founded set and show that this characterizes stutter-insensitive bisimulation. Furthermore, we show that the problem of computing a stutter-insensitive bisimulation can be rephrased to finding a classifier on states and ranking functions for the corresponding classes.

Theorem 2. *Let $\mathcal{M}$ be a deterministic labelled transition system with state space S and transition function T. Let $\simeq$ be a label-preserving partition on $\mathcal{M}$.*

Suppose that for every region $R \in S/_{\simeq}$ there exists a function $h_R\colon S \to \mathbb{N}$ such that, for every $R \neq Q \in T/_{\simeq}(R)$, the following condition holds:

$$\forall s \in R\colon T(s) \in Q \vee [T(s) \in R \wedge h_R(s) > h_R(T(s))]. \tag{1}$$

Then, $\simeq$ is a stutter-insensistive bisimulation on $\mathcal{M}$.

Proof. Let $s \simeq t$ such that $s \not\simeq s' \in T(s)$. This implies that $\exists R \neq Q \in S/_{\simeq}\colon s, t \in R$ and $s' \in Q$, and $Q \in T/_{\simeq}(R)$. Assume there exists no finite trajectory $\tau = t, t_1, \ldots, t_n, t', n \geq 0$ with $t \simeq t_i, \forall i \leq n$ and $s' \simeq t'$. As $\mathcal{M}$ is deterministic, we only need to distinguish the two cases:

- The infinite trajectory $\tau = t, t_1, t_2, \ldots$ stutters infinitely in R, i.e., $t_i \in R, \forall i \geq 1$. This contradicts the ranking property 1.
- There exists a finite trajectory $\tau = t, t_1, \ldots, t_n, t', n \geq 0$ with $t' \in V \neq Q$ and $t_i \in R, \forall i \leq n$. However $t_n \in R$ and $t' \in V \neq Q$ is a contradiction to the ranking property 1 only allowing for an exit to Q. □

The ranking functions h_R in Theorem 2 ensure that if class R has an outgoing transition to Q, for all states in R either (1) their successor is in Q or (2) their successor is in R and h_R decreases when transitioning. As the value of h_R is bounded from below and must strictly decrease along any trajectory, no trajectory can stutter in R indefinitely and must eventually enter Q (see Fig. 5).

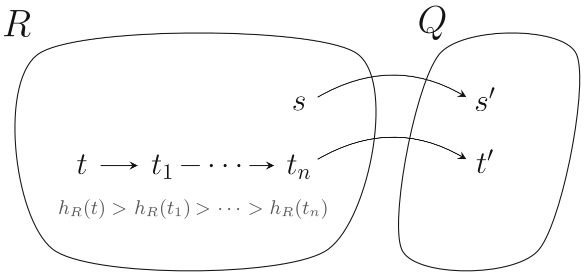

R
Q
s
s'
$t \longrightarrow t_1 - \cdots \rightarrow t_n$
t'
$h_R(t) > h_R(t_1) > \cdots > h_R(t_n)$

Fig. 5. Intuitive representation of Theorem 2. Since a state $s \in R$ has a successor in Q, the value of h_R must strictly decrease along any trajectory through R. Therefore, all states in R eventually transition to Q after possible stuttering.

Remark 3. For deterministic systems, strong bisimulations form a special case of stutter-insensitive bisimulations, not allowing for any stuttering. In our formulation, they only admit constant ranking functions h_R. Strong bisimulations preserve a system's stepwise behavior, hence, full LTL including the *next*-operator [4]. However, they may induce much larger quotients, less suitable for verifying large systems with long stuttering intervals. Furthermore, there exist infinite state systems that do not admit a finite strong bisimulation, but do admit a finite stutter-insensitive bisimulation quotient.

We aim to phrase the problem of finding a suitable partition and ranking functions that satisfy the conditions in Theorem 2 as a learning problem. For that, we introduce the notion of state classifiers.

Definition 7 (State Classifier). *A state classifier on a labelled transition system with state space S is any function $f\colon S \to C$ that maps states to a finite set of classes C. It is label-preserving if $f(s) = f(t)$ implies $\langle\!\langle s \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle$.*

We can now state Theorem 2 for a state classifier f and give sufficient conditions for f to induce a valid stutter-insensitive bisimulation.

Theorem 3. *Let $\mathcal{M}$ be a deterministic labelled transition system with state space S and transition function T. Suppose that there exists a label-preserving state classifier $f\colon S \to C$, a function $g\colon C \to C$ and functions $h_c\colon S \to \mathbb{N}$ for each $c \in C$ such that, for every $c \neq d \in C$ and $s \in S$, the following two conditions hold:*

$$f(s) = c \wedge g(c) = d \implies f(T(s)) = d \vee [f(T(s)) = c \wedge h_c(s) > h_c(T(s))], \quad (2)$$

$$f(s) = c \wedge f(T(s)) = d \implies g(c) = d. \quad (3)$$

Then, $\simeq_f$ defined as $\simeq_f = \{(s,t) \mid f(s) = f(t)\}$ is a stutter-insensitive bisimulation on $\mathcal{M}$ and $T_{\simeq_f}(f^{-1}[c]) = \{f^{-1}[g(c)]\}$.

Proof. We first show that $\simeq_f$ is a stutter-insensitive bisimulation on $\mathcal{M}$. Since f is label-preserving, $\simeq_f$ is label-preserving by definition. The quotient space is the set of non-empty pre-images of the classes C under f, i.e., $S/_{\simeq_f} = \{f^{-1}[c] \mid c \in C\} \setminus \emptyset$. By definition of $T/_{\simeq_f}$ (see Def. 4) it holds that $f^{-1}[c] \neq f^{-1}[d] \in T/_{\simeq_f}(f^{-1}[c])$ implies that there exists an $s \in f^{-1}[c]$ with $T(s) \in f^{-1}[d]$. With Condition 3 this implies that $g(c) = d$. The claim follows by Condition 2 and Theorem 2. We now show that $T_{\simeq_f}(f^{-1}[c]) = \{f^{-1}[g(c)]\}$. Since $\simeq_f$ is a stutter-insensitive bisimulation Lemma 1 implies that $T/_{\simeq}(f^{-1}[c])$ can only be a singleton for any $c \in C$. We distinguish the two cases:

- $f^{-1}[c] \neq f^{-1}[d] \in T/_{\simeq_f}(f^{-1}[c])$, then $f^{-1}[c] \neq f^{-1}[d]$ implies that $c \neq d$. By Def. 4 there must exist a $s \in f^{-1}[c]$ with $T(s) \in f^{-1}[d]$, which by Condition 3 implies that $g(c) = d$.
- $f^{-1}[c] \in T/_{\simeq_f}(f^{-1}[c])$, then some state in $s \in f^{-1}[c]$ must be $\simeq_f$-divergent by Def. 4. The only possibility for g to be a total function and not to violate Condition 2 is $g(c) = c$, as $g(c) = d \neq c$ would contradict the $\simeq_f$-divergency of s due to Condition 2. □

Remark 4. Note that Theorem 3 requires g to be well-defined, i.e., represent a deterministic transition function. However, this is not a restriction as per Lemma 1 any stutter-insensitive bisimulation on a deterministic transition system has a deterministic quotient. The fact that g has to be total additionally requires it to correctly account for the self-loops of the divergent classes.

In Theorem 3 function g takes on the role of the deterministic transition function of the quotient induced by f. Thus, f and g together provide a complete description of a stutter-insensitive bisimulation quotient of the underlying transition system. In the following, we introduce our counterexample-guided learning approach for generating appropriate functions on a given transition system.

4.1 Learner-Verifier Framework for Bisimulation Learning

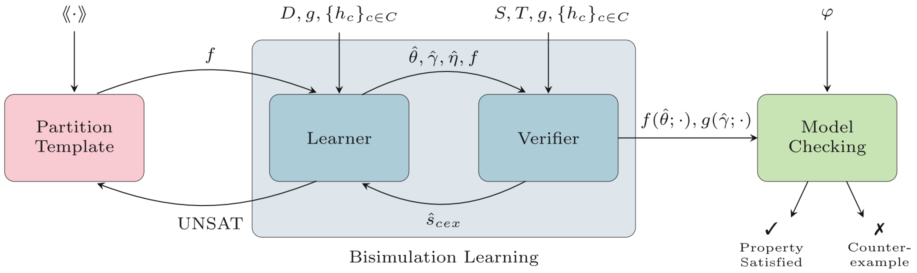


Fig. 6. Architeture of our learner-verifier framework for bisimulation learning.

Our procedure involves two communicating components, the *learner* and the *verifier*, implementing a CEGIS loop. The learner proposes candidate functions that satisfy the stutter-insensitive bisimulation conditions over a finite set of sample states. The verifier checks if a counterexample state exists for which the functions proposed by the learner violate the conditions, which are then passed back to the learner to update the functions (see Fig. 6).

Learner. We consider parametric function templates whose maps solely depend on the provided parameters. Therefore, the learner seeks suitable parameters for a label-preserving state classifier template $f\colon \Theta \times S \to C$, a transition function template $g\colon \Gamma \times C \to C$ and ranking function templates $h_c\colon H \times S \to \mathbb{N}$ for each $c \in C$, i.e., attempts to solve:

$$\exists \theta \in \Theta, \gamma \in \Gamma, \eta \in H\colon \bigwedge_{\hat{s}\in D} \Phi_1(\theta, \gamma, \eta; \hat{s}, T(\hat{s})) \wedge \Phi_2(\theta, \gamma, \eta; \hat{s}, T(\hat{s})), \tag{4}$$

where Φ_1 encodes Condition 2 of Theorem 3:

$$\begin{aligned}\Phi_1(\theta, \gamma, \eta; s, s') = \bigwedge_{c\neq d\in C} & f(\theta; s) = c \wedge g(\gamma; c) = d \implies \\ & f(\theta; s') = d \vee [f(\theta; s') = c \wedge h_c(\eta; s) > h_c(\eta; s')],\end{aligned} \tag{5}$$

and Φ_2 represents Condition 3:

$$\Phi_2(\theta, \gamma, \eta; s, s') = \bigwedge_{c \neq d \in C} f(\theta; s) = c \wedge f(\theta; s') = d \implies g(\gamma; c) = d, \tag{6}$$

for a deterministic transition system $\mathcal{M}$ and finite set of sample states $D \subseteq S$. In our instantiation, we use an SMT-solver to seek a satisfying assignment for the parameters θ, γ, and η in the quantifier-free inner formula of 4.

Verifier. The verifier checks the functions induced by the proposed candidate parameters $\hat{\theta}$, $\hat{\gamma}$ and $\hat{\eta}$ for generalisation to the entire state space, i.e., attempts to solve:

$$\exists s \in S\colon \neg\Phi_1(\hat{\theta}, \hat{\gamma}, \hat{\eta}; s, T(s)) \vee \neg\Phi_2(\hat{\theta}, \hat{\gamma}, \hat{\eta}; s, T(s)). \tag{7}$$

Similar to the learner, the verifier is an SMT-solver to which we hand the quantifier-free inner formula of 7. A found satisfying assignment for a counterexample state s is returned to the learner. If the formula is unsatisfiable, the procedure terminates and has successfully synthesised a valid stutter-insensitive bisimulation and its corresponding quotient.

4.2 Binary Decision Tree Partition Templates

From here on, our focus is on transition systems with discrete state spaces $S \subseteq \mathbb{Z}^n$ defined over the integers. We present the parametric function templates used in our instantiation of the framework. For the state classifier templates, we employ binary decision trees with real-valued decision functions in the inner nodes. We construct binary decision trees preserving the system's labelling function and automatically enlarge them when the template is not expressive enough to fit the finite set of sample states (see Fig. 6).

Definition 8 (Binary Decision Tree Templates). *The set of binary decision tree templates $\mathbb{T}$ over a finite set of classes C and parameters Θ consists of trees t, where t is either*

- *a leaf node* LEAF(c) *with $c \in C$, or*
- *a decision node* NODE(p, t_1, t_2)*, where $t_1, t_2 \in \mathbb{T}$ are the left and right subtrees, and $p\colon \Theta \times S \to \mathbb{R}$ is a parametrised real-valued function of the states.*

A parametric tree template $t \in \mathbb{T}$ over classes C and parameters Θ defines the parametric state classifier $f_t\colon \Theta \times S \to C$ given as

$$f_t(\theta, s) = \begin{cases} c & \textit{if } t = \textsc{leaf}(c) \\ f_{t_1}(\theta, s) & \textit{if } t = \textsc{node}(p, t_1, t_2) \textit{ and } p(\theta; s) \geq 0 \\ f_{t_2}(\theta, s) & \textit{if } t = \textsc{node}(p, t_1, t_2) \textit{ and } p(\theta; s) < 0. \end{cases}$$

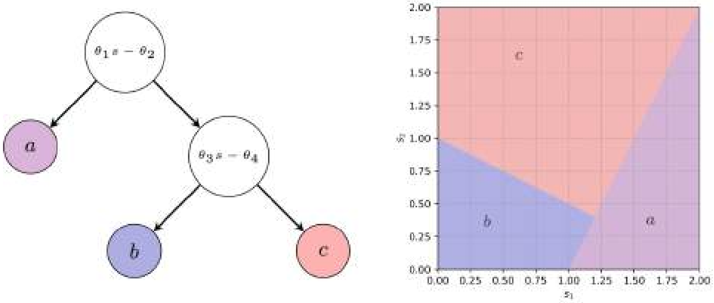


Fig. 7. Binary decision tree with parameters $\theta_1 = \begin{bmatrix}-2 & 1\end{bmatrix}, \theta_2 = -2, \theta_3 = \begin{bmatrix}\frac{1}{2} & 1\end{bmatrix}$, and $\theta_4 = 1$ for the parametrised functions, and its corresponding state classifier.

Binary decision trees appeal as state classifier templates as they are interpretable, expressive, and simple to translate into quantifier-free expressions to instantiate the formulas of the learner and the verifier, see Fig. 7. The parametric transition function template $g\colon \Gamma \times C \to C$ is simply a vector or list over classes C, indexed by C, and for the parametric ranking function templates $h_c\colon H \times S \to \mathbb{N}$ we consider linear functions of the form $h_c(\eta, s) = \eta_1 \cdot s + \eta_2$.

An important requirement for our procedure is that the synthesised state classifier is label-preserving. We guarantee this by constructing label-preserving templates which have this property by design for any parameter instantiation. We assume that any atomic proposition $a \in AP$ is associated with a real-valued function $p_a\colon S \to \mathbb{R}$, such that

$$\langle\!\langle s \rangle\!\rangle = \{a \in AP \mid p_a(s) \geq 0\}. \tag{8}$$

We construct label-preserving templates by encoding the functions corresponding to the atomic propositions into the top nodes of the binary decision tree, i.e., fixing the functions for the top nodes to represent the observation partition (see for example [47] for a canonical construction). This resembles the prerequisite of classical partition-refinement algorithms, which are initialised from label-preserving partitions. Fixing the labelling with the top nodes ensures that any instantiated state classifier is label-preserving, and the subsequent nodes further refine the label-preserving partition. Figure 8 shows the top nodes with fixed functions for a label-preserving binary tree template for the Euclidean algorithm from Figs. 1 and 2.

When the binary decision tree used is too small to fit the minimum number of regions in a quotient or if it requires more decision boundaries, the learner will return UNSAT as it cannot fit a partition with the given template on the finite set of samples. In such cases, our procedure automatically increases the size of the employed BDT template and resumes bisimulation learning with the more expressive template (see Fig. 6). Our template construction is entirely automatic and requires no user input other than the labeling function. Bisimulation learning starts from a small, automatically generated BDT template encoding the labeling

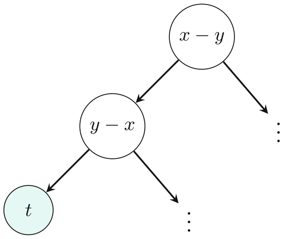


Fig. 8. Label-preserving binary decision tree template for the Euclidian algorithm from Fig. 1. The functions are assigned with respect to the loop condition $x \neq y$. The *terminated* proposition, i.e., class t, is assigned to the states satisfying $x \leq y$ and $y \leq x$. Any state satisfying either of $x > y$ or $y > x$ is labelled with *not-terminated.*

function and successively enlarges the partition template as required. We enlarge the partition template by adding an additional layer to the BDT, doubling the number of available partitions.

5 Experimental Evaluation

We implemented our approach in a software prototype and evaluated bisimulation learning on a range of benchmark systems representing two common classes of problems: verification of reactive systems and software model checking. We compare our procedure to established state-of-the-art tools: the nuXmv model checker [12,15,16] for the reactive system problems and the Ultimate [28] and CPAChecker [7] tools for software model checking benchmarks. All benchmarks, our implementation, and the used templates are publicly available. We employ the Z3 SMT-solver [41] in both learner and verifier, and the nuXmv model checker to verify the properties of interest on the obtained abstractions.

5.1 Discrete-Time Clock Synchronization

Setup. For reactive systems, we consider two distributed synchronization protocols for potentially drifted discrete clocks of distributed agents. First, the TTEthernet protocol, where all agents send their current clock value to a central synchronization master. This synchronization master computes the median clock value and sends it back to the agents, which use the received value to update their internal clocks [9]. Second, we consider an interactive convergence algorithm where the agents directly exchange clock values and compute the average to update their internal clocks while excluding received values that differ more than a given threshold from their own [34]. We check the systems for two kinds of properties: a safety invariant, which specifies that all clock valuations remain within a predefined maximum distance (G(safe)); and whether all clocks infinitely often synchronize on the same valuation (GF(sync)). Note that while the baseline procedures verify the systems regarding the given specification, our abstraction procedure is agnostic to the specification, i.e., the obtained abstraction can be used to verify arbitrary $\mathrm{LTL}_{\setminus\bigcirc}$ formulas over the atomic propositions.

To render verification with BDDs feasible, we leverage that all clock valuations remain within an interval that depends on the time discretization, as they are either continuously reset or enter a dead-lock state when violating the safety requirement. We explicitly pass this invariant to the BDD toolchain in the form of finite variable domains to allow for the construction of BDDs, whereas IC3 and our abstraction approach operate over unbounded integer variables, which would not be possible for BDDs. For both benchmarks, we consider a safe variant, where the agents use the received values to update their internal clock correctly, and an unsafe version, where they stick with their internal values and drift further. In all instances, we assess multiple instances of time discretization, i.e., the sampling frequency (number of discrete time steps) for a unit second.

Results. Table 1 presents the runtime results. Our approach depends on generating candidate parameters and counterexamples through an SMT solver. These can vary across runs of the procedure, even under identical initial conditions (i.e.,

Table 1. Results for reactive clock-synchronization benchmarks. All times are measured in seconds with "oot" denoting a timeout at 500 [sec]. The benchmark names include the used parameters, e.g., "tte-sf-1k" describes a safe TTEthernet instance with a time discretization of 1000 steps per second.

Benchmark	No. States	nuXmv (IC3)		nuXmv (BDDs)		Bisimulation Learning
		G(safe)	GF(sync)	G(safe)	GF(sync)	
tte-sf-10	250	0.1	0.7	0.1	0.1	0.3±0.4
tte-sf-100	2500	13.3	423	0.3	0.3	0.7±0.6
tte-sf-1k	2.5×10^6	oot	oot	1.8	31	1.2±0.4
tte-sf-2k	1×10^7	oot	oot	6.4	162	1.5±0.1
tte-sf-5k	6.25×10^7	oot	oot	34	417	1.6±0.4
tte-sf-10k	2.5×10^8	oot	oot	193	oot	1.6±0.2
tte-usf-10	250	0.1	0.5	0.1	0.1	0.2±0.1
tte-usf-100	2500	15.2	9.2	0.1	0.2	0.2±0.1
tte-usf-1k	2.5×10^6	421	405	1.5	10	0.3±0.1
tte-usf-2k	1×10^7	oot	oot	7.1	41	0.4±0.2
tte-usf-5k	6.25×10^7	oot	oot	32	242	0.5±0.5
tte-usf-10k	2.5×10^8	oot	oot	130	oot	0.6±0.5
con-sf-10	250	0.6	0.5	0.1	0.1	0.4±0.2
con-sf-100	2500	22	oot	0.2	0.3	0.4±0.3
con-sf-1k	2.5×10^6	oot	oot	4.6	45	0.7±0.4
con-sf-2k	1×10^7	oot	oot	17.6	210	0.7±0.2
con-sf-5k	6.25×10^7	oot	oot	95	oot	0.8±0.5
con-sf-10k	2.5×10^8	oot	oot	oot	oot	1.4±1.4
con-usf-10	250	0.2	0.3	0.1	0.1	0.2±0.2
con-usf-100	2500	31	33	0.3	0.2	0.2±0.1
con-usf-1k	2.5×10^6	oot	oot	2.8	24	0.3±0.2
con-usf-2k	1×10^7	oot	oot	8.2	154	0.4±0.3
con-usf-5k	6.25×10^7	oot	oot	36	oot	0.7±0.4
con-usf-10k	2.5×10^8	oot	oot	156	oot	0.8±0.3

provided initial samples). Since this can impact the convergence speed and overall runtime of the algorithm, we conduct each experiment 10 times and report the average runtimes and standard deviations. We only report a single outcome for our approach, as we check both properties of interest on the same abstraction and the differences in verification time are negligible on the obtained small abstractions.

Discussion. The results show that the learned bisimulations effectively and efficiently verify the specifications. Especially with decreased time discretization and, therefore, increased size of the state space, our approach clearly shows an advantageous performance. While a larger reachable state space renders verification harder for all considered approaches, a decreased time discretization is especially difficult for the IC3 toolchain based on bounded model checking, as it increases the completeness threshold and the depth of counterexamples. Since bisimulation learning generalizes from a finite set of samples, it is less susceptible to larger state spaces if the corresponding abstractions remain small. Generally, there is a trade-off between the number of provided initial samples and the number of CEGIS iterations needed to refine the initial partition. Although it may require more time to fit an initial partition on a more extensive set of uniform initial samples, it can reduce the counterexamples needed to obtain a valid stutter-insensitive bisimulation. As our instantiation leverages potentially expensive SMT solving in both the learner and the verifier, which scales in the number of considered samples, we aim at being sample-efficient: therefore, we decided to consider a fixed, small amount of uniform initial samples for all benchmarks of different sizes and leverage the generation of informative counterexamples in potentially more, but faster CEGIS cycles.

5.2 Conditional Termination

Setup. For software model checking, we consider a range of benchmarks from program termination analysis, including a selection of programs sourced from the termination category of the SV-COMP competition for software verification [6]. As is the case for the Euclidean algorithm in Fig. 1, these programs on unbounded integer variables may terminate for some inputs and enter a non-terminating loop for others. The two baseline tools determine whether a program terminates for all possible inputs. Our procedure instead goes a step further by providing an exact partition of the variable valuations, separating the inputs for which the algorithm eventually terminates from those for which it does not. As a distinguishing feature of the baseline benchmarks, we split each program into two versions: one that only allows for inputs for which the program terminates (denoted as "term") and another that includes potentially non-terminating inputs (denoted as "¬term“).

Results. Table 2 presents the runtime results. Note that we only report the analysis time for the baselines, without additional time spent on parsing or preprocessing the programs.

Table 2. Results for software termination benchmarks. All times are measured in seconds, with "oot" denoting a timeout at 500 [sec]. A non-conclusive analysis outcome is denoted by "n/c" and "-" indicates that there is no such special case of the benchmark.

Benchmark	nuXmv (IC3)		CPAChecker		Ultimate		Bisimulation Learning
	term	¬term	term	¬term	term	¬term	
term-loop-1	oot	oot	0.6	1.8	1.5	2.9	$0.2_{\pm 0.1}$
term-loop-2	oot	oot	0.6	0.3	0.7	0.2	$0.4_{\pm 0.3}$
audio-compr	3.1	< 0.1	n/c	n/c	0.9	0.6	$0.3_{\pm 0.3}$
euclid	oot	oot	n/c	0.3	1.6	0.4	$0.6_{\pm 0.2}$
greater	oot	< 0.1	0.6	0.3	1.1	0.3	$0.4_{\pm 0.2}$
smaller	oot	< 0.1	0.6	0.3	1.6	0.3	$0.2_{\pm 0.1}$
conic	oot	< 0.1	0.7	0.4	n/c	0.4	$4.2_{\pm 7.3}$
disjunction	oot	-	34	-	1.7	-	$0.3_{\pm 0.3}$
parallel	n/c	-	0.9	-	9.1	-	$0.3_{\pm 0.3}$
quadratic	0.2	-	n/c	-	n/c	-	$0.3_{\pm 0.1}$
cubic	0.2	< 0.1	n/c	n/c	n/c	0.2	$0.4_{\pm 0.2}$
nlr-cond	< 0.1	< 0.1	n/c	n/c	n/c	0.2	$0.2_{\pm 0.2}$

Discussion. The results show that bisimulation learning, while computing more informative results and solving the more complex problem of conditional termination [11,19], operates in runtimes comparable to the state-of-the-art tools for the considered benchmarks. Especially for programs that involve disjunctions over variable valuations (cf. the *disjunction* and *parallel* benchmarks), our procedure is able to prove termination more efficiently. Additionally, our approach can handle non-linear operations if the employed templates are sufficiently expressive for the corresponding partition and ranking functions. As a further surplus, it yields interpretable binary decision trees representing the derived stutter-insensitive bisimulation. These trees are valuable for system diagnostics and fault analysis, providing further insight beyond single counterexamples. Once again, this experimental evaluation shows that, while not being complete in theory, our algorithm terminates in all of the considered experiments.

Limitations. Bisimulation learning addresses a generally undecidable problem: finding finite bisimulations for systems with potentially infinite state spaces [40]. While our procedure is guaranteed to terminate on finite state systems, it must be inherently incomplete in general. Our experimental evaluation demonstrates that we can effectively and efficiently find finite bisimulations for infinite-state systems. However, there exist systems for which bisimulation learning can never successfully terminate. We give an example for such a system: Consider the infinite state space of natural numbers $S = \{0, 1, \dots\}$, where each state transitions by subtracting one, and zero loops on itself, i.e., $T = \{0 \mapsto 0\} \cup \{n \mapsto n-1, n > 0\}$. The labelling function distinguishes *zero*, *even*, and *odd* numbers (see Fig. 9).

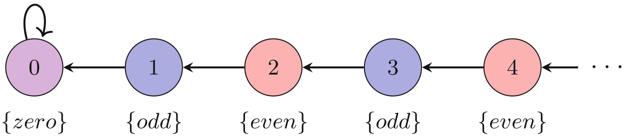


Fig. 9. A system for which bisimulation learning can never terminate, as no finite stutter-insensitive bisimulation exists.

Any infinite trajectory starting in some state i will eventually enter state zero. However, depending on the starting state, it will traverse a different sequence of even and odd states. Hence, we can construct $\mathrm{LTL}_{\setminus\bigcirc}$ formulas that distinguish each state from *smaller* states. For instance, the formula $\Diamond(\mathit{even} \wedge \Diamond(\mathit{odd} \wedge \Diamond(\mathit{zero})))$ can only be satisfied by states larger than one. Per Theorem 1, since $\mathrm{LTL}_{\setminus\bigcirc}$ can distinguish any state from smaller states, every state must be its own equivalence class with respect to stutter-insensitive bisimulation. When applying bisimulation learning to the described system, the CEGIS loop can never terminate with a finite quotient. Our procedure will keep enlarging the partition template used to fit the growing set of counterexamples, but will never be able to generalize to the entire state space. We note that this is a limitation intrinsic to bisimulations, i.e., no bisimulation algorithm could successfully terminate when applied to the stated system.

6 Conclusion

We have presented the first data-driven method to compute bisimulations. We have demonstrated that our method effectively computes finite abstractions for model checking and diagnostics. We instantiated our method to stutter-insensitive bisimulations, showcased its efficacy on $\mathrm{LTL}_{\setminus\bigcirc}$ model checking of discrete-time synchronization protocols as well as on conditional termination analysis benchmarks from the SV-COMP. On these benchmarks, our method yielded faster results than alternative model checking algorithms based on BDDs and IC3 (nuXmv), and state-of-the-art software model checking procedures (Ultimate and CPAChecker). Our benchmark sets are systems with long completeness thresholds and deep counterexamples, for which stutter-insensitive bisimulations provide succinct abstract quotients.

Our technique builds upon an existing proof rule for well-founded bisimulations. This allows us to characterise stutter-insensitive bisimulations as classifiers from infinite concrete states to finite abstract states, with an attached ranking function on each abstract state that strictly decreases as the concrete system stutters. This has enabled implementing a learner-verifier framework to compute bisimulations for deterministic systems with discrete state space. Our approach readily extends to *strong* bisimulations for deterministic systems, even though in practice these produce too large abstractions for effective model checking. Stutter-insensitive bisimulations instead are coarser and, therefore, generate smaller quotients. Not only this enables an effective verification of $\mathrm{LTL}_{\setminus\bigcirc}$ properties, but also provides succinct and interpretable abstractions.

Our result is the basis for several extensions. First, we envision extensions towards stutter-insensitive bisimulations for non-deterministic systems, which are harder because they require more general conditions on learner and verifier. Second, we target extensions towards continuous-state systems, which are harder because they offer much less flexibility in terms of numerical representation [23,24,54]. Lastly, we envision extensions towards using neural architectures for further flexibility and scalability in state classifier representation [1].

References

1. Abate, A., Edwards, A., Giacobbe, M.: Neural abstractions. In: NeurIPS (2022)
2. Abate, A., Giacobbe, M., Roy, D.: Stochastic omega-regular verification and control with supermartingales. In: CAV. LNCS. Springer (2024)
3. Ashok, P., Jackermeier, M., Jagtap, P., Kretínský, J., Weininger, M., Zamani, M.: dtcontrol: decision tree learning algorithms for controller representation. In: HSCC, pp. 17:1–17:7. ACM (2020)
4. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
5. Balcázar, J.L., Gabarró, J., Santha, M.: Deciding bisimilarity is P-Complete. Formal Aspects Comput. **4**(6A), 638–648 (1992)
6. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: TACAS (2). LNCS, vol. 13994, pp. 495–522. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
7. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
9. Bogomolov, S., Herrera, C., Steiner, W.: Verification of fault-tolerant clock synchronization algorithms. In: ARCH@CPSWeek. EPiC Series in Computing, vol. 43, pp. 36–41. EasyChair (2016)
10. Bouajjani, A., Fernandez, J.-C., Halbwachs, N.: Minimal model generation. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 197–203. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0023733
11. Bozga, M., Iosif, R., Konečný, F.: Deciding conditional termination. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 252–266. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_18
12. 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
13. Brázdil, T., Chatterjee, K., Křetínský, J., Toman, V.: Strategy representation by decision trees in reactive synthesis. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 385–407. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_21
14. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing finite kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**, 115–131 (1988)

15. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 102̂0 states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
16. 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
17. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
18. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: POPL, pp. 265–276. ACM (2007)
19. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_32
20. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
21. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. Int. J. Softw. Tools Technol. Transf. **20**(2), 157–177 (2018)
22. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. In: ESEC/SIGSOFT FSE, pp. 633–645. ACM (2022)
23. Girard, A.: Approximately bisimilar finite abstractions of stable linear systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 231–244. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71493-4_20
24. Girard, A., Pappas, G.J.: Approximate bisimulation: A bridge between computer science and control theory. Eur. J. Control. **17**(5–6), 568–578 (2011)
25. Glabbeek, R.J.: The linear time — Branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_6
26. Groote, J.F., Jansen, D.N., Keiren, J.J.A., Wijs, A.: An $O(m\log n)$ algorithm for computing stuttering equivalence and branching bisimulation. ACM Trans. Comput. Log. **18**(2), 13:1–13:34 (2017)
27. Grumberg, O., Francez, N., Makowsky, J.A., de Roever, W.P.: A proof rule for fair termination of guarded commands. Inf. Control **66**(1/2), 83–102 (1985)
28. Heizmann, M., et al.: Ultimate automizer and the commuhash normal form - (competition contribution). In: TACAS (2). LNCS, vol. 13994, pp. 577–581. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_39
29. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985)
30. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM (2004)
31. Hopcroft, J.: An n log n algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations, pp. 189–196. Academic Press (1971)
32. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Inf. Comput. **86**(1), 43–68 (1990)
33. Lamport, L.: What good is temporal logic? In: IFIP Congress, pp. 657–668. North-Holland/IFIP (1983)

34. Lamport, L., Melliar-Smith, P.M.: Byzantine clock synchronization. In: PODC, pp. 68–74. ACM (1984)
35. Lee, D., Yannakakis, M.: Online minimization of transition systems (extended abstract). In: STOC, pp. 264–274. ACM (1992)
36. Lee, I., Rajasekaran, S.: A parallel algorithm for relational coarsest partition problems and its implementation. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 404–414. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58179-0_71
37. Martens, J., Groote, J.F., van den Haak, L., Hijma, P., Wijs, A.: A linear parallel algorithm to compute bisimulation and relational coarsest partitions. In: Salaün, G., Wijs, A. (eds.) FACS 2021. LNCS, vol. 13077, pp. 115–133. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90636-8_7
38. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer (1980). https://doi.org/10.1007/3-540-10235-3
39. Milner, R.: Communication and concurrency. PHI Series in computer science. Prentice Hall (1989)
40. Moller, F.: Infinite results. In: Montanari, U., Sassone, V. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 195–216. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61604-7_56
41. 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
42. Murali, V., Trivedi, A., Zamani, M.: Closure certificates. In: HSCC, pp. 10:1–10:11. ACM (2024)
43. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: Ramesh, S., Sivakumar, G. (eds.) FSTTCS 1997. LNCS, vol. 1346, pp. 284–296. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0058037
44. Nicola, R.D., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM **42**(2), 458–487 (1995)
45. Nori, A.V., Sharma, R.: Termination proofs from tests. In: ESEC/SIGSOFT FSE, pp. 246–256. ACM (2013)
46. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
47. Pappas, G.J.: Bisimilar linear systems. Autom. **39**(12), 2035–2047 (2003)
48. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981). https://doi.org/10.1007/BFb0017309
49. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE Computer Society (1977)
50. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415. ACM (2006)
51. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) SAS 2014. LNCS, vol. 8723, pp. 302–318. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10936-7_19
52. Vardi, M.Y.: Verification of concurrent programs: The automata-theoretic framework. Ann. Pure Appl. Log. **51**(1–2), 79–98 (1991)
53. Walker, D.J.: Bisimulations and divergence. In: LICS, pp. 186–192. IEEE Computer Society (1988)
54. Zamani, M., Esfahani, P.M., Majumdar, R., Abate, A., Lygeros, J.: Symbolic control of stochastic systems via approximately bisimilar finite abstractions. IEEE Trans. Autom. Control **59**(12), 3135–3150 (2014)

Regular Reinforcement Learning

Taylor Dohmen(✉), Mateo Perez, Fabio Somenzi,
and Ashutosh Trivedi

University of Colorado, Boulder, CO 80309, USA
{taylor.dohmen,mateo.perez,fabio,
ashutosh.trivedi}@colorado.edu

Abstract. In reinforcement learning, an agent incrementally refines a behavioral policy through a series of episodic interactions with its environment. This process can be characterized as *explicit* reinforcement learning, as it deals with explicit states and concrete transitions. Building upon the concept of symbolic model checking, we propose a *symbolic* variant of reinforcement learning, in which sets of states are represented through predicates and transitions are represented by predicate transformers. Drawing inspiration from regular model checking, we choose *regular languages* over the states as our predicates, and *rational transductions* as predicate transformations. We refer to this framework as *regular reinforcement learning*, and study its utility as a symbolic approach to reinforcement learning. Theoretically, we establish results around decidability, approximability, and efficient learnability in the context of regular reinforcement learning. Towards practical applications, we develop a deep regular reinforcement learning algorithm, enabled by the use of graph neural networks. We showcase the applicability and effectiveness of (deep) regular reinforcement learning through empirical evaluation on a diverse set of case studies.

Keywords: Reinforcement Learning · Regular Model Checking · Graph Neural Networks · Symbolic Techniques for Verification and Synthesis

1 Introduction

Reinforcement learning [51] (RL) is a sampling-based approach to synthesis, capable of producing solutions with superhuman efficiency [10,44,49]. An RL agent interacts with its environment through episodic interactions while receiving scalar rewards as feedback for its performance. Following the explicit/symbolic dichotomy of model checking approaches, the interactions in classic RL can be characterized as *explicit*: each episode consists of a sequence of experiences in which the agent chooses an action from a concrete state, observes the next concrete state, and receives an associated reward for this explicit interaction.

We envision a *symbolic* approach to RL, where each experience may deal with a set of states represented by a *predicate*, and the evolution of the system

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 184–208, 2024.
https://doi.org/10.1007/978-3-031-65633-0_9

is described by *predicate transformers*. When the state space is large, symbolic representations may lead to greater efficiency and better generalization. Moreover, there are systems with naturally succinct representations—such as factored MDPs [27,32], succinct MDPs [23], and Petri nets [9]—that can benefit from symbolic manipulation of states.

The concept of symbolic interactions with an environment differs significantly from typical approximation methods used in RL, such as linear approximations [51] or deep neural networks [31]. In the context of such techniques, a learning agent attempts to generalize observations based on perceived similarities between them. In the symbolic setting, however, the generalization of a given interaction is explicitly provided by the environment itself. As a result, symbolic interactions facilitate a more direct form of generalization, where the environment ensures that similar interactions lead to similar outcomes.

This paper presents *regular reinforcement learning* (RRL), a symbolic approach to RL that employs regular languages and rational transductions, respectively, as models of predicates and their transformations. While natural languages can be used to encode symbolic interactions, we use regular languages [50] for the following reasons: (1) Regular languages enable unambiguous representation of predicates and predicate transformers. (2) Regular languages possess elegant theoretical properties including the existence of minimal canonical automata, determinizability, closure under many operations, and decidable emptiness and containment. (3) Regular languages hold a special position in machine learning, enjoying numerous efficient learnability results and active learning algorithms.

Regular languages also form the basis of a class of powerful symbolic model checking algorithm for infinite-state systems known as *regular model checking* (RMC) [2,5,13,18]. The following example introduces the concepts of RRL though a variation on the canonical token passing protocol used in the RMC literature [5,18].

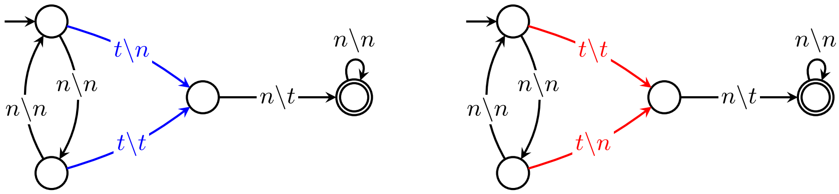
Example 1 (Token Passing). The token passing protocol involves an arbitrary number of processes arranged in a linear topology and indexed by consecutive natural numbers. At any point in time, each process can be in one of two states: t if it has a token or n if it does not. The states of the system are then strings over the alphabet $\{t, n\}$. The initial state, in which only the leftmost process has a token, is the regular language tn^*.

At each time step, an agent chooses an action from the set $\{a, b, c\}$, and each of these actions corresponds to one of the following outcomes.

(a) Each even-indexed process with a token passes it to the right. Each odd-indexed process with a token passes a copy of it to the right.

(b) Each odd-indexed process with a token passes it to the right. Each even-indexed process with a token passes a copy of it to the right.

(c) The outcome of a (resp. b) occurs with probability p (resp. $(1 - p)$).



(1.1) T_a (1.2) T_b

Fig. 1. An edge from q_0 to q_2 labelled by $t\backslash n$ denotes that if the transducer reads the symbol t from state q_0, then it outputs the symbol n and moves to q_2. Double-circled states are accepting. Such a machine is understood to only produces outputs for inputs that, once completely processed, leave the transducer in an accepting state.

Figure 1 depicts finite state transducers[1] corresponding to actions a and b. The property to be verified is that exactly one process possesses a token at any given time. The essence of this property may be captured by a reward function[2] $R : 2^{\{t,n\}^*} \to \mathbb{R}$ defined such that

$$R(L) = \begin{cases} 0 & \text{if } L \subseteq n^*tn^*, \\ -1 & \text{otherwise.} \end{cases}$$

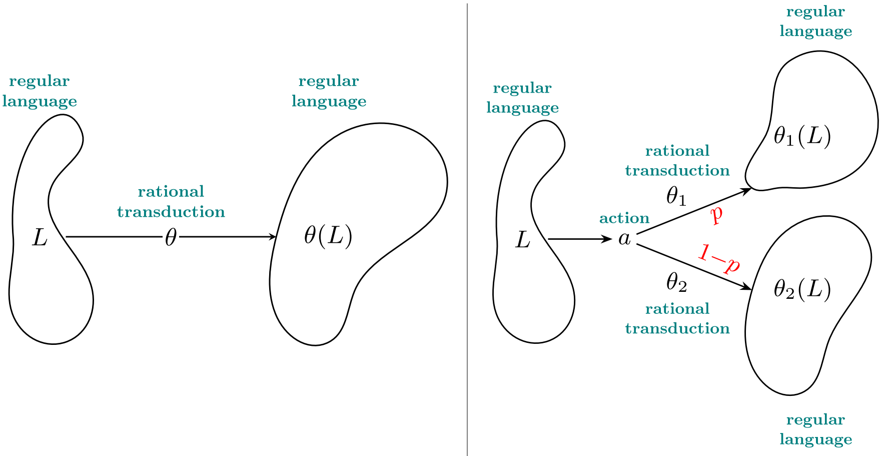
From the initial configuration tn^*, action a moves the system to state ntn^* and incurs a reward of 0, while action b transitions the system to the configuration ttn^* and incurs a reward of -1. The optimal policy selects action a when the token is with a process with an even index, and chooses action b otherwise.

In RRL, the agent initially chooses an action that it deems appropriate for the state tn^*. The environment then returns a language obtained by applying either transducer T_a or transducer T_b to transform tn^*, depending on the agent's choice. From tn^*, the two possible languages are ntn^* and ttn^*. The environment also assigns a reward to the agent. Repeated interactions of this type result in a sequence of states (regular languages) and rewards. The goal of the agent is to learn a policy (a function from regular languages to actions) that maximizes the cumulative reward.

Since there are infinitely many regular languages, the system described in Example 1 gives rise to an infinite-state decision process. As there is no known convergent RL algorithm for infinite-state environments in general, this prohibits the direct use of tabular RL algorithms for RRL. In regular model checking, techniques exist to address the difficulties of dealing with infinite state spaces, such as widening and acceleration. In regular reinforcement learning, we will leverage advances in graph neural networks to tackle this challenge.

[1] Note that the transducers in Fig. 1 are designed under the implicit assumption that their input strings will contain at most one t symbol. This is because there is no way of removing tokens from the system once they have been introduced, and, as a result, no learning can occur after a second token is introduced.

[2] Since containment is decidable for regular languages, R is computable.



(2.1) An RMC transition.

(2.2) An RMDP transition.

Fig. 2. Illustration of the difference between transitions in RMC and RMDPs.

Contributions. As in RMC, the primary application of language-theoretic modeling in RRL is the symbolic representation of states and transitions in the underlying system. We formalize RL environments that are constructed according to this principle under the name *regular Markov decision processes* (RMDPs). These environments generalize the systems modeled in RMC by incorporating controllable dynamics (through the agent selecting actions) and stochastic transition dynamics. Figure 2 provides a visual depiction of the similarities and differences between system transitions in RMC vs. RRL.

We provide a theoretical analysis of various aspects of RMDPs, focussing on issues related to decidability, finiteness, and approximability of optimal policies. This clarifies the basic limits of RRL and helps in determining when standard RL methods can or cannot be adapted to this setting. In particular, we establish the following results in Sect. 4.

- The optimal expected payoff, known as the value, of a given RMDP under an arbitrary payoff function is not computable.
- For any RMDP with computable rewards and transition probabilities, the value under a discounted payoff is approximately computable.
- For any RMDP with computable rewards and transition probabilities, the value under a discounted payoff is PAC-learnable.
- We identify several conditions under which an RMDP remains finite and present a Q-learning algorithm for such situations.

After this, we turn our attention toward practical applications of RRL. In Sect. 5 we propose a formulation of deep RRL. By representing regular languages as finite-state automata and viewing automata as labeled directed graphs, we are able to exploit graph neural networks for approximating optimal values and policies. Graph neural networks [56] are neural network architectures that process graphs as input, typically by performing repeated message passing of vectors over the graph's structure. We demonstrate through a collection of experimental case studies that deep RRL is both natural and effective.

2 Related Work

Regular Model Checking. RMC [3,5,18,55] is a verification framework based on symbolically encoding system states and transitions as regular languages and rational transductions, respectively. Despite the relative simplicity of rational transductions, allowing their arbitrary iteration produces a Turing-complete model of computation. Consequently, significant effort has been put into methods to approximate the transitive closures of rational transductions, and to compute them exactly in special cases [17,38,52].

In particular, incorporating automata learning techniques into the RMC toolbox [33,43,45] has shown promise. There is also significant work on improving the framework's expressive capabilities by extending RMC to enable the use of ω-regular languages [13,40,41], regular tree languages [4,6,16,19], and more powerful types of transductions [26]. RMC and its various extensions have been successfully applied to verification safety and liveness properties in a variety of settings related to mutual exclusion protocols, programs operating on unbounded data structures [14,15], lossy channel systems [8], and additive dynamical systems over numeric domains [11,12]. To the best of our knowledge, this paper is the first to combine deep reinforcement learning with regular model checking.

Regular Languages and Reinforcement Learning. The use of regular languages in RL has become increasingly popular to meet the increasing demand for structured, principled representations in neuro-symbolic artificial intelligence. The work closest to our own employs regular languages as a mechanism for modeling aspects of environments with certain kinds of non-Markovian, or history-dependent, dynamics.

Regular Decision Processes [20,21] are the topic of a recent line of research at the intersection of language-theoretic regularity and sequential optimization. A regular decision process is a finite state probabilistic transition system—much like a traditional Markov decision process (MDP)—except that transition probabilities and rewards are dependent on some regular property of the history. Note that while regular decision processes provide a succinct modeling framework for a subclass of non-Markovian optimization problems, they can be converted to larger, but semantically equivalent, finite-state MDPs. In contrast, the RMDPs introduced in this paper are not generally equivalent to finite MDPs. Considerable work has been done to develop the theory and practice around regular decision processes, including design and analysis of inference algorithms [1], learning efficiency analysis [47], and empirical evaluations of specific modeling tasks [42].

Reward Machines [34–36] are finite state machines used in modeling reward signals in decision processes with non-Markovian, but regular, reward dynamics Attention to the topic has resulted in inference algorithms for learning reward machines in partially observable MDPs [37], methods for jointly learning reward machines and corresponding optimal policies [57], adaptations of active grammatical inference algorithms like L* for reward machine inference [58], generalization to probabilistic machines modeling stochastic reward signals [24,29], applications to robotics [22], and more [25,59].

3 Preliminaries

Let $\mathbb{N}$ and $\mathbb{R}$ denote, respectively, the natural numbers and the real numbers. For a set X, we write 2^X to denote its powerset and $|X|$ to denote its cardinality.

3.1 Regular Languages

An alphabet Σ is a finite set of symbols, and a word w over Σ is a finite string of its symbols. The length $|w|$ of a word w is the number of its constituent symbols. The empty word, of length 0, is denoted by ε. We write Σ^n for the set of all words of length n. Further, let $\Sigma^{\leq n} = \bigcup_{k=0}^{n} \Sigma^k$ be the set of all strings of length at most n and let $\Sigma^* = \bigcup_{n=0}^{\infty} X^n$ be the set of all words over Σ. A subset $L \subseteq \Sigma^*$ is a called a language.

Definition 1 (FSA). *A (nondeterministic) finite-state automaton (FSA) $\mathcal{A}$ is given by a tuple $\langle \Sigma, Q, q_0, F, \delta \rangle$, where Σ is an alphabet, Q is a finite set of states, $q_0 \in Q$ is a distinguished initial state, $F \subseteq Q$ is a set of final states, and $\delta : Q \times \Sigma \to 2^Q$ is a transition function.*

The transition function δ may be extended to $\delta^* : Q \times \Sigma^* \to 2^Q$ such that $\delta^*(q, \varepsilon) = \{q\}$ and $\delta^*(q, \sigma w) = \bigcup_{q' \in \delta(q,\sigma)} \delta^*(q', w)$. The semantics of an FA $\mathcal{A}$ are given by a language

$$L_{\mathcal{A}} = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \emptyset\},$$

and we say that $\mathcal{A}$ recognizes *$L_{\mathcal{A}}$.*

A language is *regular* if it is recognized by an FSA.

3.2 Rational Transductions

Let Σ and Γ be alphabets. A mapping $\theta : \Sigma^* \to 2^{\Gamma^*}$, or equivalently a relation over $\Sigma^* \times \Gamma^*$, is called a *transduction*. For a language L and a transduction $\theta : \Sigma^* \to 2^{\Gamma^*}$, let $\theta(L) = \bigcup_{x \in L} \theta(x)$. The domain of $\theta : \Sigma^* \to 2^{\Gamma^*}$ is given as $\mathrm{dom}(\theta) = \{x \in \Sigma^* : \theta(x) \neq \emptyset\}$ and its image is defined as $\mathrm{im}(\theta) = \bigcup_{x \in \Sigma^*} \theta(x)$.

Given a finite set of transductions $\varTheta$ with type $\varSigma^* \to 2^{\varSigma^*}$, each finite word $\theta_1 \dots \theta_n \in \varTheta^*$ corresponds to the transduction $\theta_n \circ \dots \circ \theta_1$ where ε represents the identity mapping, i.e. $\varepsilon(x) = x$ for every $x \in \varSigma^*$. For convenience, we identify the word $\theta_1 \dots \theta_n$ with the transduction $\theta_n \circ \dots \circ \theta_1$ so that $\theta_1 \dots \theta_n(x) = \theta_n \circ \dots \circ \theta_1(x)$ holds for every $x \in \varSigma^*$. The set of languages reachable from a given language L via elements of $\varTheta^*$ is called the *orbit* of $\varTheta$ on L and is written as

$$\mathrm{Orb}_{\varTheta}(L) = \{\tau(L) : \tau \in \varTheta^*\}.$$

Definition 2 (FST). *A (nondeterministic) finite-state transducer (FST) T is given by a tuple $\langle \varSigma, \varGamma, Q, q_0, F, \delta \rangle$, where*

- *$\varSigma$ and $\varGamma$ are input and output alphabets, respectively,*
- *Q is a finite set of states,*
- *$q_0 \in Q$ is a distinguished initial state,*
- *$F \subseteq Q$ is a set of final states, and*
- *$\delta : Q \times \varSigma \to 2^{Q \times \varGamma^*}$ is a transition function that maps each state-input pair to a set of state-output pairs.*

The transition function δ may be extended to $\delta^* : Q \times \varSigma^* \to 2^{Q \times \varGamma^*}$ such that $\delta(q, \varepsilon) = \{\langle q, \varepsilon \rangle\}$ and $\delta^*(q, \sigma x) = \{\langle q_2, yz \rangle : \langle q_1, y \rangle \in \delta(q, \sigma) \wedge \langle q_2, z \rangle \in \delta^*(q_1, x)\}$. The semantics of T are the transduction $[\![T]\!] : \varSigma^* \to 2^{\varGamma^*}$ defined by

$$[\![T]\!](x) = \{y \in \varGamma^* : \exists q \in F.\ \langle q, y \rangle \in \delta^*(q_0, x)\}.$$

A *rational transduction* θ is one for which there exists an FST T such that $\theta = [\![T]\!]$. A *rational function* θ is a rational transduction such that $|\theta(x)| \leq 1$ for all $x \in \varSigma^*$. When discussing rational functions, we write the type as $\varSigma^* \to \varGamma^*$.

Remark 1. While the title of *regular language* has become standard terminology, there is no universally adopted vocabulary for their relational counterparts. The transductions we qualify here as *rational* are sometimes qualified alternatively in related work with terms such as *regular*, *FST-definable*, *GSM-definable*, etc.

3.3 Markov Decision Processes

Let $\mathrm{Dist}(X)$ be the family of all probability distributions over a set X.

Definition 3 (MDP). *A Markov decision process (MDP) M is presented by a tuple $\langle S, \hat{s}, A, p, r \rangle$, where*

- *S is a set of states,*
- *$\hat{s} \in S$ is a distinguished initial state,*
- *A is a set of actions,*
- *$p : S \times A \to \mathrm{Dist}(S)$ is a probabilistic transition function, and*
- *$r : S \times A \to \mathbb{R}$ is a reward function.*

For any states $s, t \in S$ and action $a \in A$, we write $p(t \mid s, a)$ as a shorthand for $p(s, a)(t)$. We call an MDP *finite* if both S and A are finite sets.

A *policy* over an MDP $M = \langle S, \hat{s}, A, p, r\rangle$ is a history-dependent function that determines how the next action is stochastically chosen. More formally, a policy is defined as a mapping $\pi : S\,(AS)^* \to \mathrm{Dist}\,(A)$ from the domain of interaction histories to probability distributions over the action space. Let Π_M be the set of all policies over the MDP M. Fixing a policy π on M induces a family of probability distributions $\{\mathbb{P}^n_\pi : n \in \mathbb{N}\}$ on histories $h = s_1 a_1 \ldots s_n a_n$ with $s_1 = \hat{s}$, where $\mathbb{P}^n_\pi(h) = \prod_{k=1}^{n-1} p(s_{k+1} \mid s_k, a_k)\pi(a_k \mid s_1 a_1 \ldots a_{k-1} s_k)$. There exists a unique extension $\mathbb{P}_\pi \in \mathrm{Dist}\,((SA)^\omega)$ of the family $\{\mathbb{P}^n_\pi : n \in \mathbb{N}\}$ and a corresponding expectation $\mathbb{E}_\pi$.

An *objective* over an MDP M with states S and actions A is a real-valued function J over the domain of infinite real sequences. Whenever the function $\mathsf{J} \circ r$ is $\mathbb{P}_\pi$-measurable, the expectation $\mathbb{E}_\pi(\mathsf{J}) = \int \mathsf{J} \circ r \; \mathrm{d}\mathbb{P}_\pi$ is well-defined and can be used to evaluate the quality of the policy π with respect to the environment M. The J-*value* of M is defined as $\mathrm{Val}_{\mathsf{J}}\,(M) = \sup_{\pi \in \Pi_M} \mathbb{E}_\pi(\mathsf{J})$.

Let J be a fixed objective function.

- The J-*value problem* asks, given as input (i) an MDP M and (ii) a lower bound b, to decide whether the inequality $b \leq \mathrm{Val}_{\mathsf{J}}\,(M)$ holds.
- The J-value is *computable* if, and only if, there is an algorithm that, given an MDP M as input, returns $\mathrm{Val}_{\mathsf{J}}\,(M)$.
- The J-value is *approximable* if, and only if, there exists an algorithm that, given as input (i) an MDP M and (ii) a tolerance $\epsilon > 0$, returns a value V such that $|\mathrm{Val}_{\mathsf{J}}\,(M) - V| \leq \epsilon$.

4 Regular Markov Decision Processes

Regular Markov decision processes (RMDPs) are MDPs where states have been provided with a specific structure expressed through a regular language over some alphabet Σ. An execution of an RMDP starts with an initial regular language $L_0 = I$. At each step $i \geq 0$, a decision maker or learning agent selects an action a_i from the current state L_i. The environment resolves the action by selecting a transduction θ_i from the probabilistic distribution over Θ corresponding to the action and returning the next state as $L_{i+1} = \theta_i(L_i)$ and returning the reward $\boldsymbol{r}(L_i)$. The goal of the agent is to learn a policy for selecting actions in a manner that optimizes the value of a given objective J in expectation.

Definition 4 (RMDP). *A* regular Markov decision process *(RMDP)* $\boldsymbol{R}$ *is given by a tuple* $\langle \Sigma, I, \Theta, A, \boldsymbol{p}, \boldsymbol{r}\rangle$*, where*

- Σ *is an alphabet,*
- $I \subseteq \Sigma^*$ *is an initial regular language,*
- Θ *is a finite set of rational transductions with type* $\Sigma^* \to 2^{\Sigma^*}$*,*
- A *is a finite set of actions,*

- $\boldsymbol{p} : 2^{\Sigma^*} \times A \to \mathrm{Dist}\,(\Theta)$ *is a mapping from language-action pairs to distributions over* Θ*, and*
- $\boldsymbol{r} : 2^{\Sigma^*} \to \mathbb{R}$ *is a bounded reward function.*

Semantically, $\boldsymbol{R}$ *is interpreted as a countable MDP* $[\![\boldsymbol{R}]\!] = \langle S, \hat{s}, A, p, r\rangle$*. The state set is defined as* $S = \mathrm{Orb}_\Theta\,(I)$ *with initial state* $\hat{s} = I$*, and the transition and reward functions are such that the equations*

$$p(\theta(L) \mid L, a) = \boldsymbol{p}(\theta \mid L, a) \qquad \text{and} \qquad r(L, a) = \boldsymbol{r}(L),$$

hold for all languages $L \in S$*, actions* $a \in A$*, and transductions* $\theta \in \Theta$*. The value of an objective* J *over a RMDP* $\boldsymbol{R}$ *is defined as* $\mathrm{Val}_{\mathsf{J}}\,(\boldsymbol{R}) = \mathrm{Val}_{\mathsf{J}}\,([\![\boldsymbol{R}]\!])$*.*

An RMDP $\boldsymbol{R}$ is called *finite* if the orbit $\mathrm{Orb}_\Theta\,(I)$, is finite. An RMDP is said to be *computable* if the transition probability map $\boldsymbol{p}$ and the reward function $\boldsymbol{r}$ are computable.

4.1 Undecidability of Values

Our first theoretical result establishes that value problems for RMDPs are generally undecidable.

Theorem 1. *Determining whether an arbitrary RMDP satisfies any fixed non-trivial property is undecidable.*

Proof. We construct, as depicted in Fig. 3, a deterministic FST that can simulate the transition relation of an arbitrary Turing machine (TM). Configurations of the TM, i.e. combinations of internal state and tape contents, are encoded as words in the regular language $(0 + 1)\,(0 + 1)^*\,Z\,(0 + 1)^*$, where Z is the finite set of internal states. The index i of the single element of Z occurring in each such word represents that the tape head of the TM is at position $i{-}1$. Assume that the TM in question includes an arbitrary transition instruction according to the following pair of rules.

- If 0 is read in state z, then write b_0, go to state z_0, and move the tape head.
- If 1 is read in state z, then write b_1, go to state z_1, and move the tape head.

We leave the direction of the tape head shift undetermined and show all possibilities in Fig. 3. The red edges show the construction for the above TM transition when the tape head shifts left. The blue edges show the construction when the tape head shifts right.

In combination with Rice's theorem [46]—which states that no non-trivial[3] property is decidable for the class of Turing machines— this construction implies the desired result. □

It follows from Theorem 1 that optimal values of RMDPs are not computable in general.

Corollary 1. *Under any objective, the RMDP value problem is undecidable.*

[3] Using Rice's terminology, a property is trivial with respect to class of models if it holds for all models in the class or if it holds for no models in the class.

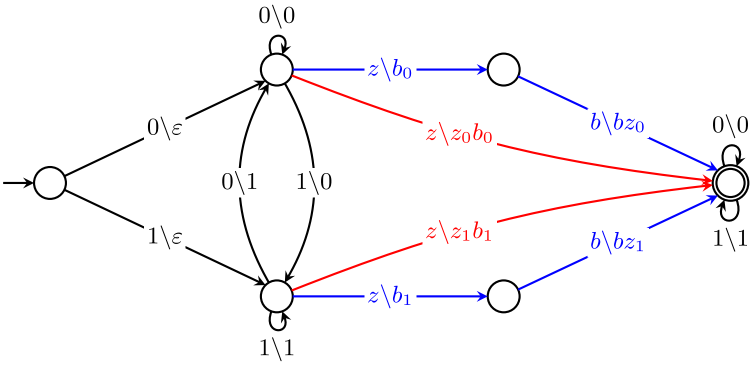


Fig. 3. The FST from the proof of Theorem 1, simulating the transition function of a Turing machine over a binary alphabet.

4.2 Discounted Optimization

We now consider RMDPs under discounted objectives. Let $x = x_1, x_2, \ldots$ be a bounded infinite sequence of real numbers. Given a discount factor $\lambda \in [0, 1)$, the λ-*discounted objective* D_λ is defined as

$$\mathsf{D}_\lambda(x) = \sum_{n=1}^{\infty} x_n \lambda^{n-1}.$$

Over computable RMDPs, it is possible to approximate the D_λ-value to an arbitrary tolerance. This result is facilitated by properties of the discounted objective, and therefore holds even when the RMDP in question is not finite. The proof uses the standard technique of finding, given a tolerance ϵ and a discount factor λ, the least n such that

$$\left| \mathsf{D}_\lambda(x) - \sum_{k=1}^{n} \lambda^{k-1} x_k \right| \leq \epsilon.$$

Theorem 2. *If $\boldsymbol{R}$ is a computable RMDP, then the D_λ-value is ϵ-approximable, for any $\lambda \in [0, 1)$ and any $\epsilon > 0$.*

Proof. For a given RMDP $\boldsymbol{R}$, the D_λ-value can be characterized by the following Bellman optimality equation:

$$D(L) = \max_{a \in A} \left\{ \boldsymbol{r}(L) + \lambda \sum_{\theta \in \Theta} \boldsymbol{p}(\theta \mid L, a) D(\theta(L)) \right\}.$$

It follows from the more general result on MDPs with countable state space, finite action space, and bounded reward [30]. Let b be an upper bound on the absolute value of the rewards. For a given $\epsilon > 0$, let n be such that

$$\frac{\lambda^{n+1} b}{1 - \lambda} \leq \epsilon.$$

Then a solution to the following recurrence characterizes an ε-optimal value and corresponding memoryful policy for the RMDP:

$$D_n(L) = \begin{cases} \max\limits_{a \in A} \left\{ \boldsymbol{r}(L) + \lambda \sum\limits_{\theta \in \Theta} \boldsymbol{p}(\theta \mid L, a) D_{n-1}(\theta(L)) \right\} & \text{if } n > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The proof is now complete. □

An RL algorithm is *probably approximately correct* (PAC) [53], with respect to parameters $\epsilon > 0$ and $\delta > 0$, if after polynomially many samples of the environment, it produces an ϵ-optimal policy with probability $1 - \delta$. Objective functions under which PAC algorithms exist are called PAC-learnable.

Theorem 3. *For every RMDP, the* D_λ*-value is PAC-learnable.*

Proof. Our approach for calculating ϵ-optimal policies for the discounted objective involves computing a policy that is optimal for a fixed number of steps, denoted by n. Given $\epsilon > 0$, we choose n such that

$$\frac{\lambda^{n+1} b}{1 - \lambda} \leq \epsilon.$$

This policy can be computed on a finite-state MDP obtained by unfolding the given RMDP n times. We can then apply existing PAC-MDP algorithms [7] to compute an $\frac{\epsilon}{2}$-optimal policy, which is also an ϵ-optimal policy for the RMDP. □

4.3 Finiteness Conditions

In this section, we provide three sufficient conditions to guarantee finiteness of the RMDP. Fix an arbitrary RMDP $\boldsymbol{R} = (\Sigma, I, \Theta, A, \boldsymbol{p}, \boldsymbol{r})$ with semantics $[\![\boldsymbol{R}]\!] = (S, \hat{s}, A, p, r)$.

Word-Based Condition. A transduction $\theta : \Sigma^* \to 2^{\Gamma^*}$ is (i) *length-preserving* if $|\theta(x)| = |x|$, (ii) *decreasing* if $|\theta(x)| < |x|$, (iii) *non-increasing* if $|\theta(x)| \leq |x|$, (iv) *non-decreasing* if $|\theta(x)| \geq |x|$, (v) *increasing* if $|\theta(x)| > |x|$, for all $x \in \Sigma^*$.

Proposition 1. *If* I *is a finite language,* Θ *is non-increasing,* $|\Sigma| = n$*, and* $\max_{x \in I} |x| = m$*, then* $\mathrm{Orb}_\Theta(I) \in 2^{O(n^m)}$.

Proof. The statement can be derived from the observation that the longest string possibly appearing in the image $\theta(I)$ of a finite language I under a non-increasing transformation θ is of length $m = \max_{w \in I} |w|$. There are n^m strings of length m over an alphabet Σ of size n, so it follows that $|\theta(I)| \leq 1 + \sum_{k=1}^{m} n^k$. More succinctly, this says that $|\theta(I)| = O(n^m)$. Since $\mathrm{Orb}_\Theta(I)$ must comprise some collection of subsets of $\Sigma^{\leq m}$, we conclude that $|\mathrm{Orb}_\Theta(I)| = 2^{O(n^m)}$. □

Language-Based Condition. A transduction $\theta : \Sigma^* \to 2^{\Sigma^*}$ is (i) *specializing* if $\theta(L) \subseteq L$, (ii) *non-specializing* if $\theta(L) \nsubseteq L$, (iii) *generalizing* if $L \subseteq \theta(L)$, (iv) *non-generalizing* if $L \nsubseteq \theta(L)$, for all $L \subseteq \Sigma^*$.

Proposition 2. *If $|I| = n$ and Θ is specializing, then* $\mathrm{Orb}_\Theta(I) \leq 2^n$.

Proof. This result can be deduced from the observation that when beginning with a finite initial language, specializing transformations can only generate languages with a cardinality that is either equal to or smaller than that of I. □

Reward-Based Condition. Let $\sim_{\boldsymbol{R}} \subseteq 2^{\Sigma^*} \times 2^{\Sigma^*}$ be an equivalence relation over languages such that $L_1 \sim_{\boldsymbol{R}} L_2$ if, and only if,

$$\boldsymbol{r}(L_1) = \boldsymbol{r}(L_2) \quad \text{and} \quad \forall \theta \in \Theta. \quad \theta(L_1) \sim_{\boldsymbol{R}} \theta(L_2).$$

This relation is often useful as a means of partitioning the state space of an RMDP into a finite set of equivalence classes that respects the structure of its dynamics. For instance, it is straightforward to deduce the following proposition.

Proposition 3. *If there exists $n \in \mathbb{N}$ such that $\boldsymbol{r}(L) = \boldsymbol{r}\left(L \cap \Sigma^{\leq n}\right)$ holds for every $L \subseteq \Sigma^*$, then* $\mathrm{Orb}_\Theta(I)$ *has finitely many $\sim_{\boldsymbol{R}}$-equivalence classes.*

4.4 Q-Learning in RMDPs

We have discussed some conditions that ensure the finiteness of $[\![M]\!]$. When any such condition is satisfied, it becomes feasible to employ off-the-shelf RL algorithms for discounted optimization. Equation (1)—in which $[L]_\sim$ denotes the equivalence class of $\sim_{\boldsymbol{R}}$ to which the language L belongs—provides an iteration scheme for a variation on the Q-learning [54] algorithm tailored for RMDPs. If learning rates $(\alpha_n)_{n\in\mathbb{N}}$ are such that $\sum_{n=1}^{\infty} \alpha_n = \infty$ and $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$, and the trajectory $\left([L_n]_\sim, a_n, r_n\right)_{n\in\mathbb{N}}$ includes each pair $[L]_\sim, a$ infinitely often, then iterating Eq. (1) converges almost surely to an optimal policy.

$$Q_{n+1}\left([L_n]_\sim, a_n\right) \coloneqq (1-\alpha_n)\, Q_n\left([L_n]_\sim, a_n\right) + \alpha_n \left(r_n + \max_{a\in A} Q_n\left([L_{n+1}]_\sim, a\right)\right) \quad (1)$$

5 Deep Regular Reinforcement Learning

Generally speaking, RMDPs may have infinite state spaces, so we cannot guarantee convergence of Q-learning. In light of this fact and the uncomputability of exact discounted values—established by Theorem 1—it makes sense to consider approximate learning methods. Accordingly, we propose a deep learning approach based on using *graph neural networks* (GNNs). Our key insight in this context is the observation that we can use automaton representations of the states of an RMDP directly as inputs to GNNs. Much like standard deep RL uses feature vectors as inputs for neural networks, this technique uses automata—which are essentially finite labeled graphs—as inputs for GNNs. We term this approach *deep regular reinforcement learning.*

Before presenting experimental results, we describe the overall architecture of our learning scheme in the next subsection.

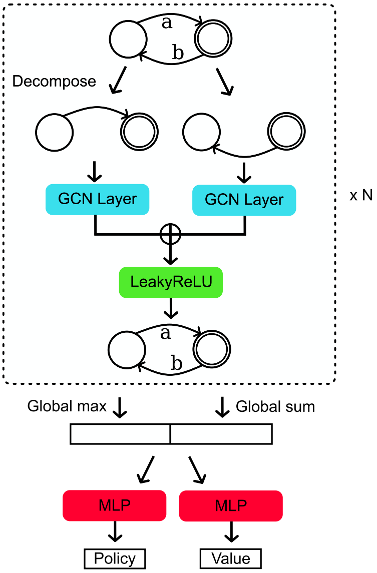


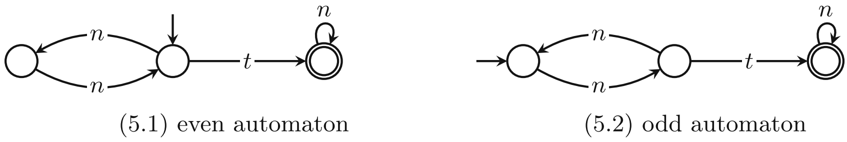
Fig. 4. Deep regular reinforcement learning architecture.

Our graph neural network architecture, is based on the graph convolution operator proposed by Kipf & Welling [39]. We perform an independent graph convolution for each letter in the input automaton— only allowing the convolution to operate over the graph connectivity for that letter—and take the mean of the resulting vectors for each node, followed by a nonlinearity. We repeat this for N layers and then concatenate the sum of all node vectors with the element-wise maximum of all node vectors. Separate multi-layer perceptrons produce the policy and state value predictions from this representation. We use proximal policy optimization (PPO) [48] for training. Figure 4 outlines this architecture. The initial embedding for each node in the automaton is a binary vector of length two, which encodes whether a node is the initial state and if a node is accepting. For all experiments, the graph neural network had 3 graph convolution layers with hidden dimension 256, and the multi-layer perceptron heads had 2 layers each with hidden dimension 256. We used the LeakyReLU nonlinearity.

In the remainder of this section, we present specific examples of regular RL problems and provide experimental results to illustrate the effectiveness of deep regular reinforcement learning.

5.1 Token Passing

We first consider the token passing scenario of Example 1 (cf. Fig. 1). Note that this example admits a partitioning of the environment via the equivalence relation defined in Sect. 4.3: there are two equivalence classes, corresponding to whether there are an even or an odd number of n symbols before the first t symbol. We compare using the GNN on the original representation (GNN) and on the representation formed by the two equivalence classes (GNN+EC). Figure 5 shows the FSA representations used for the two equivalence classes.



(5.1) even automaton

(5.2) odd automaton

Fig. 5. Automata used to represent even and odd equivalence classes.

The hyperparameters we used for PPO in this case study were 1024 steps per update, a 512 batch size, 4 optimization epochs, a clip range of $\epsilon = 0.2$, and a discount factor of $\lambda = 0.99$. The learning curves[4] are shown in Fig. 6.

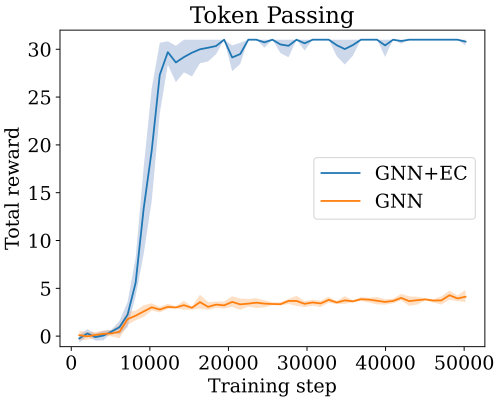


Fig. 6. Reward curves for the token passing case study.

Note that under the selected network architecture, determining whether the number of n symbols occurring before the first t symbol is even or odd is largely determined by the multi-layer perceptron components. Roughly, the number of

[4] Here, the dark lines are means and the shaded regions are the 10th to 90th percentiles over 5 training runs. All subsequent reward curves should be read this way as well.

n symbols before the first t symbol is encoded in unary in the global sum component of the graph representation. The multi-layer perceptrons must then detect parity on this unary representation, which is challenging. To encourage learning, we use a denser reward of 1 on every successful step and -1 on failure, up to a time limit of 30. Although alternative network architectures have the potential to perform better, the simple two-state equivalence class representation is still expected to result in faster learning than the unmodified representation. The learning run is shortened, and the maximum episode length is set to 30 to highlight the difference between these two setups. We see that forming equivalence classes leads to an increase in learning speed.

5.2 Duplicating Pebbles

Consider a grid world with multiple pebbles on it. The agent can select two adjacent directions, e.g., "up" and "right", and every pebble will be duplicated and moved in each of these directions. The goal of the agent is to have at least one pebble reach the goal state, while all pebbles do not accumulate a cost greater than a threshold $t = 2$. If a pebble goes over a trap cell, it incurs a cost of 1 for that pebble.

Although the number of pebbles grows exponentially, doubling after each action, the set of paths the pebbles take has an FSA representation. Namely, one can represent the growing paths by adding a state to the FSA with two transitions to the state corresponding to the two directions selected. This added state is marked as the only accepting state. The language of this FSA corresponds to all paths that pebbles have taken. A reward of -1 is given on failure and a reward of 1 is given on success. The grid layout is shown in Fig. 7, where the initial pebble begins in the top left. The agent learns the optimal policy "down, right", "down, right", "up, left", "up, left" in about 10k training steps. Figure 7 shows the execution of this optimal policy, from left to right, top to bottom. Traps are denoted by red cells and the goal is denoted by a green cell. The number in a cell counts the number of pebbles it contains.

The hyperparameters we used for PPO in this case study were 512 steps per update, a 128 batch size, 4 optimization epochs, a clip range of $\epsilon = 0.2$, and a discount factor of $\lambda = 0.99$. The resulting reward curve is shown in Fig. 8.

Since the representation of the state is an FSA of the possible trajectories, a linear program is solved at each step to find the highest cost path needed for computing the reward. Note that when "up, left" is first performed, some pebbles wrap around to the opposite side of the grid. If "down, right" was performed 3 times, instead of twice, then the agent would fail the objective since the 2 pebbles at $(1, 2)$ on the grid would duplicate and visit the trap state again after having already visited it once.

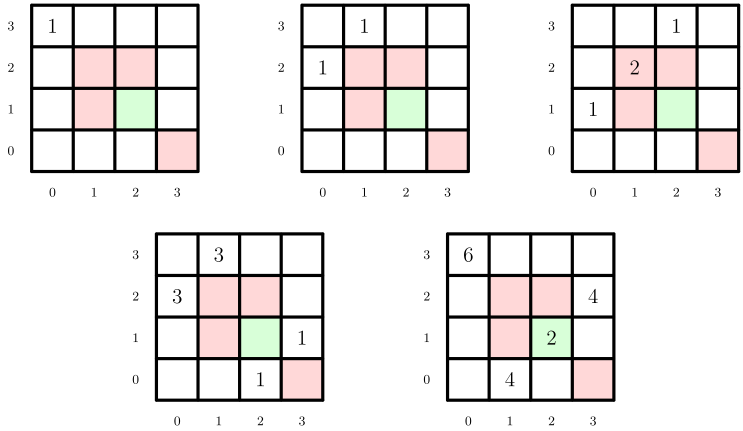


Fig. 7. Execution of the optimal policy for the duplicating pebbles case study.

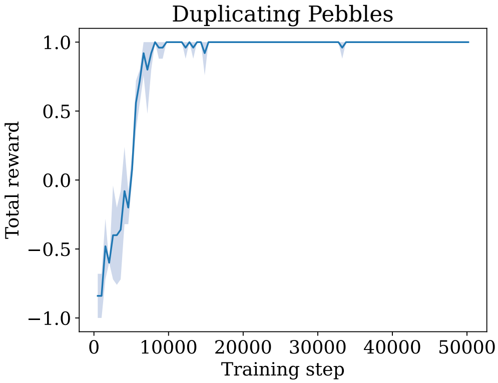


Fig. 8. Reward curve for the duplicating pebbles case study.

5.3 Shunting Yard Algorithm

To showcase representation of unbounded data structures like stacks and queues as a strength of RRL, we consider learning the shunting yard algorithm [28] which transforms an expression in infix notation to postfix notation.

We represent the input as a regular language consisting of a single string containing the concatenation of the infix notation input, the stack, and the output, each separated by a special symbol "#". The agent has three actions:

- moving the first character of the input to the output,
- pushing the first character of the input to the stack, and
- popping the top character on the stack to the output.

We generate random infix notation expressions and give a reward of -1 if the output is an invalid postfix expression, a reward of 0.5 if the output is a valid

State	Action
3*3+7+5##	Move
*3+7+5##3	Push
3+7+5#*#3	Move
+7+5#*#33	Pop
+7+5##33*	Push
7+5#+#33*	Move
+5#+#33*7	Pop
+5##33*7+	Push
5#+#33*7+	Move
#+#33*7+5	Pop
##33*7+5+	

State	Action
1+7*8+0##	Move
+7*8+0##1	Push
7*8+0#+#1	Move
*8+0#+#17	Push
8+0#+*#17	Move
+0#+*#178	Pop
+0#+#178*	Pop
+0##178*+	Push
0#+#178*+	Move
#+#178*+0	Pop
##178*+0+	

State	Action
7-1*3##	Move
-1*3##7	Push
1*3#-#7	Move
*3#-#71	Push
3#-*#71	Move
#-*#713	Pop
#-#713*	Pop
##713*-	

Fig. 9. Runs produced by the learned policy for the shunting yard algorithm.

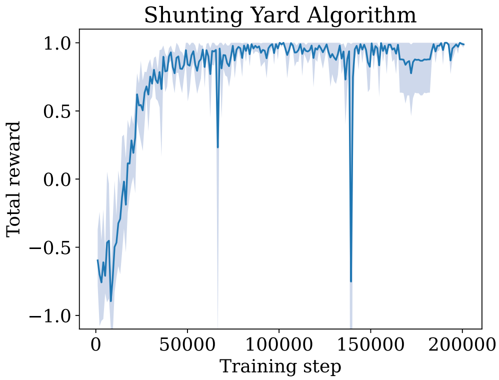


Fig. 10. Reward curve for the shunting yard algorithm case study.

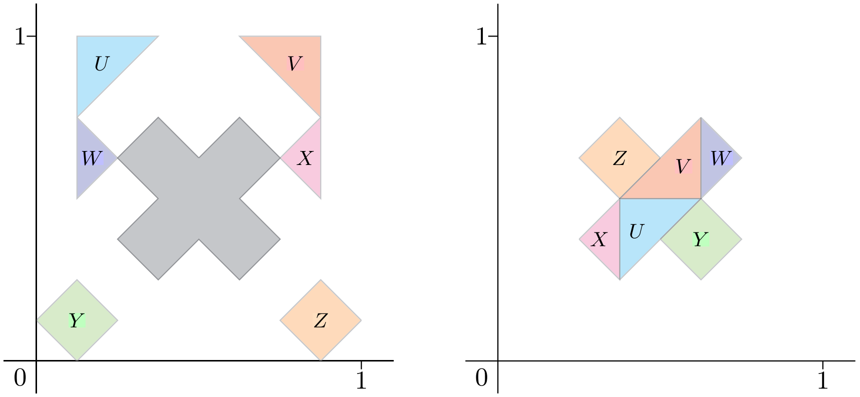
postfix expression that evaluates to the wrong value, and a reward 1 if the output evaluates to the correct value. The agent is able to learn an effective strategy in about 100k time steps.

Figure 9 shows example runs produced by the learned policy. The representation used during learning is an FSA accepting a single string, but we print the string for clarity. Actions are the actions selected upon observing that state. The last state is the final state at termination.

The hyperparameters we used for PPO in this case study were 1024 steps per update, a 128 batch size of, 10 optimization epochs, a clip range of $\epsilon = 0.2$, and a discount factor of $\lambda = 0.99$. The resulting reward curve is shown in Fig. 10.

5.4 Modified Tangrams

This case study examines the application of deep RRL to variations of geometric puzzles known as tangrams, which involve arranging a finite set of polygonal



(11.1) Tiles in their initial positions.

(11.2) Tiles arranged in a solution.

Fig. 11. A modified tangram. The goal is to cover the gray shape at the center resembling the symbol "×" by rearranging the colored tiles $\{U, V, W, X, Y, Z\}$. (Color figure online)

tiles on a flat surface to create a picture. The picture is typically a silhouette in the shape of a common object such as a building or a tree, and the puzzle is completed once the tiles have been arranged into a configuration that covers the silhouette exactly. A standard tangram set includes a collection of target pictures, 5 right triangles (2 large, 1 medium, and 2 small), a square, and a parallelogram. We consider modified tangrams, which we qualify as such because the pieces do not coincide with the standard tile set. An example is displayed in Fig. 11.

In order to cast these sorts of puzzles into the RRL framework, we apply standard notions used in positional numeration systems to connect geometric shapes and regular languages. More precisely, tiles are considered as sets of points in the unit square $[0,1] \times [0,1]$ of the Euclidean plane. Then, sets of points are encoded by regular languages consisting of digital expansions of these points.

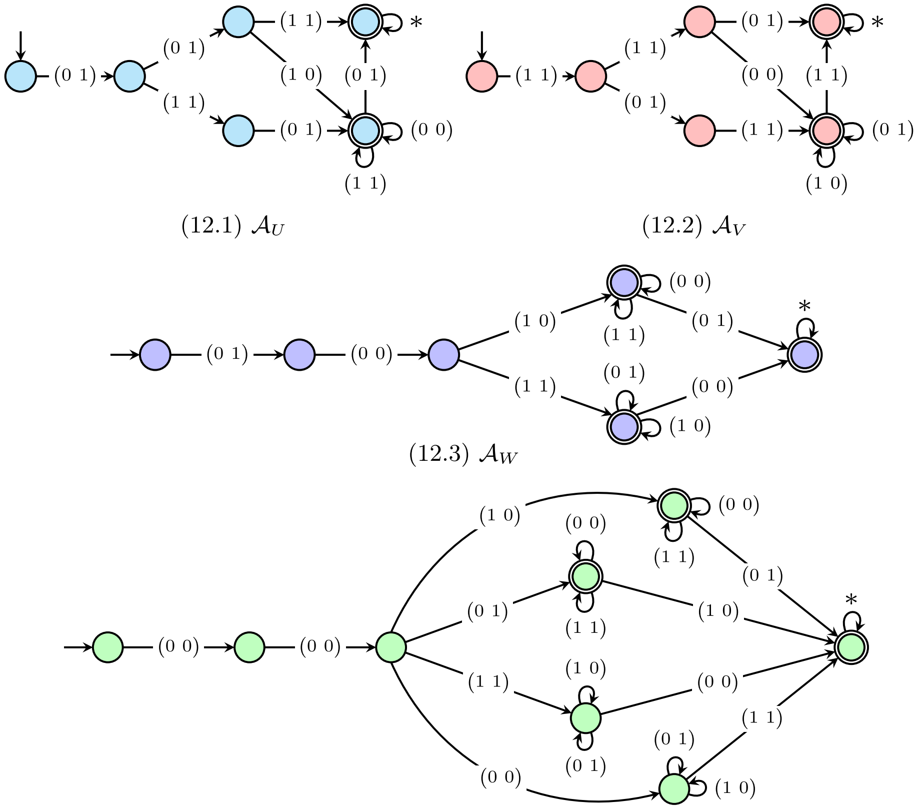
For a numeration base $b \in \mathbb{N}$, define a map $\langle\!\langle \cdot \rangle\!\rangle_b : \{0, \ldots, b-1\}^* \to (0,1)$ as

$$\langle\!\langle\, w \,\rangle\!\rangle_b = \sum_{k=1}^{|w|} \frac{w_k}{b^k}$$

to interpret each string of digits w as a base-b digital expansion (where the leftmost symbol is the most significant bit) of a number $\langle\!\langle\, w \,\rangle\!\rangle_b$ in the unit interval. Such interpretations extend to languages so that

$$\langle\!\langle\, L \,\rangle\!\rangle_b = \{\langle\!\langle\, w \,\rangle\!\rangle_b : w \in L\}\,.$$

We fix the base as $b = 2$, and consider automata over the alphabet the two-dimensional boolean alphabet $\mathbb{B}^2$ to encode points in the plane. We design automata capturing languages that represent the sets of points included in particular shapes, as illustrated in Fig. 12.



(12.1) $\mathcal{A}_U$

(12.2) $\mathcal{A}_V$

(12.3) $\mathcal{A}_W$

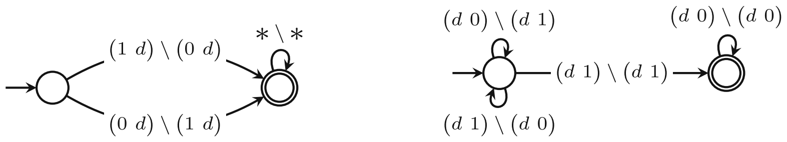
(12.4) $\mathcal{A}_Y$

Fig. 12. Automata corresponding to some of the starting tiles shown in Fig. 11.1.

Remark 2. Automata $\mathcal{A}_X$ and $\mathcal{A}_Z$ may be obtained from the automata $\mathcal{A}_W$ (Fig. 12.3) and $\mathcal{A}_Y$ (Fig. 12.4), respectively. This can be done by taking the logical complement of the x-coordinate on every non-looping transition and exchanging pairs of self loops on a common state labeled by $\begin{pmatrix}0\ 0\end{pmatrix}$ and $\begin{pmatrix}1\ 1\end{pmatrix}$ to ones labeled by $\begin{pmatrix}0\ 1\end{pmatrix}$ and $\begin{pmatrix}1\ 0\end{pmatrix}$.

We also design finite-state transducers, as illustrated in Fig. 13, for basic geometric operations such as translation by 1/2, translation by 1/4, and reflection across $x = 1/2$ and $y = 1/2$.

The agent's goal is to apply these basic transformations to move each shape from its initial position into the goal region. To reduce the number of actions, the agent selects transformations for one of the shapes at a time and uses a special "submit" action to move to the next shape. We treat the collection of automata as a single nondeterministic FSA, and specially mark the alphabet of the active automaton in the collection. Rewards are proportional to the overlap with the remaining exposed target shape when the submit action is used. On all other steps, a reward of -0.01 is given to encourage promptness.



(13.1) translate right by 1/2

(13.2) reflect about $y = 1/2$

Fig. 13. FSTs implementing some rigid transformations on the unit square. Arbitrary digits are represented by d, while $*$ represents arbitrary pairs of digits.

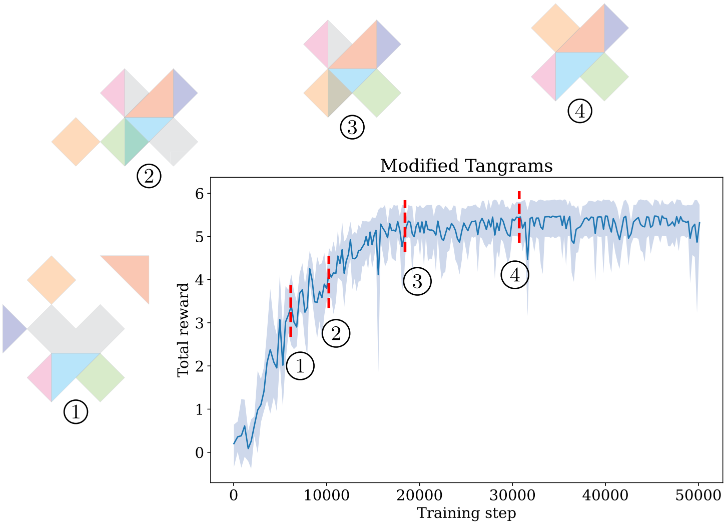


Fig. 14. Annotated reward curve for the modified tangram example.

The hyperparameters used for PPO in this case study were 256 steps per update, a 64 batch size, 10 optimization epochs, a clip range of $\epsilon = 0.1$, and a discount factor of $\lambda = 0.99$. The resulting reward curve—which we annotate at various points to show the agent's progress—is shown in Fig. 14.

6 Conclusion

This paper introduced a framework for symbolic reinforcement learning, dubbed *regular reinforcement learning*, where system states are modeled as regular languages and transition relations are modeled as rational transductions. We established theoretical results about the limitations and capabilities of this framework, proving that optimal values and policies are approximable and efficiently learnable under discounted payoffs. Furthermore, we developed an approach to deep regular reinforcement learning that combines aspects of deep learning and symbolic representation via the use of graph neural networks. Through a variety of case studies, we illustrated the effectiveness of deep regular reinforcement learning.

Acknowledgements. This work was supported in part by the NSF through grant CCF-2009022 and the NSF CAREER award CCF-2146563.

References

1. Abadi, E., Brafman, R.I.: Learning and solving regular decision processes. In: International Joint Conference on Artificial Intelligence, IJCAI, pp. 1948–1954. ijcai.org (2020). https://doi.org/10.24963/ijcai.2020/270
2. Abdulla, P.A.: Regular model checking. Int. J. Softw. Tools Technol. Transfer **14**(2), 109–118 (2012). https://doi.org/10.1007/S10009-011-0216-8
3. Abdulla, P.A.: Regular model checking: evolution and perspectives. In: Olderog, E.-R., Steffen, B., Yi, W. (eds.) Model Checking, Synthesis, and Learning. LNCS, vol. 13030, pp. 78–96. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91384-7_5
4. Abdulla, P.A., Jonsson, B., Mahata, P., d'Orso, J.: Regular tree model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 555–568. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_47
5. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_3
6. Abdulla, P.A., Legay, A., d'Orso, J., Rezine, A.: Tree regular model checking: a simulation-based approach. J. Logic Algebraic Program. **69**(1–2), 93–121 (2006). https://doi.org/10.1016/j.jlap.2006.02.001
7. Agarwal, A., Jiang, N., Kakade, S.M., Sun, W.: Reinforcement learning: Theory and algorithms. CS Department, UW Seattle, Seattle, WA, USA, Technical Report **32**, 96 (2019)
8. Baier, C., Bertrand, N., Schnoebelen, P.: On computing fixpoints in well-structured regular model checking, with applications to lossy channel systems. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 347–361. Springer, Heidelberg (2006). https://doi.org/10.1007/11916277_24
9. Bause, F., Kritzinger, P.S.: Stochastic Petri Nets - an Introduction to the Theory, 2nd edn. Vieweg (2002)
10. Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al.: Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680 (2019)
11. Boigelot, B.: On iterating linear transformations over recognizable sets of integers. Theoret. Comput. Sci. **309**(1–3), 413–468 (2003). https://doi.org/10.1016/S0304-3975(03)00314-1
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. Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying programs with dynamic 1-selector-linked structures in regular model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 13–29. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_2
15. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_5

16. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. Int. J. Softw. Tools Technol. Transfer **14**(2), 167–191 (2012). https://doi.org/10.1007/s10009-011-0205-y
17. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_29
18. 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
19. Bouajjani, A., Touili, T.: Widening techniques for regular tree model checking. Int. J. Softw. Tools Technol. Transfer **14**(2), 145–165 (2012). https://doi.org/10.1007/s10009-011-0208-8
20. Brafman, R.I., Giacomo, G.D.: Regular decision processes: A model for non-markovian domains. In: International Joint Conference on Artificial Intelligence, IJCAI, pp. 5516–5522. ijcai.org (2019). https://doi.org/10.24963/ijcai.2019/766
21. Brafman, R.I., Giacomo, G.D.: Regular decision processes: modelling dynamic systems without using hidden variables. In: International Conference on Autonomous Agents and MultiAgent Systems, AAMAS, pp. 1844–1846. International Foundation for Autonomous Agents and Multiagent Systems (2019). http://dl.acm.org/citation.cfm?id=3331938
22. Camacho, A., Varley, J., Zeng, A., Jain, D., Iscen, A., Kalashnikov, D.: Reward machines for vision-based robotic manipulation. In: International Conference on Robotics and Automation, ICRA, pp. 14284–14290. IEEE (2021). https://doi.org/10.1109/ICRA48506.2021.9561927
23. Chatterjee, K., Fu, H., Goharshady, A.K., Okati, N.: Computational approaches for stochastic shortest path on succinct MDPs. In: International Joint Conference on Artificial Intelligence, IJCAI, pp. 4700–4707. ijcai.org (2018). https://doi.org/10.24963/ijcai.2018/653
24. Corazza, J., Gavran, I., Neider, D.: Reinforcement learning with stochastic reward machines. In: Conference on Artificial Intelligence, AAAI vol. 36, no. 6, pp. 6429–6436 (2022). https://doi.org/10.1609/aaai.v36i6.20594
25. Dann, M., Yao, Y., Alechina, N., Logan, B., Thangarajah, J.: Multi-agent intention progression with reward machines. In: International Joint Conference on Artificial Intelligence, IJCAI, pp. 215–222. ijcai.org (2022). https://doi.org/10.24963/ijcai.2022/31
26. Dave, V., Dohmen, T., Krishna, S.N., Trivedi, A.: Regular model checking with regular relations. In: Bampis, E., Pagourtzis, A. (eds.) FCT 2021. LNCS, vol. 12867, pp. 190–203. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86593-1_13
27. Delgado, K.V., Sanner, S., De Barros, L.N.: Efficient solutions to factored MDPs with imprecise transition probabilities. Artif. Intell. **175**(9–10), 1498–1527 (2011). https://doi.org/10.1016/j.artint.2011.01.001
28. Dijkstra, E.W.: Algol 60 translation: An algol 60 translator for the x1 and making a translator for algol 60. Stichting Mathematisch Centrum. Rekenafdeling (MR 34/61) (1961)
29. Dohmen, T., Topper, N., Atia, G.K., Beckus, A., Trivedi, A., Velasquez, A.: Inferring probabilistic reward machines from non-Markovian reward signals for reinforcement learning. In: International Conference on Automated Planning and Scheduling, ICAPS, pp. 574–582. AAAI Press (2022). https://doi.org/10.1609/icaps.v32i1.19844

30. Feinberg, E.A.: Total expected discounted reward MDPs: existence of optimal policies (2011). https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0906
31. Goodfellow, I.J., Bengio, Y., Courville, A.C.: Deep Learning. Adaptive Computation and Machine Learning, MIT Press (2016). http://www.deeplearningbook.org/
32. Guestrin, C., Koller, D., Parr, R., Venkataraman, S.: Efficient solution algorithms for factored MDPs. J. Artif. Intell. Res. **19**, 399–468 (2003). https://doi.org/10.1613/jair.1000
33. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. In: International Workshop on Verification of Infinite-State Systems, INFINITY, pp. 21–36. Electronic Notes in Theoretical Computer Science, Elsevier (2004). https://doi.org/10.1016/j.entcs.2005.01.044
34. Icarte, R.T.: Reward Machines. Ph.D. thesis, University of Toronto, Canada (2022). http://hdl.handle.net/1807/110754
35. Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: Using reward machines for high-level task specification and decomposition in reinforcement learning. In: International Conference on Machine Learning, ICML. Proceedings of Machine Learning Research, vol. 80, pp. 2112–2121. PMLR (2018). http://proceedings.mlr.press/v80/icarte18a.html
36. Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: Reward machines: exploiting reward function structure in reinforcement learning. J. Artif. Intell. Res. **73**, 173–208 (2022). https://doi.org/10.1613/jair.1.12440
37. Icarte, R.T., Waldie, E., Klassen, T.Q., Valenzano, R.A., Castro, M.P., McIlraith, S.A.: Learning reward machines for partially observable reinforcement learning. In: Conference on Neural Information Processing Systems, NeurIPS, pp. 15497–15508 (2019). https://proceedings.neurips.cc/paper/2019/hash/532435c44bec236b471a47a88d63513d-Abstract.html
38. Jonsson, B., Saksena, M.: Systematic acceleration in regular model checking. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 131–144. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_16
39. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations, ICLR. OpenReview.net (2017). https://openreview.net/forum?id=SJU4ayYgl
40. Legay, A.: Extrapolating (omega-)regular model checking. Int. J. Softw. Tools Technol. Transfer **14**(2), 119–143 (2012). https://doi.org/10.1007/s10009-011-0209-7
41. Legay, A., Wolper, P.: On (omega-)regular model checking. ACM Trans. Computat. Logic **12**(1), 2:1–2:46 (2010). https://doi.org/10.1145/1838552.1838554
42. Lenaers, N., van Otterlo, M.: Regular decision processes for grid worlds. In: Leiva, L.A., Pruski, C., Markovich, R., Najjar, A., Schommer, C. (eds.) Benelux Conference on Artificial Intelligence, BNAIC/Benelearn. Communications in Computer and Information Science, vol. 1530, pp. 218–238. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-93842-0_13
43. 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
44. Mnih, V., et al.: Human-level control through reinforcement learning. Nature **518**, 529–533 (2015)

45. Neider, D., Jansen, N.: Regular model checking using solver technologies and automata learning. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 16–31. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_2
46. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Am. Math. Soc. **74**(2), 358–366 (1953)
47. Ronca, A., Giacomo, G.D.: Efficient PAC reinforcement learning in regular decision processes. In: International Joint Conference on Artificial Intelligence, IJCAI, pp. 2026–2032. ijcai.org (2021). https://doi.org/10.24963/ijcai.2021/279
48. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR abs/1707.06347 arxiv:1707.06347 (2017)
49. Silver, D., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**, 484–489 (2016)
50. Sipser, M.: Introduction to the Theory of Computation, chap. 1. PWS Publishing Company (1997)
51. Sutton, R.S., Barto, A.G.: Reinforcement learning - an Introduction. Adaptive Computation and Machine Learning, MIT Press (1998). https://www.worldcat.org/oclc/37293240
52. Touili, T.: Regular model checking using widening techniques. In: Verification of Parameterized Systems, VEPAS 2001, Satellite Workshop of ICALP, pp. 342–356. Electronic Notes in Theoretical Computer Science, Elsevier (2001). https://doi.org/10.1016/S1571-0661(04)00187-2
53. Valiant, L.G.: A theory of the learnable. In: Symposium on Theory of Computing, STOC, pp. 436–445. ACM (1984). https://doi.org/10.1145/800057.808710
54. Watkins, C.J.C.H., Dayan, P.: Technical note q-learning. Mach. Learn. **8**, 279–292 (1992). https://doi.org/10.1007/BF00992698
55. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028736
56. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. **32**(1), 4–24 (2021). https://doi.org/10.1109/TNNLS.2020.2978386
57. Xu, Z., Gavran, I., Ahmad, Y., Majumdar, R., Neider, D., Topcu, U., Wu, B.: Joint inference of reward machines and policies for reinforcement learning. In: International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020, pp. 590–598. AAAI Press (2020). https://doi.org/10.1609/icaps.v30i1.6756
58. Xu, Z., Wu, B., Ojha, A., Neider, D., Topcu, U.: Active finite reward automaton inference and reinforcement learning using queries and counterexamples. In: Holzinger, A., Kieseberg, P., Tjoa, A.M., Weippl, E. (eds.) CD-MAKE 2021. LNCS, vol. 12844, pp. 115–135. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84060-0_8
59. Zhou, W., Li, W.: A hierarchical Bayesian approach to inverse reinforcement learning with symbolic reward machines. In: International Conference on Machine Learning, ICML. Proceedings of Machine Learning Research, vol. 162, pp. 27159–27178. PMLR (2022). https://proceedings.mlr.press/v162/zhou22b.html

LTL Learning on GPUs

Mojtaba Valizadeh[1,2], Nathanaël Fijalkow[3], and Martin Berger[1,4](✉)

[1] University of Sussex, Brighton, UK
[2] Neubla UK Ltd., Cambridge, UK
[3] CNRS, LaBRI and Université de Bordeaux, Bordeaux, France
[4] Montanarius Ltd., London, UK
contact@martinfriedrichberger.net

Abstract. Linear temporal logic (LTL) is widely used in industrial verification. LTL formulae can be learned from traces. Scaling LTL formula learning is an open problem. We implement the first GPU-based LTL learner using a novel form of enumerative program synthesis. The learner is sound and complete. Our benchmarks indicate that it handles traces at least 2048 times more numerous, and on average at least 46 times faster than existing state-of-the-art learners. This is achieved with, among others, a branch-free implementation of LTL that has $O(\log n)$ time complexity, where n is trace length, while previous implementations are $O(n^2)$ or worse (assuming bitwise boolean operations and shifts by powers of 2 have unit costs—a realistic assumption on modern processors).

1 Introduction

Program verification means demonstrating that an implementation exhibits the behaviour required by a specification. But where do specifications come from? Handcrafting specifications does not scale. One solution is automatically to *learn* them from example runs of a system. This is sometimes referred to as trace analysis. A trace, in this context, is a sequence of events or states captured during the execution of a system. Once captured, traces are often converted into a form more suitable for further processing, such as finite state automata or logical formulae. Converting traces into logical formulae can be done with program synthesis. Program synthesis is an umbrella term for the algorithmic generation of programs (and similar formal objects, like logical formulae) from specifications, see [9,16] for an overview. Arguably, the most popular logic for representing traces is linear temporal logic (LTL) [30], a modal logic for specifying properties of finite or infinite traces. The *LTL learning problem* idealises the algorithmic essence of learning specifications from example traces, and is given as follows.

- **Input:** Two sets P and N of traces over a fixed alphabet.
- **Output:** An LTL formula ϕ that is (i) *sound*: all traces in P are accepted by ϕ, all traces in N are rejected by ϕ; (ii) *minimal*, meaning no strictly smaller sound formula exists.

All code and benchmarks are available at [1].

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 209–231, 2024.
https://doi.org/10.1007/978-3-031-65633-0_10

When we weaken minimality to minimality-up-to-ϵ, we speak of *approximate* LTL learning. Both forms of LTL learning are NP-hard [11,27]. A different and simpler problem is *noisy* LTL learning, which is permitted to learn unsound formulae, albeit only up-to a give error-rate.

LTL learning is an active research area in software engineering, formal methods, and artificial intelligence [2,5–7,12–15,19,20,23–26,28,29,32,34–36]. We refer to [6] for a longer discussion. Many approaches to LTL learning have been explored. One common and natural method involves using search-based program synthesis, often paired with templates or sketches, such as parts of formulae, automata, or regular expressions. Another leverages SAT solvers. LTL learning is also being pursued using Bayesian inference, or inductive logic programming. Learning specifically tailored small fragments of LTL often yields the best results in practice [31,32]. Learning from noisy data is investigated in [15,26,29]. All have in common is that they don't scale, and have not been optimised for GPUs. Traces arising in industrial practice are commonly long (millions of characters), and numerous (millions of traces). Extracting useful information automatically at such scale is currently a major problem, e.g., the state-of-the-art learner in [31,32] cannot reliably learn formulae greater than size 10. This is less than ideal. Our aim is to change this.

Graphics Processing Units (GPUs) are the work-horses of high-performance computing. The acceleration they provide to applications compatible with their programming paradigm can surpass CPU performance by several orders of magnitude, as notably evidenced by the advancements in deep learning. A significant spectrum of applications, especially within automated reasoning-like SAT/SMT solvers and model checkers-has yet to reap the benefits of GPU acceleration. In order for an application to be "GPU-friendly", it needs to have high parallelism, minimal data-dependent branching, and predictable data movement with substantial data locality [8,17,18]. Current automated reasoning algorithms are predominantly branching-intensive and appear sequential in nature, but it is unclear whether they are inherently sequential, or can be adapted to GPUs.

> **Research question.** *Can we scale LTL learning to at least 1000 times more traces without sacrificing trace length, learning speed or approximation ratio (cost increase of learned formula over minimum) compared to existing work, by employing suitably adapted algorithms on a GPU?*

We answer the RQ in the affirmative by developing the first GPU-accelerated LTL learner. Our work takes inspiration from [33], the first GPU-accelerated minimal regular expression inferencer. Scaling has two core orthogonal dimensions: more traces, and longer traces. We solve one problem [33] left open: scaling to more traces. Our key decision, giving up on learning minimal formula while remaining sound and complete, enables two principled algorithmic techniques.

- **Divide-and-conquer (D&C).** If a learning task has too many traces, split it into smaller specifications, learn those recursively, and combine the learned formulae using logical connectives.

- **Relaxed uniqueness checks (RUCs).** Often generate-and-test program synthesis caches synthesis results to avoid recomputation. [33] granted cache admission only after a uniqueness check. We relax uniqueness checking by (pseudo-)randomly rejecting some unique formulae.

In addition, we design novel algorithms and data structures, representing LTL formulae as contiguous matrices of bits. This allows a GPU-friendly implementation of all logical operations with linear memory access and suitable machine instructions, free from data-dependent branching. Both D&C and RUCs may lose minimality and are thus unavailable to [33]. Our benchmarks show that the approximation ratio is typically small.

Contributions. In summary, our contributions are as follows:

- A new enumeration algorithm for LTL learning, with a branch-free implementation of LTL semantics that is $O(\log n)$ in trace length (assuming unit cost for logical and shift operations).
- A CUDA implementation of the algorithm, for benchmarking and inspection.
- A parameterised benchmark suite useful for evaluating the performance of LTL learners, and a novel methodology for quantifying the loss of minimality induced by approximate LTL learning.
- Performance benchmarks showing that our implementation is both faster, and can handle orders of magnitude more traces, than existing work.

2 Formal Preliminaries

We write $\#S$ for the cardinality of set S. $\mathbb{N} = \{0, 1, 2, ...\}$, $[n]$ is for $\{0, 1, ..., n-1\}$ and $[m, n]$ for $\{m, m+1, ..., n-1\}$. $\mathbb{B}$ is $\{0, 1\}$ where 0 is falsity and 1 truth. $\mathfrak{P}(A)$ is the *powerset* of A. The *characteristic function* of a set S is the function $\mathbf{1}_S^A : A \to \mathbb{B}$ which maps $a \in A$ to 1 iff $a \in S$. We usually write $\mathbf{1}_S$ for $\mathbf{1}_S^A$. An *alphabet* is a finite, non-empty set Σ, the elements of which are *characters*. A *string of length* $n \in \mathbb{N}$ over Σ is a map $w : [n] \to \Sigma$. We write $\|w\|$ for n. We often write w_i instead of $w(i)$, and $v \cdot w$, or just vw, for the concatenation of v and w, ϵ for the empty string and Σ^* for all strings over Σ. A *trace* is a string over powerset alphabets, *i.e.*, $(\mathfrak{P}(\Sigma))^*$. We call Σ the *alphabet* of the trace and write $\mathsf{traces}(\Sigma)$ for all traces over Σ. A *word* is a trace where each character has cardinality 1. We abbreviate words to the corresponding strings, e.g., $\langle\{t\}, \{i\}, \{n\}\rangle$ to *tin*. We say v is a *suffix* of w if $w = uv$, and if $\|u\| = 1$ then v is an *immediate* suffix. We write $\mathsf{sc}(S)$ for the *suffix-closure* of S. S is *suffix-closed* if $\mathsf{sc}(S) \subseteq S$. $\mathsf{sc}^+(S)$ is the *non-empty suffix closure* of S, *i.e.*, $\mathsf{sc}(S) \setminus \{\epsilon\}$. *From now on we will speak of the suffix-closure to mean the non-empty suffix closure.* The *Hamming-distance* between two strings s and t of equal length, written $\mathsf{hamm}(s, t)$, is the number of indices i where $s(i) \neq t(i)$. We write $\mathsf{Hamm}(s, \delta)$ for the set $\{t \in \Sigma^* \mid \mathsf{hamm}(s, t) = \delta, \|s\| = \|t\|\}$.

LTL formulae over $\Sigma = \{p_1, ..., p_n\}$ are given by the following grammar.

$$\phi \quad ::= \quad p_i \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{X}\phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid \phi \,\mathsf{U}\, \phi$$

The *subformulae* of ϕ are denoted $\mathsf{sf}(\phi)$. We say $\psi \in \mathsf{sf}(\phi)$ is *proper* if $\phi \neq \psi$. A formula is in *negation normal form* (NNF) if all subformulae containing negation are of the form $\neg p$. It is U*-free* if no subformula is of the form $\phi \ \mathsf{U} \ \psi$. We write $\mathsf{LTL}(\Sigma)$ for the set of all LTL formulae over Σ. We use true as an abbreviation for $p \vee \neg p$ and false for $p \wedge \neg p$. We call $\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}$ the *temporal* connectives, $\wedge, \vee, \neg$ the *propositional* connectives, p the *atomic* propositions and, collectively name them the *LTL connectives*. Since we learn from finite traces, we interpret LTL over finite traces [10]. The satisfaction relation $tr, i \models \phi$, where tr is a trace over Σ and ϕ from $\mathsf{LTL}(\Sigma)$ is standard, here are some example clauses: $tr, i \models \mathsf{X}\phi$, if $tr, i+1 \models \phi$, $tr, i \models \mathsf{F}\phi$, if there is $i \leq j < \|tr\|$ with $tr, j \models \phi$, and $tr, i \models \phi \ \mathsf{U} \ \phi'$, if there is $i \leq j < \|tr\|$ such that: $tr, k \models \phi$ for all $i \leq k < j$, and $tr, j \models \phi'$. If $i \geq \|tr\|$ then $tr, i \models \phi$ is always false, and $tr \models \phi$ is short for $tr, 0 \models \phi$.

A *cost-homomorphism* is a map $\mathsf{cost}(\cdot)$ from LTL connectives to positive integers. We extend it to LTL formulae homomorphically: $\mathsf{cost}(\phi \ op \ \psi) = \mathsf{cost}(op) + \mathsf{cost}(\phi) + \mathsf{cost}(\psi)$, and likewise for other arities. If $\mathsf{cost}(op) = 1$ for all LTL connectives we speak of *uniform cost*. So the uniform cost of true and false is 4. *From now on all cost-homomorphisms will be uniform, except where stated otherwise.*

A *specification* is a pair (P, N) of finite sets of traces such that $P \cap N = \emptyset$. We call P the *positive* examples and N the *negative* examples. We say ϕ *satisfies, separates* or *solves* (P, N), denoted $\phi \models (P, N)$, if for all $tr \in P$ we have $tr \models \phi$, and for all $tr \in N$ we have $tr \not\models \phi$. A *sub-specification* of (P, N) is any specification (P', N') such that $P' \subseteq P$ and $N' \subseteq N$. Symmetrically, (P, N) is an *extension* of (P', N'). We can now make the *LTL learning problem* precise:

- **Input:** A specification (P, N), and a cost-homomorphism $\mathsf{cost}(\cdot)$.
- **Output:** An LTL formula ϕ that is *sound*, *i.e.*, $\phi \models (P, N)$, and *minimal*, *i.e.*, $\psi \models (P, N)$ implies $\mathsf{cost}(\phi) \leq \mathsf{cost}(\psi)$.

Cost-homomorphisms let us influence LTL learning: e.g., by assigning a high cost to a connective, we prevent it from being used in learned formulae. The *language at i* of ϕ, written $\mathsf{lang}(i, \phi)$, is $\{tr \in \mathsf{traces}(\Sigma) \mid tr, i \models \phi\}$. We write $\mathsf{lang}(\phi)$ as a shorthand for $\mathsf{lang}(0, \phi)$ and speak of the *language* of ϕ. We say ϕ *denotes* a language $S \subseteq \Sigma^*$, resp., a trace $tr \in \Sigma^*$, if $\mathsf{lang}(\phi) = S$, resp., $\mathsf{lang}(\phi) = \{tr\}$. We say two formulae ϕ_1 and ϕ_2 are *observationally equivalent*, written $\phi_1 \simeq \phi_2$, if they denote the same language. Let S be a set of traces. Then we write

$$\phi_1 \simeq \phi_2 \mod S \qquad \text{iff} \qquad \mathsf{lang}(\phi_1) \cap S = \mathsf{lang}(\phi_2) \cap S$$

and say ϕ_1 and ϕ_2 are *observationally equivalent modulo S*. The following related definitions will be useful later. Let (P, N) be a specification. The *cardinality of* (P, N), denoted $\#(P, N)$, is $\#P + \#N$. The *size* of a set S of traces, denoted $\|S\|$ is $\Sigma_{tr \in S}\|tr\|$. We extend this to specifications: $\|(P, N)\|$ is $\|P\| + \|Q\|$. The *cost* of a specification (P, N), written $\mathsf{cost}(P, N)$ is the uniform cost of a minimal sound formula for (P, N). An extension of (P, N) is *conservative* if any minimal sound formula for (P, N) is also minimal and sound for the extension. We note a useful fact: if ϕ is a minimal solution for (P, N), and also $\phi \models (P', N')$ then $(P \cup P', N \cup N')$ is a conservative extension.

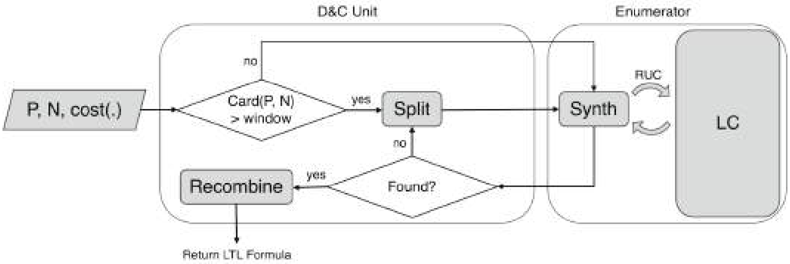


Fig. 1. High-level structure of our algorithm. LC is short for language cache.

Overfitting. It is possible to express a trace tr, respectively a set S of traces, by a formula ϕ, in the sense that $\mathsf{lang}(\phi) = \{tr\}$, resp., $\mathsf{lang}(\phi) = S$. We define the function $\mathsf{overfit}(\cdot)$ on sets of characters, traces, sets and specifications as follows.

- $\mathsf{overfit}(\{a_1, ..., a_k\}) = (\bigwedge_i a_i) \land \bigwedge_{b \in \Sigma \setminus \{a_1,...,a_k\}} \neg b.$
- $\mathsf{overfit}(\epsilon) = \neg \mathsf{X}(\mathsf{true})$ and $\mathsf{overfit}(a \cdot tr) = \mathsf{overfit}(a) \land \mathsf{X}(\mathsf{overfit}(tr))$.
- $\mathsf{overfit}(S) = \bigvee_{tr \in S} \mathsf{overfit}(tr)$
- $\mathsf{overfit}(P, N) = \mathsf{overfit}(P)$

The following are immediate from the definitions: (i) For all specifications (P, N): $\mathsf{lang}(\mathsf{overfit}(P, N)) = P$, (ii) $\mathsf{overfit}(P, N) \models (P, N)$, and (iii) the *cost of overfitting*, *i.e.*, $\mathsf{cost}(\mathsf{overfit}(P, N))$, is $O(\|P\| + \#\Sigma)$. Note that $\mathsf{overfit}(P, N)$ is overfitting only on P, and (ii) justifies this choice.

3 High-Level Structure of the Algorithm

Figure 1 shows the two main parts of our algorithm: the *divide-and-conquer unit*, short D&C-unit, and the *enumerator*. Currently, only the enumerator is implemented for execution on a GPU. For convenience, our D&C-unit is in Python and runs on a CPU. Implementing the *D&C*-unit on a GPU poses no technical challenges and would make our implementation perform better.

Given (P, N), the D&C-unit checks if the specification is small enough to be solved by the enumerator directly. If not, the specification is recursively decomposed into smaller sub-specifications. When the recursive invocations return formulae, the D&C-unit combines them into a formula separating (P, N), see §7 for details. For small enough (P, N), the enumerator performs a bottom-up enumeration of LTL formulae by increasing cost, until it finds one that separates (P, N). Like the enumerator in [33], our enumerator uses a language cache to minimise re-computation, but with a novel cache admission policy (RUCs). The language cache is append-only, hence no synchronisation is required for read-access. The key difference from [33], our use of RUCs, is discussed in §4.

The enumerator has three core parameters.

- T = maximal number of traces in the specification (P, N).
- L = number of bits usable for representing each trace from (P, N) in memory.

- W = number of bits (P, N) hashed *to* during enumeration.

We write Enum(T, L, W) to emphasise those parameters. Our current implementation hard-codes all parameters as Enum(64, 64, 128)[1], but the abstract algorithm does not depend on this. The choice of $W = 128$ is a consequence of the current limitations of WarpCore [21,22], a CUDA library for high-performance hashing of 32 and 64 bit integers. All three parameters heavily affect memory consumption. We chose $T = 64$ and $L = 64$ for convenient comparison with existing work in §8. While T, L and M are parameters of the abstract algorithm, the implementation is not parameterised: changing these parameters requires changing parts of the code. Making the implementation fully parametric is conceptually straightforward, but introduces a substantial number of new edge cases, primarily where parameters are not powers of 2, which increases verification effort.

We now sketch the high-level structure of `enum`, the entry point of the enumerator, taking a specification and a cost-homomorphism as arguments. For ease of presentation, we use LTL formulae as search space. Their representation in the implementation is discussed in §4.

```
1   language_cache = []
2
3   def enum(p, n, cost):
4       if (p, n) can be solved with Atom then return Atom
5       language_cache.append([Atom])
6       for c in range(cost(Atom)+1, cost(overfit(p, n))):
7           language_cache.append([])
8           for op in [F, U, G, X, And, Or, Not]:
9               handleOp(op, p, n, c, cost)
10      return overfit(p, n)
```

Line 4 checks if the learning problem can be solved with an atomic proposition. If not, Line 5 initialises the global language cache with the representation of atomic propositions, and search starts from the lowest cost upwards. For each cost c a new empty entry is added to the language cache. Line 8 then maps over LTL connectives and calls `handleOp` to construct all formulae of cost c using all suitable lower cost entries in the language cache. When no sound formula can be found with cost less than $\mathsf{cost}(\mathsf{overfit}(P, N))$, the algorithm terminates, returning $\mathsf{overfit}(P, N)$. This makes our algorithm *complete*, in the sense of learning a formula for every specification.

```
11  def handleOp(op, p, n, c, cost):
12      match op:
13          case F:
14              for all phi in language_cache(c-cost(F)):    # parallel
15                  phi_new = branchfree_F(phi)
16                  relaxedCheckAndCache(p, n, c, phi_new)
17          case U:
18              for all (cL, cR) in split(c-cost(U)):        # parallel, split(x) gives all (i,j) s.t. i+j
                        = x
19                  for all phi_L in language_cache(cL):     # parallel
20                      for all phi_R in language_cache(cR): # parallel
21                          phi_new = branchfree_U(phi_L, phi_R)
22                          relaxedCheckAndCache(p, n, c, phi_new)
23          case G: ...
24          case ...
```

The function `handleOp` dispatches on LTL connectives, retrieves all previously constructed formulae of suitable cost from the language cache in parallel (we use

[1] For pragmatic reasons, our implementation uses only 126 bits of W=128, and 63 bits of L = 64, details omitted for brevity.

`forall` to indicate parallel execution), calls the appropriate semantic function, detailed in the next section, e.g., `branchfree_F` for F, to construct `phi_new`, and then sends it to `relaxedCheckAndCache` to check if it already solves the learning task, and, if not, for potential caching. Most parallelism in our implementation, and the upside of the language cache's rapid growth, is the concomitant growth in available parallelism, which effortlessly saturates every conceivable processor.

```
25 def relaxedCheckAndCache(p, n, c, phi_new): # c is a concrete cost
26     if phi_new |= (p, n):                   # check if candidate is sound for (p, n)
27         exit(phi_new)                       # Terminate learning, return phi_new as learned formula
28     if relaxedUniquenessCheck(phi_new, language_cache):
29         language_cache[c].append(phi_new)   # parallel
```

This last step checks if `phi_new` satisfies (P, N). If yes, the program terminates with the formula corresponding to `phi_new`. Otherwise, Line 28 conducts a RUC, a *relaxed* uniqueness check, described in detail in §6, to decide whether to cache `phi_new` or not. Updating the language cache in Line 29 is done in parallel, and needs little synchronisation, see [33] for details. The satisfaction check in Line 26 guarantees that our algorithm is *sound*. It also makes it trivial to implement noisy LTL learning: just replace the precise check $\mathtt{phi}_\mathtt{n}\mathtt{ew} \models (\mathtt{p}, \mathtt{n})$ with a check that `phi_new` gets a suitable fraction of the specification right.

4 In-Memory Representation of Search Space

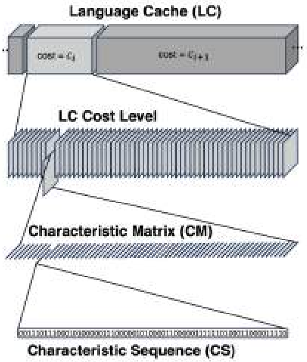


Fig. 2. Data representation in memory (simplified).

Our enumerator does generate-and-check synthesis. That means we have two problems: (i) minimising the cardinality of the search space, *i.e.*, the representation of LTL formulae during synthesis; (ii) making generation and checking as cheap as possible. For each candidate ϕ checking means evaluating the predicate

$$P \subseteq \mathsf{lang}(\phi) \text{ and } N \cap \mathsf{lang}(\phi) = \emptyset. \qquad (\dagger)$$

LTL formulae, the natural choice of search space, suffer from the redundancies of syntax: every language that is denoted by a formula at all, is denoted by infinitely many, e.g., $\mathsf{lang}(\mathsf{F}\phi) = \mathsf{lang}(\mathsf{FF}\phi)$. Even observational equivalence distinguishes too many formulae: the predicate (†) checks the language of ϕ only for elements of $P \cup N$. Formulae modulo $P \cup N$ contain exactly the right amount of information for (†), hence minimise the search space. However, the semantics of formulae on $P \cup N$ is given compositionally in terms of the non-empty suffix-closure of $P \cup N$, which would have to be recomputed at run-time for each new candidate. Since $P \cup N$ remains fixed, so does $\mathsf{sc}^+(P \cup N)$, and we can avoid such re-computation by using formulae modulo $\mathsf{sc}^+(P \cup N)$ as search space. Inspired by [33], we represent formulae ϕ by characteristic functions $\mathbf{1}_{\mathsf{lang}(\phi)} : \mathsf{sc}^+(P \cup N) \to \mathbb{B}$, which are implemented as contiguous bitvectors in memory, but with a twist. Fix a total order on $P \cup N$.

- A *characteristic sequence (CS) for ϕ over tr* is a bitvector cs such that $tr, j \models \phi$ iff $cs(j) = 1$. For Enum(64, 64, 128), CSs are unsigned 64 bit integers.
- A *characteristic matrix (CM) representing ϕ over $P \cup N$*, is a sequence cm of CSs, contiguous in memory, such that, if tr is the ith trace in the order, then $cm(i)$ is the CS for ϕ over tr.

This representation has two interesting properties, not present in [33]: (i) each CS is *suffix-contiguous*: the trace corresponding to $cs(j+1)$ is the immediate suffix of that at $cs(j)$; (ii) CMs contain *redundancies* whenever two traces in $P \cup N$ share suffixes. Redundancy is the price we pay for suffix-contiguity. Figure 2 visualises our representation in memory.

Logical Operations as Bitwise Operations. Suffix-contiguity enables the efficient representation of logical operations: if $cs = 10011$ represents ϕ over the word *abcaa*, e.g., ϕ is the atomic proposition a, then $\mathsf{X}\phi$ is 00110, *i.e.*, cs shifted one to the left. Likewise $\neg\phi$ is represented by 01100, *i.e.*, bitwise negation. As we use unsigned 64 bit integers to represent CSs, X and negation are executed as *single* machine instructions! In Python-like pseudo-code:

```
def branchfree_X(cm):
    return [cs << 1 for cs in cm]
```

```
def branchfree_Not(cm):
    return [~cs for cs in cm]
```

Conjunction and disjunction are equally efficient. More interesting is F which becomes the disjunction of shifts by *powers of two*, *i.e.*, the number of shifts is logarithmic in the length of the trace (a naive implementation of F is linear). We call this *exponential propagation*, and believe it to be novel in LTL synthesis[2]:

```
def branchfree_F(cm):
    outCm = []
    for cs in cm:
        cs |= cs << 1
        cs |= cs << 2
        cs |= cs << 4
        cs |= cs << 8
        cs |= cs << 16
        cs |= cs << 32
        outCm.append(cs)
    return outCm
```

```
def branchfree_U(cm1, cm2):
    outCm = []
    for i in range(len(cm1)):
        cs1 = cm1[i]
        cs2 = cm2[i]
        cs2 |= cs1 & (cs2 << 1)
        cs1 &= cs1 << 1
        cs2 |= cs1 & (cs2 << 2)
        cs1 &= cs1 << 2
        cs2 |= cs1 & (cs2 << 4)
        cs1 &= cs1 << 4
        cs2 |= cs1 & (cs2 << 8)
        cs1 &= cs1 << 8
        cs2 |= cs1 & (cs2 << 16)
        cs1 &= cs1 << 16
        cs2 |= cs1 & (cs2 << 32)
        outCm.append(cs2)
    return outCm
```

To see why this works, note that $\mathsf{F}\phi$ can be seen as the infinite disjunction $\phi \vee \mathsf{X}\phi \vee \mathsf{X}^2\phi \vee \mathsf{X}^3\phi \vee ...$, where $\mathsf{X}^n\phi$ is given by $X^0\phi = \phi$ and $X^{n+1}\phi = \mathsf{X}\mathsf{X}^n\phi$. Since we work with finite traces, $tr, i \not\models \phi$ whenever $i \geq \|tr\|$. Hence checking $tr, 0 \models \mathsf{F}\phi$ for tr of length n amounts to checking

$$tr, 0 \models \phi \vee \mathsf{X}\phi \vee \mathsf{X}^2\phi \vee ... \vee \mathsf{X}^{n-1}\phi$$

The key insight is that the imperative update $\texttt{cs} \mid= \texttt{cs} \ll \texttt{j}$ propagates the bit stored at $cs(i+j)$ into $cs(i)$ without removing it from $cs(i+j)$. Consider the

[2] By representing ϕ as a CS, *i.e.*, unsigned integer, we can also read $\mathsf{F}\phi$ as rounding up ϕ to the next bigger power of 2 and then subtracting 1, cf. [3].

flow of information stored in $cs(n-1)$. At the start, this information is only at index $n-1$. This amounts to checking $tr, n-1 \models \mathsf{X}^{n-1}\phi$. Thus assigning $\texttt{cs} \ \ |\!= \texttt{cs} \ll 1$ puts that information at indices $n-2, n-1$. This amounts to checking $tr, 0 \models \mathsf{X}^{n-2}\phi \vee \mathsf{X}^{n-1}\phi$. Likewise, then assigning $\texttt{cs} \ \ |\!= \texttt{cs} \ll 2$ puts that information at indices $n-4, n-3, n-2, n-1$. This amounts to checking $tr, 0 \models \mathsf{X}^{n-4}\phi \vee \mathsf{X}^{n-3}\phi \vee \mathsf{X}^{n-2}\phi \vee \mathsf{X}^{n-1}\phi$, and so on. In a logarithmic number of steps, we reach $tr, 0 \models \phi \vee \mathsf{X}\phi \vee ... \vee \mathsf{X}^{n-1}\phi$. This works uniformly for all positions, not just $n-1$. In the limit, this saves an exponential amount of work over naive shifting.

We can implement U using similar ideas, with the number of bitshifts also logarithmic in trace length. As with F, this works because we can see $\phi \,\mathsf{U}\, \psi$ as an infinite disjunction

$$\psi \vee (\phi \wedge \mathsf{X}\psi) \vee (\phi \wedge \mathsf{X}(\phi \wedge \mathsf{X}\psi)) \vee \dots$$

We define (informally) $\phi \,\mathsf{U}_{\leq p}\, \psi$ as: ϕ holds until ψ does within the next p positions, and $\mathsf{G}_{\geq p}\phi$ if ϕ holds for the next p positions. The additional insight allowing us to implement exponential propagation for U is to compute both, $\mathsf{G}_{\geq 2^i}\phi$ and $\phi \,\mathsf{U}_{\leq 2^i}\, \psi$, for increasing values of i at the same time.

In addition to saving work, exponential propagation maps directly to machine instructions, and is essentially branch-free code for all LTL connectives[3], thus maximises GPU-friendliness of our learner. In contrast, previous learners like Flie [28], Scarlet [32] and Syslite [4], implement the temporal connectives naively, e.g., checking $tr, i \models \phi \,\mathsf{U}\, \psi$ by iterating from i as long as ϕ holds, stopping as soon as ψ holds. Likewise, Flie encodes the LTL semantics directly as a propositional formula. For U this is quadratic in the length of tr for Flie and Syslite.

5 Correctness and Complexity of the Branch-Free Implementation of Temporal Operators

We reproduce here a slightly more general version (for any length) of the exponentially propagating algorithms for computing F and U.

```
def branchfree_F(cs):
    L = len(cs)
    for i in range(log(L)+1):
        cs |= cs << 2**i
    return cs
```

```
def branchfree_U(cs1, cs2):
    L = len(cs1)
    for i in range(log(L)+1):
        cs2 |= cs1 & (cs2 << 2**i)
        cs1 &= cs1 << 2**i
    return cs2
```

They are lifted to CMs pointwise. As a warm-up, let us start with F.

Lemma 1. *Let cs be the characteristic sequence for ϕ over a trace of length L. The algorithm above computes the characteristic sequence for $\mathsf{F}\phi$. Assuming bitwise boolean operations and shifts by powers of two have unit costs, the complexity of the algorithm is $O(\log(L))$.*

[3] Including, *mutatis mutandis*, past-looking temporal connectives.

To ease notations, let us introduce $\mathsf{F}_{\leq p}$ with the semantics $tr, j \models \mathsf{F}_{\leq p}\phi$ if there is $j \leq k < \min(j+p, \|tr\|)$ with $tr, k \models \phi$. The parameter $p \in \mathbb{N}$ in $\mathsf{F}_{\leq p}\phi$ can be read as the number of positions in tr where ϕ will be evaluated at, starting from the current position j in tr.

Proof. Let us write cs for the characteristic sequence for ϕ over tr, and cs_i for the characteristic sequence after the ith iteration. We write $\log(x)$ as a shorthand for $\lfloor \log_2(x) \rfloor$. We write $\mathsf{F}cs$ for $\mathsf{F}\phi$, and $\mathsf{F}_{\leq p}cs$ for $\mathsf{F}_{\leq p}\ \phi$. We show by induction that for all $i \in [0, \log(L)+1]$, for all tr of length L we have:

$$\forall j \in [0, L],\ tr, j \models cs_i \iff tr, j \models \mathsf{F}_{\leq 2^i} cs.$$

This is clear for $i = 0$, as it boils down to $cs_0 = cs$. Assuming it holds for i, by definition $cs_{i+1}(j) = cs_i(j) \vee (cs_i \ll 2^i)(j) = cs_i(j) \vee cs_i(j + 2^i)$, hence

$$\begin{aligned} tr, j \models cs_{i+1} &\iff tr, j \models cs_i, \text{or } tr, j + 2^i \models cs_i \\ &\iff tr, j \models \mathsf{F}_{\leq 2^i} cs, \text{or } tr, j + 2^i \models \mathsf{F}_{\leq 2^i} cs \\ &\iff tr, j \models \mathsf{F}_{\leq 2^{i+1}} cs. \end{aligned}$$

This concludes the induction proof. For $i = \log(L)$ we obtain

$$\forall j \in [0, L],\ tr, j \models cs_i \iff tr, j \models \mathsf{F}_{\leq L} cs \iff tr, j \models \mathsf{F}cs,$$

since clearly $\mathsf{F} = \mathsf{F}_{\leq L}$, when restricted to traces not exceeding L in length. □

We now move to U.

Lemma 2. *Let cs_1, cs_2 the characteristic sequences for ϕ_1 and ϕ_2, both over traces of length L. The algorithm above computes the characteristic sequence for $\phi_1\ \mathsf{U}\ \phi_2$. Assuming bitwise boolean operations and shifts by powers of two have unit costs, the complexity of the algorithm is $O(\log(L))$.*

Again to ease notations, let us introduce $\mathsf{U}_{\leq p}$ with the semantics $tr, j \models \phi_1\ \mathsf{U}_{\leq p}\ \phi_2$ if there is $j \leq k < \min(j + p, \|tr\|)$ such that $tr, k \models \phi_2$ and for all $i \leq k' < k$ we have $tr, k' \models \phi_1$. We will also need $\mathsf{G}_{\leq p}$ defined with the semantics $tr, j \models \mathsf{G}_{\leq p}\phi$ if for all $j \leq k < \min(j + p, \|tr\|)$ we have $tr, k \models \phi$.

Proof. Let us write $cs_{1,i}$ and $cs_{2,i}$ for the respective characteristic sequences after the ith iteration. We show by induction that for all $i \in [0, \log(L)+1]$, for all tr of length L, for all $j \in [0, L]$, we have:

- $tr, j \models cs_{1,i} \iff tr, j \models \mathsf{G}_{\leq 2^i} cs_1$, and
- $tr, j \models cs_{2,i} \iff tr, j \models cs_1\ \mathsf{U}_{\leq 2^i}\ cs_2$.

This is clear for $i = 0$, as it boils down to $cs_{1,0} = cs_1$ and $cs_{2,0} = cs_2$. Assume it holds for i. Let us start with $cs_{1,i+1}$: by definition

$$\begin{aligned} cs_{1,i+1}(j) &= cs_{1,i}(j) \wedge (cs_{1,i} \ll 2^i)(j) \\ &= cs_{1,i}(j) \wedge cs_{1,i}(j + 2^i), \end{aligned}$$

hence it is the case that

$$\begin{aligned} tr, j \models cs_{1,i+1} &\iff tr, j \models cs_{1,i}, \text{and } tr, j + 2^i \models cs_{1,i} \\ &\iff tr, j \models \mathsf{G}_{\leq 2^i} cs_1, \text{ and } tr, j + 2^i \models \mathsf{G}_{\leq 2^i} cs_1 \\ &\iff tr, j \models \mathsf{G}_{\leq 2^{i+1}} cs_1. \end{aligned}$$

Now, by definition $cs_{2,i+1}(j) = cs_{2,i}(j) \vee (cs_{1,i}(j) \wedge (cs_{2,i} \ll 2^i)(j))$, which is equal to $cs_{2,i}(j) \vee (cs_{1,i}(j) \wedge cs_{2,i}(j + 2^i))$. Hence

$$\begin{aligned} tr, j \models cs_{2,i+1} &\iff tr, j \models cs_{2,i}, \text{or } (tr, j \models cs_{1,i}, \text{ and } tr, j + 2^i \models cs_{2,i}) \\ &\iff tr, j \models cs_1 \mathsf{U}_{\leq 2^i} cs_2, \text{or} \\ &\qquad \big(tr, j \models \mathsf{G}_{\leq 2^i} cs_1, \text{and } tr, j + 2^i \models cs_1 \mathsf{U}_{\leq 2^i} cs_2\big) \\ &\iff tr, j \models cs_1 \mathsf{U}_{\leq 2^{i+1}} cs_2. \end{aligned}$$

This concludes the induction proof. For $i = \log(L)$ we obtain

$$\forall j \in [0, L],\ tr, j \models cs_{2,i} \iff tr, j \models cs_1 \mathsf{U}_{\leq L} cs_2 \iff tr, j \models cs_1 \mathsf{U} cs_2,$$

since clearly $\mathsf{U} = \mathsf{U}_{\leq L}$ for all sufficiently short traces. □

6 Relaxed Uniqueness Checks

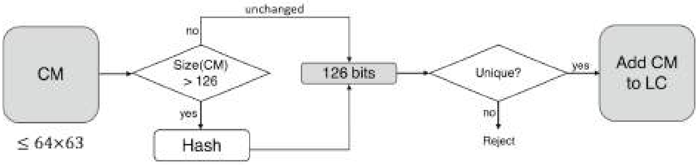
Our choice of search space, formulae modulo $\mathsf{sc}^+(P \cup N)$, while more efficient than bare formulae, still does not prevent the explosive growth of candidates: uniqueness of CMs is *not* preserved under LTL connectives. [33] recommends storing newly synthesised formulae in a "language cache", but only if they pass a uniqueness check. Without this cache admission policy, the explosive growth of redundant CMs rapidly swamps the language cache with useless repetition. While uniqueness improves scalability, it just delays the inevitable: there are simply too many unique CMs. Worse: with Enum(64, 64, 128), CMs use up-to 32 times more memory than language cache entries in [33]. We improve memory consumption of our algorithm by relaxing strictness of uniqueness checks: we allow false positives (meaning that CMs are falsely classified as being already in the language cache), but not false negatives. We call this new cache admission policy *relaxed uniqueness checks* (RUCs). False positives mean that less gets cached. False positives are sound: every formula learned in the presence of false positives is separating, but no longer necessarily minimal—every minimal solution might have some of its subformulae missing from the language cache, hence cannot be constructed by the enumeration. False positives also do not affect completeness: in the worst case, our algorithm terminates by overfitting.

We implement the RUC using non-cryptographic hashing in several steps.

- We treat each CM as a big bitvector, *i.e.*, ignore its internal structure. Now there are two possibilities.
    - The CM uses more than 126 bits. Then we hash it to 126 bits using a variant of MuellerHash from WarpCore.

  - Otherwise we leave the CM unchanged (except padding it with 0 s to 126 bits where necessary).
- Only if this 126 bit sequence is unique, it is added to the language cache.

If the CM is $\leq$ 126 bits, then the RUC is precise and enumeration performs a full bottom-up enumeration of CMs, so any learned formula is minimal cost. This becomes useful in benchmarking.



Note that RUCs implemented by hashing amount to a (pseudo-)random cache admission policy. Using RUCs essentially means that hash-collisions (pseudo-) randomly prevent formulae from being subformulae of any learned solution. It is remarkable that this works well in practice, but it probably means that LTL has sufficient redundancy in formulae vis-a-vis the probability of hash collisions. We leave a detailed theoretical analysis as future work.

7 Divide & Conquer

The D&C-unit's job is, recursively, to split specifications until they are small enough to be solved by Enum(T, L, W) in one go, and, afterwards recombine the results. A naive D&C-strategy could split (P, N), when needed, into four smaller specifications (P_i, N_j) for $i, j = 1, 2$, such that P is the disjoint union of P_1 and P_2, and N of N_1 and N_2. Then it learned the ϕ_{ij} recursively from the (P_i, N_j), and finally combine all into

$$(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$$

which is sound for (P, N), but is not necessarily minimal. E.g., whenever ϕ_{11} implies ϕ_{12}, then $\phi_{11} \vee (\phi_{21} \wedge \phi_{22})$ is lower cost[4]. Thus it might be tempting to minimise D&C-steps. Alas, the enumerator may run out-of-memory (OOM): the parameters in Enum(T, L, W) are *static* constraints, pertaining to data structure layout, and do not guarantee successful termination. Let us call the maximal cardinality $\#(P, N)$ that the D&C-unit sends directly to the enumerator, the *split window*. In order to navigate the trade-offs between avoiding OOM and minimising the approximation ratio, our refined D&C-units below use search to find as large as possible a split window. We write *win* for the split window parameter. Both implementations split specifications until they fit into the split window, *i.e.*, $\#(P, N) \leq win$, and then invoke the enumerator. The split window is then successively halved, until the enumerator no longer runs OOM but returns a sound formula.

[4] Such redundancies can be eliminated, for example, by using theorem provers.

Deterministic Splitting. The idea behind $\mathsf{detSplit}(P, N, win)$ is: if $\#(P, N) \leq win$, we send (P, N) directly to the enumerator. Otherwise, assume P is $\{p_1, ..., p_n\}$. Then $P_1 = \{p_1, ..., p_{n/2}\}$ and $P_2 = \{p_{n/2+1}, ..., p_n\}$ are the new positive sets, and likewise for N. (If the specification is given as two lists of traces, this is deterministic.) We then make 4 recursive calls, but remove redundancies in the calls' arguments.

- $\phi_{11} = \mathsf{detSplit}(P_1, N_1, win)$,
- $\phi_{12} = \mathsf{detSplit}(P_1, N_2 \cap \mathsf{lang}(\phi_{11}), win)$,
- $\phi_{21} = \mathsf{detSplit}(P_2 \setminus L, N_1, win)$,
- $\phi_{22} = \mathsf{detSplit}(P_2 \setminus L, N_2 \cap \mathsf{lang}(\phi_{21}), win)$,

Here $L = \mathsf{lang}(\phi_{11}) \cup \mathsf{lang}(\phi_{12})$. Assuming that none of the 4 recursive calls returns OOM, the resulting formula is $(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$. Otherwise we recurse with $\mathsf{detSplit}(P, N, win/2)$.

Random Splitting. This variant of the algorithm, written $\mathsf{randSplit}(P, N, win)$, is based on the intuition that often a small number of traces already contain enough information to learn a formula for the whole specification. (E.g., the traces are generated by running the same system multiple times.)

- $\phi_{11} = \mathsf{aux}(P, N, win)$
- $\phi_{12} = \mathsf{randSplit}(P \cap \mathsf{lang}(\phi_{11}), N \cap \mathsf{lang}(\phi_{11}), win)$
- $\phi_{21} = \mathsf{randSplit}(P \setminus \mathsf{lang}(\phi_{11}), N \setminus \mathsf{lang}(\phi_{11}), win)$
- $\phi_{22} = \mathsf{randSplit}(P \setminus \mathsf{lang}(\phi_{11}), N \cap \mathsf{lang}(\phi_{11}), win)$

The function $\mathsf{aux}(P, N, win)$ first construct a sub-specification (P_0, N_0) of (P, N) as follows. Select two random subsets $P_0 \subseteq P$ and $N_0 \subseteq N$, such that the cardinality of (P_0, N_0) is as large as possible but not exceeding win; in addition we require the cardinalities of P_0 and N_0 to be as equal as possible. Then (P_0, N_0) is sent to the enumerator. If that returns OOM, $\mathsf{aux}(P, N, win/2)$ is invoked, then $\mathsf{aux}(P, N, win/4), ...$ until the enumerator successfully learns a formula. Once ϕ_{11} is available, the remaining ϕ_{ij} can be learned in parallel. Finally, we return $(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$.

Our benchmarks in §8 show that deterministic and random splitting display markedly different behaviour on some benchmarks.

8 Evaluation of Algorithm Performance

This section quantifies the performance of our implementation. We are interested in a comparison with existing LTL learners, but also in assessing the impact on LTL learning performance of different algorithmic choices. We benchmark along the following quantitative dimensions: number of traces the implementation can handle, speed of learning, and cost of inferred formulae. In our evaluation we are facing several challenges.

- We are comparing a CUDA program running on a GPU with programs, sometimes written in Python, running on CPUs.

- We run our benchmarks in *Google Colab Pro*. It is unclear to what extent Google Colab Pro is virtualised. We observed variations in CPU and GPU running times, for all implementations measured.
- Existing benchmarks are too easy. They neither force our implementation to learn costly formulae, nor terminate later than the *measurement threshold* of around 0.2 s, a minimal time the Colab-GPU would take on any task, including toy programs that do nothing at all on the GPU.
- Lack of ground-truth: how can we evaluate the price we pay for scale, *i.e.*, the loss of formula minimality guarantees from algorithmic choices, when we do not know what this minimum is?

Hardware and Software Used for Benchmarking. Benchmarks below run on Google Colab Pro. We use Colab Pro because it is a widely used industry standard for running and comparing ML workloads. **Colab CPU parameters:** Intel Xeon CPU ("cpu family 6, model 79"), running at 2.20GHz, with 51 GB RAM. **Colab GPU parameters:** Nvidia Tesla V100-SXM2, with System Management Interface 525.105.17, with 16 GB memory. We use Python version 3.10.12, and CUDA version: 12.2.140. All our timing measurements are end-to-end (from invocation of $\mathsf{learn}(P, N)$ to its termination), using Python's `time` library.

Benchmark Construction. Since existing benchmarks for LTL learning are too easy for our implementation, we develop new ones. A good benchmark should be tunable by a small number of explainable parameters that allows users to achieve hardness levels, from trivial to beyond the edge-of-infeasibility, and any point in-between. We now describe how we construct our new benchmarks.

- By $\textsc{BenchBase}(\Sigma, k, lo, hi)$ we denote the specifications generated using the following process: uniformly sample $2 \cdot k$ traces from $\{tr \in \mathsf{traces}(\Sigma) \mid lo \leq \|tr\| \leq hi\}$. Split them into two sets (P, N), each containing k traces.
- By $\textsc{Scarlet}(\Sigma, \phi, k, lo, hi)$ we mean using the sampler coming with Scarlet [32] to sample specifications (P, N) that are separated by ϕ, where ϕ is a formula over the alphabet Σ. Both, P and N, contain k traces each, and for each $tr \in P \cup N$ we have $lo \leq \|tr\| \leq hi$. The probability distribution Scarlet implements is detailed in [32].
- By $\textsc{Sampling}(i, k, c)$, where $i, k \in \mathbb{N}$ and $c \in \{\mathit{conservative}, \neg \mathit{conservative}\}$, we mean the following process, which we also call *extension by sampling*.
  1. Generate (P, N) with $\textsc{BenchBase}(\mathbb{B}, i, 2, 5)$.
  2. Use our implementation to learn a minimal formula ϕ for (P, N) that is U-free and in NNF (for easier comparison with Scarlet, which can neither handle general negation nor U).
  3. Next we sample a specification (P', N') from $\textsc{Scarlet}(\mathbb{B}, \phi, k, 63, 63)$.
  4. The final specification is given as follows:
     - $(P \cup P', N \cup N')$ if $c = \mathit{conservative}$. Hence the final specification is a conservative extension of (P, N) and its cost is $\mathsf{cost}(P, N)$.
     - (P', N') otherwise.

Note that the minimal formula required in Step 2. exists because for $i \leq 8$, any (P, N) generated by $\textsc{BenchBase}(\mathbb{B}, i, 2, 5)$ has the property that $\#\mathsf{sc}^+(P \cup N) \leq 80 < 126$, so our algorithm uses neither RUCs nor D&C, but, by construction, does an exhaustive bottom-up enumeration that is guaranteed to learn a minimal sound formula.

- By $\textsc{Hamming}(\Sigma, l, \delta)$, with $l, \delta \in \mathbb{N}$, we mean specifications $(\{tr\}, \mathsf{Hamm}(tr, \delta))$, where tr is sampled uniformly from all traces of length l over Σ.

Benchmarks from $\textsc{Sampling}(k, i, c)$ are useful for comparison with existing LTL learners, and to hone in on specific properties of our algorithm. But they don't fully address a core problem of using random traces: they tend to be too easy. One dimension of "too easy" is that specifications (P, N) of random traces often have tiny sound formulae, especially for large alphabets. Hence we use binary alphabets, the hardest case in this context. That alone is not enough to force large formulae. $\textsc{Hamming}(\Sigma, l, \delta)$ works well in our benchmarking: it generates benchmarks that are hard even for the GPU. We leave a more detailed investigation why as future work. Finally, in order to better understand the effectiveness of MuellerHash in our RUC, we use the following deliberately simple map from CMs to 126 bits.

First-k-percent (FKP). This scheme simply takes the first $k\%$ of each CS in the CM. All remaining bits are discarded. The percentage k is chosen such that the result is as close as possible to 126 bits. e.g., for a 64*63 bit CM, $k = 3$.

Comparison with Scarlet. In this section we compare the performance of our implementation against Scarlet [32], in order better to understand how much performance we gain in comparison with a state-of-the-art LTL learner. Our comparison with Scarlet is implicitly also a comparison with Flie [28] and Syslite [4] because [32] already benchmarks Scarlet against them, and finds that Scarlet performs better. We use the following benchmarks in our comparison.

- All benchmarks from [32], which includes older benchmarks for Flie and Syslite.
- Two new benchmarks for evaluating scalability to high-cost formulae, and to high-cardinality specifications.

In all cases, we learn U-free formulae in NNF for easier comparison with Scarlet. This restriction hobbles our implementation which can synthesise cheaper formulae in unrestricted LTL.

Scarlet on Existing Benchmarks. We run our implementation in 12 different modes: D&C by deterministic, resp., random splitting, with two different hash functions (MuellerHash and FKP), and three different split windows (16, 32, and 64). The results are visualised in Table 1. We make the following observations. On existing benchmarks, our implementation usually returns formulae that are roughly of the same cost as Scarlet. They are typically only larger on

Table 1. Comparison of Scarlet with our implementation on existing benchmarks. Timeout is 2000 s. On the existing benchmarks our implementation never runs OOM or out-of-time (OOT), while Scarlet runs OOM in 5.9% of benchmarks and OOT in 3.8%. In computing the average speedup we are conservative: we use 2000 s whenever Scarlet runs OOT, if Scarlet runs OOM, we use the time to OOM. The "Lower Cost" column gives the percentages of instances where our implementation learns a formula with lower cost than Scarlet, and likewise for "Equal" and "Higher". Here and below, "Ave" is short for the *arithmetic mean*. The column on the right reports the average speedup over Scarlet of our implementation.

D&C	/ Hsh	/ Win	Lower Cost	Equal Cost	Higher Cost	Ave Speed-up
Rand Split	FKP	64	11.2%	81.9%	6.9%	>**515x**
		32	10.7%	79.8%	9.5%	>**483x**
		16	10.9%	76.3%	12.8%	>**320x**
	Mueller	64	12.3%	77.9%	9.8%	>**58x**
		32	11.4%	72.7%	15.9%	>**433x**
		16	11.2%	67.2%	21.6%	>**236x**
Det Split	FKP	64	11.4%	74.4%	14.2%	>**173x**
		32	10.4%	70.5%	19.2%	>**103x**
		16	10.5%	66.1%	23.3%	>**52x**
	Mueller	64	12.4%	78.1%	9.5%	>**263x**
		32	10.5%	72.5%	16.9%	>**114x**
		16	10.5%	66.5%	23.0%	>**46x**

benchmarks with a sizeable specification, e.g., 100000 traces, which forces our implementation into D&C, with the concomitant increase in approximation ratio due to the cost of recombination. However the traces are generated by sampling from trivial formulae (mostly $\mathsf{F}p, \mathsf{G}p$ or $\mathsf{G}\neg p$). Scarlet handles those well. [32] defines a parameterised family ϕ^n_{seq} that can be made arbitrarily big by letting n go to infinity. However in [32] $n < 6$, and even on those Scarlet run OOM/OOT, while our implementation handles all in a short amount of time. On existing benchmarks, our implementation runs on average at least 46 times faster. We believe that this surprisingly low worst-case average speedup is largely because the existing benchmarks are too easy, and the timing measurements are dominated by GPU startup latency. The comparison on harder benchmarks below shows this.

Scarlet and High-Cost Specifications. The existing benchmarks can all be solved with small formulae. This makes it difficult to evaluate how our implementation scales when forced to learn high-cost formulae. In order to ameliorate this problem, we create a new benchmark using $\textsc{Hamming}(\mathbb{B}, l, \delta)$ for $l = 3, 6, 9, ..., 48$ and $\delta = 1, 2$. We benchmark with the aforementioned 12 modes. The left of Table 2 summarises the results. This benchmark clearly shows that Scarlet is

Table 2. On the left, comparison between Scarlet and our implementation on $\textsc{Hamming}(\mathbb{B}, l, \delta)$ benchmarks with $l = 3, 6, 9, ..., 48$ and $\delta = 1, 2$. Timeout is 2000 sec. Reported percentage is fraction of specifications that were successfully learned. On the right, comparison on benchmarks from $\textsc{Sampling}(5, 2^k, \mathit{conservative})$ for $k = 3, 4, 5, ..., 17$. All benchmarks were run to conclusion, Scarlet's OOMs occurred between 1980.21 sec for $(2^{17}, 2^{17})$, and 16568.7 sec for $(2^{13}, 2^{13})$. Recall from §2 that $\#S$ denotes the cardinality of set S.

D&C	/ Hsh	/ Win	Delta=1	Delta=2
Rand Split	FKP	64	100%	100%
		32	100%	100%
		16	100%	100%
	Mueller	64	100%	75%
		32	100%	75%
		16	100%	100%
Det Split	FKP	64	100%	100%
		32	100%	100%
		16	100%	100%
	Mueller	64	100%	88%
		32	100%	100%
		16	100%	100%
Scarlet			**7%**	**7%**

(# P, # N)	Our impl. Time (Cost)	Scarlet Time (Cost)
$(2^3, 2^3)$	0.31s (12)	1532.85s (19)
$(2^4, 2^4)$	0.32s (12)	1463.67s (17)
$(2^5, 2^5)$	0.36s (12)	2867.47s (17)
$(2^6, 2^6)$	0.34s (12)	5691.98s (17)
$(2^7, 2^7)$	0.63s (20)	OOM
$(2^8, 2^8)$	0.95s (19)	OOM
$(2^9, 2^9)$	0.72s (19)	OOM
$(2^{10}, 2^{10})$	1.09s (19)	OOM
$(2^{11}, 2^{11})$	1.32s (19)	OOM
$(2^{12}, 2^{12})$	1.66s (19)	OOM
$(2^{13}, 2^{13})$	2.46s (19)	OOM
$(2^{14}, 2^{14})$	4.62s (20)	OOM
$(2^{15}, 2^{15})$	8.35s (19)	OOM
$(2^{16}, 2^{16})$	15.52s (19)	OOM
$(2^{17}, 2^{17})$	30.49s (19)	OOM

mostly unable to learn bigger formulae, while our implementation handles all swiftly.

Scarlet and High-Cardinality Specifications. The previous benchmark addresses scalability to high-cost specifications. The present comparison with Scarlet seeks to quantify an orthogonal dimension of scalability: high-cardinality specifications. Our benchmark is generated by $\textsc{Sampling}(i, 2^k, \mathit{conservative})$ for $i = 5$, and $k = 3, 4, 5, ..., 17$. Using $i = 5$ ensures getting a few concrete times from Scarlet rather than just OOM/OOT; $k \leq 17$ was chosen as Scarlet's sampler makes benchmark generation too time-consuming otherwise. The choice of parameters also ensures that the cost of each benchmark is moderate (≤ 20). This means any difficulty with learning arises from the sheer number of traces. Unlike the previous two benchmarks, we run our implementation only in one configuration: using MuellerHash, and random splitting with window size 64 (the difference between the variants is too small to affect the comparison with Scarlet in a substantial way). The results are also presented on the right of Table 2. This benchmark clearly shows that we can handle specifications at least 2048 times larger, despite Scarlet having approx. 3 times more memory available. Moreover, not only is our implementation much faster and can handle more traces, it also finds substantially smaller formulae in all cases where a comparison is possible.

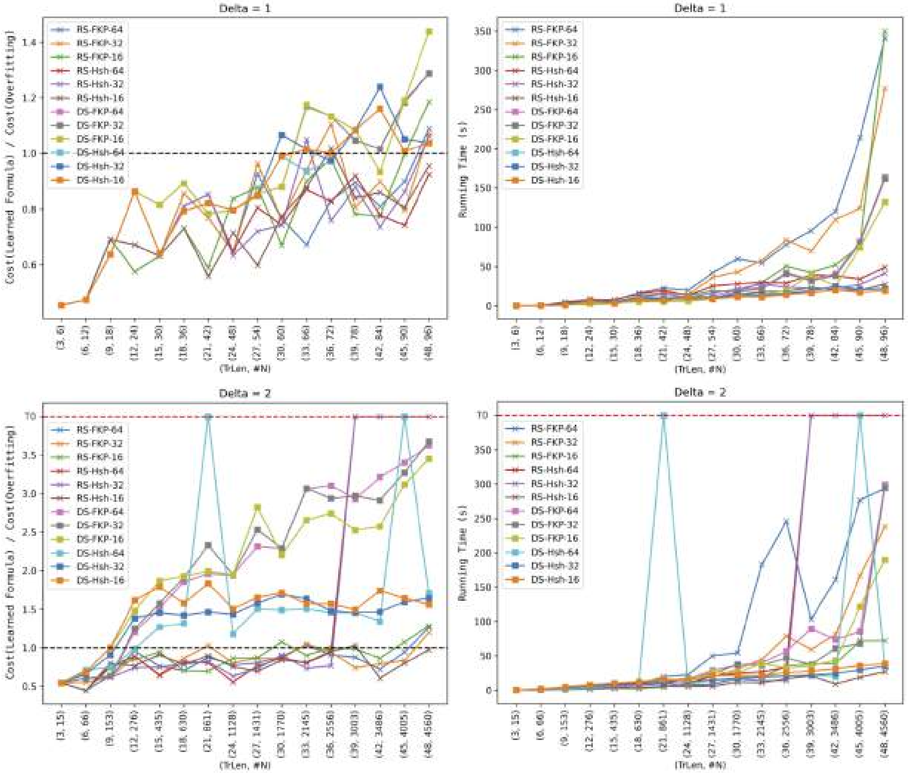


Fig. 3. Here RS means random-splitting, DS deterministic-splitting. The numbers 16, 32, 64 are the used splitting window. Hsh is short for MuellerHash. The x-axis is annotated by $(trLen, \#N)$, giving the length of the single trace in P, and the cardinality $\#N$ of N. TO denotes timeout. Timeout is 2000 s. On the left, the y-axis gives the ratio $\frac{\text{cost of learned formula}}{\text{cost of overfitting}}$, the dotted line at 1.0 is the cost of overfitting.

Hamming Benchmarks. We have already used Hamming(...) in our comparison with Scarlet. Now we abandon existing learners, and delve deeper into the performance of our implementation by having it learn costly formulae. This benchmark is generated using $\text{Hamming}(\mathbb{B}, l, \delta)$ for $l = 3, 6, 9, ..., 48$ and $\delta = 1, 2$. As above, the implementation learns U-free formulae in NNF. Figure 3 gives a more detailed breakdown of the results. The uniform cost of overfitting on each benchmark is given for comparison[5]:

Length of tr	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48
Cost of overfitting	22	38	55	73	92	111	129	147	164	182	201	219	238	256	274	292

[5] For this benchmark we disable returning $\mathsf{overfit}(P, N)$ when the cost of target formulae matches $\mathsf{cost}(\mathsf{overfit}(P, N))$, cf. Line 10 in the sketch of `enum` in §3.

Table 3. All run times are below the measurement threshold.

(# P, # N)	FKP		MuellerHash	
	AveExtraCost	OOM	AveExtraCost	OOM
(8, 8)	0.0%	0.0%	0.0%	0.0%
(12, 12)	2.4%	12.8%	1.9%	24.5%
(16, 16)	4.1%	19.3%	0.7%	32.6%
(20, 20)	4.1%	22.4%	0.4%	32.9%
(24, 24)	3.9%	24.0%	0.1%	33.6%
(28, 28)	3.6%	24.9%	0.0%	32.9%
(32, 32)	2.7%	26.3%	0.2%	40.3%

This benchmark shows the following. Hamming benchmarks are hard for our implementation, and sometimes run for >3 min: we successfully force our implementation to synthesise large formulae, and that has an effect on running time. The figures on the left show that random splitting typically leads to smaller formulae in comparison with deterministic splitting, especially for $\delta = 2$. Indeed, we may be seeing a sub-linear increase in formula cost (relative to the cost of overfitting) for random splitting, while for deterministic splitting, the increase seems to be linear. In contrast, the running time of the implementation seems to be relatively independent from the splitting mechanism. It is also remarkable that the maximal cost we see is only about 3.5 times the cost of overfitting: the algorithm processes P and N, yet over-fitting happens only on P, which contains a single trace. Hence the cost of over-fitting (292 in the worst case) is not affected by N, which contains up to 4560 elements (of the same length as the sole positive trace).

Benchmarking RUCs. Our algorithm uses RUCs, a novel cache admission policy, and it is interesting to gain a more quantitative understanding of the effects of (pseudo-)randomly rejecting some CMs. We cannot hope to come to a definitive conclusion here. Instead we simply compare MuellerHash with FKP, which neither distributes values uniformly across the hash space (our 126 bits) to minimise collisions, nor has the avalanche effect where a small change in the input produces a significantly different hash output. This weakness is valuable for benchmarking because it indicates how much a hash function can degrade learning performance. (Note that for the edge case of specifications that can be separated from just the first k% alone, FKP should perform better, since it leaves the crucial bits unchanged.) The benchmark data is generated with $\textsc{Sampling}(8, 24, \mathit{conservative})$. Table 3 summarises our measurements. We note the following. The loss in formula cost is roughly constant for each hash: it stabilises to around 0.2% for MuellerHash, and a little above 2.5% for FKP. Hence MuellerHash is an order of magnitude better. Nevertheless, even 2.5% should be irrelevant in practice, and we conjecture that replacing MuellerHash with a cryptographic hash will have only a moderate effect on learning performance.

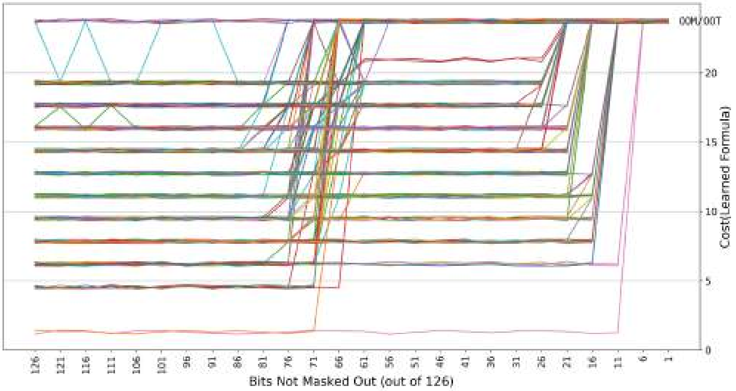


Fig. 4. Effects of masking on formula cost. Timeout is 200 s. Colours correspond to different (P, N). The slight 'wobble' on all graphs is deliberately introduced for readability, and is not in the data.

A surprising number of instances run OOM, more so as specification size grows, with MuellerHash more than FKP. We leave a detailed understanding of these phenomena as future work.

Masking. The previous benchmarks suggested that naive hash functions like FKP sometimes work better than expected. Our last benchmark seeks to illuminate this in more detail and asks: can we relate the information loss from hashing and the concomitant increase in formula cost? A precise answer seems to be difficult. We run a small experiment: after MuellerHashing CMs of size 64*63 bits to 126 bits, we add an additional information loss phase: we *mask out k* bits, *i.e.*, we set them to 0. This destroys all information in the k masked bits. After masking, we run the uniqueness check. We sweep over $k = 1, ..., 126$ with stride 5 to mask out benchmarks generated with $\textsc{Sampling}(8, 32, \neg conservative)$. Figure 4 shows the results. Before running the experiments, the authors expected a gradual increase of cost as more bits are masked out. Instead, we see a phase transition when approx. 75 to 60 bits are not masked out: from minimal cost formulae before, to OOM/OOT after, with almost no intermediate stages. Only a tiny number of instances have 1 or 2 further cost levels between these two extremes. We leave an explanation of this surprising behaviour as future work.

9 Conclusion

The present work demonstrates the effectiveness of carefully tailored algorithms and data structures for accelerating LTL learning on GPUs. We close by summarising the reasons why we achieve scale: high degree of parallelism inherent in generate-and-test synthesis; application of divide-and-conquer strategies; relaxed uniqueness checks for (pseudo-)randomly curtailing the search space; and succinct, suffix-contiguous data representation, enabling exponential propagation where LTL connectives map directly to branch-free machine instructions with predictable data movement. All but the last are available to other learning tasks that have suitable operators for recombination of smaller solutions.

LTL and GPUs, a match made in heaven.

Acknowledgement. The first author thanks the University of Sussex, School of Engineering and Informatics for their generous funding, making this work possible. The second author acknowledges the support of the French PEPR Intelligence Artificielle SAIF project (ANR-23-PEIA-0006).

References

1. Github repository. https://github.com/MojtabaValizadeh/ltl-learning-on-gpus (2024)
2. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 4–16. POPL '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/503272.503275
3. Anderson, S.E.: Bit twiddling hacks: round up to the next highest power of 2 (2005). https://graphics.stanford.edu/~seander/bithacks.html
4. Arif, M.F., Larraz, D., Echeverria, M., Reynolds, A., Chowdhury, O., Tinelli, C.: SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces. In: Proceedings of the International Conference on Formal Methods in Computer Aided Design, FMCAD, pp. 93–103 (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_16
5. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. In: Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019, Berkeley, CA, USA, July 11-15, 2019, pp. 621–630. AAAI Press (2019). https://ojs.aaai.org/index.php/ICAPS/article/view/3529
6. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. In: International Conference on Automated Planning and Scheduling, ICAPS. vol. 29, pp. 621–630 (2019). https://ojs.aaai.org/index.php/ICAPS/article/view/3529
7. Chen, Y.F., Farzan, A., Clarke, E.M., Tsay, Y.K., Wang, B.Y.: Learning minimal separating DFA's for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 31–45. Springer, Berlin Heidelberg, Berlin, Heidelberg (2009)
8. Dally, W.J., Turakhia, Y., Han, S.: Domain-specific hardware accelerators. Commun. ACM **63**(7), 48–57 (2020). https://doi.org/10.1145/3361682
9. David, C., Kroening, D.: Program Synthesis: Challenges and Opportunities. Philos. Trans. A **375**(2104), 20150403 (2017)
10. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, pp. 854–860. IJCAI '13, AAAI Press (2013)
11. Fijalkow, N., Lagarde, G.: The complexity of learning linear temporal formulas from examples. In: Proceedings of the Fifteenth International Conference on Grammatical Inference. Proceedings of Machine Learning Research, vol. 153, pp. 237–250. PMLR (2021). https://proceedings.mlr.press/v153/fijalkow21a.html

12. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 339–349. SIGSOFT '08/FSE-16, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1453101.1453150
13. Gabel, M., Su, Z.: Symbolic Mining of Temporal Specifications. In: Proceedings of the 30th International Conference on Software Engineering, pp. 51–60. ICSE '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1368088.1368096
14. Gabel, M., Su, Z.: Online Inference and Enforcement of Temporal Properties. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, pp. 15–24. ICSE '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1806799.1806806
15. Gaglione, J., Neider, D., Roy, R., Topcu, U., Xu, Z.: Maxsat-based temporal logic inference from noisy data. Innovations Syst. Softw. Eng. **18**(3), 427–442 (2022). https://doi.org/10.1007/S11334-022-00444-8
16. Gulwani, S., Polozov, O., Singh, R.: Program Synthesis. Now Foundations and Trends (2017). http://ieeexplore.ieee.org/document/8187066
17. Hennessy, J., Patterson, D.: Computer Architecture: a quantitative approach. The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann (2017)
18. Hwu, W.M.W., Kirk, D.B., Hajj, I.E.: Programming Massively Parallel Processors, Morgan Kaufmann (2022)
19. Ielo, A., Law, M., Fionda, V., Ricca, F., De Giacomo, G., Russo, A.: Towards ILP-Based LTL$_f$ Passive Learning. In: Inductive Logic Programming, pp. 30–45. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-49299-0_3
20. Jeppu, N., Melham, T., Kroening, D., O'Leary, J.: Learning Concise Models from Long Execution Traces. In: Proceedings of the 57th ACM/IEEE Design Automation Conference, DAC. pp. 1–6 (2020).https://doi.org/10.1109/DAC18072.2020.9218613
21. Jünger, D.: WARPCORE: hashing at the speed of light on modern CUDA-accelerators (2022). https://github.com/sleeepyjack/warpcore
22. Jünger, D., et al.: WarpCore: a library for fast hash tables on GPUs. In: Proceedings of the 27th International Conference on High Performance Computing, Data, and Analytics, HiPC, pp. 11–20 (2020). https://doi.org/10.1109/HiPC50609.2020.00015
23. Kim, J., Muise, C., Shah, A., Agarwal, S., Shah, J.: Bayesian inference of linear temporal logic specifications for contrastive explanations. In: International Joint Conference on Artificial Intelligence, IJCAI (2019). https://doi.org/10.24963/ijcai.2019/776
24. Lemieux, C., Beschastnikh, I.: Investigating program behavior using the texada LTL specifications miner. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 870–875. IEEE Computer Society, Los Alamitos, CA, USA (2015). https://doi.org/10.1109/ASE.2015.94
25. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 81–92. IEEE Computer Society, Los Alamitos, CA, USA (2015). https://doi.org/10.1109/ASE.2015.71

26. Luo, W., Liang, P., Du, J., Wan, H., Peng, B., Zhang, D.: Bridging LTLf inference to GNN inference for learning LTLf formulae. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36(9), 9849–9857 (2022). https://doi.org/10.1609/aaai.v36i9.21221
27. Mascle, C., Fijalkow, N., Lagarde, G.: Learning temporal formulas from examples is hard (2023). https://doi.org/10.48550/arXiv.2312.16336
28. Neider, D., Gavran, I.: Learning linear temporal properties. In: Formal Methods in Computer Aided Design, FMCADm, pp. 1–10 (2018). https://doi.org/10.23919/FMCAD.2018.8603016
29. Peng, B., et al.: PURLTL: mining LTL specification from imperfect traces in testing. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1766–1770. IEEE Computer Society, Los Alamitos, CA, USA (2023). https://doi.org/10.1109/ASE56229.2023.00202
30. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, FOCS, pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
31. Raha, R., Rajarshi, R., Fijalkow, N., Neider, D.: Scarlet: scalable anytime algorithms for learning fragments of linear temporal logic (2024)
32. Raha, R., Roy, R., Fijalkow, N., Neider, D.: Scalable anytime algorithms for learning fragments of linear temporal logic. In: TACAS 2022. LNCS, vol. 13243, pp. 263–280. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_14
33. Valizadeh, M., Berger, M.: Search-based regular expression inference on a GPU. Proc. ACM Program. Lang. **7**(PLDI), 1317–1339 (2023). https://doi.org/10.1145/3591274, technical report available at https://arxiv.org/abs/2305.18575, implementation: https://github.com/MojtabaValizadeh/paresy
34. Weimer, W., Necula, G.C.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_30
35. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proceedings of the 28th International Conference on Software Engineering, pp. 282–291. ICSE '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1134285.1134325
36. Yogananda Jeppu, N.: Learning symbolic abstractions from system execution traces. Ph.D. thesis, University of Oxford (2022)

Safe Exploration in Reinforcement Learning by Reachability Analysis over Learned Models

Yuning Wang and He Zhu(✉)

Rutgers University, New Brunswick, NJ, USA
{yw895,hz375}@cs.rutgers.edu

Abstract. We introduce VELM, a reinforcement learning (RL) framework grounded in verification principles for safe exploration in unknown environments. VELM ensures that an RL agent systematically explores its environment, adhering to safety properties throughout the learning process. VELM learns environment models as symbolic formulas and conducts formal reachability analysis over the learned models for safety verification. An online shielding layer is then constructed to confine the RL agent's exploration solely within a state space verified as safe in the learned model, thereby bolstering the overall safety profile of the RL system. Our experimental results demonstrate the efficacy of VELM across diverse RL environments, highlighting its capacity to significantly reduce safety violations in comparison to existing safe learning techniques, all without compromising the RL agent's reward performance.

Keywords: Controller Synthesis · Reinforcement Learning · Safety Verification · Safe Exploration

1 Introduction

Deep reinforcement learning (RL) is a promising approach for synthesizing controllers [19] to govern cyber-physical systems like autonomous vehicles. State-of-the-art RL algorithms can autonomously acquire motor skills through trial and error, either in simulated environments or even in unknown terrains, thus circumventing the need for laborious manual engineering. However, during training in most RL algorithms, agents perform a significant number of exploratory steps that can lead to dangerous behavior. In many real-world scenarios where ensuring high assurance is crucial, it becomes imperative for the RL agent to behave safely during environment interactions, even in training scenarios when the agent is not yet optimal [37,43].

To facilitate safe exploration, it is essential to have a mechanism that determines the safety of executing an action in a given environment state. Several existing approaches utilize prior knowledge about system dynamics [5,6,52] to make such assessments. When the environment dynamics are not known a priori,

This work is supported by the National Science Foundation under grant CCF-2007799.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 232–255, 2024.
https://doi.org/10.1007/978-3-031-65633-0_11

existing safe RL methods utilize learned predictors in the shape of neural networks [1,7,15,47] to predict the safety implications of particular control action. Training these neural predictors may require numerous potentially unsafe environment interactions.

There are also model-based safe RL techniques that leverage learned environment models in unknown environments to filter out unsafe actions [4,34]. In the recent CRABS framework [34], a barrier certificate and a model for environment dynamics are co-trained in conjunction with a controller. The learned neural barrier certificate serves as a predictive tool to assess whether a control action from the policy aligns with the safety requirement. In cases where it does not, a safeguard policy, trained on the environment model, is executed. While rooted in formal methods concepts, CRABS cannot rigorously verify the accuracy of a learned barrier certificate. This challenge arises from the fact that both the certificate and the underlying environment models are deep neural networks, making formal verification a complex task. Another recent work SPICE [4] uses weakest preconditions [16] to generate, from a learned environment model, a predicate that decides if an action is safe to take at a current environment state concerning a short time horizon H. However, H cannot be extended to cover the entire horizon of an RL task, primarily because of the inherent challenge in constructing precise weakest precondition transformers for neural networks. As a result, although grounded in Hoare logic, SPICE still suffers from notable safety violations in its environment exploration.

We present VELM, a model-based safe reinforcement learning framework that engages in formally **v**erified safe **e**xploration through **l**earned environment **m**odels, covering the entire horizon of an RL task. VELM learns a symbolic environment model linking the system's future states, past states, and the controller's actions. Most non-linear control systems are characterized by dynamics dictated by mathematical equations involving operators such as trigonometric functions (like sine and cosine) and power functions. By leveraging this prior knowledge of common operators that could appear in environment dynamics, VELM searches a symbolic environment model in the space of interpretable mathematical expressions by symbolic regression techniques. Symbolic regression methods have demonstrated remarkable extrapolation capabilities in recent studies and have proven valuable across diverse domains including physics [10,28,30]. More importantly, unlike neural environment models, symbolic environment models are conducive to long-horizon reachability analysis, enabling the computation of the reachable set of a control system across the entire task horizon. VELM leverages this capability to establish a safe exploration regime for verified safe learning.

VELM can be instantiated on top of any model-based reinforcement learning algorithms. It involves a two-step procedure repeated until convergence: (a) interact with the true environment to collect a dataset of environment transitions and learn from the data an environment state transition model F (i.e. a function that maps current state s_t and action a_t to next state s_{t+1}) and (b) derive a controller π from this learned model. In each learning iteration, VELM aims to ensure that the data collection process of using the current controller π

to interact with the true environment in step (a) is safe. One way to do so is by verifying the safety of π according to the learned model. However, conducting reachability analysis of neural networks in a closed-loop control system remains a challenging research problem [27]. Alternatively, VELM considers π as an oracle and derives a much simpler and verification-friendly *time-varying* linear controllers π' to approximate the policy actions executed by π at each time step within the RL task horizon. While alternative methods such as approximating a neural controller as a polynomial function exist [48], our objective is to achieve a balance between expressiveness and verifiability. Time-varying linear controllers provide computational efficiency for reachability analysis, making verification over learned models feasible in a learning loop. VELM solves a constrained optimization problem aimed at optimizing the behavior of π' to closely match that of π while simultaneously ensuring that π' can be formally verified as safe for the learned environment model. Leveraging π' as a reference, VELM computes a safety shield that restricts the neural policy π to explore solely within the state space where π' is verified as safe in the learned model. The shield intervenes whenever the neural policy π proposes a potentially unsafe control action that could result in a next state outside the safe state space. It then substitutes this action with a safe alternative provided by π'. The environment state transition model F is repeatedly updated during the learning process using data safely collected using the shielded neural controller. The computation of the shield is accordingly repeated, leading to a more refined shield with each update to the controller.

While there exists prior work that explored shielding for safe RL, they require a calibrated piecewise linear dynamics model [5,52] or an abstract model of the agent's safe behavior [24], whereas VELM automatically learns a dynamics model and a safe shielding policy. Adapting these techniques to learned environment models that evolve across training iterations is challenging, given the inherent difficulty of approximating nonlinear models as piecewise linear functions. Compared with SPICE [4], VELM is computationally efficient as it only computes a shield once for a policy while SPICE requires calling a QP (Quadratic Programming) procedure at every timestep.

Across a suite of challenging continuous control benchmarks, VELM exhibits reward performance comparable to fully neural approaches and significantly fewer safety violations during training compared to state-of-the-art safe RL techniques.

In summary, this paper makes the following contributions:

- We propose a novel approach for model-based safe reinforcement learning. Our approach learns an environment model as a symbolic formula and constructs a shielding layer to confine an RL agent to explore within a state space formally verified as safe for the learned model, thereby enhancing the overall safety profile of the RL system.
- We present VELM as an efficient instantiation of this approach. The experiment results show that VELM offers much greater safety than prior model-based safe RL approaches without suffering a loss in reward performance.

2 Problem Setup

Safety Specification. We define a safety specification as a logical formula specifying the safe states of a control system.

Definition 1 (Safety Specification). *A safety specification φ is a quantifier-free Boolean combination of linear inequalities over the environment state variables x:*

$$\langle\varphi\rangle ::= \langle P\rangle \text{ — } \varphi \wedge \varphi \text{ — } \varphi \vee \varphi;$$
$$\langle P\rangle ::= \mathcal{A} \cdot x \leq b \text{ where } \mathcal{A} \in \mathbb{R}^{|x|},\ b \in \mathbb{R};$$

A state $s \in S$ satisfies a safety specification φ, denoted as $s \models \varphi$, iff $\varphi(s)$ is true.
MDP. We formalize an RL system as a Markov decision process (MDP). Specifically, an MDP is a structure $M[\cdot] = (S, A, P, R, S_0, H, \cdot)$ where S is an infinite set of *continuous real-vector* environment states which are valuations of the state variables $x_1, x_2, \ldots, x_n$ of dimension n ($S \subseteq \mathbb{R}^n$), A is a set of *continuous real-vector* control actions which are valuations of the action variables $u_1, u_2, \ldots, u_m$ of dimension m. $R : S \times A \rightarrow \mathbb{R}$ is a reward function that returns the immediate reward after the transition from an environment state $s \in S$ with action $a \in A$. $P(s_{t+1} \mid s_t, a_t)$ is an (unknown) probabilistic state transition function where $s_{t+1}, s_t \in S$ and $a_t \in A$ and t is a time step index. S_0 is a set of initial states. H is the time horizon of the control task (i.e. the maximum number of timesteps of a trajectory). An MDP $M[\cdot]$ is parameterized with an (unknown) controller.
Controller (Policy). A controller is a stochastic function $\pi : S \rightarrow A$ mapping states to distributions over actions. We explicitly model the deployment of a (learned) controller π in $M[\cdot]$ as a closed-loop system $M[\pi]$. $M[\pi]$ generates trajectories (or rollouts) $\zeta = s_0, a_0, s_1, a_1, \ldots, a_{H-1}, s_H$ where $s_0 \in S_0$, each $a_t \sim \pi(s_t)$, and each $s_{t+1} \sim P(s_t, a_t)$. Given a discount factor $0 \leq \beta < 1$, the long-term reward of a policy π is $R(\pi) = \mathbb{E}_{(\zeta = s_0, a_0, \ldots, s_H) \sim M[\pi]}[\sum_{t=0}^{H} \beta^t R(s_i, a_i)]$.
Problem Formulation. The goal of reinforcement learning is to find a policy $\pi^* = \arg\max_\pi R(\pi)$. To achieve this goal, the learning process of (model-free or model-based) reinforcement learning algorithms progressively refines and optimizes policies $\pi_0, \pi_1, \ldots, \pi_T$ over successive iterations. At each iteration, the current policy is evaluated, and adjustments are made to improve its performance. This learning process continues until the policy converges to the optimal policy π^*. Given a bound δ, we define safe exploration as a learning process $\pi_0, \pi_1, \ldots, \pi_T$ such that

$$\pi_T = \pi^* \text{ and } \forall 1 \leq j \leq T,\ 0 \leq t \leq H.\ P_{\zeta \sim \pi_j, s_t \in \zeta}(\neg\varphi(s_t)) < \delta \tag{1}$$

Essentially, the end goal is for the final policy π_T in the sequence to optimize long-term rewards, while each intermediate policy (excluding π_0) is constrained to a limited probability δ of unsafe behavior. This definition does not place safety constraints on π_0 as the environment dynamics is not known and hence π_0 can exhibit arbitrary (unsafe) behavior.

Algorithm 1. VELM: **V**erified **E**xploration based on **L**earned Models.

```
1: procedure VELM($M$, $\varphi$)
2:     Initialize an empty dataset $D$ and a random NN policy $\pi_{\texttt{NN}}$
3:     for epoch in $0, \ldots, T$ do
4:         if epoch $= 0$ then
5:             $\pi_{\mathcal{S}} \leftarrow \lambda s.\lambda t.\ \pi_{\texttt{NN}}(s)$
6:         else
7:             $\pi_{\mathcal{S}} \leftarrow$ SHIELD($\hat{M}$, $\pi_{\texttt{NN}}$, $\varphi$)          ▷ Algorithm 2
8:         Unroll real rollouts $\{(s_t, a_t, s_{t+1})\}$ in the real environment $M$ under $\pi_{\mathcal{S}}$
9:         $D \leftarrow D \cup \{(s_t, a_t, s_{t+1})\}$
10:        $\hat{M} \leftarrow$ LEARNMODEL($D$)
11:        Optimize $\pi_{\texttt{NN}}$ using the learned environment $\hat{M}$ via any RL algorithm
```

3 Verified Exploration Through Learned Models

The Main Algorithm. Our overall framework, Verified Exploration through Learned Models (VELM), employs a learned environment model to facilitate safety analysis during the training phase. Akin to existing model-based safe RL techniques [4,5,12,26,34], VELM utilizes the learned environment model to delineate safety regions for the underlying control policy. While VELM can also be applied to safe model-based planning, a policy is in general more efficient than a planner. The primary training procedure is outlined in Algorithm 1. It operates within an *unknown* environment M and takes as input a safety property φ. The algorithm concurrently learns an environment model represented as an MDP $\hat{M}$ and a stochastic neural control policy $\pi_{\texttt{NN}}$[1]. The algorithm maintains a dataset D comprising observed environment transitions, each of which is a tuple of future and past states along with the controller's actions (s_t, a_t, s_{t+1}). This dataset is acquired by interaction with the real environment M (Line 9). Subsequently, VELM utilizes this dataset to learn a symbolic environment model $\hat{M}$ (Line 10) and optimizes the neural policy $\pi_{\texttt{NN}}$ on this learned model via any model-free RL algorithm of the user's choice (Line 11). Notably, VELM uses a shielded policy $\pi_{\mathcal{S}}$ for exploring the real environment to construct D. $\pi_{\mathcal{S}}$ takes a state s at a timestep t as input and generates a safe action for the RL agent to execute at t. This is necessary because directly executing the neural controller $\pi_{\texttt{NN}}$ in the real environment M could result in safety violations. The shield policy $\pi_{\mathcal{S}}$ is constructed in Line 7 via the SHIELD procedure (Algorithm 2). This procedure leverages reachability analysis on the learned environment model $\hat{M}$ to establish a safe exploration regime covering the entire task horizon. $\pi_{\mathcal{S}}$ constrains $\pi_{\texttt{NN}}$ to only explore the real environment within the established safe region.

In the following, we describe in detail the procedures to learn symbolic environment models and construct shielded policies for verified safe exploration.

[1] VELM integrates a stochastic policy for exploring the environment to seek high-reward signals. This is not a strict requirement and VELM can also integrate any deterministic policy learning algorithms.

$$\alpha ::= \alpha \ + \ \alpha \mid \alpha \ - \ \alpha \mid \alpha \ \times \ \alpha \mid \alpha \ / \ \alpha \mid \sin(\alpha) \mid \cos(\alpha) \mid x \mid n$$

Fig. 1. Context-free grammar for defining state-transition functions.

3.1 Symbolic Environment Models

The LearnModel procedure (Line 10 in Algorithm 1) follows the conventional model-based RL framework [25] to learn an environment MDP model $\hat{M}[\cdot] = (S, A, F, R, S_0, H, \cdot)$ where $F : S \times A \to S$ is learned using the dataset D to approximate the unknown probabilistic state transition P in the real environment[2]. VELM distinguishes itself from existing methods by learning a *symbolic* environment state transition function F instead of a deep neural network model.

Given a dataset $D = \{(s_t, a_t, s_{t+1})\}$ of real environment state transitions, the LearnModel procedure learns an approximate model f of the environment's dynamics to fit D:

$$f = \operatorname*{argmax}_{f \in \mathcal{F}_\alpha} \mathbb{E}_{(s_t, a_t, s_{t+1}) \in D} \| f(s_t, a_t) - s_{t+1} \| \tag{2}$$

where $\mathcal{F}_\alpha$ is a family of expressions that can be articulated using the grammar outlined in Fig. 1. This grammar accommodates common mathematical operators such as trigonometric functions. The metavariables x and n represent state variables and constants respectively. The symbolic function f establishes the relation between the next state s_{t+1} and the system's past state s_t, as well as the controller's action a_t.

Why Symbolic Environment Models? First, we observe that the dynamics of non-linear control systems often follow mathematical equations. Second, symbolic environment models are suitable for long-horizon reachability analysis to verify the safety of a control system. In contrast, performing reachability analysis over neural network models suffers from large accumulation errors arising from over-approximation [27].

To infer a symbolic formula f to fit D in Eq. 2, the LearnModel procedure employs off-the-shelf symbolic regression techniques [10]. Symbolic regression is a machine learning approach that can learn the governing formulas of data. As demonstrated in recent studies [10,28,30], symbolic regression exhibits excellent extrapolation capabilities and has already proved useful in a variety of domains such as physics. VELM uses it to search over the space of mathematical expression by manipulating the operators, constants, and variables in the grammar depicted in Fig. 1.

Nondeterministic Environment Model. It is important to note that VELM does not directly use the deterministic function f as the state transition function F for learned models $\hat{M}[\cdot] = (S, A, F, R, S_0, H, \cdot)$. In cases where control environments are stochastic (common in RL tasks), deterministic state transition

[2] If the real reward function is unknown, an approximate reward function $R : S \times A \to \mathbb{R}$ can also be learned from data [25] by recording in $D = \{(s_t, a_t, s_{t+1}, r_t)\}$ the immediate reward r_t of taking an action a_t. We omit this detail in the paper.

functions are not adequate. For stochastic environments, we aim to bound the deviation between f and the real environment. We identify ϵ such that for all s_t and a_t, $\|f(s_t, a_t) - s_{t+1}\| \leq \epsilon$ where $s_{t+1} \sim P(\cdot|s_t, a_t)$ is sampled from the true environment transition at s_t by taking action a_t. We then express the state transition function of a learned model $\hat{M}[\cdot]$ as a nondeterministic function:

$$F(s_t, a_t) = f(s_t, a_t) + [-\epsilon, \epsilon]$$

When used for simulation, F generates a next state at time step t by adding an error vector uniformly sampled from $[-\epsilon, \epsilon]$ to the result of $f(s_t, a_t)$. When used for reachability analysis and verification, we consider all possible error terms within $[-\epsilon, \epsilon]$ as an overapproximation to account for the worst-case deviation.

In practice, we estimate ϵ from data and choose the most permissible ϵ such that $\forall(s_t, a_t, s_{t+1}) \in D.\ \|f(s_t, a_t) - s_{t+1}\| \leq \epsilon$. *Given* f, with sufficient data in D, the model learning procedure LearnModel returns a model that is close to the actual environment with high probability $1 - \delta_M$. That is, for all $s \in S, a \in A$,

$$\Pr_{s' \sim P(\cdot|s,a)}\left[s' \notin F(s, a)\right] < \delta_M$$

In this paper, we learn F as a discrete dynamics system model. With an Ordinary Differential Equation solver, we can also leverage symbolic regression to learn a more accurate continuous-time dynamics model. This is left for future work.

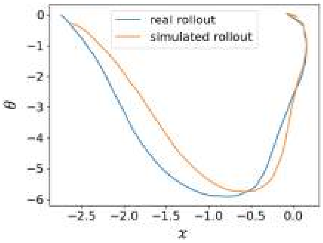


Fig. 2. Executing a random policy on the real CartPole environment and a learned model.

Example 1. Consider the classic CartPole environment [8]. The system's state is described by $(x, \dot{x}, \theta, \dot{\theta})$ where x (resp. $\dot{x}$) denotes the position (resp. speed) of the cart along the x-axis and θ (resp. $\dot{\theta}$) is the angle (resp. angular velocity) of the pole with respect to the cart. The goal is to balance the pole straight up and bound the deviation of the cart. VELM learns the following equation to describe the state transition function of the system using the Operon [9] symbolic regression tool where u represents the control action (we ignore ϵ for simplicity):

$$\begin{aligned}
&x = x + 0.02\dot{x} \qquad\qquad \theta = \theta + 0.02\dot{\theta}\\
&\dot{x} = \dot{x} + 0.019u - (0.001u \cdot \sin(0.999\theta) + 0.001)\sin(\theta) - 0.007\sin(2\theta)\\
&\dot{\theta} = \dot{\theta} - (0.029u + (0.001\dot{x} + 0.002\dot{\theta}^2 + 0.001\dot{\theta} - 0.02)\sin(\theta))\cos(\theta) + 0.3\cos(\theta - 1.58)
\end{aligned}$$

Figure 2 depicts the rollouts in the real environment and simulated in the learned model by executing a random policy from (0,0,0,0). The learned model can reasonably capture real trajectories within a small error bound.

3.2 Shielding for Verified Safe Exploration

With a learned environment model $\hat{M}[\cdot]$, under the assumption of its high-probability approximate accuracy, the verification of a neural controller $\pi_{\texttt{NN}}$ can be directly pursued through reachability analysis over the closed-loop neural network controlled system $\hat{M}[\pi_{\texttt{NN}}]$ (NNCS). However, the verification of NNCS remains a significant challenge in the research literature [27].

Time-varying Linear Controllers. VELM instead distills a neural controller $\pi_{\texttt{NN}}$ into a time-varying linear controller that is as similar as possible to $\pi_{\texttt{NN}}$. Simultaneously, this process ensures that the safety of the time-varying linear controller can be formally verified concerning the learned model $\hat{M}[\cdot]$ and a safety property φ. Principally, a time-varying linear controller can provide an accurate local approximation of a neural controller at each time step (if the time step is small) and incur a much-reduced verification cost owing to the linearity of the representation. A time-varying linear controller $\pi_\theta(s,t)$ with trainable parameters θ for a time horizon H $(0 \leq t < H)$ can be expressed mathematically as:

$$\pi_\theta(s,t) = \theta_k(t)^T \cdot s + \theta_b(t)$$

$\pi_\theta(s,t)$ generates the control input at time t when observing the current environment state s at t. The time-varying nature of the controller is captured by the dependence of the time-varying gain matrix $\theta_k(t)$ and the time-varying bias term $\theta_b(t)$, reflecting the dynamic adjustments in the control strategy over different time instances t. The overall objective of distilling $\pi_{\texttt{NN}}$ into a time-varying linear controller π_θ is:

$$\begin{aligned} &\min_\theta \mathbb{E}_{s_0,s_1,\ldots,s_H \sim \hat{M}[\pi_\theta]} \; \|\pi_\theta(s_t,t) - \pi_{\texttt{NN}}(s_t)\|_2 \\ &\text{subject to } \textsc{Verify}(\hat{M}, \pi_\theta, \varphi) = \text{True} \end{aligned} \tag{3}$$

where $\|\cdot\|_2$ is a loss function using the L^2 norm.

Verifying Time-varying Linear Controllers. VELM verifies the safety of a time-varying linear controller π_θ for a learned model $\hat{M}[\cdot]$ using abstract interpretation. While there exist other approaches such as synthesizing barrier certificates for controller verification, the techniques have difficulty handling non-polynomial system dynamics. VELM soundly performs reachability analysis to approximate the set of reachable states of a control system at each timestep:

Definition 2 (Symbolic Rollouts). *Given an environment model $\hat{M}[\pi] = (S, A, F, R, S_0, H, \pi)$ deployed with a controller π, an abstract domain $\mathcal{D}$, an abstract transformer $F^{\mathcal{D}}$ for the state transition function F over $\mathcal{D}$, a symbolic rollout of $M[\pi]$ over $\mathcal{D}$ is $\zeta^{\mathcal{D}} = S_0^{\mathcal{D}}, S_1^{\mathcal{D}}, \ldots, S_H^{\mathcal{D}}$ where $S_0^{\mathcal{D}} = \alpha(S_0)$ is the abstraction of the initial states S_0 in $\mathcal{D}$ and α is the abstraction function of $\mathcal{D}$. Each symbolic state $S_t^{\mathcal{D}}$ over-approximates the set of reachable states from an initial state in S_0 at timestep t. We have $S_{t+1}^{\mathcal{D}} = F^{\mathcal{D}}\big(S_t^{\mathcal{D}}, A_t^{\mathcal{D}}\big)$ where $A_t^{\mathcal{D}}$ overapproximates the set of actions at t. γ is the concretization function of $\mathcal{D}$ for obtaining the set of concrete states represented by an abstract state $S_t^{\mathcal{D}}$.*

Algorithm 2. Synthesize a shield $\pi_{\mathcal{S}}$ for safe exploration of $\pi_{\texttt{NN}}$. $\pi_{\mathcal{S}}$ intervenes to override potentially unsafe actions by $\pi_{\texttt{NN}}$.

```
1: procedure Shield($\hat{M}[\cdot] = \{S, A, F, R, S_0, H, \cdot\}$, $\pi_{\texttt{NN}}$, $\varphi$)
2:     $\pi_\theta \leftarrow$ Approximate($\hat{M}[\cdot]$, $\pi_{\texttt{NN}}$, $\varphi$)                    ▷ Algorithm 3
3:     $S_0^{\mathcal{D}}, S_1^{\mathcal{D}}, \ldots, S_{H-1}^{\mathcal{D}}, S_H^{\mathcal{D}} \leftarrow$ ReachSet($\hat{M}[\pi_\theta]$)
4:     $\pi_{\mathcal{S}} \leftarrow \lambda s.\lambda t.$ let $a_{\texttt{NN}} = \pi_{\texttt{NN}}(s)$ in
5:                 if $\exists 0 \le \boldsymbol{i} \le t+1.\ F(s, a_{\texttt{NN}}) \subset \gamma(S_{\boldsymbol{i}}^{\mathcal{D}})$ then $a_{\texttt{NN}}$
6:                 else let $\boldsymbol{i} = \max\big(\{\boldsymbol{i} \mid s \in \gamma(S_{\boldsymbol{i}}^{\mathcal{D}})\}\big)$ in $\pi_\theta(s, \boldsymbol{i})$
7:     return $\pi_{\mathcal{S}}$
```

The abstract interpreter $F^{\mathcal{D}}$ in VELM uses Taylor Model (TM) flowpipes as the abstract domain $\mathcal{D}$ to reason about the safety of $\hat{M}[\pi_\theta]$. For reachability analysis of $\hat{M}[\pi_\theta]$ at each timestep t (where $t > 0$), VELM gets the TM flowpipe $S_t^{\mathcal{D}}$ for the reachable set of states of $\hat{M}[\pi_\theta]$ at timestep $t-1$. To obtain a TM representation for the output set of the time-varying linear controller π_θ at timestep t, VELM uses TM arithmetic to evaluate a TM flowpipe $A_t^{\mathcal{D}}$ for $\pi_\theta(s,t) = \theta_k(t)^T \cdot s + \theta_b(t)$ for all states $s \in S_t^{\mathcal{D}}$. The resulting TM representation $A_t^{\mathcal{D}}$ can be viewed as an overapproximation of the controller's output at timestep t. Finally, we use Flow* [11] to construct the TM flowpipe overapproximation $S_{t+1}^{\mathcal{D}}$ for all reachable states at timestep t by reachability analysis over the state transition function $F^{\mathcal{D}}(S_t^{\mathcal{D}}, A_t^{\mathcal{D}})$. To verify $\hat{M}(\pi_\theta)$ against a safety property φ, VELM uses Flow* to check if for each abstract state $S_t^{\mathcal{D}}$ in the symbolic rollout of $\hat{M}(\pi_\theta)$, the concretized states in $\gamma(S_t^{\mathcal{D}})$ does not violate φ.

Verified Shielding. The safety of a distilled controller π_θ does not imply its oracle neural controller $\pi_{\texttt{NN}}$ is safe. For safe exploration using $\pi_{\texttt{NN}}$, VELM constructs a shield for $\pi_{\texttt{NN}}$ based on π_θ. The high-level algorithm for shield synthesis is presented in Algorithm 2.

Given a learned environment model $\hat{M}[\cdot]$, a neural controller $\pi_{\texttt{NN}}$, and a safety specification φ, at Line 2, Algorithm 2 invokes Approximate to construct a distillation of $\pi_{\texttt{NN}}$ as a time-varying linear controller π_θ. We describe Approximate in detail in Algorithm 3 and Sect. 3.3. At Line 3, Algorithm 2 uses the symbolic rollout $\zeta^{\mathcal{D}} = S_0^{\mathcal{D}}, S_1^{\mathcal{D}}, \ldots, S_H^{\mathcal{D}}$ of $\hat{M}[\pi_\theta]$ to derive the reachable set of states of π_θ for the learned environment model $\hat{M}[\cdot]$. As this reachable set of states has been verified safe for $\hat{M}[\pi_\theta]$, the shield constrains $\pi_{\texttt{NN}}$ to only explore within the reachable set $\cup_{0 \le i \le H} \gamma(S_i^{\mathcal{D}})$ to remain safe. Algorithm 2 returns a shield $\pi_{\mathcal{S}}$ for $\pi_{\texttt{NN}}$ in the form of a lambda function that takes an environment state s_t at time step t and t as inputs. We show that assuming the learning model soundly approximates the unknown state transition distribution P of the real environment (Sect. 3.1), the shield is provably safe in the following lemma.

Lemma 1. *Assume a learned environment model* $\hat{M}[\cdot] = \{S, A, F, R, S_0, H, \cdot\}$ *is a sound nondeterministic approximation of the true environment:* $\forall s \in S, a \in A.\ s' \sim P(\cdot|s,a) \Rightarrow s' \in F(s,a)$*. Given a safety property* φ*, a neural policy* π_{NN}*, and its shield* $\pi_{\mathcal{S}} =$ Shield($\hat{M}[\cdot]$, π_{NN}, φ)*, for any rollouts* $s_0, a_0, s_1, \ldots, s_H$

collected by π_S *in the true environment where* $s_0 \in S_0$*,* $a_t = \pi_{\mathcal{S}}(s_t, t)$*, and* $s_{t+1} \sim P(\cdot|s_t, a_t)$*, we have* $s_t \models \varphi$ *(i.e.* s_t *is safe) for all* $0 \leq t \leq H$*.*

Proof. Since $\pi_{\mathcal{S}} = \textsc{Shield}(\hat{M}[\cdot],\ \pi_{\texttt{NN}},\ \varphi)$, there exists a π_θ (Line 2 in Algorithm 2) whose symbolic rollouts $S_0^{\mathcal{D}}, S_1^{\mathcal{D}}, \ldots, S_H^{\mathcal{D}}$ can be verified safe with respect to φ (Line 3 of Algorithm 2). We show that for all $0 \leq t \leq H$, we have $\bigvee_{0 \leq i \leq t} s_t \in \gamma(S_i^{\mathcal{D}})$. This invariant implies that s_t is safe. We prove the invariant by induction. When $t = 0$, the invariant holds as $s_0 \in \gamma(S_0^{\mathcal{D}})$ by construction. Given an s_t that satisfied the invariant, if $\exists 0 \leq i \leq t+1$. $F(s_t, \pi_{\texttt{NN}}(s_t)) \subset \gamma(S_i^{\mathcal{D}})$ (Line 5), then $a_t = \pi_{\text{NN}}(s_t)$ and by assumption $s_{t+1} \sim P(\cdot|s_t, a_t) \in F(s_t, a_t) \subset \gamma(S_i^{\mathcal{D}})$, which means the invariant holds on s_{t+1} in this case. Otherwise (Line 6), $a_t = \pi_\theta(s_t, i)$ where $i = \max\left(\{i \mid s_t \in \gamma(S_i^{\mathcal{D}}\}\right)$. Such i must exist as we assume s_t satisfied the invariant. Since $s_{t+1} \sim P(\cdot|s_t, a_t) \in F(s_t, a_t)$ and the soundness of the abstract interpreter $F^{\mathcal{D}}$ ensures that if $s_t \in \gamma(S_i^{\mathcal{D}})$, then $F(s_t, a_t) \subseteq \gamma(S_{i+1}^{\mathcal{D}})$, which means the invariant holds on s_{t+1} in this case as well. By induction, the invariant is true for all $0 \leq t \leq H$.

Theorem 1 (Shield (Algorithm 2) is probabilistically safe). *For a learned environment model* $\hat{M}[\cdot] = \{S, A, F, R, S_0, H, \cdot\}$*, let* δ_M *be the probability bound of the model:* $\Pr_{s' \sim P(\cdot|s,a)}\big[s' \notin F(s,a)\big] < \delta_M$*. Given a safety property* φ*, a neural policy* π_{NN}*, and its shield* $\pi_{\mathcal{S}} = \textsc{Shield}(\hat{M}[\cdot],\ \pi_{\mathit{NN}},\ \varphi)$*, for any rollouts* $s_0, a_0, s_1, \ldots, s_H$ *collected by* π_S *in the true environment where* $s_0 \in S_0$*,* $a_t = \pi_{\mathcal{S}}(s_t, t)$*, and* $s_{t+1} \sim P(\cdot|s_t, a_t)$*, we have* $s_t \models \varphi$ *(i.e.* s_t *is safe) with probability at least* $(1 - \delta_M)^t$ *for all* $0 \leq t \leq H$*.*

Proof. By Lemma 1, if $s_{t+1} \in F(s_t, a_t)$, then s_{t+1} is safe for all $0 \leq t < H$. By assumption, at each time step, we have $s_{t+1} \in F(s_t, a_t)$ with probability at least $1 - \delta_M$. After t time steps, the probability that the assumption is valid is at least $(1 - \delta_M)^t$, which means that s_t is safe with probability at least $(1 - \delta_M)^t$.

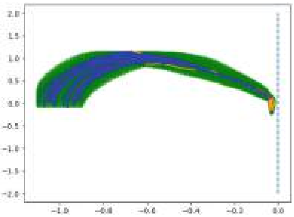


Fig. 3. Executing a shielded neural policy in ACC. The green region denotes the safe space verified on a learned model. The yellow regions denote the control steps where intervention takes place. (Color figure online)

We can relate the probability guarantee in Theorem 1 with our overall objective in Eq. 1 by bounding $\delta_M < 1 - (1 - \delta)/\exp(H)$. This theorem illustrates that VELM only allows for safety violations when there's an inaccuracy in the environment model. In contrast, existing approaches to safe exploration are susceptible to safety violations stemming from both modeling inaccuracies and actions that are not safe even considering the environment model. For example, SPICE [4] applies weakest precondition generation from safety constraints to a linearization of the learned environment model to determine safe control actions. However, this linearization process introduces substantial approximation errors, compromising the safety of the computed actions on the learned environment model. CRABS [34] uses neural networks for representing environment models and control barrier certificates to identify safe exploration regions. However, a neural control barrier certificate may converge to a suboptimal model and CRABS does not have a procedure to rigorously guarantee its correctness. This may result in delayed or absent intervention for unsafe behaviors.

Example 2. Consider an adaptive cruise control (ACC) system [5]. The goal is to control an ego car to closely follow a lead car without collision. The lead car can apply acceleration to itself at any time. Figure 3 shows the rollouts (blue) of a *shielded* neural controller $\pi_{\texttt{NN}}$ in the real environment. The x-axis shows the distance to the lead car while the y-axis shows the relative velocities of the two cars. The rollouts start by accelerating to close the gap with the lead car and subsequently decelerating to prevent a collision. The green region denotes the reachable set of a distilled time-varying linear controller π_θ verified as safe on a learned model of the ACC environment. The yellow regions indicate interventions where π_θ constrains $\pi_{\texttt{NN}}$ to stay within the safe region. Without such intervention, the neural controller alone would fail to decelerate rapidly enough and crash into the lead car (the dashed line on the right side). At times, π_θ needs to intervene well before the final steps to ensure the feasibility of avoiding a crash later.

3.3 Neural Controller Approximation

This section formalizes the Approximate procedure invoked by Algorithm 2 (Line 2) for distilling a neural controller $\pi_{\texttt{NN}}$ to a time-varying linear controller π_θ that can be verified safe according to a learned environment model.

Minimizing the gap between π_θ and a (fixed) neural controller $\pi_{\texttt{NN}}$ as two functions can be straightforwardly achieved by optimizing θ through gradient descent. However, a binary verification result (true or false) does not offer guidance on how θ should be optimized to ensure that π_θ can be verified safe. Following previous research [45], when facing verification failures, our approach utilizes verification feedback, indicating the extent of safety violations, to guide the optimization process for π_θ. We first formalize the concept of safety violation within the concrete environment state space and then lift it to abstract state spaces.

Definition 3 (State Safety Loss Function). *For a safety specification φ over states $s \in S$, we define a non-negative loss function $\mathcal{L}(s, \varphi)$ such that $\mathcal{L}(s, \varphi) = 0$ iff s satisfies φ, i.e. $s \models \varphi$. We define $\mathcal{L}(s, \varphi)$ recursively, based on the possible shapes of φ (Definition 1):*

- $\mathcal{L}(s, \mathcal{A} \cdot x \le b) := \max(\mathcal{A} \cdot s - b, 0)$
- $\mathcal{L}(s, \varphi_1 \wedge \varphi_2) := \max(\mathcal{L}(s, \varphi_1), \mathcal{L}(s, \varphi_2))$
- $\mathcal{L}(s, \varphi_1 \vee \varphi_2) := \min(\mathcal{L}(s, \varphi_1), \mathcal{L}(s, \varphi_2))$

Notice that $\mathcal{L}(s, \varphi_1 \wedge \varphi_2) = 0$ iff $\mathcal{L}(s, \varphi_1) = 0$ and $\mathcal{L}(s, \varphi_2) = 0$, and similarly $\mathcal{L}(\varphi_1 \vee \varphi_2) = 0$ iff $\mathcal{L}(\varphi_1) = 0$ or $L(\varphi_2) = 0$.

We extend the safety loss definition (Definition 3) to the abstract state space employed in a verification procedure.

Definition 4 (Abstract State Safety Loss Function). *Given an abstract state $S^{\mathcal{D}}$ and a safety specification φ, we define an abstract safety loss function:*

$$\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi) = \max_{s \in \gamma(S^{\mathcal{D}})} \mathcal{L}(s, \varphi)$$

It quantifies the worst-case safety loss of φ across all concrete states encompassed by $S^{\mathcal{D}}$. For an abstract domain $\mathcal{D}$, we typically can approximate the concretization of an abstract state $\gamma(S^{\mathcal{D}})$ using a tight interval $\gamma_I(S^{\mathcal{D}})$. For example, it is straightforward to represent Taylor model flowpipes as intervals in Flow. Based on the potential structure of φ, we redefine $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi)$ as:*

- $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \mathcal{A} \cdot x \leq b) := \max_{s \in \gamma_I(S^{\mathcal{D}})} \big(\max(\mathcal{A} \cdot s - b, 0)\big)$
- $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_1 \wedge \varphi_2) := \max(\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_1), \mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_2))$
- $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_1 \vee \varphi_2) := \min(\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_1), \mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_2))$

By definition, we have $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi) = 0 \implies \forall s \in \gamma_I(S^{\mathcal{D}}).\ s \models \varphi$.

We further lift the definition of safety loss over abstract states (Definition 4) to the symbolic rollout of an MDP (Definition 2).

Definition 5 (Symbolic Rollout Safety Loss). *Given an environment model $\hat{M}[\pi_\theta]$ and a safety specification φ, assuming the symbolic rollout of $\hat{M}[\pi_\theta]$ over an abstract domain $\mathcal{D}$ is $\zeta^{\mathcal{D}}_{0:H} = S^{\mathcal{D}}_0, \ldots, S^{\mathcal{D}}_H$, we define an abstract safety loss function to measure the degree to which φ is violated by $\hat{M}[\pi_\theta]$:*

$$\mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta],\ \varphi)\ =\ \mathcal{L}_{\mathcal{D}}(\zeta_{0:H},\ \varphi)\ =\ \max_{0 \leq i \leq H}(\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}_i, \varphi))$$

Definition 5 enables a quantitative metric for the safety loss of a controller π_θ in the abstract state space of a safety verifier. By definition, we have

$$\mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi) = 0 \implies \hat{M}[\pi_\theta] \models \varphi.$$

We rewrite the overall objective of distilling a neural controller $\pi_{\texttt{NN}}$ into a time-varying linear controller π_θ in Eq. 3 as:

$$\begin{aligned} &\min_\theta \mathbb{E}_{s_0, s_1, \ldots, s_H \sim \hat{M}[\pi_\theta]}\ \|\pi_\theta(s_t, t) - \pi_{\texttt{NN}}(s_t)\|_2 \\ &\quad \text{subject to } \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi) = 0 \end{aligned} \tag{4}$$

The objective described in Eq. 4 frames a constraint optimization problem. To address this, we employ Lagrangian optimization, which provides a principled way to seamlessly incorporate the verification constraint ($\mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi) = 0$) into the distillation objective. We introduce a Lagrangian function that incorporates a Lagrange multiplier λ to account for constraint violation and transform Eq. 4 into an unconstrained optimization problem:

$$L(\theta, \lambda) = \mathcal{L}_S(\pi_\theta, \pi_{\texttt{NN}}) + \lambda \cdot \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi)$$

Algorithm 3. Approximate a neural control policy $\pi_{\mathtt{NN}}$ with a time-varying linear controller π_θ while ensuring π_θ is formally verified safe for the learned model $\hat{M}$.

```
 1: procedure APPROXIMATE($\hat{M}[\cdot] = \{S, A, F, R, S_0, H, \cdot\}$, $\pi_{\mathtt{NN}}$, $\varphi$)
 2:     Initialize a time-varying linear policy $\pi_\theta$ over $H$ timesteps
 3:     $\theta \leftarrow$ all parameters in $\pi_\theta$ for optimization
 4:     while true do
 5:         $\ell_S \leftarrow \mathcal{L}_S(\pi_\theta, \pi_{\mathtt{NN}})$
 6:         $\ell_{\mathcal{D}} \leftarrow \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi)$
 7:         if $\ell_{\mathcal{D}} = 0$ and $\ell_S$ converges then
 8:             return $\pi_\theta$
 9:         $\theta \leftarrow \theta - \eta_\theta \cdot \big(\nabla_\theta \mathcal{L}_S(\pi_\theta, \pi_{\mathtt{NN}}) + \lambda \cdot \nabla_\theta \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi)\big)$
10:         $\lambda \leftarrow \lambda + \eta_\lambda \cdot \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi)$
```

where $\mathcal{L}_S(\pi_\theta, \pi_{\mathtt{NN}}) = \mathbb{E}_{s_0, s_1, \ldots, s_H \sim \hat{M}[\pi_\theta]} \; \|\pi_\theta(s_t, t) - \pi_{\mathtt{NN}}(s_t)\|_2$. VELM seeks to optimize the primal parameter θ and the Lagrange multiplier λ to minimize the function $L(\theta, \lambda)$, effectively reducing both the L^2 loss for the distillation objective and safety violations for the verification constraint.

Algorithm 3 outlines the procedure for distilling $\pi_{\mathtt{NN}}$ to π_θ. It iteratively performs the following two gradient-based update rules to minimize $L(\theta, \lambda)$:

$$\theta \leftarrow \theta - \eta_\theta \cdot \big(\nabla_\theta \mathcal{L}_S(\pi_\theta, \pi_{\mathtt{NN}}) + \lambda \cdot \nabla_\theta \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi)\big)$$
$$\lambda \leftarrow \lambda + \eta_\lambda \cdot \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi)$$

where η_θ is a learning rate for θ and η_λ is a learning rate for λ. The Lagrange multiplier λ is increased during the optimization process to penalize deviations from satisfying the verification constraint. As such, even though the verification procedure may introduce approximation errors, VELM can reduce this error by conducting optimization in the abstract state space [45]. VELM repeats the iterative update until the distillation loss (ℓ_S) converges and the safety violation loss (ℓ_D) converges to 0 (Line 8).

Gradient Estimation for $\mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi)$. Deriving the gradients of the verification constraint $\mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi)$ directly poses a challenge, as it requires the verification procedure to be differentiable, a feature not practical. To address this obstacle, following prior research [45], VELM estimates the gradients of $\mathcal{L}_{\mathcal{D}}$ through random search [36]. In each training iteration, given a closed-loop environment $\hat{M}[\pi_\theta]$, we generate perturbed systems $\hat{M}[\pi_{\theta+\nu\omega}]$ and $\hat{M}[\pi_{\theta-\nu\omega}]$ by introducing sampled Gaussian noise ω to the current controller π_θ's parameters θ in both directions. Here, ν represents a small positive real number. By assessing the abstract safety losses of the symbolic rollouts for $\hat{M}[\pi_{\theta+\nu\omega}]$ and $\hat{M}[\pi_{\theta-\nu\omega}]$, we update θ using a finite difference approximation along an unbiased estimator of the gradient:

$$\nabla_\theta \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_\theta], \varphi) \leftarrow \frac{1}{N} \sum_{k=1}^{N} \frac{\Big(\mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_{\theta+\nu\omega_k}],\ \varphi) - \mathcal{L}_{\mathcal{D}}(\hat{M}[\pi_{\theta-\nu\omega_k}], \varphi)\Big)}{\nu} \omega_k$$

Performance Guarantees. We conclude the technical section by discussing the reward performance of VELM. One important concern is whether shielding a neural control policy hinders the RL algorithm's ability to learn the optimal policy. Previous studies [4,5,44] have established the following regret bound concerning the reward performance of a shielded policy for safe exploration compared to the optimal policy that does not seek to restrict safety violations during the learning process. Let $\pi_S^i = \textsc{Shield}(\hat{M}^i[\cdot], \pi_{\mathtt{NN}}^i, \varphi)$ for $1 \leq i \leq T$ be a sequence of policies learned in Algorithm 1 where φ is the safety property, $\hat{M}^i[\cdot]$ and $\pi_{\mathtt{NN}}^i$ are the learned environment model and neural controller at the i^{th} iteration. Introduce a safety indicator Z that takes the value 1 when $\pi_S^i(s,t) = \pi_{\mathtt{NN}}^i(s)$ and 0 otherwise, and let $\xi = \mathbb{E}[1 - Z]$ be the frequency with which π_S^i intervenes in neural policy controls. Assume the reward function is Lipschitz on the controller parameter space and let L_R be the corresponding Lipschitz constant. Let β and τ^2 be the bias and variance in the gradient estimate that is incurred due to sampling. Let $\epsilon_{\mathcal{S}}$ be an upper bound on the imprecision incurred by distilling $\pi_{\mathtt{NN}}^i$ to a linear time-varying controller. Let ϵ_m be an upper bound for the Kullback-Leibler divergence between the learned environment model and the true environment dynamics at all time steps. Let ϵ_π be an upper bound on the total variational divergence between the policy used to gather data and the policy being trained at all time steps. Set the learning rate η of the RL algorithm for updating $\pi_{\mathtt{NN}}^i$ as $\sqrt{\frac{1}{\tau^2}(\frac{1}{T} + \epsilon_{\mathcal{S}})}$. Assuming π^* is the (unknown) the optimal safe control policy, we have the following regret bound [4,5,44] for Algorithm 1: $R(\pi^*) - \mathbb{E}\big[\frac{1}{T}\sum_{i=1}^{T} R(\pi_S^i)\big] = O\big(\sqrt{\frac{1}{\tau^2}(\frac{1}{T} + \epsilon_{\mathcal{S}})} + \beta + L_R \cdot \xi + \epsilon_m + \epsilon_\pi\big)$. VELM does not impose a significant penalty on the agent's reward performance for achieving safety as the regret bound becomes tighter when the frequency of interventions in the decision of the neural controllers ξ decreases during training. As the environment model improves during training (i.e. ϵ_m and ϵ_π decrease), the controller converges to higher rewards. The remaining terms are associated with the standard error by using sampling to approximate policy update gradients.

4 Experiments

In our implementation of VELM[3], we use SAC [23], a state-of-the-art reinforcement learning algorithm, as the base algorithm to optimize neural network controllers. We build the abstract interpreter for reachability analysis of a time-varying linear controller against a learned model on top of Flow* [11] for reasoning about nonlinear state transition functions. We use Operon [9] to learn a symbolic environment model for the LearnModel procedure at Line 10 in Algorithm 1. In the implementation, we invoke the LearnModel procedure only when the existing environment model is invalid for the newly collected trajectories from the real environment. Recall that our learned model is nondeterministic (Sect. 3.1). Given a current state, it outputs a range for the next state. If the

[3] VELM is available at https://github.com/RU-Automated-Reasoning-Group/VELM.

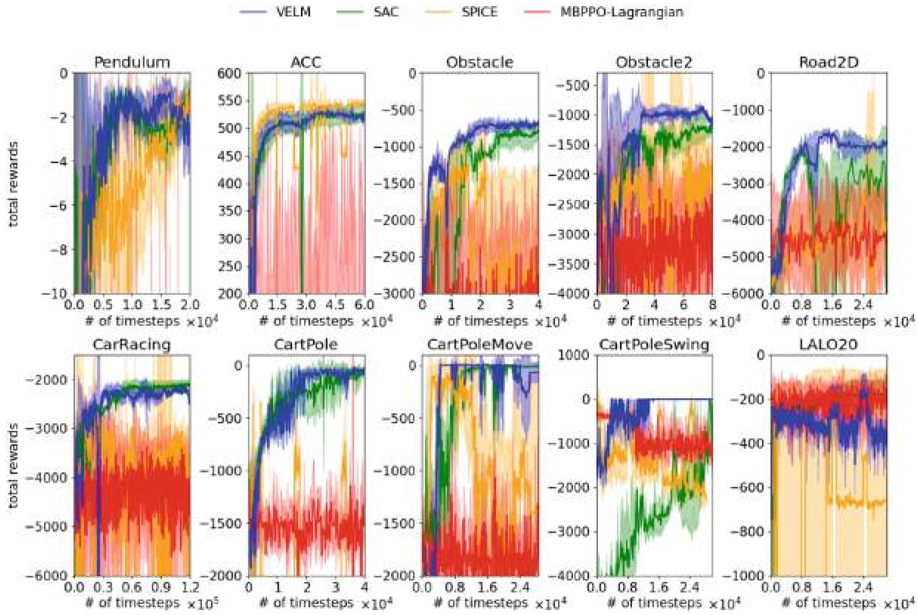


Fig. 4. Rewards for all the tools throughout the training phase. The solid curve represents the mean across 5 random seeds. The shaded area indicates the standard deviation.

actual next state is not within the range, we consider the model invalid (i.e., $\exists t. s_{t+1} \notin F(s_t, a_t)$). This strategy significantly accelerates the learning process.

Baselines. We compared VELM with three baselines: SAC, SPICE [4], and MBPPO-Lagrangian [26]. The SAC baseline acts as an upper bound on reward performance since the agent does not need to explicitly handle safety constraints. The other safe RL baselines are relevant because they are all model-based as VELM. However, they all use neural networks for learning environment state transition dynamics. SPICE applies weakest precondition generation from safety constraints to a linearized form of learned environment models to ascertain safe control actions for shielding. The linearization step may introduce approximation errors. MBPPO-Lagrangian finds a safety-constraint-satisfying policy by using the Lagrangian method to reduce the cumulative safety violations throughout trajectories executed on the learned model. This method does not consider shielding to ensure safe exploration. We also tried to use CRABS [34] as another model-based safe-learning baseline. In addition to a neural environment model, CRABS uses another neural network to learn a control barrier certificate to identify a safe region on the neural environment model for shielding. However, we found that CRABS is excessively time-consuming to execute, completing only an average of 10 episodes within a day. Therefore, we have excluded CRABS in the results presented in this section. In summary, these baselines suffer from safety violations stemming from both (1) environment modeling imprecision and (2)

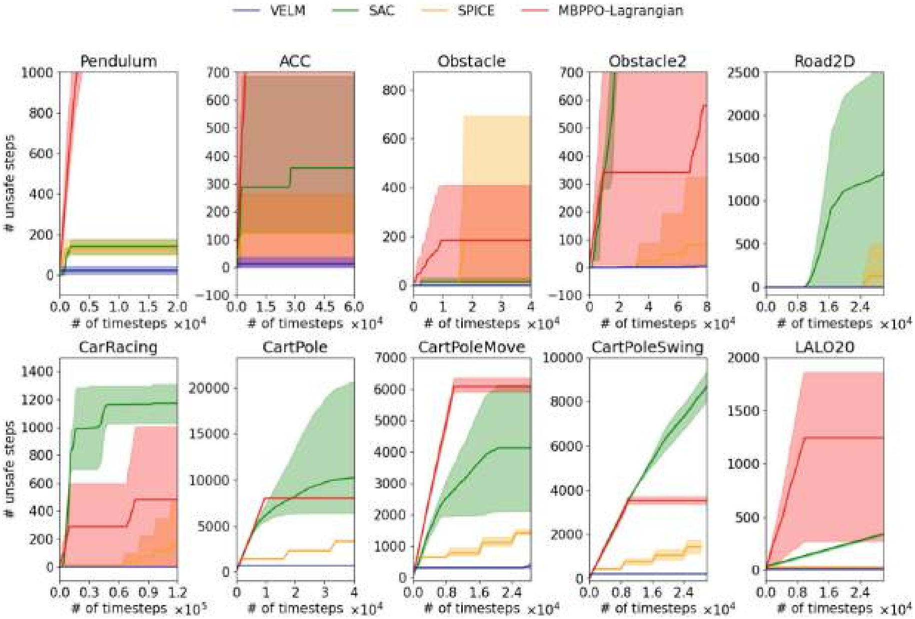


Fig. 5. Cumulative safety violations for all the tools throughout the training phase. The solid curve represents the mean across 5 random seeds. The shaded area indicates the range between the minimum and maximum values.

control policies that are not safe even considering the environment model. VELM eliminates the second source of errors. Our experiments aim to answer the question - *How does the performance of VELM compare to representative baseline approaches, considering metrics such as rewards, number of unsafe steps, and overall efficiency?*

Benchmarks. We used the benchmarks considered in related work. Pendulum, ACC, Obstacle, Obstacle2, Road2D, and CarRacing are taken from the SPICE benchmarks [4,5]. In Road2D, an autonomous vehicle is controlled to reach a designated destination while adhering to a specified speed limit. Obstacle and Obstacle2 pose a challenge for a 2D robot to reach a specified goal while avoiding an obstacle. In Obstacle, the obstruction is positioned to the side, affecting the agent only during exploration, without cutting the shortest path to the goal. In Obstacle2, the obstruction is placed between the initial region and the goal region, requiring the learned controller to navigate around it. In the Pendulum task, the objective is to maintain a pendulum in the upright position. The goal of ACC (adaptive cruise control) is to closely follow a leading vehicle without collision, with the lead car selecting acceleration randomly from a truncated normal distribution at each time step. The CarRacing environment is similar to Obstacle2 but the goal is to reach a goal region on the opposite side of the obstacle and then return to the initial region. This requires the agent to complete a loop around the obstacle to fulfill the objective. Cartpole is from Open AI Gym [8]. The nonlinear benchmarks CartPoleMove and CartPoleSwing are taken from CRABS [34]. The CartPoleMove task is challenging as high-reward policies must carefully explore near the safety boundary. The user-specified safety set is $\{(x, \theta) : |\theta| \leq \theta_{\mathtt{max}} = 0.2, |x| \leq 0.9\}$ where x is the cart horizontal position and θ is the pole angle. $\theta_{\mathtt{max}}$ corresponds to approximately $11°$. The reward function of the task is $r(s, a) = x^2$. Consequently, the optimal policy must delicately move the cart and pole toward the boundary of unsafe regions but remain safe. Similarly, the CartPoleSwing task is also high-risk, high-reward environment. The reward function is $r(s, a) = \theta^2$ and the user-specified safety set is $\{(x, \theta) : |\theta| \leq \theta_{\mathtt{max}} = 1.5, |x| \leq 0.9\}$. So the optimal policy will swing back and forth to some degree close to $90°$ but prevent the pole from falling. LALO20 is a challenging 7-dimensional nonlinear benchmark modeling a molecular network taken from prior work [48]. This task is difficult because the initial states are situated near the boundary of the unsafe region.

Results. We report the mean reward performance of the learned controllers as well as cumulative safety violations over time during training of each benchmark for VELM and each baseline in Fig. 4 and Fig. 5. The shield intervention rates of VELM and SPICE are listed in Table 1. These results are averaged over 5 random seeds.

Table 1. Comparison of Shield Intervention Rates between VELM and SPICE.

Benchmarks	VELM	SPICE
Pendulum	0.07	0.00
ACC	0.23	0.78
Obstacle	0.11	0.70
Obstacle2	0.26	0.34
Road2D	0.03	0.18
CarRacing	0.17	0.44
CartPole	0.12	0.81
CartPoleMove	0.16	0.55
CartPoleSwing	0.01	0.78
LALO20	0.49	0.79

Figure 5 demonstrates that VELM exhibits superior safety performance as it experiences a significantly lower frequency of unsafe steps compared to the baseline methods. Except for the initial controller π_0, the controllers learned by VELM demonstrate nearly zero safety violations when interacting with the real environment in training. SPICE accumulates safety violations more quickly compared to VELM. Overall, VELM achieves a 99.7% reduction in unsafe steps compared to SPICE. SPICE incurs significantly more safety violations in highly nonlinear environments such as CartPole. This suggests that the model linearization step in SPICE introduces significant approximation errors, resulting in either unnecessary interventions or a lack of intervention when there is truly unsafe behavior. This kind of approximation error also limits SPICE to use a bounded-time analysis to determine potential safety violations within the next few time steps (5, as recommended in SPICE [4]). VELM instead can predict the long-term safety of an action far into the future. For example, on CartPole, the average shield intervention rate for SPICE over all the rollouts in the real environment is 81%, while VELM only has an intervention rate of 12%. Similarly, VELM is safer than MBPPO-Lagrangian in every benchmark. As can be seen from Fig. 4, MBPPO-Lagrangian continues to violate the safety property more over time than VELM. Principally, in contrast to VELM, MBPPO-Lagrangian seeks to limit safety violations in expectation and does not assure safety for all visited states.

Table 2. Training time in seconds for model, network and shield updates

Benchmarks	Model (s)	Network (s)	Shield (s)
Pendulum	6.4	184.0	17.1
ACC	489.9	616.4	57.0
Obstacle	18.6	657.0	33.3
Obstacle2	47.7	661.8	436.5
Road2D	19.8	884.0	62.0
CarRacing	41.2	648.3	209.4
CartPole	21.0	577.5	176.8
CartPoleMove	13.3	302.1	384.5
CartPoleSwing	13.2	311.1	10.5
LALO20	47.8	302.3	808.1

Figure 4 also demonstrates that in most cases, VELM attains comparable (or slightly superior) reward performance to SAC. SPICE imposes a substantial penalty on reward performance compared to SAC. This is because SPICE in general exhibits significantly higher shield intervention rates than VELM. As discussed in the performance guarantee analysis in Sect. 3, frequent shield interventions hinder the RL algorithm from converging to the optimal policy. LALO20 is the only benchmark that VELM does not achieve a comparable reward performance to SAC. This is because, in this benchmark, the average shield intervention rate for VELM over all the rollouts in the real environment is relatively high at 49%. However, VELM achieves nearly 0 safety violations during learning. The modest performance penalty is an acceptable trade-off for safety. Although SPICE also achieves almost 0 safety violations on LALO20, its shield intervention rate is 79%, preventing the neural policy from achieving high reward performance.

We present the execution times for each component of VELM across all benchmarks in Table 2, averaged over five random seeds. The Network column in the table reports the time spent training a neural network controller using the base RL algorithm. The Model and Shield columns report the time spent on learning a symbolic environment model and constructing a formally verified shield, respectively. On average, VELM dedicates approximately 9% of its execution time to model learning and 28% to shield construction. This modest overhead is justified by the substantial safety guarantees provided.

5 Related Work

Prior Safe RL works consider constrained Markov decision processes (CMDP), where observed safety violations should be bounded. Lagrangian methods are widely used to solve CMDP with the Lagrangian multiplier controlled adaptively [41] or by PID [40]. Trust region methods [1,47,51] project a current control policy to a feasible safe space around the current policy in each learning iteration. The goal is to bound the number of safety violations under a threshold *in expectation*, while VELM aims to ensure safety for all visited states. Combining these methods with learning a dynamics model can further improve their data efficiency [26,50]. There exist works that learn conservative safety critics to underestimate the long-term safety cost of taking a particular action in a particular state and use the conservative safety critics for safe exploration and policy optimization [7,46,49]. However, training neural safety critics models may require numerous potentially unsafe environment interactions. VELM instead uses symbolic reachability analysis over learned environment models to identify safe regions of the state space. Other approaches involve pre-training a policy in a simpler environment and fine-tuning it in a more challenging setting [39], or leveraging existing offline data and co-training a recovery policy [42]. Integrating VELM with pretraining and offline data is an interesting avenue for future research.

Another research direction explores Lyapunov functions and barrier certificates. The work in [6] uses Lyapunov functions to identify policy attraction regions where safe operation is guaranteed for discretized deterministic control systems, provided that certain Lipschitz continuity conditions hold. However, this method requires access to system dynamics models. Additionally, a neural network controller may not exhibit Lipschitz continuity with a reasonable coefficient. In [13], it is shown that Lyapunov functions can be co-learned with controllers for discrete action spaces. This work was extended to continuous action spaces by utilizing the Deterministic Policy Gradient theorem [38]. The work by Chow et al. [14] projects control actions to guarantee a decrease in the Lyapunov function after each timestep. In contrast, Donti et al. [17] construct sets of stabilizing actions using a Lyapunov function and then project actions onto this set. A handcrafted barrier function is leveraged in [12] to secure safe exploration in reinforcement learning. A line of research, exemplified by a prior study [48], focuses on verifying an RL controller upon convergence against safety and reachability properties by inferring barrier certificates but does not address safety during training. Combining VELM with such work is promising for future investigation.

Model-based safe reinforcement learning approaches ensure the safety of an RL agent through a model of its environment. When a pre-established model of environmental dynamics is available, a safety shield and a backup controller can be constructed from the model using formal methods to regulate agent behavior [3]. To enforce the safety of a deep neural network controller, the backup controller is run in tandem with the neural controller [2,5,18,20–22,24,29,31,52]. Whenever the neural controller is about to leave the provably safe state space governed by the backup controller, the backup controller overrides the potentially unsafe neural actions to enforce the neural controller to stay within the certified safe space. When environment dynamics models are not known a priori, several works [26,32,33,35] maintain a learned environment model and employ various statistical techniques to devise a policy that is likely to be safe according to the model. This gives rise to two sources of unsafe conduct: the policy may exhibit unsafety in relation to the model, or the model could provide an imprecise depiction of the environment. VELM addresses the first source of error by assuring control policies are safe within the confines of an environment model. REVEL [5] involves an iterative learning approach where a neural policy is trained, potentially resulting in unsafety. Subsequently, the learned neural policy is distilled into a piecewise linear policy. Automatic verification is then applied to certify the piecewise linear policy, a process akin to constructing a barrier function. First, this certification method assumes a calibrated dynamics model, whereas VELM, in contrast, learns the dynamics model. Second, the verification algorithm in REVEL requires a piecewise linear environment model to be manually constructed to approximate the calibrated dynamics model, a condition not practical in VELM (learned environment models evolve across learning iterations in VELM). CRABS [34] iteratively learns a barrier certificate, a dynamics model, and a control policy where the barrier certificate, learned via adversarial training, ensures the policy's safety assuming the learned dynamics model. Yet, formally verifying the correctness of the barrier certificate faces challenges as both the certificate and the underlying environment model are complex, deep neural network models. SPICE [4] determines action safety using weakest preconditions derived from a learned neural environment model within a short time horizon H. However, extending H to cover the entire horizon of an RL task faces challenges due to the difficulty of constructing precise weakest precondition transformers for neural networks and the accumulation of approximation errors inherent in linearizing a neural environment model. Instead, VELM conducts formally verified exploration for RL agents, covering the entire horizon of an RL task through learned environment models.

6 Conclusion

In summary, we present VELM, a novel framework for ensuring verified safe exploration in model-based reinforcement learning. VELM learns environment models as symbolic formulas. Through formal reachability analysis over learned models, VELM constructs an online shielding layer that acts as a safeguard, confining RL agent exploration to a state space verified as safe in the learned model. The results of our experiments in various RL environments, alongside comparisons with state-of-the-art safe RL techniques, highlight the efficacy of VELM in significantly mitigating safety violations during online exploration while maintaining strong learning performance. VELM thus establishes a foundation for building trustworthy and secure RL systems capable of navigating complex environments while adhering to stringent safety constraints.

References

1. Achiam, J., Held, D., Tamar, A., Abbeel, P.: Constrained policy optimization. In: Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6–11 August 2017. Proceedings of Machine Learning Research, vol. 70 (2017)
2. Akametalu, A.K., Kaynama, S., Fisac, J.F., Zeilinger, M.N., Gillula, J.H., Tomlin, C.J.: Reachability-based safe learning with gaussian processes. In: 53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, 15–17 December, 2014 (2014)
3. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, 2–7 February, 2018 (2018)
4. Anderson, G., Chaudhuri, S., Dillig, I.: Guiding safe exploration with weakest preconditions. In: The Eleventh International Conference on Learning Representations (2023)
5. Anderson, G., Verma, A., Dillig, I., Chaudhuri, S.: Neurosymbolic reinforcement learning with formally verified exploration. In: Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020)
6. Berkenkamp, F., Turchetta, M., Schoellig, A.P., Krause, A.: Safe model-based reinforcement learning with stability guarantees. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4–9 December, 2017, Long Beach, CA, USA (2017)
7. Bharadhwaj, H., Kumar, A., Rhinehart, N., Levine, S., Shkurti, F., Garg, A.: Conservative safety critics for exploration. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, 3–7 May, 2021 (2021)
8. Brockman, G., et al.: Openai gym (2016)
9. Burlacu, B., Kronberger, G., Kommenda, M.: Operon c++: an efficient genetic programming framework for symbolic regression. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO 2020 (2020)
10. Cava, W.G.L., et al.: Contemporary symbolic regression methods and their relative performance. In: Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, Virtual, December 2021
11. 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
12. Cheng, R., Orosz, G., Murray, R.M., Burdick, J.W.: End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, 27 January–1 February, 2019 (2019)
13. Chow, Y., Nachum, O., Duéñez-Guzmán, E.A., Ghavamzadeh, M.: A lyapunov-based approach to safe reinforcement learning. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada (2018)
14. Chow, Y., Nachum, O., Faust, A., Duéñez-Guzmán, E.A., Ghavamzadeh, M.: Safe policy learning for continuous control. In: 4th Conference on Robot Learning, CoRL

2020, 16–18 November 2020, Virtual Event/Cambridge, MA, USA. Proceedings of Machine Learning Research, vol. 155 (2020)

15. Dalal, G., Dvijotham, K., Vecerík, M., Hester, T., Paduraru, C., Tassa, Y.: Safe exploration in continuous action spaces. CoRR **abs/1801.08757** (2018)
16. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
17. Donti, P.L., Roderick, M., Fazlyab, M., Kolter, J.Z.: Enforcing robust control guarantees within neural network policies. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, 3–7 May, 2021 (2021)
18. Fisac, J.F., Akametalu, A.K., Zeilinger, M.N., Kaynama, S., Gillula, J.H., Tomlin, C.J.: A general safety framework for learning-based control in uncertain robotic systems. IEEE Trans. Autom. Control. **64**(7) (2019)
19. François-Lavet, V., Henderson, P., Islam, R., Bellemare, M.G., Pineau, J.: An introduction to deep reinforcement learning. Found. Trends. Mach. Learn. **11**(3-4) (2018)
20. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, 2–7 February, 2018 (2018)
21. Fulton, N., Platzer, A.: Verifiably safe off-model reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 413–430. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_28
22. Gillula, J.H., Tomlin, C.J.: Guaranteed safe online learning via reachability: tracking a ground target using a quadrotor. In: IEEE International Conference on Robotics and Automation, ICRA 2012, 14–18 May, 2012, St. Paul, Minnesota, USA (2012)
23. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80 (2018)
24. Hunt, N., Fulton, N., Magliacane, S., Hoang, T.N., Das, S., Solar-Lezama, A.: Verifiably safe exploration for end-to-end reinforcement learning. In: HSCC '21: 24th ACM International Conference on Hybrid Systems: Computation and Control, Nashville, Tennessee, 19–21 May, 2021 (2021)
25. Janner, M., Fu, J., Zhang, M., Levine, S.: When to trust your model: Model-based policy optimization. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8–14 December, 2019, Vancouver, BC, Canada (2019)
26. Jayant, A.K., Bhatnagar, S.: Model-based safe deep reinforcement learning via a constrained proximal policy optimization algorithm. In: NeurIPS (2022)
27. Johnson, T.T., et al.: ARCH-COMP21 category report: artificial intelligence and neural network control systems (AINNCS) for continuous and hybrid systems plants. In: 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21), Brussels, Belgium, July 9, 2021. EPiC Series in Computing, vol. 80 (2021)
28. Kamienny, P., d'Ascoli, S., Lample, G., Charton, F.: End-to-end symbolic regression with transformers. In: NeurIPS (2022)
29. Koller, T., Berkenkamp, F., Turchetta, M., Krause, A.: Learning-based model predictive control for safe exploration. In: 57th IEEE Conference on Decision and Control, CDC 2018, Miami, FL, USA, 17–19 December, 2018 (2018)

30. Kronberger, G., de França, F.O., Burlacu, B., Haider, C., Kommenda, M.: Shape-constrained symbolic regression - improving extrapolation with prior knowledge. Evol. Comput. **30**(1) (2022)
31. Li, S., Bastani, O.: Robust model predictive shielding for safe reinforcement learning with stochastic dynamics. In: 2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020 (2020)
32. Li, Y., Li, N., Tseng, H.E., Girard, A., Filev, D.P., Kolmanovsky, I.V.: Safe reinforcement learning using robust action governor. In: Proceedings of the 3rd Annual Conference on Learning for Dynamics and Control, L4DC 2021, 7–8 June 2021, Virtual Event, Switzerland. Proceedings of Machine Learning Research, vol. 144 (2021)
33. Liu, Z., Zhou, H., Chen, B., Zhong, S., Hebert, M., Zhao, D.: Safe model-based reinforcement learning with robust cross-entropy method. CoRR **abs/2010.07968** (2020)
34. Luo, Y., Ma, T.: Learning barrier certificates: towards safe reinforcement learning with zero training-time violations. In: Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, 6–14 December, 2021, virtual (2021)
35. Ma, Y.J., Shen, A., Bastani, O., Jayaraman, D.: Conservative and adaptive penalty for model-based safe reinforcement learning. In: Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Virtual Event, February 22 - March 1, 2022 (2022)
36. Mania, H., Guy, A., Recht, B.: Simple random search of static linear policies is competitive for reinforcement learning. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada (2018)
37. Moldovan, T.M., Abbeel, P.: Safe exploration in Markov decision processes. In: Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26–July 1, 2012 (2012)
38. Sikchi, H., Zhou, W., Held, D.: Lyapunov barrier policy optimization. CoRR **abs/2103.09230** (2021)
39. Srinivasan, K., Eysenbach, B., Ha, S., Tan, J., Finn, C.: Learning to be safe: Deep RL with a safety critic. CoRR **abs/2010.14603** (2020)
40. Stooke, A., Achiam, J., Abbeel, P.: Responsive safety in reinforcement learning by PID Lagrangian methods. In: Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13–18 July 2020, Virtual Event. Proceedings of Machine Learning Research, vol. 119 (2020)
41. Tessler, C., Mankowitz, D.J., Mannor, S.: Reward constrained policy optimization. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019 (2019)
42. Thananjeyan, B., et al.: Safe reinforcement learning with learned recovery zones. IEEE Robot. Autom. Lett. **6**(3) (2021)
43. Turchetta, M., Berkenkamp, F., Krause, A.: Safe exploration in finite Markov decision processes with gaussian processes. In: Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, 5–10 December, 2016, Barcelona, Spain (2016)
44. Verma, A., Le, H.M., Yue, Y., Chaudhuri, S.: Imitation-projected programmatic reinforcement learning. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8–14 December, 2019, Vancouver, BC, Canada (2019)

45. Wang, Y., Zhu, H.: Verification-guided programmatic controller synthesis. In: Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, 22–27 April, 2023, Proceedings, Part II. LNCS, vol. 13994 (2023)
46. Yang, Q., Simão, T.D., Tindemans, S.H., Spaan, M.T.J.: WCSAC: worst-case soft actor critic for safety-constrained reinforcement learning. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event, 2–9 February, 2021 (2021)
47. Yang, T., Rosca, J., Narasimhan, K., Ramadge, P.J.: Projection-based constrained policy optimization. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, 26–30 April, 2020 (2020)
48. Yang, Z., Zhang, L., Zeng, X., Tang, X., Peng, C., Zeng, Z.: Hybrid controller synthesis for nonlinear systems subject to reach-avoid constraints. In: Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I (2023)
49. Yu, D., Ma, H., Li, S., Chen, J.: Reachability constrained reinforcement learning. In: International Conference on Machine Learning, ICML 2022, 17–23 July 2022, Baltimore, Maryland, USA. Proceedings of Machine Learning Research, vol. 162 (2022)
50. Zanger, M.A., Daaboul, K., Zöllner, J.M.: Safe continuous control with constrained model-based policy optimization. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - October 1, 2021 (2021)
51. Zhang, Y., Vuong, Q., Ross, K.W.: First order constrained optimization in policy space. In: Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, 6–12 December, 2020, virtual (2020)
52. Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, 22–26 June, 2019 (2019)

Cyberphysical and Hybrid Systems

Using Four-Valued Signal Temporal Logic for Incremental Verification of Hybrid Systems

Florian Lercher(✉) and Matthias Althoff

School for Computation, Information and Technology,
Technical University of Munich, 85748 Garching, Germany
{florian.lercher,althoff}@tum.de

Abstract. Hybrid systems are often safety-critical and at the same time difficult to formally verify due to their mixed discrete and continuous behavior. To address this issue, we propose a novel incremental verification algorithm for hybrid systems based on online monitoring techniques and reachability analysis. To this end, we develop a four-valued semantics for signal temporal logic that allows us to distinguish two types of uncertainty: one arising from set-based evaluation and another one from the incremental nature of our algorithm. Using these semantics to continuously update the verification verdict, our verification algorithm is the first to run alongside the reachability analysis of the system to be verified. This makes it possible to stop the reachability analysis as soon as we obtain a conclusive verdict. We demonstrate the usefulness of our novel approach by several experiments.

Keywords: Hybrid systems verification · Many-valued temporal logic · Online verification

1 Introduction

Hybrid systems are a powerful modeling concept, as they can exhibit both continuous and discrete dynamics. As such, they are applicable in many contexts, including autonomous vehicles, power systems, robotics, and systems biology. As the typical application areas indicate, hybrid systems are often safety-critical and thus require formal verification. Specifications for the formal verification of hybrid systems are often formalized using signal temporal logic (STL) [26], which is evaluated on real-valued signals over continuous time. STL *monitoring* algorithms can determine whether a concrete execution of a system satisfies an STL specification [8]. By considering the *reachable set*, i.e., the set of states reached by at least one execution, rather than single executions, monitoring algorithms can be adapted to verify a specification for all executions [22,35]. While [22,35] focus on offline monitoring algorithms, we adapt an online algorithm for incremental verification. This allows us to stop the computation of the reachable set as soon as the specification can be verified or falsified. Thus, we can use our novel method online to, e.g., verify motion plans of autonomous vehicles [3].

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 259–281, 2024.
https://doi.org/10.1007/978-3-031-65633-0_12

1.1 Related Work

Online Monitoring of Real-Time Temporal Logics: The paper that originally introduced STL also presents an offline monitoring algorithm propagating satisfaction signals of atomic subformulas up the syntax tree of the specification [26]. Later, [27] calls this method *offline marking* and adapts it for online monitoring. The new procedure, called *incremental marking*, essentially performs offline marking for each new observation and discards the already propagated parts of the signals. The tool AMT [28] implements both algorithms. Other online monitoring approaches for real-time logics rely on translating the formula to timed [10,17] or untimed [20] automata. The algorithm in [34] rewrites the monitored metric temporal logic formula to represent remaining constraints whenever an observation is made. For robust monitoring, [15] adapts incremental marking to quantitative semantics of STL [16].

Many-Valued Semantics of Temporal Logics: In the context of monitoring, [10] employs three-valued semantics for linear temporal logic (LTL) and timed LTL to handle uncertainty due to finite traces. To obtain a more expressive monitoring result for finite traces, [9] extends this to a four-valued semantics that distinguishes *presumably* true or false finite traces. The authors of [13] use a five-valued semantics of LTL to deal with uncertainties arising from finite traces and race conditions in parallel systems. Most closely related to our approach are [22,35], which employ three-valued semantics for verification of hybrid systems. Based on reachable sets, previous work constructs three-valued satisfaction signals for the atomic predicates of an STL formula implicitly [22] or explicitly [35]. The third truth value indicates that both satisfying and violating states are reachable. To decide whether the specification is met, these are propagated akin to offline marking. Moreover, [35] employs statically determined *masks* to evaluate atomic predicates only where they are relevant; masking is an orthogonal approach to our proposed incremental verification.

Hybrid Systems Verification: Besides the aforementioned approaches based on three-valued semantics of STL [22,35], there are other verification methods using only the usual two truth values. The authors of [32] introduce a variant of STL called *reachset temporal logic* that is interpreted directly over the reachable set. They provide a sound transformation of STL into their logic, which is complete if all intervals in the STL formula range from 0 to a globally fixed time step. In [7], the authors propose a syntactic separation procedure for STL, which splits a formula into subformulas referring to disjoint time intervals. Based on the separated formula, they use satisfiability modulo theories (SMT) techniques to search for counterexamples bounded in length and the number of value changes; the SMT encoding is improved in [25]. Finally, there are deductive verification approaches that adapt dynamic logic suitable for software verification into *differential dynamic logic* suitable for hybrid systems [29,30]. Differential dynamic logic has been augmented with a fragment of STL to derive temporal properties [1] and assumption-commitment reasoning to handle parallel hybrid systems [12].

1.2 Contributions

We propose a novel algorithm for verifying STL specifications on hybrid systems based on reachability analysis. Following the idea of the incremental marking procedure for STL monitoring [27, Sect. 3.2], our algorithm runs alongside the reachability analysis. As soon as the reachability algorithm determines the reachable set for new time steps, our algorithm uses the new information to update its verdict on specification satisfaction. Thus, we can terminate the reachability analysis as soon as we obtain a conclusive verdict. The theoretical foundation of our algorithm is a novel four-valued semantics for STL. The two new truth values handle uncertainty arising from set-based (sets might contain both states satisfying and violating a predicate) and incremental (the entire reachable set over time is not immediately available) computation.

This paper is organized as follows: After discussing preliminaries and our problem statement in Sect. 2, we give an overview of our solution concept in Sect. 3. We present our four-valued semantics for STL in Sect. 4, followed by the novel incremental verification algorithm in Sect. 5. In Sect. 6, we apply a prototype implementation to systems occurring in autonomous driving and systems biology before coming to a conclusion in Sect. 7.

2 Preliminaries and Problem Statement

After introducing the necessary interval operations, we establish the required truth values. We then define signals as functions over time and briefly discuss set-based reachability analysis of hybrid systems. Finally, we recapitulate the syntax and Boolean semantics of STL before providing our problem statement.

2.1 Intervals

We work with intervals over $\mathbb{R}$, admitting ∞ and $-\infty$ as endpoints if the interval is open. The *left-closure* $\mathrm{cl}_{\mathtt{l}}(I)$ of an interval I always includes its left endpoint, except if the endpoint is infinite (e.g., $\mathrm{cl}_{\mathtt{l}}((a,b]) = [a,b]$ if $a \neq -\infty$). Analogously, the *right-closure* $\mathrm{cl}_{\mathtt{r}}(I)$ always includes the right endpoint.

For sets $\mathcal{A}$ and $\mathcal{B}$, their *Minkowski sum* $\mathcal{A} \oplus \mathcal{B}$ is $\{a + b \mid a \in \mathcal{A}, b \in \mathcal{B}\}$. We will write $a \oplus \mathcal{B}$ instead of $\{a\} \oplus \mathcal{B}$. We also use $\mathcal{A} \oplus (-\mathcal{B})$ for *back shifting* [26], where $-\mathcal{B} := \{-b \mid b \in \mathcal{B}\}$. If $\mathcal{A}$ and $\mathcal{B}$ are intervals, so are $\mathcal{A} \oplus \mathcal{B}$ and $-\mathcal{B}$.

2.2 Truth Values

We use the values $\mathbb{B} := \{\top, \bot\}$ to denote *truth* $\top$ and *falsehood* $\bot$. By extending the semantics of the usual Boolean connectives to handle a third value $\dashv_1$ denoting *unknown*, we can indicate that a statement could be true or false. This results in a three-valued propositional logic, such as that of Kleene [23]. For uncertainty arising from incremental computations, we add a fourth value $\dashv_2$ to denote *inconclusive*, indicating that the statement is either true, false, or

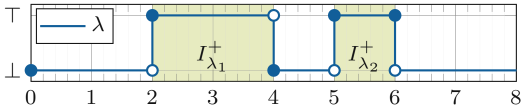


Fig. 1. A Boolean signal λ and its unitary decomposition $\{\lambda_1, \lambda_2\}$

unknown. In other words, $\dashv_1$ means "we know that we don't know," while $\dashv_2$ means "we don't know whether we don't know." We define the sets of truth values $\mathbb{U}_1 := \mathbb{B} \cup \{\dashv_1\}$ and $\mathbb{U}_2 := \mathbb{U}_1 \cup \{\dashv_2\}$. Moreover, we introduce the *truth order* $\sqsubseteq_{\mathtt{t}}$, where $v \sqsubseteq_{\mathtt{t}} v'$ for $v, v' \in \mathbb{U}_2$ means that v is "less true" than v'. Thus, we define $\bot \sqsubseteq_{\mathtt{t}} \dashv_1 \sqsubseteq_{\mathtt{t}} \top$ and $\bot \sqsubseteq_{\mathtt{t}} \dashv_2 \sqsubseteq_{\mathtt{t}} \top$; $\dashv_1$ and $\dashv_2$ are incomparable.

2.3 Signals

Let us fix $\mathbb{R}_{\geq 0}$ as our *time domain*. A *signal* over the domain $\mathcal{D}$, or $\mathcal{D}$-signal for short, is a function $\sigma : \mathbb{R}_{\geq 0} \to \mathcal{D}$. A *partial* $\mathcal{D}$-signal $\tilde{\sigma} : \mathcal{T} \to \mathcal{D}$ is only defined over a subset $\mathcal{T} \subseteq \mathbb{R}_{\geq 0}$ of the time domain. We refer to signals over $\mathbb{B}$, $\mathbb{U}_1$, and $\mathbb{U}_2$ as *logical signals*; in particular, *Boolean signals* are logical signals over $\mathbb{B}$. We adopt the following naming convention for logical signals: λ indicates Boolean signals, Λ indicates $\mathbb{U}_1$-signals, and $\tilde{\Lambda}$ indicates $\mathbb{U}_2$-signals.

A Boolean signal λ is *unitary* if there is one contiguous interval $I^+_\lambda \subseteq \mathbb{R}_{\geq 0}$ such that $\lambda(t) = \top$ for all $t \in I^+_\lambda$ and $\lambda(t) = \bot$ everywhere else [26]. Every Boolean signal can be represented as a disjunction of unitary signals, as shown in Fig. 1 [26]. In this work, we require this *unitary decomposition* to be minimal, i.e., the number of involved unitary signals must be minimal.

2.4 Reachability Analysis of Hybrid Systems

The literature provides numerous methods for describing hybrid systems. Our verification method is independent of the chosen description method as long as the system model is amenable to set-based reachability analysis (see [4] for an overview). Given the mixed continuous and discrete *state space* $\mathcal{X}$ of the hybrid system $\mathcal{H}$, an *execution* of $\mathcal{H}$ is a signal $\xi : \mathbb{R}_{\geq 0} \to \mathcal{X}$.

We are interested in the *reachable set* of the system $\mathcal{H}$, i.e., the set of all states that are part of at least one execution of $\mathcal{H}$. Formally, the reachable set of $\mathcal{H}$ is a signal $\mathcal{R} : \mathbb{R}_{\geq 0} \to 2^{\mathcal{X}}$ given by

$$\mathcal{R}(t) := \{\xi(t) \mid \xi \text{ is an execution of } \mathcal{H}\}.$$

Since determining the exact reachable set is often computationally infeasible, tools like CORA [2], JuliaReach [11], and SpaceEx [18] typically return a discrete-time overapproximation when performing reachability analysis. To this end, they represent $\mathcal{R}$ as a sequence of sets so that the set $\mathcal{R}_I$ for the time interval I subsumes $\bigcup_{t \in I} \mathcal{R}(t)$. Our verification algorithm assumes that this sequence is incrementally computed for consecutive time intervals, as is the case with the tools mentioned. To handle Taylor model representations (e.g., as used by Flow* [14]), a preprocessing step would be required to obtain a sequence of sets.

2.5 Signal Temporal Logic with Boolean Semantics

Suppose $\mathcal{AP}$ is a fixed set of atomic predicates, where each predicate is a function $a : \mathcal{X} \to \mathbb{B}$. An STL *formula* φ over $\mathcal{AP}$ is constructed according to the grammar

$$\varphi ::= \mathit{true} \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \, \mathbf{U}_I \, \varphi_2,$$

where $a \in \mathcal{AP}$ and I is an interval over $\mathbb{R}_{\geq 0}$ with rational endpoints [26]. We use the common abbreviations $\varphi_1 \vee \varphi_2 := \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\mathbf{F}_I \, \varphi := \mathit{true} \, \mathbf{U}_I \, \varphi$ (finally), and $\mathbf{G}_I \, \varphi := \neg \, \mathbf{F}_I \, \neg\varphi$ (globally). Note that we define *true* as basic syntax rather than introducing it as an abbreviation for $a \vee \neg a$, because the law of excluded middle does not transfer well to the four-valued semantics we define later.

In Boolean semantics, an STL formula φ is interpreted over an execution $\xi : \mathbb{R}_{\geq 0} \to \mathcal{X}$ to obtain a yes-or-no answer whether ξ satisfies φ [26,27]. We define the Boolean *satisfaction signal* $[\![\varphi]\!]_\xi : \mathbb{R}_{\geq 0} \to \mathbb{B}$ of φ over ξ inductively as

$$\begin{aligned}
[\![\mathit{true}]\!]_\xi(t) &:= \top, \\
[\![a]\!]_\xi(t) &:= a(\xi(t)), \\
[\![\neg\varphi]\!]_\xi(t) &:= \neg[\![\varphi]\!]_\xi(t), \\
[\![\varphi_1 \wedge \varphi_2]\!]_\xi(t) &:= [\![\varphi_1]\!]_\xi(t) \wedge [\![\varphi_2]\!]_\xi(t), \\
[\![\varphi_1 \, \mathbf{U}_I \, \varphi_2]\!]_\xi(t) &:= \begin{cases} \top & \text{if } \exists t' \in t \oplus I : [\![\varphi_2]\!]_\xi(t') = \top \\ & \qquad \text{and } \forall t'' \in (t, t') : [\![\varphi_1]\!]_\xi(t'') = \top \\ \bot & \text{otherwise} \end{cases},
\end{aligned}$$

where $a \in \mathcal{AP}$. The value of the satisfaction signal at time t indicates whether φ holds at t. Thus, an execution ξ *satisfies* φ, denoted by $\xi \models \varphi$, if and only if $[\![\varphi]\!]_\xi(0) = \top$. A hybrid system $\mathcal{H}$ *satisfies* φ, written as $\mathcal{H} \models \varphi$, if $\xi \models \varphi$ for all executions ξ of $\mathcal{H}$. Note that we use the strict until semantics from [27], which does not require φ_1 to hold at t or t', unlike the version in [26,35]. This semantics is more expressive, as we can recover the until from [26,35] as $\varphi_1 \wedge \varphi_1 \, \mathbf{U}_I (\varphi_1 \wedge \varphi_2)$.

To exclude Zeno behavior, we limit ourselves to executions ξ such that, for all $a \in \mathcal{AP}$, the Boolean signal given by point-wise application of a to ξ is of *finite variability*. That is, it changes its value only a finite number of times in any finite time interval [6, Sect. 2.3.5]. This is a common assumption in related work [17,26,27,35], albeit not always under this name; we refer the reader to [26, Sect. 4] and [27, Sect. 4] for a discussion.

2.6 Problem Statement

Given an STL formula φ and a hybrid system $\mathcal{H}$, we want to determine whether $\mathcal{H} \models \varphi$ based on the reachable set of $\mathcal{H}$. As reachability analysis is often incremental, we need to interpret φ over a reachable set that is only known for some time intervals to form a preliminary verification verdict. Formally, this means we want to define and compute a $\mathbb{U}_2$-satisfaction signal $[\![\varphi]\!]_{\tilde{\mathcal{R}}} : \mathbb{R}_{\geq 0} \to \mathbb{U}_2$ over a partial reachable set $\tilde{\mathcal{R}} : \mathcal{T} \to 2^{\mathcal{X}}$, where $\mathcal{T} \subseteq \mathbb{R}_{\geq 0}$. If our preliminary verdict is inconclusive due to partial knowledge of the reachable set, we aim to efficiently update our verdict as soon as more information becomes available. If the final verdict turns out to be $\dashv_1$, we need to refine our overapproximation of the reachable set to verify or falsify the specification.

This paper focuses on the four-valued semantics and the efficient update of the preliminary verdict. We only briefly discuss refinements of the overapproximation, as this often amounts to tuning the parameters of the reachability analysis. Moreover, it is a common step in verification approaches based on reachable sets [22,32,35]. An automatic refinement technique for the reachset temporal logic approach [32] is presented in [24].

3 Basic Idea and Solution Concept

As a motivating example, consider the intentionally simple dynamical system given by the differential equation $\dot{x} = u$, where the input u lies somewhere in $[0.9, 1.1]$ and initially $x \in [-0.5, 0.5]$. Suppose we want to verify that x eventually becomes larger than 1 within the next five seconds, which we formalize in STL as $\varphi := \mathbf{F}_{[0,5]}\, x > 1$. To this end, we exploit that the reachable set $\mathcal{R}$ of our system encloses all executions of the system. Thus, if we can prove that there exists a $t \in [0, 5]$ such that $x > 1$ for all states $x \in \mathcal{R}(t)$, we have shown φ for all executions of our system.

Recall that the reachable set is usually computed incrementally for consecutive time intervals. We are thus dealing with a partial reachable set $\tilde{\mathcal{R}}$, which is only determined for a subset of the time domain. For example, a reachability algorithm might first compute the reachable set for up to 1 s (top left in Fig. 2) and then continue to determine the reachable set in the next 0.8 s (top right).

If we are able to interpret our specification φ over such partial reachable sets, we can re-evaluate its satisfaction with every newly determined time interval and terminate the algorithm once we obtain a conclusive result. To this end, we derive a set-based version $\hat{a} : 2^{\mathcal{X}} \to \mathbb{U}_1$ of every atomic predicate $a \in \mathcal{AP}$ so that

$$\hat{a}(\mathcal{X}') := \begin{cases} \top & \text{if } \mathcal{X}' \neq \emptyset \text{ and } \mathcal{X}' \subseteq [\![a]\!] \\ \bot & \text{if } \mathcal{X}' \neq \emptyset \text{ and } \mathcal{X}' \cap [\![a]\!] = \emptyset \\ \dashv_1 & \text{otherwise} \end{cases}$$

for $\mathcal{X}' \subseteq \mathcal{X}$, where $[\![a]\!] := \{x \in \mathcal{X} \mid a(x) = \top\}$ denotes the set of states satisfying a. This enables us to construct a $\mathbb{U}_2$-satisfaction signal for atomic predicates over $\tilde{\mathcal{R}}$ by assigning $\dashv_2$ at times where the reachable set has not yet been determined. As shown in the second row of Fig. 2, this signal becomes $\dashv_1$ as soon as the reachable set starts intersecting with our atomic predicate and changes to $\top$ once it lies fully inside. Thus, $\dashv_1$ means that we do not know whether a predicate is

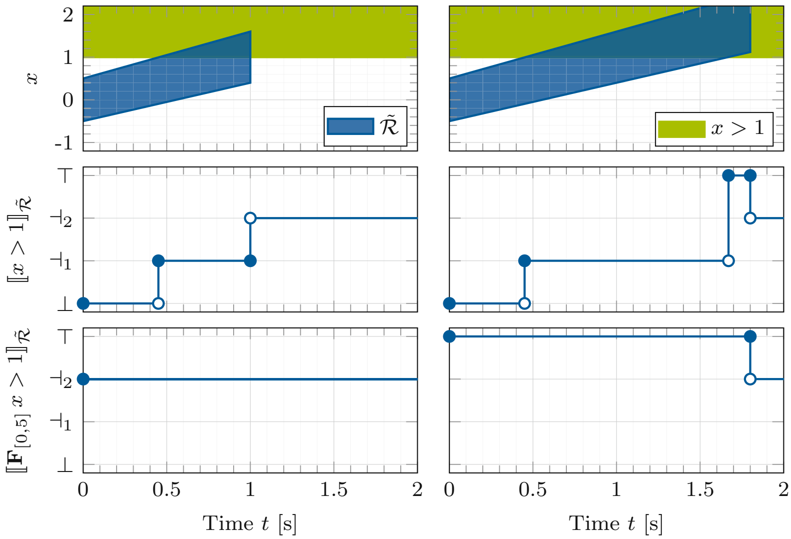


Fig. 2. Reachable set $\tilde{\mathcal{R}}$ and $\mathbb{U}_2$-satisfaction signals of $x > 1$ and $\mathbf{F}_{[0,5]}\, x > 1$ for the simple example system after reachability analysis for up to 1 s (left) and 1.8 s (right)

true or false since the reachable set contains satisfying and violating states; $\dashv_2$ means that we do not know as we have not yet computed the set for this time.

Now that we have satisfaction signals for the atomic predicates, it remains to combine them in order to obtain satisfaction signals for compound formulas. For this, we develop operators that preserve the intended meaning of our two uncertain values $\dashv_1$ and $\dashv_2$ in Sect. 4. The third row of Fig. 2 shows the resulting satisfaction signal $[\![\varphi]\!]_{\tilde{\mathcal{R}}}$ for our example specification: After reachability analysis for up to 1 s, the verification verdict is inconclusive, since $[\![\varphi]\!]_{\tilde{\mathcal{R}}}(0) = \dashv_2$. Once we update the satisfaction signal to incorporate information about the next 0.8 s, we can verify that φ holds and terminate early. In Sect. 5, we use ideas of the incremental marking procedure [27] to perform this update efficiently.

4 Four-Valued Signal Temporal Logic

To compute the $\mathbb{U}_2$-satisfaction signal for an STL formula φ with respect to a partial reachable set, we proceed by structural recursion on φ. The base case for *true* is clear, and we treat atomic predicates as described in Sect. 3. For compound formulas, i.e., negation, conjunction, and until, we combine the recursively computed satisfaction signals of their subformulas using suitable operators on signals. Instead of defining these operators directly on $\mathbb{U}_2$-signals, we under- and overapproximate a $\mathbb{U}_2$-signal using $\mathbb{U}_1$-signals. Similarly, we represent $\mathbb{U}_1$-signals by Boolean signals. Utilizing this representation, we can use the negation,

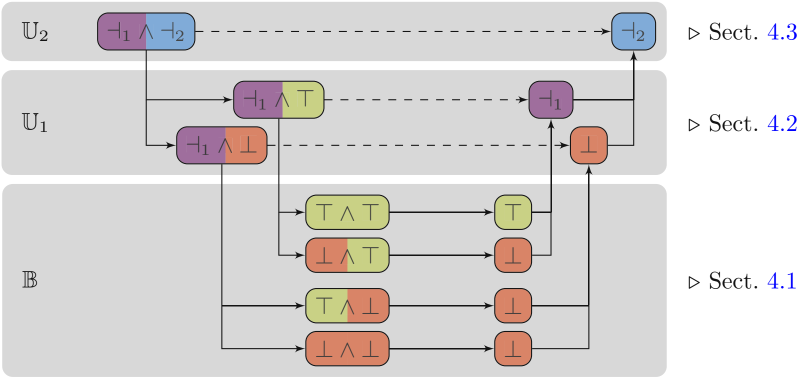


Fig. 3. Evaluating $\dashv_1 \wedge \dashv_2$ in $\mathbb{U}_2$-semantics: On the way down, we under- and overapproximate the input values with respect to the truth order by replacing the uncertain value of the current layer with $\bot$ and $\top$. In the Boolean layer, we evaluate the conjunction as usual. To move back up, we check whether the results for both approximations agree, and assign the appropriate uncertain value if this is not the case.

conjunction, and until operators defined for Boolean signals to combine $\mathbb{U}_2$-satisfaction signals. Figure 3 shows this concept simplified to the propositional case, in which we are dealing with truth values instead of logical signals.

4.1 Computing Boolean Satisfaction Signals

To define the operators for combining Boolean satisfaction signals, we closely follow the procedure from [27]. For negation and conjunction, we lift $\neg$ and $\wedge$ from Boolean values to Boolean signals by point-wise application [27, Sect. 3.1.1]. Unlike [27], we handle until directly instead of expressing it as a combination of untimed until and timed finally, thus avoiding the rather involved specification rewriting of [27, Lemma 1]. To this end, we adapt the method from [26, Sect. 3] for arbitrary intervals and strict semantics.

We first define an until operator that works only for unitary Boolean signals and generalize it later using the unitary decomposition. Unitary signals have the helpful property that $\lambda(t) = \top = \lambda(t')$ for times $t \leq t'$ implies $\lambda(t'') = \top$ for all $t'' \in [t, t']$. Given unitary signals λ_1, λ_2 and an interval I over $\mathbb{R}_{\geq 0}$, we define the unitary until $\overline{\mathbf{U}}_I$ so that $(\lambda_1 \,\overline{\mathbf{U}}_I\, \lambda_2)(t) = \top$ if and only if $t \in I_1 \cup I_2$, where

$$I_1 := \left[\left(I^+_{\lambda_2} \cap \mathrm{cl}_{\mathtt{r}}(I^+_{\lambda_1})\right) \oplus \left(-(I \setminus \{0\})\right)\right] \cap \mathrm{cl}_{\mathtt{l}}(I^+_{\lambda_1}), \quad I_2 := \begin{cases} I^+_{\lambda_2} & \text{if } 0 \in I \\ \emptyset & \text{otherwise} \end{cases}. \quad (1)$$

Here, $I \setminus \{0\}$ is always an interval, since $I \subseteq \mathbb{R}_{\geq 0} = [0, \infty)$. We prove that $\overline{\mathbf{U}}_I$ implements the until semantics for unitary signals.

Lemma 1. *Suppose* λ_1 *and* λ_2 *are* unitary *Boolean signals, and* I *is an interval over* $\mathbb{R}_{\geq 0}$*. For all* $t \in \mathbb{R}_{\geq 0}$*, we have*

$$(\lambda_1 \,\overline{\mathbf{U}}_I\, \lambda_2)(t) = \begin{cases} \top & \textit{if } \exists t' \in t \oplus I : \lambda_2(t') = \top \textit{ and } \forall t'' \in (t, t') : \lambda_1(t'') = \top \\ \bot & \textit{otherwise} \end{cases}.$$

Proof. Let I_1 and I_2 be given as in (1) so that $(\lambda_1 \,\overline{\mathbf{U}}_I\, \lambda_2)(t) = \top$ if and only if $t \in I_1 \cup I_2$. For an arbitrary $t \in \mathbb{R}_{\geq 0}$, we first prove that I_1 treats the case $t' > t$:

$$\begin{aligned} t \in I_1 &\iff t \in \left[\left(I^+_{\lambda_2} \cap \mathrm{cl}_{\mathtt{r}}(I^+_{\lambda_1})\right) \oplus (-(I \setminus \{0\}))\right] \cap \mathrm{cl}_{\mathtt{l}}(I^+_{\lambda_1}) \\ &\iff \exists t' \in t \oplus (I \setminus \{0\}) : t' \in I^+_{\lambda_2} \text{ and } t' \in \mathrm{cl}_{\mathtt{r}}(I^+_{\lambda_1}) \text{ and } t \in \mathrm{cl}_{\mathtt{l}}(I^+_{\lambda_1}) \\ &\iff \exists t' \in t \oplus (I \setminus \{0\}) : \lambda_2(t') = \top \text{ and } \forall t'' \in (t, t') : \lambda_1(t'') = \top. \end{aligned}$$

For the second step, observe that $\mathcal{A} \oplus (-\mathcal{B}) = \{x \mid \exists a \in x \oplus \mathcal{B} : a \in \mathcal{A}\}$. The last equivalence uses that λ_1 is unitary: If λ_1 is $\top$ both immediately after t, i.e., $t \in \mathrm{cl}_{\mathtt{l}}(I^+_{\lambda_1})$, and immediately before t', i.e., $t' \in \mathrm{cl}_{\mathtt{r}}(I^+_{\lambda_1})$, the signal must also be $\top$ throughout (t, t'), since $I^+_{\lambda_1}$ is contiguous. For the converse, note that $\emptyset \subsetneq (t, t') \subseteq I^+_{\lambda_1}$ implies $t \in \mathrm{cl}_{\mathtt{l}}(I^+_{\lambda_1})$ and $t' \in \mathrm{cl}_{\mathtt{r}}(I^+_{\lambda_1})$. If $0 \notin I$, we are done, as $I = I \setminus \{0\}$ and $I_2 = \emptyset$. Otherwise, I_2 handles the case $t' = t$, where $t \oplus \{0\} = \{t\}$ and the universal quantifier is vacuously satisfied:

$$\begin{aligned} t \in I_2 &\iff t \in I^+_{\lambda_2} \\ &\iff \lambda_2(t) = \top \\ &\iff \exists t' \in t \oplus \{0\} : \lambda_2(t') = \top \text{ and } \forall t'' \in (t, t') : \lambda_1(t'') = \top. \end{aligned} \qquad \square$$

Using that all Boolean signals admit a unitary decomposition, we generalize $\overline{\mathbf{U}}_I$ to Boolean signals λ_1 and λ_2 that are not necessarily unitary. We define

$$(\lambda_1 \,\mathbf{U}_I\, \lambda_2)(t) := \bigvee_{1 \leq i \leq n_1} \bigvee_{1 \leq j \leq n_2} (\lambda_{1,i} \,\overline{\mathbf{U}}_I\, \lambda_{2,j})(t),$$

where $t \in \mathbb{R}_{\geq 0}$ and $\{\lambda_{k,1}, \ldots, \lambda_{k,n_k}\}$ is the unitary decomposition of λ_k for $k \in \{1, 2\}$. We prove that this is a general implementation of the until semantics.

Lemma 2. *Let* λ_1 *and* λ_2 *be Boolean signals and* I *be an interval over* $\mathbb{R}_{\geq 0}$*. For all* $t \in \mathbb{R}_{\geq 0}$*, we have*

$$(\lambda_1 \,\mathbf{U}_I\, \lambda_2)(t) = \begin{cases} \top & \textit{if } \exists t' \in t \oplus I : \lambda_2(t') = \top \textit{ and } \forall t'' \in (t, t') : \lambda_1(t'') = \top \\ \bot & \textit{otherwise} \end{cases}.$$

In particular, we obtain $[\![\varphi_1 \,\mathbf{U}_I\, \varphi_2]\!]_\xi = [\![\varphi_1]\!]_\xi \,\mathbf{U}_I [\![\varphi_2]\!]_\xi$*.*

Proof. Observe that the second statement is a specialization of the first by the Boolean semantics of STL. Let $t \in \mathbb{R}_{\geq 0}$ be arbitrary. If $(\lambda_1 \, \mathbf{U}_I \, \lambda_2)(t) = \top$, there must be i and j so that $(\lambda_{1,i} \, \overline{\mathbf{U}}_I \, \lambda_{2,j})(t) = \top$, and we can immediately conclude with Lemma 1. Conversely, let $t' \in t \oplus I$ so that $\lambda_2(t') = \top$ and $\lambda_1(t'') = \top$ for all $t'' \in (t, t')$. Since $\lambda_2(t') = \top$, there exists j such that $\lambda_{2,j}(t') = \top$. As the unitary decomposition of λ_1 is minimal, the union of two or more of the $I^+_{\lambda_{1,i}}$ cannot yield a contiguous interval (otherwise, we could merge them for a smaller decomposition). Hence, there exists i such that $(t, t') \subseteq I^+_{\lambda_{1,i}}$, or, in other words, $\lambda_{1,i}(t'') = \top$ for all $t'' \in (t, t')$. Applying Lemma 1 again concludes the proof. □

4.2 Computing Three-Valued Satisfaction Signals

To compute the $\mathbb{U}_1$-satisfaction signal of an STL formula over a reachable set, we represent $\mathbb{U}_1$-signals using Boolean signals and then reuse the techniques from Sect. 4.1. Recall from Sect. 2.2 that $\dashv_1$ means that we do not know whether a statement is true or false. Thus, every $\mathbb{U}_1$-signal $\varLambda$ induces a set of Boolean signals, called *refinements* of $\varLambda$, in which the occurrences of $\dashv_1$ are replaced with $\top$ or $\bot$. Formally, a Boolean signal λ *refines* $\varLambda$, denoted as $\lambda \prec \varLambda$, if $\varLambda(t) \neq \dashv_1$ implies $\lambda(t) = \varLambda(t)$ for all $t \in \mathbb{R}_{\geq 0}$. Since the set of refinements is unique for each $\mathbb{U}_1$-signal $\varLambda$, we use it to represent $\varLambda$. We argue that the two special refinements

$$\lfloor \varLambda \rfloor(t) := \begin{cases} \bot & \text{if } \varLambda(t) = \dashv_1 \\ \varLambda(t) & \text{otherwise} \end{cases} \quad \text{and} \quad \lceil \varLambda \rceil(t) := \begin{cases} \top & \text{if } \varLambda(t) = \dashv_1 \\ \varLambda(t) & \text{otherwise} \end{cases}$$

adequately characterize this set. Lifting the truth order $\sqsubseteq_{\mathtt{t}}$ to a partial order on logical signals by point-wise application, we find that the Boolean signal λ refines $\varLambda$ if and only if $\lfloor \varLambda \rfloor \sqsubseteq_{\mathtt{t}} \lambda \sqsubseteq_{\mathtt{t}} \lceil \varLambda \rceil$. Hence, $\lfloor \varLambda \rfloor$ *underapproximates* the refinements of $\varLambda$, while $\lceil \varLambda \rceil$ *overapproximates* them. We can recover $\varLambda(t)$ as $\lfloor \varLambda \rfloor(t) \sqcup_1 \lceil \varLambda \rceil(t)$, where $v \sqcup_1 v'$ with $v, v' \in \mathbb{B}$ yields v if and only if $v = v'$ and $\dashv_1$ otherwise.

The operator $\mathbf{U}_I$ on Boolean signals is *monotone*, i.e., we have $\lambda_1 \, \mathbf{U}_I \, \lambda_2 \sqsubseteq_{\mathtt{t}} \lambda'_1 \, \mathbf{U}_I \, \lambda'_2$ given that $\lambda_i \sqsubseteq_{\mathtt{t}} \lambda'_i$ for $i \in \{1, 2\}$. Intuitively, this means that if we set the inputs to $\top$ at more time points, the output signal will also be $\top$ more often. Thus, $\lfloor \varLambda_1 \rfloor \, \mathbf{U}_I \lfloor \varLambda_2 \rfloor$ is a faithful underapproximation of $\lambda_1 \, \mathbf{U}_I \, \lambda_2$, given that $\lambda_i \prec \varLambda_i$ for $i \in \{1, 2\}$. Similarly, $\lceil \varLambda_1 \rceil \, \mathbf{U}_I \lceil \varLambda_2 \rceil$ is an overapproximation. Figure 4 visualizes this for a derived finally operator $\mathbf{F}_I \, \lambda := \lambda_\top \, \mathbf{U}_I \, \lambda$, where $\lambda_\top$ is $\top$ everywhere. To show monotonicity of $\mathbf{U}_I$, we apply Lemma 2 and use that $\lambda_i(t) = \top$ implies $\lambda'_i(t) = \top$. The operator $\wedge$ is also monotone. In contrast, $\neg$ is *antitone*, i.e., $\lambda \sqsubseteq_{\mathtt{t}} \lambda'$ implies $\neg\lambda' \sqsubseteq_{\mathtt{t}} \neg\lambda$. So, $\neg\lceil \varLambda \rceil$ is an underapproximation, while $\neg\lfloor \varLambda \rfloor$ is an overapproximation. We define the operators $\neg$, $\wedge$, and $\mathbf{U}_I$ on $\mathbb{U}_1$-signals such that they recover a $\mathbb{U}_1$-signal from these over- and underapproximations:

$$\begin{aligned} (\neg\varLambda)(t) &:= (\neg\lfloor \varLambda \rfloor)(t) \sqcup_1 (\neg\lceil \varLambda \rceil)(t), \\ (\varLambda_1 \wedge \varLambda_2)(t) &:= (\lfloor \varLambda_1 \rfloor \wedge \lfloor \varLambda_2 \rfloor)(t) \sqcup_1 (\lceil \varLambda_1 \rceil \wedge \lceil \varLambda_2 \rceil)(t), \\ (\varLambda_1 \, \mathbf{U}_I \, \varLambda_2)(t) &:= (\lfloor \varLambda_1 \rfloor \, \mathbf{U}_I \lfloor \varLambda_2 \rfloor)(t) \sqcup_1 (\lceil \varLambda_1 \rceil \, \mathbf{U}_I \lceil \varLambda_2 \rceil)(t). \end{aligned}$$

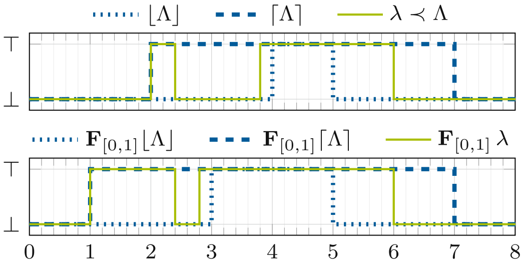


Fig. 4. Top: All refinements λ of a $\mathbb{U}_1$-signal Λ lie between the underapproximation $\lfloor \Lambda \rfloor$ and the overapproximation $\lceil \Lambda \rceil$. Bottom: After applying the monotone finally operator to all three Boolean signals, the refinement is still between the approximations. We omit the markers at the jumps, as they are irrelevant to the point of this example.

Finally, we define the $\mathbb{U}_1$-satisfaction signal $[\![\varphi]\!]_{\mathcal{R}} : \mathbb{R}_{\geq 0} \to \mathbb{U}_1$ of an STL formula φ with respect to a reachable set $\mathcal{R} : \mathbb{R}_{\geq 0} \to 2^{\mathcal{X}}$ using our operators. For all $t \in \mathbb{R}_{\geq 0}$, we define

$$[\![\mathit{true}]\!]_{\mathcal{R}}(t) := \top \qquad\qquad \text{and} \qquad\qquad [\![a]\!]_{\mathcal{R}}(t) := \hat{a}(\mathcal{R}(t)),$$

where $a \in \mathcal{AP}$ and $\hat{a}$ is defined as in Sect. 3. Moreover, we define

$$\begin{aligned} [\![\neg\varphi]\!]_{\mathcal{R}} &:= \neg[\![\varphi]\!]_{\mathcal{R}}, \\ [\![\varphi_1 \wedge \varphi_2]\!]_{\mathcal{R}} &:= [\![\varphi_1]\!]_{\mathcal{R}} \wedge [\![\varphi_2]\!]_{\mathcal{R}}, \\ [\![\varphi_1 \, \mathbf{U}_I \, \varphi_2]\!]_{\mathcal{R}} &:= [\![\varphi_1]\!]_{\mathcal{R}} \, \mathbf{U}_I [\![\varphi_2]\!]_{\mathcal{R}}. \end{aligned}$$

To relate this to the Boolean semantics, we show that $[\![\varphi]\!]_\xi$ refines $[\![\varphi]\!]_{\mathcal{R}}$, if $\mathcal{R}$ covers the execution ξ. We say a reachable set $\mathcal{R}$ *covers* an execution ξ, denoted as $\xi \prec \mathcal{R}$, if $\xi(t) \in \mathcal{R}(t)$ for all $t \in \mathbb{R}_{\geq 0}$.

Theorem 1. *Suppose $\mathcal{R}$ is a reachable set and φ an STL formula. For every execution ξ covered by $\mathcal{R}$, we have $[\![\varphi]\!]_\xi \prec [\![\varphi]\!]_{\mathcal{R}}$. In other words, if $[\![\varphi]\!]_{\mathcal{R}}(t) = v$, we have $[\![\varphi]\!]_\xi(t) = v$ for all $\xi \prec \mathcal{R}$, $v \in \mathbb{B}$, and $t \in \mathbb{R}_{\geq 0}$.*

Proof. We proceed by structural induction on φ. Let $t \in \mathbb{R}_{\geq 0}$ be arbitrary.

Base Cases: The case for $\varphi = \mathit{true}$ is straightforward, as *true* always evaluates to $\top$ in Boolean and $\mathbb{U}_1$-semantics. For an atomic predicate $a \in \mathcal{AP}$, we find

$$\hat{a}(\mathcal{R}(t)) = \top \iff \mathcal{R}(t) \neq \emptyset \text{ and } \mathcal{R}(t) \subseteq [\![a]\!] \implies \forall \xi \prec \mathcal{R} : a(\xi(t)) = \top,$$

and thus $[\![a]\!]_{\mathcal{R}}(t) = \top \implies \forall \xi \prec \mathcal{R} : [\![a]\!]_\xi(t) = \top$. The argument for $\bot$ is similar.

Direct Semantics: Turning to the inductive cases, we notice that our operators on $\mathbb{U}_1$-signals depend on their Boolean counterparts. This makes them easy to implement, but difficult to handle in proofs. Therefore, we first show that they adhere to the following direct semantics that work without this dependency:[1]

$$(\neg \Lambda)(t) = \begin{cases} \top & \text{if } \Lambda(t) = \bot \\ \bot & \text{if } \Lambda(t) = \top \\ \dashv_1 & \text{otherwise} \end{cases},$$
$$(\Lambda_1 \wedge \Lambda_2)(t) = \begin{cases} \top & \text{if } \Lambda_1(t) = \top \text{ and } \Lambda_2(t) = \top \\ \bot & \text{if } \Lambda_1(t) = \bot \text{ or } \Lambda_2(t) = \bot \\ \dashv_1 & \text{otherwise} \end{cases}, \tag{2}$$
$$(\Lambda_1 \,\mathbf{U}_I\, \Lambda_2)(t) = \begin{cases} \top & \text{if } \exists t' \in t \oplus I : \Lambda_2(t') = \top \text{ and } \forall t'' \in (t, t') : \Lambda_1(t'') = \top \\ \bot & \text{if } \forall t' \in t \oplus I : \Lambda_2(t') = \bot \text{ or } \exists t'' \in (t, t') : \Lambda_1(t'') = \bot \\ \dashv_1 & \text{otherwise} \end{cases}.$$

Proof of the Direct Semantics: First, consider the case where $(\Lambda_1 \,\mathbf{U}_I\, \Lambda_2)(t)$ is $\top$. Recalling that $v \sqcup_1 v'$ only yields $\top$ if $v = \top = v'$, we derive

$$\begin{aligned} (\Lambda_1 \,\mathbf{U}_I\, \Lambda_2)(t) = \top &\iff (\lfloor \Lambda_1 \rfloor \,\mathbf{U}_I \lfloor \Lambda_2 \rfloor)(t) = \top = (\lceil \Lambda_1 \rceil \,\mathbf{U}_I \lceil \Lambda_2 \rceil)(t) \\ &\iff \forall \lambda_1 \prec \Lambda_1 : \forall \lambda_2 \prec \Lambda_2 : (\lambda_1 \,\mathbf{U}_I\, \lambda_2)(t) = \top. \end{aligned}$$

The second equivalence is due to the monotonicity of $\mathbf{U}_I$ over Boolean signals: If $\lambda_i \prec \Lambda_i$, we know that $\lfloor \Lambda_i \rfloor \sqsubseteq_{\mathtt{t}} \lambda_i \sqsubseteq_{\mathtt{t}} \lceil \Lambda_i \rceil$ for $i \in \{1, 2\}$. Thus, we also have $\lfloor \Lambda_1 \rfloor \,\mathbf{U}_I \lfloor \Lambda_2 \rfloor \sqsubseteq_{\mathtt{t}} \lambda_1 \,\mathbf{U}_I\, \lambda_2 \sqsubseteq_{\mathtt{t}} \lceil \Lambda_1 \rceil \,\mathbf{U}_I \lceil \Lambda_2 \rceil$. Hence, $(\lambda_1 \,\mathbf{U}_I\, \lambda_2)(t)$ must be $\top$ due to the antisymmetry of $\sqsubseteq_{\mathtt{t}}$. The converse is clear, as $\lfloor \Lambda_i \rfloor$ and $\lceil \Lambda_i \rceil$ are particular refinements of Λ_i. We continue our derivation by applying Lemma 2 and find

$$\begin{aligned} \ldots \iff & \forall \lambda_1 \prec \Lambda_1 : \forall \lambda_2 \prec \Lambda_2 : \\ & \qquad \exists t' \in t \oplus I : \lambda_2(t') = \top \text{ and } \forall t'' \in (t, t') : \lambda_1(t'') = \top \\ \iff & \exists t' \in t \oplus I : \Lambda_2(t') = \top \text{ and } \forall t'' \in (t, t') : \Lambda_1(t'') = \top. \end{aligned}$$

To explain the forward direction of the last equivalence, we consider the refinements $\lfloor \Lambda_1 \rfloor$ and $\lfloor \Lambda_2 \rfloor$. Instantiating the universal quantifiers, we find that

$$\exists t' \in t \oplus I : \lfloor \Lambda_2 \rfloor (t') = \top \text{ and } \forall t'' \in (t, t') : \lfloor \Lambda_1 \rfloor (t'') = \top.$$

Since $\lfloor \Lambda_i \rfloor (t) = \top$ if and only if $\Lambda_i(t) = \top$, this establishes the forward direction. For $(\Lambda_1 \,\mathbf{U}_I\, \Lambda_2)(t) = \bot$, we argue analogously, and then the case for $\dashv_1$ follows by elimination. The reasoning for the remaining operators is similar.

[1] These semantics arise naturally from the Boolean semantics in Sect. 2.5 by making the Boolean "otherwise" case explicit and adding a new "otherwise" to handle $\dashv_1$.

Inductive Cases: We are now equipped to prove the inductive cases for our main statement. Using the direct semantics (2), we exemplarily show the case for until:

$$\begin{aligned}
[\![\varphi_1 \,\mathbf{U}_I\, \varphi_2]\!]_{\mathcal{R}}(t) = \top \iff\; & \exists t' \in t \oplus I : [\![\varphi_2]\!]_{\mathcal{R}}(t') = \top \\
& \quad \text{and } \forall t'' \in (t, t') : [\![\varphi_1]\!]_{\mathcal{R}}(t'') = \top \\
\implies\; & \exists t' \in t \oplus I : (\forall \xi \prec \mathcal{R} : [\![\varphi_2]\!]_{\xi}(t') = \top) && \text{(IH)} \\
& \quad \text{and } \forall t'' \in (t, t') : \forall \xi \prec \mathcal{R} : [\![\varphi_1]\!]_{\xi}(t'') = \top \\
\implies\; & \forall \xi \prec \mathcal{R} : \exists t' \in t \oplus I : [\![\varphi_2]\!]_{\xi}(t') = \top && (*) \\
& \quad \text{and } \forall t'' \in (t, t') : [\![\varphi_1]\!]_{\xi}(t'') = \top \\
\iff\; & \forall \xi \prec \mathcal{R} : [\![\varphi_1 \,\mathbf{U}_I\, \varphi_2]\!]_{\xi}(t) = \top.
\end{aligned}$$

To derive $[\![\varphi_1 \,\mathbf{U}_I\, \varphi_2]\!]_{\mathcal{R}}(t) = \bot \implies \forall \xi \prec \mathcal{R} : [\![\varphi_1 \,\mathbf{U}_I\, \varphi_2]\!]_{\xi}(t) = \bot$, we argue similarly. Note that the step marked with $(*)$ is not an equivalence since we need to swap an existential and a universal quantifier. Intuitively, this means that the Boolean semantics allow us to choose the time when φ_2 becomes true for each refinement individually, while we have to choose the same time for all refinements in the $\mathbb{U}_1$-semantics. A similar step is necessary in the $\bot$ case of conjunction, where we have to choose which subformula is false. □

4.3 Computing Four-Valued Satisfaction Signals

In the previous section, we used two Boolean signals to over- and underapproximate the refinements of a $\mathbb{U}_1$-signal. This enabled us to reuse the operators defined on Boolean signals for computing a $\mathbb{U}_1$-satisfaction signal of an STL formula with respect to a reachable set. Following the same pattern, we can over- and underapproximate the refinements of a $\mathbb{U}_2$-signal by two $\mathbb{U}_1$-signals to compute $\mathbb{U}_2$-satisfaction signals over partial reachable sets. Hence, many concepts, definitions, and proofs are analogous to Sect. 4.2, so we only sketch them here.

Given a $\mathbb{U}_2$-signal $\tilde{\varLambda}$, the $\mathbb{U}_1$-signal $\varLambda$ *refines* $\tilde{\varLambda}$, denoted by $\varLambda \prec \tilde{\varLambda}$, if $\tilde{\varLambda}(t) \neq \dashv_2$ implies $\varLambda(t) = \tilde{\varLambda}(t)$ for all $t \in \mathbb{R}_{\geq 0}$. We define the *underapproximation* $\lfloor \tilde{\varLambda} \rfloor$ and the *overapproximation* $\lceil \tilde{\varLambda} \rceil$ analogously to Sect. 4.2, i.e., by replacing $\dashv_2$ with $\bot$ and $\top$, respectively. Again, we have $\lfloor \tilde{\varLambda} \rfloor \sqsubseteq_{\mathtt{t}} \varLambda \sqsubseteq_{\mathtt{t}} \lceil \tilde{\varLambda} \rceil$ if and only if $\varLambda \prec \tilde{\varLambda}$. Moreover, we can reconstruct $\tilde{\varLambda}(t)$ as $\lfloor \tilde{\varLambda} \rfloor(t) \sqcup_2 \lceil \tilde{\varLambda} \rceil(t)$. Here, $v \sqcup_2 v'$ with $v, v' \in \mathbb{U}_1$ is v if and only if $v = v'$ and $\dashv_2$ otherwise.

To define the operators for negation, conjunction, and until on $\mathbb{U}_2$-signals, we first show that the $\mathbb{U}_1$-operators are monotone or antitone.

Lemma 3. *The operators $\wedge$ and $\mathbf{U}_I$ on $\mathbb{U}_1$-signals are monotone; $\neg$ is antitone.*

Proof. Using the direct semantics (2) of the operators shown in the proof of Theorem 1 and case distinction, the proof is straightforward. We exemplarily consider the case $(\varLambda_1 \,\mathbf{U}_I\, \varLambda_2)(t) = \dashv_1$, where we need to show $\dashv_1 \sqsubseteq_{\mathtt{t}} (\varLambda_1' \,\mathbf{U}_I\, \varLambda_2')(t)$ given $\varLambda_i \sqsubseteq_{\mathtt{t}} \varLambda_i'$ for $i \in \{1, 2\}$. From the direct semantics, we know

$$\exists t' \in t \oplus I : \varLambda_2(t') \neq \bot \text{ and } \forall t'' \in (t, t') : \varLambda_1(t'') \neq \bot.$$

Since $\varLambda_i'$ can only be $\bot$ where $\varLambda_i$ is also $\bot$, the same statement holds for $\varLambda_1'$ and $\varLambda_2'$. Thus, $(\varLambda_1' \, \mathbf{U}_I \, \varLambda_2')(t)$ cannot be $\bot$. Consequently, it must be either $\dashv_1$ or $\top$, and we know that $\dashv_1 \sqsubseteq_{\mathtt{t}} \top$. □

Due to Lemma 3, it is justified to define the operators $\neg$, $\wedge$, and $\mathbf{U}_I$ on $\mathbb{U}_2$-signals like in Sect. 4.2 using $\sqcup_2$ instead of $\sqcup_1$. With these, we can define the *satisfaction signal* $[\![\varphi]\!]_{\tilde{\mathcal{R}}} : \mathbb{R}_{\geq 0} \to \mathbb{U}_2$ of an STL formula φ with respect to a partial reachable set $\tilde{\mathcal{R}} : \mathcal{T} \to 2^{\mathcal{X}}$, where $\mathcal{T} \subseteq \mathbb{R}_{\geq 0}$. For atomic formulas *true* and $a \in \mathcal{AP}$, we define

$$[\![\mathit{true}]\!]_{\tilde{\mathcal{R}}}(t) := \top \qquad \text{and} \qquad [\![a]\!]_{\tilde{\mathcal{R}}}(t) := \begin{cases} \hat{a}(\tilde{\mathcal{R}}(t)) & \text{if } t \in \mathcal{T} \\ \dashv_2 & \text{otherwise} \end{cases}$$

for all $t \in \mathbb{R}_{\geq 0}$. The satisfaction signal for compound formulas is defined analogously to Sect. 4.2 using our operators. Finally, we state the equivalent of Theorem 1 to relate $\mathbb{U}_2$- and $\mathbb{U}_1$-satisfaction signals: We show that $[\![\varphi]\!]_{\mathcal{R}}$ refines $[\![\varphi]\!]_{\tilde{\mathcal{R}}}$, if $\mathcal{R}$ is an extension of $\tilde{\mathcal{R}}$. Given $\tilde{\mathcal{R}} : \mathcal{T} \to 2^{\mathcal{X}}$ with $\mathcal{T} \subseteq \mathbb{R}_{\geq 0}$, we say that the reachable set $\mathcal{R}$ *extends* $\tilde{\mathcal{R}}$, denoted as $\mathcal{R} \prec \tilde{\mathcal{R}}$, if $\mathcal{R}(t) = \tilde{\mathcal{R}}(t)$ for all $t \in \mathcal{T}$.

Theorem 2. *Suppose $\tilde{\mathcal{R}}$ is a partial reachable set and φ an STL formula. For every reachable set $\mathcal{R}$ that extends $\tilde{\mathcal{R}}$, we have $[\![\varphi]\!]_{\mathcal{R}} \prec [\![\varphi]\!]_{\tilde{\mathcal{R}}}$. In other words, if $[\![\varphi]\!]_{\tilde{\mathcal{R}}}(t) = v$, we have $[\![\varphi]\!]_{\mathcal{R}}(t) = v$ for all $\mathcal{R} \prec \tilde{\mathcal{R}}$, $v \in \mathbb{U}_1$, and $t \in \mathbb{R}_{\geq 0}$.*

Sketch of Proof. The proof proceeds by structural induction on φ. For the most part, it is analogous to the proof of Theorem 1, except that we now have to consider an additional case for $v = \dashv_1$. In the base case for $\varphi = a$ with $a \in \mathcal{AP}$, we derive:

$$\begin{aligned} [\![a]\!]_{\tilde{\mathcal{R}}}(t) = \dashv_1 &\iff t \in \mathcal{T} \text{ and } \hat{a}(\tilde{\mathcal{R}}(t)) = \dashv_1 \\ &\implies \forall \mathcal{R} \prec \tilde{\mathcal{R}} : \hat{a}(\mathcal{R}(t)) = \dashv_1 \\ &\iff \forall \mathcal{R} \prec \tilde{\mathcal{R}} : [\![a]\!]_{\mathcal{R}}(t) = \dashv_1, \end{aligned}$$

where the partial reachable set $\tilde{\mathcal{R}}$ is defined over $\mathcal{T} \subseteq \mathbb{R}_{\geq 0}$. For $\top$ and $\bot$, we argue similarly. For the inductive cases with compound formulas, we can determine direct semantics for our operators on $\mathbb{U}_2$-signals similar to those in the proof of Theorem 1. Like (2), they follow the pattern of making the “otherwise” case of the $\mathbb{U}_1$-semantics explicit and introducing a new “otherwise” case to handle $\dashv_2$. As for Theorem 1, the crucial points in the proof are those where we need to use a statement about all refinements of a $\mathbb{U}_2$-signal $\tilde{\varLambda}$ to infer something about $\tilde{\varLambda}$ itself. In particular, we need variations of the following property

$$\forall \varLambda \prec \tilde{\varLambda} : \exists t \in \mathbb{R}_{\geq 0} : \varLambda(t) \in \mathcal{U} \iff \exists t \in \mathbb{R}_{\geq 0} : \tilde{\varLambda}(t) \in \mathcal{U},$$

where $\mathcal{U} \subsetneq \mathbb{U}_1$. To prove the forward direction, we choose a value $v \in \mathbb{U}_1 \setminus \mathcal{U}$, which must exist since $\mathcal{U}$ is a proper subset of $\mathbb{U}_1$. We consider the refinement $\varLambda_v \prec \tilde{\varLambda}$ in which all occurrences of $\dashv_2$ are replaced by v and find that $\varLambda_v(t) \in \mathcal{U}$ if and only if $\tilde{\varLambda}(t) \in \mathcal{U}$ to establish the forward direction. Using the direct semantics, we show the inductive cases analogously to the proof of Theorem 1.□

5 Incremental Verification of Hybrid Systems

Theorems 1 and 2 provide a sound, but incomplete, method of proving or disproving that a hybrid system $\mathcal{H}$ satisfies an STL specification φ. We note that incompleteness is unavoidable to some extent since the reachability problem for hybrid systems is undecidable in general [19, Sect. 4]. We summarize the verification method in the following corollary.

Corollary 1. *Let φ be an STL formula and $\tilde{\mathcal{R}}$ a partial reachable set. Let $\mathcal{R}$ be an extension of $\tilde{\mathcal{R}}$. If $[\![\varphi]\!]_{\tilde{\mathcal{R}}}(0)$ is*

- *$\top$, we have $\xi \models \varphi$ for all executions ξ covered by $\mathcal{R}$.*
- *$\bot$, we have $\xi \not\models \varphi$ for all executions ξ covered by $\mathcal{R}$.*
- *$\dashv_1$, the result is unknown. Based on the reachable set $\mathcal{R}$, we cannot make a statement about all covered executions.*
- *$\dashv_2$, the result is inconclusive. The partial reachable set $\tilde{\mathcal{R}}$ does not contain enough information to support a claim about all its extensions.*

If $\tilde{\mathcal{R}}$ is a partial overapproximation of the reachable set of a hybrid system $\mathcal{H}$, $[\![\varphi]\!]_{\tilde{\mathcal{R}}}(0) = \top$ implies $\mathcal{H} \models \varphi$, while $[\![\varphi]\!]_{\tilde{\mathcal{R}}}(0) = \bot$ implies $\mathcal{H} \not\models \varphi$.

The last claim above holds because there must exist some extension of $\tilde{\mathcal{R}}$ covering all executions of the system by definition of the reachable set.

5.1 Incremental Verification Algorithm

Algorithm 1 implements the approach outlined in Corollary 1. It incrementally computes a $\mathbb{U}_2$-satisfaction signal of the specification φ as the reachability analysis of the hybrid system $\mathcal{H}$ progresses. At its core, the algorithm alternates between computing the reachable states of the system, which allows it to observe the satisfaction or violation of predicates in new time intervals, and propagating these observations up the syntax tree of φ to update the satisfaction signal. The algorithm terminates as soon as the satisfaction signal provides a conclusive verdict. Below, we explain Algorithm 1 in more detail.

Before entering the main loop, we initialize the reachability analysis of $\mathcal{H}$ and construct the syntax tree of φ. We assume that φ is written without syntactic sugar. Each node n of the syntax tree stores the $\mathbb{U}_2$-satisfaction signal of the subformula of φ that its subtree represents. Initially, the satisfaction signal, which we refer to as $n.signal$, is $\dashv_2$ on the entire time domain, except for nodes representing *true*, where it is $\top$ everywhere.

In the main loop, we first compute one step of the reachability analysis, which yields the set of states $\mathcal{R}_I$ that the system can reach in the time interval I. For each node representing an atomic predicate $a \in \mathcal{AP}$, we determine $\hat{a}(\mathcal{R}_I)$ based on the observed set (function Eval) and update its satisfaction signal during I accordingly. Afterward, we propagate the new observations up the syntax tree. Note that all occurring signals are guaranteed to be of finite variability as long as the computed sequence of reachable sets does not exhibit Zeno behavior. If

Algorithm 1. Incremental Verification

Input: STL formula φ, hybrid system $\mathcal{H}$, time horizon $t_{\mathtt{h}}$
Output: Verdict from $\mathbb{U}_2$ on whether $\mathcal{H}$ satisfies φ

```
$reach \leftarrow$ InitializeReachabilityAnalysis($\mathcal{H}$, $t_{\mathtt{h}}$)
$tree \leftarrow$ SyntaxTree($\varphi$)
while $\neg reach$.Done() do                                  ▷ Did we reach $t_{\mathtt{h}}$?
    $\mathcal{R}_I \leftarrow reach$.NextStep()               ▷ Reachable states in time interval $I$
    for $ap \in tree.aps$ do                                  ▷ Iterate nodes representing atomic predicates
        $v \leftarrow ap$.Eval($\mathcal{R}_I$)               ▷ Set-based predicate evaluation to $v \in \mathbb{U}_1$
        $ap.signal$.Set($I$, $v$)
    end for
    $tree.root$.Propagate()                                   ▷ See Algorithm 2
    $verdict \leftarrow tree.root.signal(0)$
    if $verdict \neq \dashv_2$ then                           ▷ Is the verdict conclusive?
        return $verdict$
    end if
end while
return $tree.root.signal(0)$
```

we obtain a conclusive verdict on the satisfaction of the top-level formula φ at time 0, we return the verdict early. Otherwise, we continue until the reachable set is determined up to a given time horizon $t_{\mathtt{h}}$. After reaching $t_{\mathtt{h}}$, we return the current verdict, even if it is the inconclusive $\dashv_2$.

Algorithm 2 propagates new observations up the syntax tree. It closely follows the incremental marking procedure of Maler and Ničković [27, Algorithm 2]. The algorithm traverses the syntax tree in post-order and applies the operators developed in Sect. 4.3 to update the satisfaction signal of each node based on the satisfaction signals of its children (function Combine).

Since we only admit future connectives, the satisfaction of a formula at time t depends only on the truth values of its subformulas at times $t' \geq t$ [27, Sect. 3.2]. To exploit this, every node n stores a time interval $n.irr$ of the form $[0, t)$ or $[0, t]$, indicating the prefix of $n.signal$ that is irrelevant for updates of the parent node. Initially, $n.irr$ is empty. After updating the signal of n, we also need to revise the irrelevant prefixes of its children. To this end, we find the largest interval I containing 0 such that $n.signal$ has a conclusive value, i.e., not $\dashv_2$, everywhere in $I \setminus n.irr$; if $n.signal(0) = \dashv_2$ and $0 \notin n.irr$, we return an empty interval (function ConclusiveInterval). We exclude $n.irr$ from consideration because $n.signal$ itself is irrelevant at these times. Then, we can drop the irrelevant prefix from memory by overwriting it with $\dashv_2$.

Remark 1 (Propagation Frequency). If we conduct the reachability analysis with a small time step size, we obtain numerous observations. To reduce the overhead incurred by signal propagation, we can accumulate the observations of several reachability steps and then propagate them all at once. However, in doing so, we might compute more reachability steps than required to reach a conclusive

Algorithm 2. Signal Propagation

Input: Syntax tree node n

```
function $n$.Propagate()
    if $n$.IsLeaf() then
        return
    end if
    for $c \in n.children$ do
        $c$.Propagate()
    end for
    $\Delta \leftarrow n$.Combine($\{c.signal \mid c \in n.children\}$)
    $n.signal$.Merge($\Delta$)          ▷ Overwrite $n.signal$ with $\Delta$ where $\Delta$ is not $\dashv_2$
    for $c \in n.children$ do
        $c.irr \leftarrow$ ConclusiveInterval($n.signal$, $n.irr$)
        $c.signal$.Set($c.irr$, $\dashv_2$)          ▷ Drop the irrelevant prefix from memory
    end for
end function
```

verdict. Choosing a propagation frequency is thus a trade-off similar to the one mentioned in [27, Sect. 5.3]. In the extreme case, where we never propagate before reaching the time horizon, we obtain an offline method similar to [22,35].

5.2 Refinement via Branching the Reachability Analysis

Since we use the reachable set as the basis for verification, our approach works best if all executions of the system under scrutiny behave roughly similarly. The intuition for this is given at the end of the proof of Theorem 1: To verify that $\varphi_1 \, \mathbf{U}_I \, \varphi_2$ holds, we require that all executions covered by the reachable set satisfy the eventuality φ_2 at the same time. However, hybrid systems can have executions with vastly different behavior, e.g., due to discrete transitions changing the continuous dynamics. Moreover, the system behavior might strongly depend on the initial state. For these systems, our algorithm would often return $\dashv_1$, as the executions covered by the reachable set do not synchronize as required.

The underlying problem is that we compute just one reachable set to cover all system executions. If these executions have significant differences, the reachable set also covers many additional *spurious* executions that are infeasible according to the system dynamics. To alleviate this issue, we can perform the reachability analysis in multiple *branches* so that each branch only covers similar executions. For example, we could start a new branch whenever a discrete transition occurs. In addition, we could partition the set of initial states so that we analyze initial states that lead to vastly different behavior in separate branches.

To adapt Algorithm 1 for several branches, we clone the syntax tree whenever a new branch starts. We then process each branch using its copy of the syntax tree and combine the verdicts. If the verdicts are conclusive for all branches, we merge them using $\sqcup_1$, i.e., we return $\top$ or $\bot$ if all branches agree on the verdict, and $\dashv_1$ otherwise. If the analysis is inconclusive for at least one branch after reaching the time horizon $t_{\mathtt{h}}$, the combined verdict is also $\dashv_2$. In this case, we need to extend the time horizon only for the inconclusive branches.

6 Evaluation

We implemented our algorithm in MATLAB using CORA[2] [2] for reachability analysis. First, we demonstrate the capabilities and limitations of our method on a simple hybrid system. Then, we apply it to autonomous driving and systems biology. For all experiments, our algorithm is configured to accumulate 20 observations before propagation. Note that CORA continues the reachability analysis in a new branch (cf. Sect. 5.2) after the system takes a discrete transition.

6.1 Bouncing Ball

First, let us consider a bouncing ball, which is a classic example of a hybrid system. The bouncing ball has two state variables: height h and velocity v. It accelerates under the influence of gravity and bounces back up once it hits the ground ($h = 0$). Bouncing is modeled as a discrete transition that reduces the velocity and flips its sign (see, e.g., [33, Sect. 2.2.3] for a full derivation). Initially, we have $h \in [0.95, 1.05]$ and $v \in [-0.05, 0.05]$, which means that the first bounce happens after about 0.5 s. The time horizon for our verification algorithm is $t_{\mathtt{h}} := 2.5\,s$ and the reachability analysis uses a time step size of 0.01 s.

Table 1 summarizes the results of our experiments. The *future reach* of a formula is the amount of time it maximally looks into the future [21, Sect. 3]. Thus, approaches like [22,35] need to perform reachability analysis up to the future reach of the specification. We can successfully verify the first two specifications and falsify the third while terminating the reachability analysis well before their future reach. Even though the fourth property also holds for the system (recall that the ball bounces after about 0.5 s), we cannot verify it since the reachable set is not sufficiently accurate. However, as $\dashv_1$ is already returned after 0.7 s, we know early that we need to refine the reachable set. After refining the reachability analysis by using a time step size of 0.002 s, we can also prove this specification. The final property demonstrates the drawbacks of accumulating observations: While we could reject it already after observing the initial set, our algorithm only returns $\bot$ after 20 time steps.

For the last column of Table 1, we applied the verification algorithm by Roehm et al. [32] to the bouncing ball. Since this algorithm requires the reachable set for the entire future reach of the specification in order to start the verification, it is not applicable to the first property. We cannot compute the reachable set for an infinite time horizon here, as no fixed point is detected. As shown by the fourth specification, [32] returns $\bot$ whenever it fails to verify a property. Therefore, in contrast to our algorithm, it does not distinguish between insufficient accuracy of the reachable set and actual falsification of the property by the system. For the other properties, its verdict is the same as ours.

[2] https://cora.in.tum.de.

Table 1. Application of our approach and the approach from [32] to the bouncing ball

		Our approach		[32]
Specification	Future reach	Verdict	Termination	Verdict
$\mathbf{F}_{[0.2,\infty)}\, h < 0.5$	∞	$\top$	0.4 s	N/A
$\mathbf{F}_{[0,0.1]}(v < 0\, \mathbf{U}_{[0,1]}\, h < 0.25)$	1.1 s	$\top$	0.6 s	$\top$
$\mathbf{F}_{[0,1]}(h < 0.1 \wedge \mathbf{G}_{[0,2]}\, h < 0.3)$	3.0 s	$\bot$	1.3 s	$\bot$
$\mathbf{F}_{[0,0.1]}(v < 0\, \mathbf{U}_{[0,1]}\, h < 0.01)$	1.1 s	$\dashv_1$	0.6 s	$\bot$
$\mathbf{G}_{[0,1]}\, h < 0.1$	1.0 s	$\bot$	0.2 s	$\bot$

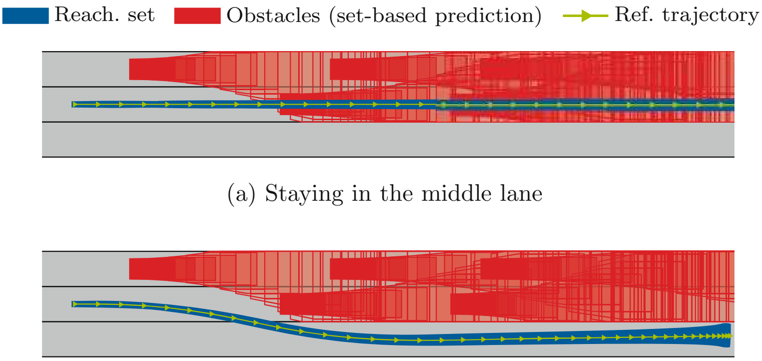
6.2 Autonomous Driving

Next, we examine an application for autonomous driving in the context of the CommonRoad[3] [5] framework. In our example scenario, an autonomous vehicle is driving in the middle lane of an interstate with another vehicle in front indicating to change from the left to the middle lane (see Fig. 5; CommonRoad scenario ID: `ZAM_HW-1_1_S-1`). Suppose the motion planner has determined two reference trajectories that the autonomous vehicle could follow for the next five seconds: one where it stays in its current lane and another where it changes to the right lane. We want to verify that the autonomous vehicle avoids collisions with other vehicles for the entire planned trajectory, even if it cannot precisely follow the trajectory due to disturbances. Moreover, it should eventually enter the right lane and stay there for at least 1 s. We formalize these requirements in STL as $(\mathbf{G}_{[0,4]}(x, y) \notin \mathcal{O}) \wedge \mathbf{F}_{[0,4]}\, \mathbf{G}_{[0,1]}(x, y) \in \mathcal{L}$, where (x, y) is the position of the autonomous vehicle, $\mathcal{O}$ is the area occupied by other vehicles according to a set-based prediction, and $\mathcal{L}$ is the right lane. Here, obtaining a verdict as soon as possible is particularly important since the autonomous vehicle has limited time to decide which trajectory to follow. With our algorithm, the vehicle can quickly reject the "stay" trajectory after computing the reachable set up to 1 s. Then, it can spend the remaining time on verifying the "change" trajectory, which yields the verdict $\top$ after performing reachability analysis up to 4 s. For the reachability analysis, we adopted a kinematic single-track model [31, Sect. 2.2] of the vehicle and assumed it tracks the reference trajectory using a P controller.

6.3 Genetic Oscillator

Finally, we consider the 9-dimensional genetic oscillator example (state variables $x_1, \ldots, x_9$) from [35, Sect. 5]. In [35], the authors verified the specification $\mathbf{G}_{[0,1]}(a_1 \vee \mathbf{G}_{[3,3.5]}\, a_2)$, where $a_1 := x_6 - 1 > 0$ and $a_2 := 0.032 - 125^2(x_4 - 0.003)^2 - 3(x_6 - 0.5)^2 > 0$. We could not verify the original property since the reachable sets computed by CORA were not accurate enough,

[3] https://commonroad.in.tum.de.



(a) Staying in the middle lane

(b) Changing to the right lane

Fig. 5. Reachable set projected to the position domain for two reference trajectories

resulting in the verdict $\dashv_1$. After slightly relaxing the property by using $a_2' := 0.04 - 125^2(x_4 - 0.003)^2 - 3(x_6 - 0.5)^2 > 0$ instead of a_2, verification succeeded. The reachability analysis was stopped after 4.5 s, matching the future reach of the formula. If we change the specification to $\mathbf{G}_{[0,2]}(a_1 \vee \mathbf{G}_{[0,0.5]}\, a_2')$, we can reject it early after computing the reachable set for up to 1.3 s.

7 Conclusion

We proposed an incremental STL verification algorithm for hybrid systems based on reachability analysis and a four-valued semantics for STL. Due to its incremental nature, our algorithm can run alongside the reachability analysis and continuously update its verdict. Consequently, it can stop the computation of the reachable set as soon as the verdict becomes conclusive. This makes our approach particularly worthwhile for high-dimensional systems, for which reachability analysis is computationally expensive. The evaluation of our prototype showed promising results across several application domains.

Acknowledgments. The authors gratefully acknowledge funding from the German Research Foundation (DFG) under grant numbers AL 1185/20-1 and GRK 2428. Moreover, they thank Benedikt Seidl for implementing the system models used for the evaluation in CORA.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ahmad, H., Jeannin, J.B.: A program logic to verify signal temporal logic specifications of hybrid systems. In: Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 1–11 (2021). https://doi.org/10.1145/3447928.3456648
2. Althoff, M.: An introduction to CORA 2015. In: Proc. of the 1st and 2nd Workshop on Applied Verification for Continuous and Hybrid Systems, pp. 120–151 (2015). https://doi.org/10.29007/zbkv
3. Althoff, M., Dolan, J.M.: Online verification of automated road vehicles using reachability analysis. IEEE Trans. Rob. **30**(4), 903–918 (2014). https://doi.org/10.1109/TRO.2014.2312453
4. Althoff, M., Frehse, G., Girard, A.: Set propagation techniques for reachability analysis. Annual Rev. Control Robot. Autonom. Syst. **4**(1), 369–395 (2021). https://doi.org/10.1146/annurev-control-071420-081941
5. Althoff, M., Koschi, M., Manzinger, S.: CommonRoad: composable benchmarks for motion planning on roads. In: Proceedings of the IEEE Intelligent Vehicles Symposium (IV), pp. 719–726 (2017). https://doi.org/10.1109/IVS.2017.7995802
6. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM **43**(1), 116–146 (1996). https://doi.org/10.1145/227595.227602
7. Bae, K., Lee, J.: Bounded model checking of signal temporal logic properties using syntactic separation. Proc. ACM Program. Lang. **3**(POPL), 51:1–51:30 (2019). https://doi.org/10.1145/3290364
8. 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: Introductory and Advanced Topics, pp. 135–175 (2018). https://doi.org/10.1007/978-3-319-75632-5_5
9. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. **20**(3), 651–674 (2010). https://doi.org/10.1093/logcom/exn075
10. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011). https://doi.org/10.1145/2000799.2000800
11. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: JuliaReach: a toolbox for set-based reachability. In: Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 39–44 (2019). https://doi.org/10.1145/3302504.3311804
12. Brieger, M., Mitsch, S., Platzer, A.: Dynamic logic of communicating hybrid programs (2023). https://doi.org/10.48550/arXiv.2302.14546
13. Chai, M., Schlingloff, B.H.: Online monitoring of distributed systems with a five-valued LTL. In: Proceedings of the IEEE International Symposium on Multiple-Valued Logic (ISMVL), pp. 226–231 (2014). https://doi.org/10.1109/ISMVL.2014.47
14. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification (CAV), pp. 258–263 (2013). https://doi.org/10.1007/978-3-642-39799-8_18
15. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. Formal Methods Syst. Design **51**(1), 5–30 (2017). https://doi.org/10.1007/s10703-017-0286-7

16. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) Formal Modeling and Analysis of Timed Systems (FORMATS), pp. 92–106 (2010). https://doi.org/10.1007/978-3-642-15297-9_9
17. Ferrère, T., Maler, O., Ničković, D., Pnueli, A.: From real-time logic to timed automata. J. ACM **66**(3), 19:1–19:31 (2019). https://doi.org/10.1145/3286976
18. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification (CAV), pp. 379–395 (2011). https://doi.org/10.1007/978-3-642-22110-1_30
19. Henzinger, T.A.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998). https://doi.org/10.1006/jcss.1998.1581
20. Ho, H.M., Ouaknine, J., Worrell, J.: Online monitoring of metric temporal logic. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification (RV), pp. 178–192 (2014). https://doi.org/10.1007/978-3-319-11164-3_15
21. Hunter, P., Ouaknine, J., Worrell, J.: Expressive completeness for metric temporal logic. In: Proceedings of the ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 349–357 (2013). https://doi.org/10.1109/LICS.2013.41
22. Ishii, D., Yonezaki, N., Goldsztejn, A.: Monitoring temporal properties using interval analysis. IEICE Trans. Fundament. Electr. Commun. Comput. Sci. **E99-A**(2), 442–453 (2016). https://doi.org/10.1587/transfun.E99.A.442
23. Kleene, S.C.: On notation for ordinal numbers. J. Symbolic Logic **3**(4), 150–155 (1938). https://doi.org/10.2307/2267778
24. Kochdumper, N., Bak, S.: Fully automated verification of linear time-invariant systems against signal temporal logic specifications via reachability analysis. Nonlinear Anal. Hybrid Syst **53**, 101491 (2024). https://doi.org/10.1016/j.nahs.2024.101491
25. Lee, J., Yu, G., Bae, K.: Efficient SMT-based model checking for signal temporal logic. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 343–354 (2021). https://doi.org/10.1109/ASE51524.2021.9678719
26. Maler, O., Ničković, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems (FORMATS/FTRTFT) pp. 152–166 (2004). https://doi.org/10.1007/978-3-540-30206-3_12
27. Maler, O., Ničković, D.: Monitoring properties of analog and mixed-signal circuits. Int. J. Softw. Tools Technol. Transf. **15**(3), 247–268 (2013). https://doi.org/10.1007/s10009-012-0247-9
28. Ničković, D., Lebeltel, O., Maler, O., Ferrère, T., Ulus, D.: AMT 2.0: Qualitative and quantitative trace analysis with extended signal temporal logic. Inter. J. Software Tools Technol. Transf. **22**(6), 741–758 (2020). https://doi.org/10.1007/s10009-020-00582-z
29. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reason. **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
30. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-63588-0
31. Rajamani, R.: Vehicle Dynamics and Control. Springer (2012). https://doi.org/10.1007/978-1-4614-1433-9
32. Roehm, H., Oehlerking, J., Heinz, T., Althoff, M.: STL model checking of continuous and hybrid systems. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis (ATVA), pp. 412–427 (2016). https://doi.org/10.1007/978-3-319-46520-3_26

33. van der Schaft, A., Schumacher, H.: An Introduction to Hybrid Dynamical Systems. Springer (2000). https://doi.org/10.1007/BFb0109998
34. Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Electr. Notes Theoretical Comput. Sci. **113**, 145–162 (2005). https://doi.org/10.1016/j.entcs.2004.01.029
35. Wright, T., Stark, I.: Property-directed verified monitoring of signal temporal logic. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification (RV), pp. 339–358 (2020). https://doi.org/10.1007/978-3-030-60508-7_19

Optimization-Based Model Checking and Trace Synthesis for Complex STL Specifications

Sota Sato[1,3](✉), Jie An[1,4](✉), Zhenya Zhang[1,2](✉), and Ichiro Hasuo[1,3](✉)

[1] National Institute of Informatics, Tokyo, Japan
{sotasato,jiean,hasuo}@nii.ac.jp
[2] Kyushu University, Fukuoka, Japan
zhang@ait.kyushu-u.ac.jp
[3] SOKENDAI (The Graduate University for Advanced Studies), Tokyo, Japan
[4] Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract. Techniques of light-weight formal methods, such as monitoring and falsification, are attracting attention for quality assurance of cyber-physical systems. The techniques require formal specs, however, and writing right specs is still a practical challenge. Commonly one relies on *trace synthesis*—i.e. automatic generation of a signal that satisfies a given spec—to examine the meaning of a spec. In this work, motivated by 1) complex STL specs from an automotive safety standard and 2) the struggle of existing tools in their trace synthesis, we introduce a novel trace synthesis algorithm for STL specs. It combines the use of MILP (inspired by works on controller synthesis) and a *variable-interval encoding* of STL semantics (previously studied for SMT-based STL model checking). The algorithm solves model checking, too, as the dual of trace synthesis. Our experiments show that only ours has realistic performance needed for the interactive examination of STL specs by trace synthesis.

1 Introduction

Safety and quality assurance of *cyber-physical systems (CPSs)* is an important and multifaceted problem. The pervasiveness and safety-critical nature of CPSs makes the problem imminent and pressing; at the same time, the problem comes with very different flavors in different application domains, calling for different solutions. For example, in the aerospace domain, full formal verification all the way up from the codebase seems feasible [33]. Such is a luxury that the automotive domain may not afford, however, because of short product cycles, dependence on third-party (thus black-box) components, heterogeneous environmental uncertainties, and fierce competition (thus tight budget).

The authors are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), the START Grant No. JPMJST2213, the ASPIRE grant No. JPMJAP2301, JST. S.S. is supported by KAKENHI No. 23KJ1011, JSPS. Z.Z. is supported by JSPS KAKENHI Grant No. JP23K16865 and No. JP23H03372.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 282–306, 2024.
https://doi.org/10.1007/978-3-031-65633-0_13

The above limitations in the automotive domain point, in the formal methods terms, to the *absence of white-box system models*. This has led to the flourish of *light-weight formal methods*, such as monitoring [8], runtime verification, and hybrid system falsification [16]. These are logic-based methods that operate on *formal specifications*, often given in *signal temporal logic (STL)* [24]. These methods give up comprehensive guarantee due to the absence of white-box system models; yet their values in practical usage scenarios are widely acknowledged.

Trace Synthesis and Model Checking. In this paper, we are motivated by some automotive instances of the *trace synthesis* problem: it asks to synthesize an execution trace σ of a system $\mathcal{M}$ that satisfies a given STL specification φ. There are two major approaches to trace synthesis for CPSs.

One common approach is via *hybrid system falsification* [16]: here, we try many input signals τ for $\mathcal{M}$, iteratively modifying them in the direction of satisfying φ; the quantitative *robust semantics* of STL [17] serves as an objective function that allows hill-climbing optimization. It is notable that the system model $\mathcal{M}$ can be *black-box*: we do not need to know its internal working; it is enough to compute the execution trace $\mathcal{M}(\tau)$ under given input τ. Falsification has attracted a lot of interest especially in the automotive domain; see e.g. [16].

We take the other approach to trace synthesis, namely as the *dual of the model checking problem*. Here model checking decides if, under *any* input τ, the execution trace $\mathcal{M}(\tau)$ satisfies φ. Our choice of this approach may be puzzling—it requires a white-box model $\mathcal{M}$, but it is rare in the automotive domain.

Analyzing Specifications (Rather Than Models). Our choice of the model checking approach to trace synthesis comes from the following basic scope of the paper: *we use trace synthesis to analyze the quality of specifications (specs).*

This is in stark contrast with many falsification tools whose scope is analyzing *models*. There, a model $\mathcal{M}$ is extensive and complex (typically a Simulink model of an actual product), and counterexample traces are used for "debugging" $\mathcal{M}$.

In this paper, instead, a model $\mathcal{M}$ is simple and white-box (it can even be the trivial model, where the input and output are the same), but a spec φ tends to be complex. One typical usage scenario for our framework is when φ is a *normative rule*—such as a law, a traffic rule, or a property required in an international standard—in which case φ is imposed on many different systems (e.g. different vehicle models). Then $\mathcal{M}$ should be a simple overapproximation of a variety of systems, rather than a detailed system model.

Another typical usage scenario of our framework is an early "requirement development" phase of the *V-model* of the automotive system design. Here, engineers fix specs that pin down the later development efforts, in which those specs get refined and realized. They want to confirm that the specs are sensible (e.g. there is no mutual conflict) and faithful to their intentions. Since a system is yet to be developed, a system model $\mathcal{M}$ cannot be detailed.

Motivating Example. More specifically, the current work is motivated by the work [30] on formalizing disturbance scenarios in the ISO 34502 standard for automated driving vehicles. There, a vehicle dynamics model is simple (the scenarios should apply to different vehicle models—see above), but STL formulas are complex. It is observed that existing algorithms have a hard time handling the complexity of specs (see §6 for experiments). This motivated our current technical development, namely a trace synthesis algorithm that exploits *white-box models* and *MILP optimization* for efficiency.

Fig. 1. Rear-end near collision

The following example illustrates the challenge encountered in [30].

Example 1.1 (rear-end near collision). We would like to express, in STL, a *rear-end near collision* scenario for two cars. It refers to those driving situations where a rear car $\mathrm{Car_r}$ comes too close to a front car $\mathrm{Car_f}$. We assume a single-lane setting (Fig. 1), so we can ignore lateral dynamics.

Consider the following STL formulas. Here, $x_\mathrm{f}, v_\mathrm{f}, a_\mathrm{f}$ are the variables for the position, velocity, and acceleration of $\mathrm{Car_f}$; the other variables are for $\mathrm{Car_r}$.

$$\begin{aligned} \texttt{danger} &:\equiv x_\mathrm{f} - x_\mathrm{r} \le 10 \\ \texttt{dyn\_inv} &:\equiv x_\mathrm{f} - x_\mathrm{r} \ge 0 \wedge 2 \le v_\mathrm{f} \le 27 \wedge 2 \le v_\mathrm{r} \le 27 \\ \texttt{trimming} &:\equiv (\Diamond\texttt{danger}) \Rightarrow \big((\Box_{[0,0.2]} a_\mathrm{r} \ge 0.5)\,\mathcal{U}\,\texttt{danger}\big) \\ \texttt{RNC1} &:\equiv \Box(\texttt{dyn\_inv} \wedge \texttt{trimming}) \wedge \Diamond_{[0,9]}\Box_{[0,1]}\texttt{danger} \end{aligned} \qquad (1)$$

The last formula RNC1 formalizes rear-end near collision; in particular, its subformula $\Diamond_{[0,9]}\Box_{[0,1]}\texttt{danger}$ requires that danger occurs within 9 s and persists for at least one second.

The formula RNC1 comes with two auxiliary conditions: dyn_inv and trimming. We shall now exhibit their content and why they are needed. In fact, these conditions arose naturally in the course of *trace synthesis*, the problem of our focus.

Specifically, in [30], we conducted trace synthesis repeatedly in order to 1) *illustrate* the meaning of STL specifications and 2) *confirm* that they reflect informal intentions. The generated traces were animated for graphical illustration. This workflow is much like in the tool STLInspector [31].

The formula dyn_inv imposes basic constraints on the dynamics of the cars. In the trace synthesis in [30], without this basic constraint, we obtained a number of nonsensical example traces in which a car warps and instantly passes the other, drives much faster than the legal maximum, and so on.

The formula trimming requires $\mathrm{Car_r}$ to accelerate until danger occurs. It was added to limit a generated trace to an interesting part. For example, a trace can have danger only after a 8-s pacific journey; animating this whole trace can easily bore users. The condition trims such a trace to the part where $\mathrm{Car_r}$ is accelerating towards danger.

The dynamics model used in [30] is the following simple one:

$$\dot{x_{\mathrm{f}}} = v_{\mathrm{f}},\ \dot{v_{\mathrm{f}}} = a_{\mathrm{f}}; \qquad \dot{x_{\mathrm{r}}} = v_{\mathrm{r}},\ \dot{v_{\mathrm{r}}} = a_{\mathrm{r}}. \tag{2}$$

This relates x, v and a in the spec (1). The double integrator model is certainly simplistic, but it suffices the purpose in [30] of illustrating and confirming specs.

Remark 1.2. In [30], after illustrating and confirming STL specs through trace synthesis, the final goal was to use them for monitoring actual driving data. Neither the dynamics model (2) nor the condition `dyn_inv` is really relevant to monitoring—actual driving data should comply with them anyway. In contrast, `trimming` is important, in order to extract only relevant parts of the data.

Technical Solution: MILP-Based Trace Synthesis. We present a novel trace synthesis algorithm. Note that it also solves the dual problem, namely STL model checking. It originates from two recent lines of work: MILP-based optimal control [14,28,29] and SMT-based STL model checking [7,23,34].

The controller synthesis techniques in [14,28,29] exploit *mixed-integer linear programming (MILP)* for efficiency. The optimal control problem that they solve can be specialized to our trace synthesis problem (detailed discussions come later). But we found their capability of handling complex specs (as in Ex. 1.1) limited, largely because of their *constant-interval encoding* to MILP.

We solve this challenge by our novel *variable-interval encoding* of the STL semantics to MILP. It is inspired by the *stable partitioning* technique introduced in [7]: the technique is used in [7,23,34] for *logical* encoding towards SMT-based model checking; we use it for *numerical* encoding to MILP. This way we will solve the *bounded* trace synthesis problem—in the sense that variability of the truth values of the relevant formulas is bounded—much like in [7,23,34]. For our MILP encoding, however, we need special care since MILP does not accommodate strict inequalities (partitions such as $\dots, (\gamma_{i-1}, \gamma_i), \{\gamma_i\}, (\gamma_i, \gamma_{i+1}), \dots$ in [7] cannot be expressed). We therefore use a novel technique called *δ-stable partitioning*.

Overall, our algorithm works as follows. We assume that a system model $\mathcal{M}$ can be MILP-encoded, either exactly or approximately. Some model families are discussed in §5. This assumption, combined with our key technique of *variable-interval MILP encoding* of STL, reduces trace synthesis to an MILP problem, which we solve by Gurobi Optimizer [20]. We conduct experimental evaluation to confirm the scalability of our algorithm, especially for complex specs (§6).

Our algorithm is *anytime* (i.e. *interruptible*): even if the budget runs out in the course of optimization, a best-effort result (the trace that is the closest to a solution so far) is obtained. A similar benefit is there in case there is no execution trace σ that satisfies the spec φ: we obtain a trace σ' that is the closest to satisfy φ. Accommodation of *parameters* is another advantage thanks to our use of MILP; we exploit it for *parameter mining* for PSTL formulas. See §3.

Both controller synthesis techniques [14,28,29] and SMT-based model checking techniques [7,23,34] can be used for trace synthesis. The methodological differences are discussed later in §1; experimental comparison is made in §6.

Contributions and Organization. We summarize our contributions.

- We introduce an optimization-based algorithm for bounded trace synthesis for STL specs. It assumes that a system model is white-box and MILP-encodable; it also solves the dual problem (namely bounded model checking).
- As a key element, we introduce a *variable-interval* encoding of STL to MILP.
- MILP encodings of some system models, notably rectangular hybrid automata and double integrator dynamics (suited for the automotive domain).
- We experimentally confirm scalability of our algorithm, especially for complex specs. Comparison is made with MILP-based optimal control [14], SMT-based model checking [34], and optimization-based falsification [11,37].
- Through the algorithm, case studies and experiments, we argue for the importance and feasibility of *spec analysis* for CPSs.

After exhibiting preliminaries on STL and stable partitioning in §2, we formulate our problems (bounded trace synthesis, model checking, etc.) in §3. In §4 we present a novel *variable-interval MILP encoding* of STL; in §5 we discuss MILP encoding of a few families of models. Our main algorithm combines these two encodings. In §6 we present experiment results.

Related Work I: Optimal STL Control with MILP. The works [14,28,29] inspire our use of MILP for STL. Their problem is *optimal controller synthesis under STL constraints*, i.e. to find an input signal τ to a system model $\mathcal{M}$ so that 1) the output signal $\mathcal{M}(\tau)$ satisfies a given STL spec φ and 2) it optimizes $J(\mathcal{M}(\tau))$, where J is a given objective function. This problem subsumes our problem of trace synthesis, by taking a constant function as J.

The algorithms in [14,28,29] reduce their problem to MILP by a *constant-interval encoding* of the robust semantics [13,17] of STL (an enhanced encoding is presented in [22]). Specifically, their system model is discrete-time dynamics $x(t + \Delta t) = f_{\mathrm{d}}(x(t), u(t), w(t))$ with a constant interval Δt.

In contrast, in our *variable-interval* encoding (§4), continuous time is discretized into the intervals $\dots, (\gamma_{i-1}, \gamma_i), \{\gamma_i\}, (\gamma_i, \gamma_{i+1}), \dots$ where the end points γ_i are also variables in MILP. This is advantageous not only in modeling precision but also in scalability: for system models that are largely continuous, constant-interval discretization incurs more integer variables in MILP, hampering the performance of MILP solvers. See §6 for experimental comparison.

Related Work II: SMT-Based STL Model Checking. Our key technical element (a variable-interval MILP encoding of STL) uses the idea of stable partitioning from [7,23,34]. They solve bounded STL model checking, and also its dual (trace synthesis). The main difference is the class of system models $\mathcal{M}$ accommodated. SMT solvers accommodate more theories than MILP solving, and thus allows encoding of a greater class of models. In contrast, by restricting the model class to MILP-encodable, our algorithm benefits speed and scalability (MILP is faster than SMT). Iterative optimization in MILP also makes our algorithm an anytime one. Native support of parameter synthesis is another plus.

Other Related Work. *Optimization-based falsification* has its root in the quantitative robust semantics of STL [13,17]; the successful combination with stochastic optimization metaheuristics has made falsification an approach of both scientific and industrial interest. See the ARCH competition report [16] for state-of-the-art. Falsification is most of the time thought of as *search-based testing*; therefore, unlike the model checking approach, the absence of counterexamples is usually not proved. Exceptions are [25,35] where they strive for probabilistic guarantees for such absence.

The current work is motivated by the observation that falsification solvers often struggle in trace synthesis for complex STL specs, even if a system model is simple. It is known that specs with more connectives pose a performance challenge, and many countermeasures are proposed, including [2] (for temporal operators) and [36,37] (for Boolean connectives).

2 Preliminaries

We let $\mathbb{N}, \mathbb{R}$ denote the sets of natural numbers and reals, respectively; $\mathbb{R}_{\geq 0}$ denotes an obvious subset. The set $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$ is that of extended reals. The set $\mathbb{B} = \{\top, \bot\}$ is for Boolean truth values. The *powerset* of a set X is denoted by $\wp(X)$. An *interval* is a subset of $\mathbb{R}_{\geq 0}$ of the form (a, b), $[a, b)$, $(a, b]$, or $[a, c]$, where $a < b$ and $a \leq c$. Therefore a singleton $\{a\}$ is an interval.

Definition 2.1 (linear predicate $p$and $[\![p]\!], \pi_p$). Given a set V of variables, a *(closed) linear predicate* is a function $p\colon \mathbb{R}^V \to \mathbb{B}$ defined as follows, using some $c \in \mathbb{R}^V$ and $b \in \mathbb{R}$: $p(x) = \top$ if and only if $c^\top x + b \geq 0$. We write $[\![p]\!]$ for the closed half-space $\{x \mid p(x) = \top\} \subseteq \mathbb{R}^V$.

For the above p, we define a function $\pi_p(x)\colon \mathbb{R}^V \to \mathbb{R}$ by $\pi_p(x) \coloneqq c^\top x + b$. This is understood as the degree of satisfaction (or violation, if negative) of a linear predicate p by $x \in \mathbb{R}^V$. Indeed, $\pi_p(x)$ is the (signed) Euclidean distance to the boundary of $[\![p]\!]$, assuming that the Euclidean norm of c is $\|c\| = 1$.

Definition 2.2 (signal). Let V be a finite set of variables and T a positive real. A *signal* over V with a time horizon T is a function $\sigma : [0, T] \to \mathbb{R}^V$. We write $\mathbf{Signal}_V^T$ for the set of all signals over V with time horizon T, or simply $\mathbf{Signal}_V$ when T is clear from the context.

If necessary, the domain $[0, T]$ of σ can be extended to $\mathbb{R}_{\geq 0}$ by setting $\sigma(t) \coloneqq \sigma(T)$ for all $t > T$. This allows us to define the notion of *t-postfix*, which will serve as the basis of the STL semantics (§2.1). Precisely, the *t-postfix* of σ is a signal σ^t defined by $\sigma^t(t') \coloneqq \sigma(t + t')$. The domain of σ^t can be chosen freely but we set it to $[0, T]$ for consistency.

Definition 2.3 (system model, trace set $\mathcal{L}(\mathcal{M})$). Let V, V' be finite sets of variables. A *system model* $\mathcal{M}$ from V' to V with a time horizon T is a function $\mathcal{M}\colon \mathbf{Signal}_{V'}^T \to \wp(\mathbf{Signal}_V^T)$. The *trace set* $\mathcal{L}(\mathcal{M}) \coloneqq \bigcup_{\tau \in \mathbf{Signal}_{V'}^T} \mathcal{M}(\tau)$ is the set of all output signals of $\mathcal{M}$ where an input signal τ can vary.

We allow system models to be nondeterministic (note the the powerset construction $\wp$); the models in §1 were deterministic for simplicity. A special case of the above is when $V' = \emptyset$, that is, when $\mathcal{M}$ does not have any input. In this case, a system model $\mathcal{M}$ can be identified with a subset $\mathcal{L}(\mathcal{M}) \subseteq \mathbf{Signal}_V$.

Example 2.4 ($\mathcal{M}_{\mathrm{RNC}}$). The dynamics model in Ex. 1.1 is formalized as a system model $\mathcal{M}_{\mathrm{RNC}}$ whose input variables (in V') are $a_{\mathrm{f}}, v_{\mathrm{f}}^{\mathrm{init}}, x_{\mathrm{f}}^{\mathrm{init}}, a_{\mathrm{r}}, v_{\mathrm{r}}^{\mathrm{init}}, x_{\mathrm{r}}^{\mathrm{init}}$, and output variables (in V) are $a_{\mathrm{f}}, v_{\mathrm{f}}, x_{\mathrm{f}}, a_{\mathrm{r}}, v_{\mathrm{r}}, x_{\mathrm{r}}$. Here, the input is acceleration rates $(a_{\mathrm{f}}, a_{\mathrm{r}})$ and the initial values of velocities and positions (modeled using signals $v_{\mathrm{f}}^{\mathrm{init}}$ etc. for convenience). The time horizon T of $\mathcal{M}$ represents its simulation time; here we set $T = 20$. Given an input signal τ, the output $\mathcal{M}(\tau)$ is a singleton $\mathcal{M}(\tau) = \{\sigma\}$, and σ is determined by the ODE (2). Specifically, $\sigma(t)(a_{\mathrm{f}}) = \tau(t)(a_{\mathrm{f}})$, $\sigma(t)(v_{\mathrm{f}}) = \tau(0)(v_{\mathrm{f}}^{\mathrm{init}}) + \int_0^t \tau(t')(a_{\mathrm{f}})\,\mathrm{d}t'$, and so on.

2.1 Signal Temporal Logic

Definition 2.5 (signal temporal logic (STL)). In STL, an *atomic proposition* over a variable set V is represented as $p :\equiv (f(\vec{w}) \geq 0)$, where $f : \mathbb{R}^V \to \mathbb{R}$ is a function that maps a V-dimensional vector $\vec{w}$ to a real. The syntax of an STL formula φ (over V) is defined as follows: $\varphi :\equiv p \mid \bot \mid \top \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \Diamond_I \varphi \mid \Box_I \varphi \mid \varphi_1 \,\mathcal{U}_I\, \varphi_2 \mid \varphi_1 \,\mathcal{R}_I\, \varphi_2$, where I is a nonsingular closed time interval, and $\Diamond_I$, $\Box_I, \mathcal{U}_I$, $\mathcal{R}_I$ are temporal operators *eventually*, *always*, *until* and *release*. Implication is defined: $\varphi_1 \Rightarrow \varphi_2 :\equiv \neg\varphi_1 \vee \varphi_2$. We write temporal operators without the subscript I when $I = [0, \infty]$ (e.g., $\Diamond$). Note that we do not lose generality by restricting the inequality in $p :\equiv (f(\vec{w}) \geq 0)$. Indeed, $\leq, <, >$ can be encoded using (a combination of) $-f$ and $\neg$.

The set $\mathrm{Sub}(\varphi)$ collects all subformulas of an STL formula φ; the set $\mathrm{AP}(\varphi)$ collects all atomic propositions α occurring in φ.

Proposition 2.6. Every STL formula has a formula in the *negation normal form (NNF)*—i.e. one in which negation $\neg$ appears only in front of atomic propositions—that is semantically equivalent. □

Assumption 2.7. We assume that each atomic proposition p is a linear predicate (Def. 2.1), that is, $f(x) = c^\top x + b$ with some $c \in \mathbb{R}^V, b \in \mathbb{R}$ in each $p :\equiv (f(\vec{w}) \geq 0)$.

The Boolean semantics $\sigma \models \varphi$ and robust semantics $[\![\sigma, \varphi]\!] \in \overline{\mathbb{R}}$ of STL are standard. See [32, Appendix A].

PSTL is a parametric extension of STL. It is from [4]; see also [9]. Its definition is in [32, Appendix A]. The semantics of PSTL formula is defined naturally by fixing $\vec{u}, \vec{v}$; see Prob. 3.3 for the specific forms we use.

2.2 Finite Variability

The satisfiability checking problem for STL—this is equivalent to the model checking problem under the trivial (identity) system model—is already

EXPSPACE-complete [3]. To ease computational complexity, *bounded model checking* has been a common approach [23,26]. Its main idea is to bound the number of time-points at which the truth value of each subformula can vary.

Definition 2.8 (finite variability [27]). A (finite) *partition* $\mathcal{P}$ of an interval $D \subseteq \mathbb{R}$ is a sequence $\mathcal{P} = (J_i)_{i=1}^{N}$ of nonempty and mutually disjoint intervals such that $\bigcup_{i=1}^{N} J_i = D$, and $\sup(J_i) \leq \inf(J_{i'})$ for any $i < i'$. A Boolean signal $q\colon \mathbb{R}_{\geq 0} \to \mathbb{B}$ is *constant* on an interval $J \subseteq \mathbb{R}_{\geq 0}$ if $q(t) = q(t')$ for any $t, t' \in J$. We say $q(t)$ has *N-bounded variability* if there exists a partition $\mathcal{P}$ of $[0, \infty)$ and $q(t)$ is constant on every interval $J \in \mathcal{P}$.

Let $\sigma\colon [0, T] \to \mathbb{R}^V$ be a signal and φ be an STL formula over V. We say that σ has the *N-bounded variability* with respect to φ if the Boolean (truth value) signal $t \mapsto (\sigma^t \models \varphi)$ has the N-bounded variability. We say σ is *finitely variable* with respect to φ if it has the N-bounded variability for some N.

Finally, we say σ has the *hereditary N-bounded variability* with respect to φ if, for each subformula $\psi \in \mathrm{Sub}(\varphi)$, σ has the N-bounded variability with respect to ψ. We write *N-HBV* for the hereditary N-bounded variability.

Lemma 2.9 ([7]). Let φ be an STL formula. A signal σ has the N-HBV with respect to φ for some N if and only if it is finitely variable with respect to each atomic proposition $p \in \mathrm{AP}(\varphi)$ occurring in φ. □

The following is the basis of bounded model checking in [7,23].

Definition 2.10 (stable partition). Let σ be a signal, φ be an STL formula, and $\mathcal{P}$ be a partition of $[0, T]$ such that every $J \in \mathcal{P}$ is singular or open. Intuitively, $\mathcal{P}$ looks like $\{\gamma_0\}, (\gamma_0, \gamma_1), \{\gamma_1\}, (\gamma_1, \gamma_2), \dots, \{\gamma_N\}$. We say $\mathcal{P}$ is a *stable partition* for σ and φ if $t \mapsto \sigma^t \models \psi$ is constant on J for each $J \in \mathcal{P}$, $\psi \in \mathrm{Sub}(\varphi)$.

3 Problem Formulation

We formulate our problems and discuss their mutual relationship. The next problem is studied in [7,23,34].

Problem 3.1 (bounded STL model checking). Given an STL formula φ (over V), a system model $\mathcal{M}$ (from V' to V) with time horizon T, and a variability bound $N \in \mathbb{N}$, decide if the following is true or not: $\sigma \models \varphi$ holds for an arbitrary trace $\sigma \in \mathcal{L}(\mathcal{M})$ (cf. Def. 2.3) that has the hereditary N-bounded variability (N-HBV) with respect to φ.

The following is the dual of Prob. 3.1, and is our main scope.

Problem 3.2 (bounded STL trace synthesis). Given $\varphi, \mathcal{M}, T$ and N as in Prob. 3.1, find a trace $\sigma \in \mathcal{L}(\mathcal{M})$ such that 1) σ has the N-HBV with respect to φ and 2) $\sigma \models \varphi$ holds, or prove that such σ does not exist.

Prob. 3.2 resembles the *falsification problem* [17]: given $\mathcal{M}$ (that can be black-box) and φ', find a *counterexample input* τ such that $\mathcal{M}(\tau) \not\models \varphi'$. The emphases and the settings are often different though; see §1.

The following is a special case of the *STL parameter mining* problem; see e.g. [9, § 3.5]. Recall from [32, Def. A.3] that $\varphi_{\vec{u},\vec{v}}$ instantiates parameters $\vec{p},\vec{q}$ in φ with real values $\vec{u},\vec{v}$ from the domains P, Q, respectively.

Problem 3.3 (bounded existential parameter mining). Let φ be a PSTL formula over parameters $(\vec{p},\vec{q})$, and $\mathcal{M}, T$ and N be as in Prob. 3.1. Find the set $\bigl\{ (\vec{u},\vec{v}) \in P \times Q \bigm| \sigma \models \varphi_{\vec{u},\vec{v}}$ for some $\sigma \in \mathcal{L}(\mathcal{M})$ that has the N-HBV wrt. $\varphi \bigr\}$.

In §6, we study a further special case where there is only one parameter p and the goal is to find the maximum p in the above set.

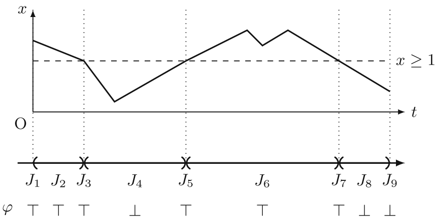


Fig. 2. A stable partition (cf. [7]) for σ and $\varphi :\equiv x \geq 1$. The symbols $\top$ and $\bot$ denote the (constant) truth value of φ each interval J_i.

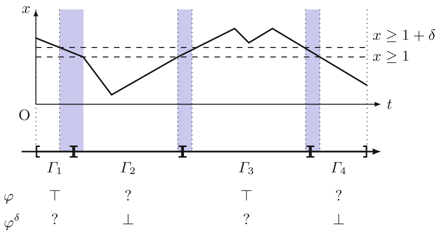


Fig. 3. A δ-stable partition (Def. 4.7) for σ and φ. Here $\varphi^{\delta} \equiv (x \geq 1+\delta)$. $\top$ and $\bot$ are much like in Fig. 2; the symbol ? indicates that the truth value is not necessarily constant. In some regions (shaded), $\sigma^t \models \varphi$ is true but $\sigma^t \models \varphi^{\delta}$ is not.

4 Variable-Interval Encoding of STL to MILP

4.1 δ-Stable Partitions

We shall adapt the idea of stable partitioning [7], reviewed in §2.2, to the current MILP setting. A major difference we need to address is that SMT is symbolic while MILP is numerical: most MILP solvers do not distinguish $<$ from $\leq$ and do not accommodate strict inequalities. See e.g. [20].

In order to address this difference, we develop a theory of *δ-stable partitions.* Here is its outline. Firstly, we replace partitions $\dots, (\gamma_{i-1}, \gamma_i), \{\gamma_i\}, (\gamma_i, \gamma_{i+1}), \dots$ used in [7] (see also Def. 2.10) with new "partitions" $\dots, [\gamma_{i-1}, \gamma_i], [\gamma_i, \gamma_{i+1}], \dots$. The latter can be expressed only using $\leq$; but they have overlaps (at γ_i). The original stability notion (see §2.2) does not fit the new partition notion—it requires "constantly true" or "never true," and prohibits overlaps. Therefore we introduce *δ-stability*; it requires either "constantly true" or "never *robustly* true."

Example 4.1. Let σ be a continuous signal. Suppose that a sequence $\mathcal{P} = (J_i)_{i=1}^{M}$ is a stable partition for σ and an STL formula φ, as illustrated in Fig. 2.

In this paper, since MILP solvers do not accommodate strict inequalities, we are forced to use closed intervals; see $\varGamma_1, \dots, \varGamma_4$ in Fig. 3. Notice that the truth value of the formula φ not constant in $\varGamma_2$ or $\varGamma_4$. To regain stability, we introduce the *δ-tightening* φ^δ of the formula φ with some $\delta > 0$ (Def. 4.4); here $\varphi^\delta \equiv (x \geq 1 + \delta)$. Then the truth value of φ^δ (instead of φ) is constantly false in $\varGamma_2$ and $\varGamma_4$, that is, φ is "never δ-robustly true" in $\varGamma_2$ and $\varGamma_4$.

Definition 4.2 (timed state sequence). A *time sequence* of $[0, T]$ is a sequence $\varGamma = (0 = \gamma_0 < \cdots < \gamma_N = T)$. Such a time sequence induces a "partition of $[0, T]$ with singular overlaps," namely $\varGamma = \bigl([\gamma_{i-1}, \gamma_i]\bigr)_{i=1}^{N}$. We identify it with the original time sequence, writing $\varGamma_i$ for the interval $[\gamma_{i-1}, \gamma_i]$.

Given a time sequence, a *timed state sequence* over V is a sequence $\varsigma = \bigl((x_0, \gamma_0), \dots, (x_N, \gamma_N)\bigr)$, where $x_0, \dots, x_N$ in $\mathbb{R}^V$.

In MILP, it is efficient to represent signals as (continuous) *piecewise-linear signals*, so that values within an interval can be deduced by linear interpolation.

Definition 4.3 (piecewise-linear signal). Given a timed state sequence $\varsigma = ((x_0, \gamma_0), \dots, (x_N, \gamma_N))$, the signal $\varsigma^{\mathrm{pwl}} \colon [0, \gamma_N] \to \mathbb{R}^V$ is defined by the following linear interpolation: $\varsigma^{\mathrm{pwl}}(t) := (1 - \lambda)x_{i-1} + \lambda x_i$ if $\gamma_{i-1} \leq t \leq \gamma_i$ (where $\lambda = \frac{1}{\gamma_i - \gamma_{i-1}}(t - \gamma_{i-1})$).

In this paper, a *piecewise-linear signal* is a signal of the form ς^{pwl} for some timed state sequence ς. Note that it is continuous everywhere, and is linear everywhere except for only finitely many points. Obviously, ς^{pwl} is finitely variable with respect to any linear predicate p (Def. 2.1).

Definition 4.4 (δ-tightening of linear predicates). Let $\delta > 0$ be a positive real and p be a linear predicate defined by $p(x) = \top \iff c^\top x + b \geq 0$. The *$\delta$-tightening* of p is a linear predicate defined by $p^\delta(x) = \top \iff c^\top x + b \geq \delta$.

Note that p^δ is stronger than p, i.e., $[\![p^\delta]\!] \subsetneq [\![p]\!]$. We further extend the concept of δ-tightening for general STL formulas in NNF (cf. Prop. 2.6). Let p^- be the linear predicate defined by $p^-(x) = \top \iff -c^\top x - b \geq 0$.

Definition 4.5 (δ-tightening of STL formulas in NNF). Let φ be an STL formula in NNF. The *δ-tightening* φ^δ of φ is the STL formula obtained from φ by replacing all occurrences of atomic predicates p (resp. $\neg p$) by p^δ (resp. $(p^-)^\delta$).

The δ-tightening construction is related to robust semantics [32, Def. A.2].

Proposition 4.6. Let σ be a signal, φ be an STL formula in NNF, and $\delta > 0$. Then $\sigma \models \varphi^\delta$ implies $[\![\sigma, \varphi]\!] \geq \delta$. □

Since the closed halfspace $[\![p^-]\!]$ coincides with the closure of the open halfspace $\mathbb{R}^V \setminus [\![p]\!]$, the robust semantics is not affected by the difference between p^- and $\neg p$. For simplicity, in the following, we assume that any STL formula in NNF does not contain negation, i.e., $\neg p$ is replaced by a new atomic proposition p^-.

We are ready to define *δ-stability*.

Definition 4.7 (δ-stability). Let φ be an STL formula over V in NNF, $\sigma \in \mathbf{Signal}_V^T$ be a signal, and $\mathit{\Gamma} = (\gamma_0, \dots, \gamma_N)$ be a time sequence (Def. 4.2) with $\gamma_N = T$. We say $\mathit{\Gamma}$ is *δ-stable* for σ and φ if, for each $i \in [1, N]$ and each subformula $\psi \in \mathrm{Sub}(\varphi)$, either of the following holds: 1) $\sigma^t \models \psi$ for each $t \in \mathit{\Gamma}_i$, or 2) $\sigma^t \not\models \psi^\delta$ for each $t \in \mathit{\Gamma}_i$.

In the above definition, in each interval $\mathit{\Gamma}_i$, a subformula ψ is either 1) always true or 2) never robustly true. The two conditions are not mutually exclusive—both hold if $\sigma^t \models \psi \wedge \neg\psi^\delta$ for all $t \in \mathit{\Gamma}_i$.

The next notion of conservative valuation records which of 1) and 2) is true in each interval. It conservatively approximates the actual truth of φ (Fig. 3).

Definition 4.8 (conservative valuation). Let φ be an STL formula in NNF, and $\mathit{\Gamma} = (\gamma_0, \dots, \gamma_N)$ be a time sequence . A *valuation* of φ in $\mathit{\Gamma}$ is a function $\mathit{\Theta} : \mathrm{Sub}(\varphi) \times [1, N] \to \mathbb{B}$ that assigns, to each subformula and index of the intervals of $\mathit{\Gamma}$, a Boolean truth value. Let σ be a signal with a time horizon $T = \gamma_N$. We say that $\mathit{\Theta}$ is a *conservative valuation* of φ in $\mathit{\Gamma}$ on σ (up to δ) if 1) $\mathit{\Theta}(\psi, i) = \top$ implies that, for each $t \in \mathit{\Gamma}_i$, $\sigma^t \models \psi$ holds; and 2) $\mathit{\Theta}(\psi, \mathit{\Gamma}_i) = \bot$ implies, for each $t \in \mathit{\Gamma}_i$, $\sigma^t \not\models \psi^\delta$.

We simply write $\langle\psi\rangle_i$ for $\mathit{\Theta}(\psi, i)$ when $\mathit{\Theta}$ is clear from context.

Suppose there exists a conservative valuation $\mathit{\Theta}$ of an STL formula φ in a time sequence $\mathit{\Gamma}$ on a signal σ up to δ. Then $\mathit{\Gamma}$ is δ-stable for σ and φ.

We shall argue in §4.2 that, for each piecewise-linear signal σ (Def. 4.3), an STL formula φ, there is a time sequence $\mathit{\Gamma}$ in which φ is δ-stable on σ. We start with a special case where φ is an atomic proposition p.

Definition 4.9. Let $x, x' \in \mathbb{R}^V$, and p be a linear predicate on V. We say (x, x') is a *δ-crossing pair* with respect to p if $x \in [\![p^\delta]\!]$ and $x' \notin [\![p^\delta]\!]$ (cf. Def. 2.1), or vice versa. A δ-crossing pair is *stationary* if $x \in [\![p]\!]$ and $x' \in [\![p]\!]$.

Lemma 4.10. *Let p be a linear predicate and σ be a piecewise-linear signal. There is a time sequence $\mathit{\Gamma} = (\gamma_0, \dots, \gamma_N)$ such that, for any $i \in [1, N]$, 1) σ is linear in the interval $[\gamma_{i-1}, \gamma_i]$, and 2) if $(\sigma(\gamma_{i-1}), \sigma(\gamma_i))$ is a δ-crossing pair, it is stationary. It follows that there is a conservative valuation $\mathit{\Theta}$ of p in $\mathit{\Gamma}$ on σ.*

Proof. The lemma argues that, whenever σ enters or leaves $[\![p^\delta]\!]$, it has to do so via $[\![p]\!] \setminus [\![p^\delta]\!]$. See Fig. 4. This can be enforced by adding suitable points to $\mathit{\Gamma}$, exploiting continuity of σ (Def. 4.3) and the intermediate value theorem. □

We note another advantage of δ-stable partitions: the number of invervals is roughly halved compared to (original) stable partitions (see Figs. 2 and 3). This advantage may be exploited also in SMT-based approaches [7] for scalability.

4.2 Variable-Interval MILP Encoding

Our MILP encoding of STL relies on the constructs in §4.1. For the purpose of trace synthesis for an STL formula φ, our basic strategy is to search for 1) a time sequence $\Gamma = (\gamma_0, \dots, \gamma_N)$ (i.e. a "partition," see Def. 4.2) and 2) a valuation $\Theta : \mathrm{Sub}(\varphi) \times [1, N] \to \mathbb{B}$, such that

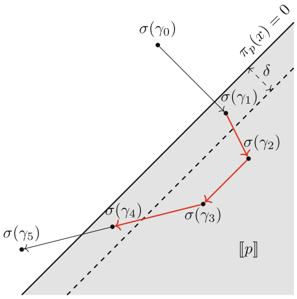


Fig. 4. A conservative valuation Θ of a linear predicate p on σ. The red segments are assigned $\top$ by Θ. (Color figure online)

- Θ is *consistent* in the sense that the truth values assigned to subformulas comply with the STL semantics (§2.1);
- Θ is *fulfilling* in the sense that it assigns $\top$ to the top-level formula φ in Γ_1 (the first interval); and
- Θ is *realizable* in the sense that there is a piecewise-linear trace $\sigma \in \mathcal{L}(\mathcal{M})$ of $\mathcal{M}$ that *yields* Θ. That is, precisely, Θ must be a conservative valuation of φ in Γ on σ (Def. 4.8).

The entities Γ, Θ we search for are expressed as MILP variables, and the above three conditions are expressed as MILP constraints. We describe these MILP variables and constraints in the rest of the section. The constraints expressing $\sigma \in \mathcal{L}(\mathcal{M})$ require system model encoding and are thus deferred to later sections.

Variables. We use the following MILP variables. Their collection is denoted by $\mathbf{Var}(\varphi, N)$. Here $N \in \mathbb{N}$ is a constant for variability bound (Prob. 3.2).

- Real-valued variables $\{\gamma_0, \dots, \gamma_N\}$ for a time sequence Γ.
- Boolean variables $\{\langle\psi\rangle_i \mid 1 \le i \le N, \psi \in \mathrm{Sub}(\varphi)\}$ for the value $\Theta(\psi, i)$ of a valuation Θ that we search for.
- Real-valued variables $\{x_{i,v} \mid 0 \le i \le N, v \in V\}$ for the values of a piecewise-linear trace $\sigma \in \mathcal{L}(\mathcal{M})$.
- Boolean variables $\{\zeta_i^p, \zeta_i^{\delta,p} \mid 0 \le i \le N, p \in \mathrm{AP}(\varphi)\}$ for the truth values of p and p^δ at time γ_i. These variables are used to detect crossing pairs (Def. 4.9).
- Real-valued variables $\{S_i^\psi \mid 0 \le i \le N, \Box_I\psi \in \mathrm{Sub}(\varphi)\}$. This auxiliary variable records for how long ψ has been true before γ_i.
- Real-valued variables $\{P_i^\psi \mid 0 \le i \le N, \Diamond_I\psi \in \mathrm{Sub}(\varphi)\}$. This auxiliary variable records for how long ψ has been false before γ_i.

By an *assignment* we refer to a function $\boldsymbol{v} \colon \mathbf{Var}(\varphi, N) \to \mathbb{R}$ such that $\boldsymbol{v}(y) \in \{0, 1\}$ for each Boolean variable y. The MILP problem is to find an assignment $\boldsymbol{v}$ that optimizes an objective under given constraints.

Notation 4.11. In what follows, as a notational convention, we simply write a variable y for the value $\boldsymbol{v}(y)$ when the assignment $\boldsymbol{v}$ is clear from context. We further write ς for the timed state sequence composed of the time sequence $\{\gamma_0, \dots, \gamma_N\}$ and the trace values $\{x_{j,v} \mid 0 \le j \le N, v \in V\}$.

Note that, in this paper, we encode the *Boolean* semantics of STL [32, Def. A.1], unlike [28,29] where the *robust* semantics is encoded in a constant-interval manner. The combination of variable-interval encoding and quantitative robust semantics is future work; for example, a quantitative extension of δ-stable partitions (§4.1) seems quite nontrivial.

Shorthands for Propositional Connectives. We use standard shorthands for Boolean connectives in MILP constraints (such as $\neg A$, $A \wedge B$ where A, B are Boolean variables). See [32, Appendix B] for the formal encodings.

Realizability Constraints: Traces and Atomic Propositions. We need to constrain $\gamma_0, \dots, \gamma_N$ to be a time sequence (Def. 4.2), using some constant $\varepsilon > 0$ and letting $\dots \ge \varepsilon$ stand for $\dots > 0$.

$$\gamma_0 = 0, \quad \gamma_N = T, \quad \gamma_i - \gamma_{i-1} \ge \varepsilon \quad \text{for all } i \in [1, N] \tag{3}$$

For each i and $p \in \mathrm{AP}(\varphi)$ (say p is defined by $c^\top x + b \ge 0$), the variables $\zeta_i^p, \zeta_i^{\delta,p}$ are constrained as follows,

$$\begin{aligned} &\zeta_i^p = 1 \;\Rightarrow\; c^\top x_i + b \ge 0 && \zeta_i^p = 0 \;\Rightarrow\; c^\top x_i + b \le -\varepsilon \\ &\zeta_i^{\delta,p} = 1 \;\Rightarrow\; c^\top x_i + b \ge \delta && \zeta_i^{\delta,p} = 0 \;\Rightarrow\; c^\top x_i + b \le \delta - \varepsilon \end{aligned} \tag{4}$$

Moreover, we impose the following to ensure that Γ is the one in Lem. 4.10:

$$\zeta_i^{\delta,p} = 0 \wedge \zeta_{i+1}^{\delta,p} = 1 \;\Rightarrow\; \zeta_i^p = 1, \quad \zeta_i^{\delta,p} = 1 \wedge \zeta_{i+1}^{\delta,p} = 0 \;\Rightarrow\; \zeta_{i+1}^p = 1 \tag{5}$$

Under constraints (3) to (5), Γ is δ-stable for ς^{pwl} (cf. Def. 4.3) and p, by Lem. 4.10. By the definition of δ-stability, we can now constrain the variable $\langle p \rangle_i$ by $\langle p \rangle_i = \zeta_{i-1}^{p,\delta} \vee \zeta_i^{p,\delta}$ for each i and $p \in \mathrm{AP}(\varphi)$.

Remark 4.12. Note that ε must be chosen to be small enough for the completeness of the encoding (Thm. 4.18). Thereafter we assume that, given a piecewise-linear signal σ and an STL formula φ, ε is small enough to find a δ-stable partition for σ and φ, and we omit ε from the constraints for simplicity.

Consistency Constraints I: Boolean Connectives. We can directly encode conjunction $\bigwedge_{j=1}^m \psi_j$ in STL by recursively applying the shorthand $\wedge$ in [32, Appendix B]: $\langle \bigwedge_{j=1}^m \psi_j \rangle_i = \langle \psi_1 \rangle_i \wedge \langle \bigwedge_{j=2}^m \psi_j \rangle_i$ for each $i \in [1, N]$. It is known that the following alternative encoding avoids auxiliary variables $\langle \bigwedge_{j=k}^m \psi_j \rangle_i$ (where k varies): for each $i \in [1, N]$, $\langle \bigwedge_{j=1}^m \psi_j \rangle_i \ge 1 - m + \sum_{j=1}^m \langle \psi_j \rangle_i$ and $\langle \bigwedge_{j=1}^m \psi_j \rangle_i \le \langle \psi_j \rangle_i$. An encoding for disjunction is given similarly: $\langle \bigvee_{j=1}^m \psi_j \rangle_i \le \sum_{j=1}^m \langle \psi_j \rangle_i$, $\langle \bigvee_{j=1}^m \psi_j \rangle_i \ge \langle \psi_j \rangle_i$.

Consistency Constraints II: Unbounded Temporal Modalities. For temporal operators with $I = [0, \infty)$, the following encodings are straightforward.

$$\begin{array}{ll}\langle \psi_1 \mathcal{U}\ \psi_2 \rangle_i = \langle \psi_2 \rangle_i \vee (\langle \psi_1 \mathcal{U}\ \psi_2 \rangle_{i+1} \wedge \langle \psi_1 \rangle_i), & \\ \langle \psi_1 \mathcal{R}\ \psi_2 \rangle_i = \langle \psi_2 \rangle_i \wedge (\langle \psi_1 \mathcal{R}\ \psi_2 \rangle_{i+1} \vee \langle \psi_1 \rangle_i) & \text{for each}\, i \in [1, N-1], \\ \langle \psi_1 \mathcal{U}\ \psi_2 \rangle_N = \langle \psi_2 \rangle_N, \quad \langle \psi_1 \mathcal{R}\ \psi_2 \rangle_N = \langle \psi_2 \rangle_N & \text{for } i = N. \end{array} \tag{6}$$

The encodings for $\Diamond, \Box$ are special cases.

Consistency Constraints III: Bounded Temporal Modalities. This is the most technically involved part. The challenge here is that the stability for $\Box_{[a,b]}\psi$ is not guaranteed by the stability for ψ (similarly for $\Diamond_{[a,b]}\psi$). Therefore we need additional MILP constraints for the stability for $\Box_{[a,b]}\psi$.

Our encoding is inspired by the results from [26]; ours is simpler thanks to our theory in §4.1 where intervals are all closed.

Recall that we use the variables S_i^ψ, P_i^ψ for this purpose. We focus on $\Box_{[a,b]}\psi$; the encoding of $\Diamond_{[a,b]}\psi$ is similar. The constraints on S_i^ψ are as follows.

$$\begin{array}{ll} S_0^\psi = 0, \qquad \langle \psi \rangle_i = 0 \ \Rightarrow\ S_i^\psi = 0, & \\ \langle \psi \rangle_i = 1 \ \Rightarrow\ S_i^\psi \geq S_{i-1}^\psi + (\gamma_i - \gamma_{i-1}) & \text{for each } i \in [1, N]. \end{array}$$

It follows that, for any non-negative real number $L \in [0, \gamma_j)$, we have $S_j^\psi \leq L$ if and only if there exists $k \in [1, j]$ such that $\langle \psi \rangle_k = 0$ and $\gamma_j - \gamma_k \leq L$.

We proceed to the constraints that describe the relationship between S_i^ψ and the semantics of $\Box_I \psi$. Suppose $\Gamma = (\gamma_0, \dots, \gamma_N)$ is δ-stable for a signal σ and ψ. Let us write $\gamma_{N+1} = \infty$ and $\langle \psi \rangle_{N+1} = \langle \psi \rangle_N$ for simplicity.

We consider consistency for the positive and negative cases separately. For the positive one (i.e. $\langle \Box_{[a,b]}\psi \rangle_i = 1$), the following observation is used.

Proposition 4.13. Let $\varphi \equiv \Box_I \psi$ be an STL formula in NNF, and Θ be a conservative valuation of ψ in $\Gamma = (\gamma_0, \dots, \gamma_N)$ on a signal σ. Given $i \in [1, N]$, suppose $(\Gamma_i + I) \cap (\gamma_{j-1}, \gamma_j] \neq \emptyset$ implies $\langle \psi \rangle_j = 1$ for each $j \in [i, N+1]$. Then $\sigma^t \models \varphi$ holds for any $t \in \Gamma_i$. □

Prop. 4.13 leads to the following MILP constraint:

$$\neg \langle \varphi \rangle_i \vee (\gamma_i + b \leq \gamma_{j-1}) \vee (\gamma_{i-1} + a > \gamma_j) \vee \langle \psi \rangle_j \ \ \text{for each}\, i \in [1, N], j \in [i, N+1].$$

The constraint itself does not follow the MILP format; we can nevertheless express it in MILP using an auxiliary Boolean variable Z_f. Specifically, an inequality $f(x) \geq 0$ in a disjunctive constraint is constrained by $Z_f = 1 \Rightarrow f(x) \geq 0$.

For the consistency in the negative case (i.e. $\langle \Box_{[a,b]}\psi \rangle_i = 0$), the counterpart of Prop. 4.13 also involves S_j^ψ. See below; it leads to an MILP constraint much like Prop. 4.13 does.

Proposition 4.14. Suppose φ, σ, $\mathit{\Gamma}$, and $\mathit{\Theta}$ are as in Prop. 4.13. For any $t \in \mathit{\Gamma}_i$, $\sigma^t \not\models \varphi^\delta$ holds if the following conditions are satisfied for each $j \in [i, N]$:

$$\begin{cases} S^\psi_j \leq b - a & \text{if } \gamma_j \in (\gamma_{i-1} + b, \gamma_i + b), \\ S^\psi_j \leq \gamma_j - \gamma_i - a & \text{if } \gamma_i + b \in [\gamma_{j-1}, \gamma_j], \\ S^\psi_N \leq \max(0, \gamma_N - \gamma_i - a) & \text{if } \gamma_i + b > \gamma_N. \end{cases} \tag{7}$$

Proof. Let $j_t \in [i, N+1]$ be the unique index such that $t+b \in [\gamma_{j_t-1}, \gamma_{j_t})$. When $j_t \leq N$ and $\gamma_{j_t} < \gamma_i + b$, we have $\gamma_{j_t} \in (\gamma_{i-1} + b, \gamma_i + b)$ and by assumption $S^\psi_{j_t} \leq b - a$. There is $k \in [1, j_t]$ such that $\langle\psi\rangle_k = 0$ and $\gamma_k \geq \gamma_{j_t} - b + a > t + a$. We obtain $\mathit{\Gamma}_k \cap (t + [a, b]) \neq \emptyset$ and then $\sigma^t \not\models \varphi^\delta$ holds. The other cases can be checked in a similar manner. □

Remark 4.15. For Prop. 4.13, the converse of the statement does not hold. This is because $\sigma^t \models \psi$ does not guarantee $\langle\psi\rangle_i \coloneqq \mathit{\Theta}(\psi, i) = 1$ where $t \in \mathit{\Gamma}_i$—we allow $\langle\psi\rangle_i = 0$ when $\sigma^t \models \psi \wedge \neg\psi^\delta$. It is similar for Prop. 4.14. However, this does not affect the completeness of the encoding (Thm. 4.18): while the converse of Prop. 4.13 does not hold for *fixed* $\mathit{\Gamma}$, in our workflow we also search for $\mathit{\Gamma}$, in which case it is easily shown that the MILP constraints derived from Prop. 4.13 are complete. The same is true for Prop. 4.14.

The remaining cases ($\varphi \equiv \psi_1 \,\mathcal{U}_I\, \psi_2$ and $\varphi \equiv \psi_1 \,\mathcal{R}_I\, \psi_2$) can be reduced to the cases for $\Box_I$ and $\Diamond_I$. It is by the rewriting techniques shown in [12]:

$$\psi_1 \,\mathcal{U}_{[a,b]}\, \psi_2 \quad \sim \quad \Diamond_{[a,b]}\psi_2 \wedge \Box_{[0,a]}(\psi_1 \mathcal{U}\ \psi_2), \tag{8}$$

$$\psi_1 \,\mathcal{R}_{[a,b]}\, \psi_2 \quad \sim \quad \Box_{[a,b]}\psi_2 \vee \Diamond_{[0,a]}(\psi_1 \mathcal{R}\ \psi_2). \tag{9}$$

These equivalences hold in both Boolean and robust semantics.

Correctness of Encoding. Let $\mathbf{Enc_{STL}}(\varphi, N, T, \delta)$ denote the polyhedron defined by the above MILP constraints. It is correct in the following sense; see also the goal we announced in the beginning of §4.2. Its proof is by induction on φ.

Lemma 4.16. Let φ be an STL formula in NNF, $N \in \mathbb{N}$, $T > 0$ and $\delta > 0$. Given an assignment $\boldsymbol{v}\colon \mathbf{Var}(\varphi, N) \to \mathbb{R}$ that lies in $\mathbf{Enc_{STL}}(\varphi, N, T, \delta)$, let $\mathit{\Gamma}$, ς be the time sequence and the timed state sequence determined by $\boldsymbol{v}$, and define a valuation $\mathit{\Theta}$ by $\mathit{\Theta}(\psi, i) \coloneqq \langle\psi\rangle_i$ (cf. Def. 4.8). Then $\mathit{\Theta}$ is a conservative valuation of φ in $\mathit{\Gamma}$ on the signal ς^{pwl}. □

We define $\mathbf{Enc}(\varphi, \mathcal{M}, N, T, \delta)$ by the intersection of $\mathbf{Enc_{STL}}(\varphi, N, T, \delta)$, the MILP encoding $\mathbf{Enc_{model}}(\mathcal{M}, N, T)$ of a system model $\mathcal{M}$, and $\langle\varphi\rangle_1 = 1$.

Theorem 4.17 (soundness). Let φ be an STL formula in NNF, $\mathcal{M}$ be a model with a time horizon T, $N \in \mathbb{N}$ and $\delta > 0$. If an assignment $\boldsymbol{v}$ lies in $\mathbf{Enc}(\varphi, \mathcal{M}, N, T, \delta)$, the induced ς^{pwl} has $\varsigma^{\mathrm{pwl}} \in \mathcal{L}(\mathcal{M})$ and $[\![\varsigma^{\mathrm{pwl}}, \varphi]\!] \geq 0$. □

Theorem 4.18 (completeness). Assume the setting of Thm. 4.17. If there is piecewise-linear $\sigma \in \mathcal{L}(\mathcal{M})$ such that $[\![\sigma, \varphi]\!] \geq \delta$, there is an assignment $\boldsymbol{v}$ that lies in $\mathbf{Enc}(\varphi, \mathcal{M}, N, T, \delta)$ for some $N \in \mathbb{N}$. □

5 System Models and Their MILP Encoding

We introduce the MILP encoding $\mathbf{Enc_{model}}(\mathcal{M}, N, T)$ for some families of models $\mathcal{M}$. We introduce an exact encoding for *rectangular hybrid automata (RHAs)*, and an approximate one for *HAs with closed-form solutions*. We also introduce a refinement of the latter—it is more precise and efficient—restricting to *double integrator dynamics*. The last is useful for automotive examples such as Ex. 1.1.

We defer the discussion of RHAs for the space reason; see [32, Appendix C]. We thus focus on the other two families.

5.1 HAs with Closed-Form Solutions

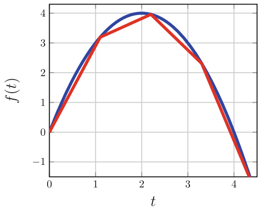


Fig. 5. MILP encoding of $f(t)$

Here we are interested in hybrid automata (HAs) whose continuous flow dynamics at each control mode has a closed-form solution. The basic idea is simple and it is illustrated in Fig. 5, where the solution $f(t)$ of dynamics (blue) is approximated by a piecewise linear function (red). Such MILP encoding is standard; see e.g. [5].

We formalize this intuition. Firstly, to accommodate input signals $\tau \in \mathbf{Signal}_{V'}$ (Def. 2.3), we extend the HA definition so that some variables x^{in} can be designated to be *input variables*. This means that there are no ODEs whose left-hand side is $\dot{x}^{\text{in}}$, and that the variable updates associated with mode transitions never change x^{in}.

Then the above "closed-form solution" assumption on an HA $\mathcal{H}$ is precisely described as follows. Let $\vec{x^{\text{in}}} = (x^{\text{in}}_1, \dots, x^{\text{in}}_k)$ enumerate $\mathcal{H}$'s input variables, and $\vec{x} = (x_1, \dots, x_l)$ enumerate its other variables. We assume that, for the flow dynamics at each control mode u, there is a *closed-form solution*

$$\vec{x}(t) \;=\; f_u(t, \vec{x^{\text{in}}}, \vec{x_0}) \qquad \text{such that, for each } t_0 \in \mathbb{R}_{\geq 0},\ f_u(t_0, \vec{x^{\text{in}}}, \vec{x_0}) \text{ is a} \quad (10)$$
$$\text{linear function over the variables } \vec{x^{\text{in}}}, \vec{x}_0.$$

Here, the variable t is the elapsed time since the arrival at the current control mode u; the variables $\vec{x^{\text{in}}}$ refer to the input variables (their values are assumed to be constant within the same mode); and the variables $\vec{x_0}$ refer to the *initial* values of $\vec{x}$ on the arrival at u. The assumption holds in many examples, such as polynomial dynamics.

Let us motivate the assumption. A closed-form solution f_u helps precision: in piecewise linear approximation such as in Fig. 5, errors do not accumulate over time; in contrast, if a closed-form solution is not given, our alternative will be numerical integration e.g. by the Euler method, where errors accumulate. The linearity assumption in (10) is there for MILP encoding; see below.

Our approximate MILP encoding poses the closed-form solution assumption and follows the intuition of Fig. 5. Specifically, 1) it fixes a constant $\Delta t \in \mathbb{R}_{\geq 0}$ as a sampling interval; 2) it obtains a family $\bigl(f_u(k\cdot \Delta t, \vec{x^{\mathrm{in}}}, \vec{x}_0)\bigr)_k$ of linear functions over the variables $\vec{x^{\mathrm{in}}}, \vec{x}_0$; and 3) the value of $\vec{x}$ at the elapsed time t is expressed by the linear interpolation

$$\tfrac{(k+1)\Delta t - t}{\Delta t} f_u(k\Delta t, \vec{x^{\mathrm{in}}}, \vec{x}_0) + \tfrac{t - k\Delta t}{\Delta t} f_u\bigl((k+1)\Delta t, \vec{x^{\mathrm{in}}}, \vec{x}_0\bigr), \tag{11}$$

where k is such that $k\Delta t \le t \le (k+1)\Delta t$. This encoding of flow dynamics is combined with the HA structure, much like in [32, Appendix C], yielding an approximate MILP encoding of the whole HA.

The above encoding has two sources of numerical errors. One is *linear interpolation*. Errors caused by it are illustrated in Fig. 5 as the vertical margin between blue and red.

The other source is *binary expansion* [18,19], a standard MILP technique for encoding bilinear functions. Indeed, in (11), $t, \vec{x^{\mathrm{in}}}, \vec{x}_0$ are all continuous variables in MILP, and the expression (11) can contain their products. The linearity assumption in (10) has been posed to restrict (11) to bilinear.

5.2 HAs with Double Integrator Dynamics

Our next focus is a special case of the model family of §5.1, where each continuous flow is *double integrator dynamics*. This is important because 1) it gets rid of one of the two error sources in §5.1, namely linear interpolation, by the *trapezoidal rule*, and 2) it can be used for many automotive dynamics models (cf. Ex. 1.1).

The *trapezoidal rule* is a basic technique in numerical integration [6], where $\int_a^b g(t)\,\mathrm{d}t$ is approximated by $(b-a)\frac{g(a)+g(b)}{2}$. For double integrator dynamics, we apply the trapezoidal rule to the velocity v, and it is exact since v's evolution is linear. This allows us to express the position x in the bilinear form $x = t\cdot \frac{v_0+v}{2}$, using the variables t (elapsed time), v_0 (initial velocity), and v (current velocity). Thus we can dispose of the sampling points and their interpolation (11) in §5.1.

We exploit this encoding for our automotive case studies such as Ex. 1.1.

6 Implementation and Experiments

We implemented, in Python, our MILP encodings of the STL semantics (§4) and two model families, namely RHAs [32, Appendix C] and double integrator dynamics (§5.2; multiple modes are not supported since our benchmarks do not need them). The hyperparameter δ in our encoding is fixed at 0.1 for all benchmarks. The resulting MILP constraints are solved by Gurobi Optimizer [20]. This prototype implementation is called *STLts*—STL trace synthesizer.

Our experiments are designed to address the following research questions.

RQ1 Assess the effect of variability bounds N (Prob. 3.2) on the performance.
RQ2 Compare the performance of STLts with optimization-based falsification, and with SMT-based model checking.
RQ3 Assess the performance of STLts for real-world complex scenarios.

RQ4 Assess the performance of STLts in parameter mining (Prob. 3.3).

We used three classes of benchmarks: *rear-end near collision (RNC)*, *navigation (NAV)*, and *disturbance scenarios in ISO 34502 (ISO)*. In each class, we have multiple STL specs, resulting in benchmarks such as RNC1, RNC2, etc.

Rear-End Near Collision (RNC1–3). As discussed in Ex. 1.1, these automotive benchmarks are simplifications of the ISO benchmarks below. The spec `RNC1` is presented in Ex. 1.1. The system model (2) (see also $\mathcal{M}_{\mathrm{RNC}}$ in Ex. 2.4) is double integrator dynamics (§5.2) and is shared by the benchmarks RNC1–3.

The other two specs `RNC2`, `RNC3` are defined as follows, using formulas in (1):

$$\begin{aligned} \texttt{RNC2} \quad &:\equiv \big(\Box(x_{\mathrm{f}} - x_{\mathrm{r}} \geq 0)\big) \wedge \\ &\qquad \Diamond_{[0,9]} \big((\Box_{[0,1]} \texttt{danger}) \wedge (\Box_{[0,1]} a_{\mathrm{r}} \geq 1) \wedge (\Diamond_{[1,5]} \neg \texttt{danger}) \big) \\ \texttt{trimming2} \quad &:\equiv (\Diamond \texttt{danger}) \Rightarrow \big((\Box_{[0,1]} a_{\mathrm{r}} \geq 1)\, \mathcal{U}\, \texttt{danger} \big) \\ \texttt{RNC3} \quad &:\equiv \Box(\texttt{dyn\_inv} \wedge \texttt{trimming2}) \wedge \Diamond_{[0,9]} \Box_{[0,1]} \texttt{danger} \end{aligned} \tag{12}$$

Navigation (NAV1–2). Here we use a system model that adapts NAV-2 from [15]. The latter is a standard example of an RHA [32, Appendix C], used e.g. in [10].

Our system model $\mathcal{M}_{\mathrm{NAV}}$ is an RHA that describes the motion of a point robot in a 2×2 grid where each region has a rectangular vector field, with a time horizon $T = 40$. See Fig. 6. We have 4 regions $\ell_1, \ldots, \ell_4$, each associated with rectangular bounds for $\dot{x}, \dot{y}$ and invariants; besides, we set an unsafe region `unsafeR` ($x \in [9, 10]$) and a goal region `goalR` ($x \in [4, 6] \wedge y \in [2, 5]$). The robot starts from an initial position (x_0, y_0) where $x_0 \in [0, 3] \wedge y_0 = 0$.

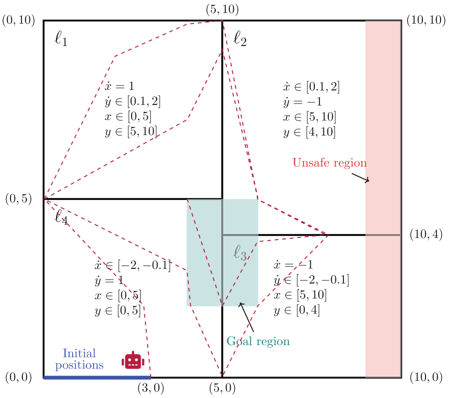


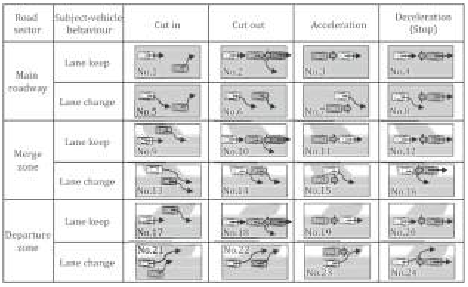
Fig. 6. The RHA $\mathcal{M}_{\mathrm{NAV}}$ for NAV1–2

We consider two specs: `NAV1` $:\equiv \Diamond(\Box_{[0,3]}((x, y) \in \texttt{goalR})) \wedge \Box(x \notin \texttt{unsafeR})$ and `NAV2` $\equiv \Box((x, y) \in \ell_3 \rightarrow \Diamond_{[0,3]}(x, y) \in \ell_4)$. `NAV1` is almost a standard reach-avoid constraint, but it additionally requires the *persistence* to the goal region for three seconds. Such specifications are not accommodated in many control and model checking frameworks specialized in reach-avoid constraints (see e.g. [10]). `NAV2` is a *response specification*—the trigger (being in ℓ_3) must be responded by moving to ℓ_4 within a three-second deadline. Such specs are common in manufacturing; see e.g. [36].

ISO 34502 Disturbance Scenarios for Automated Driving (ISO1, ISO3, . . . , ISO8). These benchmarks motivated the current work. As discussed in §1 (see Ex. 1.1), we obtained in [30] complex STL specs as the formalization of the disturbance scenarios in the ISO 34502 standard, but in our illustration efforts by trace synthesis, we found that existing techniques such as optimization-based falsification struggle.

In our experiments, the system model is similar to $\mathcal{M}_{\mathrm{RNC}}$ (Ex. 1.1 and 2.4), while lateral dynamics is added and the time horizon is 10 time units here. As for specs, we use seven STL specs $\mathtt{ISO1}, \mathtt{ISO3}, \dots, \mathtt{ISO8}$; these are obtained in [30] as the formalization of the *disturbance scenarios No. 1,3,...,8* in the ISO 34502 standard for automated driving vehicles. See Table 1. Scenario No. 2 was omitted in [30] since it involves three vehicles; we omit Scenarios No. 9–24 since they are the same with No. 1–8 except in the road shape.

Table 1. Disturbance scenarios in the ISO 34502 standard. Table from [21]



Road sector	Subject-vehicle behaviour	Cut in	Cut out	Acceleration	Deceleration (Stop)
Main roadway	Lane keep	No.1	No.2	No.3	No.4
	Lane change	No.5	No.6	No.7	No.8
Merge zone	Lane keep	No.9	No.10	No.11	No.12
	Lane change	No.13	No.14	No.15	No.16
Departure zone	Lane keep	No.17	No.18	No.19	No.20
	Lane change	No.21	No.22	No.23	No.24

Specifically, the specs $\mathtt{ISO}i$ follow the common format shown below [30]:

$$\begin{aligned} \mathtt{ISO}i \;&\equiv \mathtt{initSafe} \wedge \mathtt{disturb}_i, \\ \mathtt{disturb}_i \;&\equiv \mathtt{initialCondition}_i \wedge \mathtt{behaviourSV}_i \wedge \mathtt{behaviourPOV}_i \end{aligned}$$

where SV refers to the subject ("ego") vehicle and POV refers to the principal other vehicle. The component formulas $\mathtt{initialCondition}_i$, $\mathtt{behaviourSV}_i$ and $\mathtt{behaviourPOV}_i$ vary for different scenarios (No. i). Going into their definitions are beyond the scope of this paper; we highlight $\mathtt{ISO5}$ as an example to demonstrate the complexity of the specs $\mathtt{ISO}i$.

$$\begin{aligned} \mathtt{initialCondition}_5 \;&\equiv \top \qquad\qquad \mathtt{behaviourSV}_5 \;\equiv\; \mathtt{leavingLane}(\mathtt{SV}, L) \\ \mathtt{behaviourPOV}_5 \;&\equiv \mathtt{cutIn}(\mathtt{POV}, \mathtt{SV}) \\ \mathtt{leavingLane}(a, L) \;&\equiv \mathtt{atLane}(a, L) \wedge \Diamond(\neg \mathtt{atLane}(a, L)) \\ \mathtt{cutIn}(\mathtt{POV}, \mathtt{SV}, L) \;&\equiv \neg \mathtt{sameLane}(\mathtt{POV}, \mathtt{SV}, L) \wedge \Diamond\big(\mathtt{danger}(\mathtt{SV}, \mathtt{POV}) \\ &\quad \wedge \Diamond_{[0,\mathtt{minDanger}]}(\mathtt{sameLane}(\mathtt{SV}, \mathtt{POV}, L) \wedge \mathtt{aheadOf}(\mathtt{SV}, \mathtt{POV}))\big) \\ \mathtt{danger}(\mathtt{SV}, \mathtt{POV}) \;&\equiv \Box_{[0,\mathtt{minDanger}]} \mathtt{rssViolation}(\mathtt{SV}, \mathtt{POV}) \end{aligned} \tag{13}$$

The formulas not defined here are suitably defined atomic propositions.

Experiment Settings. Our implementation *STLts* is compared with the following tools: 1) a widely used optimization-based falsification tool *Breach* [11]; 2) another falsification tool *ForeSee* [1,37] that emphasizes optimized treatment of Boolean connectives in STL; 3) an MILP-based STL optimal control tool *bluSTL* [14]; and 4) *STLmc*, an SMT-based bounded STL model checker [34].

The experiments were conducted on an Amazon EC2 c4.4xlarge instance (2.9 GHz Intel Xeon E502666 v3, 30.0GB RAM) running Ubuntu Server 20.04.

RQ1: the Effect of the Variability Bound N. There is an obvious trade-off about the choice of a variability bound N (Prob. 3.2): bigger N means the search is more extensive, but it incurs greater computational cost.

This tendency is confirmed in our experiments; the result for the `ISO6` benchmark is in Fig. 7 for illustration. Here, synthesis was successful for $N = 4$ for the first time.

We also observe in the figure that computational cost is low when trace synthesis is unsuccessful. This suggests the following strategy: we start with small N and increment it if trace synthesis is unsuccessful. We might waste time by trying too small N's; but the wasted time should be small.

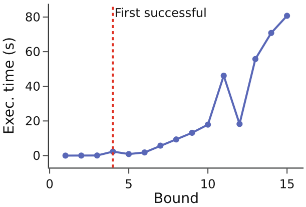


Fig. 7. Execution time of STLts for different var. bd. N, on `ISO6`

Experimental Results, Overview. Our experimental results are in summarized in Table 2, where the best performers are highlighted by color.

We explain the missing entries. In ($*$), the tool is not applicable due to the nondeterminism of the benchmark. In (†), we did not conduct experiments since the performance comparison with STLts is already clear with simpler `RNC` benchmarks. In (‡), bluSTL does not support multiple control modes. (¶) is because bluSTL (at least its implementation available to us) does not support the until $\mathcal{U}$ modality.

Overall, our STLts is clearly the best performer in all benchmarks but one. The other tools time out, or takes tens of seconds. For our motivation of *illustrating* STL specs by trace synthesis in close interaction with users, tens of seconds is prohibitively long. The results adequately demonstrate satisfactory performance of our algorithm, in trace synthesis for complex STL specs.

Table 2. Experimental results for trace synthesis, showing execution time (seconds). (N) for STLts is the first successful bound. Timeout (t/o) is 600 s.

	STLts		Breach	ForeSee	bluSTL	STLmc
`RNC1`	0.1	(3)	59.4	546.8	(¶)	t/o
`RNC2`	0.3	(4)	9.3	104.3	14.3	t/o
`RNC3`	0.1	(3)	81.3	197.4	(¶)	t/o
`NAV1`	32.5	(17)	($*$)	($*$)	(‡)	16.5
`NAV2`	2.1	(11)				10.0
`ISO1`	0.4	(3)	8.9	t/o	(†)	(†)
`ISO3`	0.2	(2)	t/o	t/o		
`ISO4`	0.4	(2)	t/o	t/o		
`ISO5`	9.9	(4)	31.2	435.8		
`ISO6`	2.4	(4)	t/o	58.9		
`ISO7`	0.6	(3)	33.6	187.2		
`ISO8`	1.5	(3)	38.8	t/o		

RQ2: Comparison with Other Approaches. A summary of comparison is in Table 3. The comparison with optimization-based falsification tools is as we expected—their struggle with complex specs motivated this work (§1). Boolean connectives in STL specs have been found problematic in falsification: this is called the *scale problem* [36,37]. The results in Table 2 show that our benchmark specs are even beyond the capability of ForeSee, a tool that incorporates Monte Carlo tree search to specifically handle the scale problem. After all, one can say

Table 3. Comparison of our approach (STLts) with baselines (Breach, ForeSee, bluSTL, STLmc). Highlited cells represent positive features.

Feature	**STLts**	**Breach/ForeSee**	**bluSTL**	**STLmc**
Trace synthesis for analyzing specs	● Successful in all benchmarks with large STL formulas	◑ Good for falsifying models but not good with large STL formulas	- Timeout in most of benchmarks	◔ Timeout except for linear dynamics
Model checking	◑ Complete up to N and δ	-	◑ Control synthesis with guarantee	● Complete up to N
Parameter mining	◑ By MILP	-	◑ By MILP	◔ By binary search
Continuous STL semantics	● Variable-interval encoding	- Discretized	- Discretized	● Variable-interval encoding
Accommodated class of nonlinear dynamics	MILP-encodable, can be nondeterministic	Black-box, deterministic	MILP-encodable, can be nondeterministic	SMT-encodable, can be nondeterministic

● = full support; ◑ = partial support; ◔ = very limited support; - = not supported.

that falsification tools are aimed at complex *models*, while our STLts is aimed at complex *specs*.

STLmc has a similar ("dual") scope and utilizes a similar technique (stable partitioning) to our STLts; the main difference is that STLmc is SMT-based while STLts is MILP-based. Therefore STLts accommodates a smaller class of models, but it can be faster on them exploiting numeric optimization. Table 2 suggests the advantage of STLts for common STL specs in manufacturing.

RQ3: Performance in Real-World Scenarios. For this RQ, we refer to STLts's performance on the ISO benchmarks. Illustrating the specs `ISO`i by trace synthesis is a real-world problem about safety standards for automated driving (§1), and Table 2 shows that STLts has sufficient performance and scalability to handle complex specs there (see (13)).

RQ4: Performance in Parameter Mining. We conducted parameter mining experiments with the `ISO8` benchmark. Its specification has a subformula $\texttt{fasterThan}(SV, POV, p)$ that requires that SV's velocity is bigger than POV's by at least a parameter p. We used STLts to solve Prob. 3.3, that is, to find the maximum p for which a satisfying trace exists.

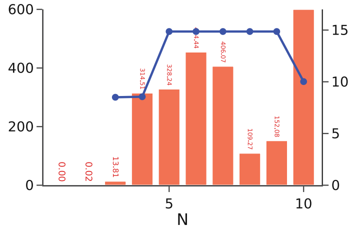


Fig. 8. STLts for parameter synthesis. Red is execution time (axis left, seconds); blue is the maximum p (axis right). (Color figure online)

Figure 8 shows the results with varying variability bound N. Parameter mining is generally more expensive than trace synthesis. This is because the former has a nontrivial objective function (namely p in this example), while the latter does not (it is thus a constraint satisfaction problem). We observe the optimization with $N \geq 10$ resulted in a timeout. The tendency, much like in trace synthesis, is that the result (max p) improves but execution time gets larger as N becomes bigger (there are some exceptions such as $N = 8, 9$ though). Taking the same strategy as above (incrementing N), it takes roughly 10 min to obtain a largely converged value ($\sim$ 14.9 for the maximum p). Overall, we believe this is a realistic performance for practical usage.

References

1. ForeSee falsification solver (2021). https://github.com/choshina/ForeSee
2. Akazaki, T., Hasuo, I.: Time robustness in MTL and expressivity in hybrid system falsification. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 356–374. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_21
3. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM **43**(1), 116–146 (1996). https://doi.org/10.1145/227595.227602
4. Asarin, E., Donzé, A., Maler, O., Nickovic, D.: Parametric identification of temporal properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 147–160. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_12
5. Asghari, M., Fathollahi-Fard, A.M., Mirzapour Al-e hashem, S.M.J., Dulebenets, M.A.: Transformation and linearization techniques in optimization: a state-of-the-art survey. Mathematics **10**(2), 283 (2022). https://doi.org/10.3390/math10020283
6. Atkinson, K.E.: An Introduction to Numerical Analysis. Wiley, New York, second edn. (1989). http://www.worldcat.org/isbn/0471500232
7. Bae, K., Lee, J.: Bounded model checking of signal temporal logic properties using syntactic separation. Proc. ACM Program. Lang. **3**(POPL), 51:1–51:30 (2019). https://doi.org/10.1145/3290364
8. 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
9. Bartocci, E., Mateis, C., Nesterini, E., Nickovic, D.: Survey on mining signal temporal logic specifications. Inf. Comput. **289**(Part), 104957 (2022). https://doi.org/10.1016/J.IC.2022.104957
10. Bu, L., Frehse, G., Kundu, A., Ray, R., Shi, Y., Zaffanella, E.: Arch-comp22 category report: Hybrid systems with piecewise constant dynamics and bounded model checking. 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. 44–57. EasyChair (2022). https://doi.org/10.29007/lnzf
11. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
12. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_19

13. 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
14. Donzé, A., Raman, V.: BluSTL: controller synthesis from signal temporal logic specifications. In: ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems, pp. 160–150. https://doi.org/10.29007/g39q
15. Duggirala, P.S., Mitra, S.: Abstraction refinement for stability. In: 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems, pp. 22–31. IEEE (2011). https://doi.org/10.1109/ICCPS.2011.24
16. Ernst, G., et al.: ARCH-COMP 2021 category report: falsification with validation of results. In: Frehse, G., Althoff, M. (eds.) 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21), Brussels, Belgium, July 9, 2021. EPiC Series in Computing, vol. 80, pp. 133–152. EasyChair (2021). https://doi.org/10.29007/XWL1
17. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoret. Comput. Sci. **410**(42), 4262–4291 (2009). https://doi.org/10.1016/j.tcs.2009.06.021
18. Glover, F.: Improved linear integer programming formulations of nonlinear integer problems. Manag. Sci. **22**, 455–460 (1975). https://doi.org/10.1287/mnsc.22.4.455
19. Gupte, A., Ahmed, S., Cheon, M.S., Dey, S.: Solving mixed integer bilinear problems using MILP formulations. SIAM J. Optim. **23**(2), 721–744 (2013). https://doi.org/10.1137/110836183
20. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2023). https://www.gurobi.com
21. Road vehicles - Test scenarios for automated driving systems - Scenario based safety evaluation framework. Standard, International Organization for Standardization, Geneva, CH (2022)
22. Kurtz, V., Lin, H.: A more scalable mixed-integer encoding for metric temporal logic. IEEE Control. Syst. Lett. **6**, 1718–1723 (2022). https://doi.org/10.1109/LCSYS.2021.3132839
23. Lee, J., Yu, G., Bae, K.: Efficient SMT-based model checking for signal temporal logic. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 343–354 (2021). https://doi.org/10.1109/ASE51524.2021.9678719
24. 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
25. Pedrielli, G., et al.: Part-X: A family of stochastic algorithms for search-based test generation with probabilistic guarantees. IEEE Trans. Autom. Sci. Eng. 1–22 (2023). https://doi.org/10.1109/TASE.2023.3297984
26. Prabhakar, P., Lal, R., Kapinski, J.: Automatic trace generation for signal temporal logic. In: 2018 IEEE Real-Time Systems Symposium (RTSS), pp. 208–217. IEEE, Nashville, TN (2018). https://doi.org/10.1109/RTSS.2018.00038
27. Rabinovich, A.M.: On the decidability of continuous time specification formalisms. J. Log. Comput. **8**(5), 669–678 (1998). https://doi.org/10.1093/logcom/8.5.669
28. Raman, V., Donzé, A., Maasoumy, M., Murray, R.M., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Model predictive control with signal temporal logic specifications. In: 53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles,

CA, USA, December 15-17, 2014, pp. 81–87. IEEE (2014). https://doi.org/10.1109/CDC.2014.7039363

29. Raman, V., Donzé, A., Sadigh, D., Murray, R.M., Seshia, S.A.: Reactive synthesis from signal temporal logic specifications. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, pp. 239–248. ACM, Seattle Washington (2015). https://doi.org/10.1145/2728606.2728628
30. Reimann, J., et al.: Temporal logic formalisation of ISO 34502 critical scenarios: modular construction with the RSS safety distance. In: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing (SAC 2024) to appear (2024). arXiv:2403.18764
31. Roehm, H., Heinz, T., Mayer, E.C.: STLInspector: STL Validation with Guarantees. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 225–232. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_11
32. Sato, S., An, J., Zhang, Z., Hasuo, I.: Optimization-based model checking and trace synthesis for complex STL specifications (extended version) (2024). available on arXiv
33. Souyris, J., Wiels, V., Delmas, D., Delseny, H.: Formal verification of avionics software products. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 532–546. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_34
34. Yu, G., Lee, J., Bae, K.: Stlmc: Robust STL model checking of hybrid systems using SMT. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. LNCS, vol. 13371, pp. 524–537. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_26
35. Zhang, Z., Arcaini, P.: Gaussian process-based confidence estimation for hybrid system falsification. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 330–348. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_18
36. Zhang, Z., Hasuo, I., Arcaini, P.: Multi-armed bandits for Boolean connectives in hybrid system falsification. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 401–420. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_23
37. Zhang, Z., Lyu, D., Arcaini, P., Ma, L., Hasuo, I., Zhao, J.: Effective hybrid system falsification using monte carlo tree search guided by QB-robustness. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 595–618. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_29

Inner-Approximate Reachability Computation via Zonotopic Boundary Analysis

Dejin Ren[1,2](✉), Zhen Liang[3], Chenyu Wu[1,2], Jianqiang Ding[4], Taoran Wu[1,2], and Bai Xue[1,2](✉)

[1] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
`{rendj,wucy,wutr,xuebai}@ios.ac.cn`

[2] University of Chinese Academy of Sciences, Beijing, China

[3] College of Computer Science and Technology, National University of Defense Technology, Changsha, China
`liangzhen@nudt.edu.cn`

[4] Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
`jianqiang.ding@aalto.fi`

Abstract. Inner-approximate reachability analysis involves calculating subsets of reachable sets, known as inner-approximations. This analysis is crucial in the fields of dynamic systems analysis and control theory as it provides a reliable estimation of the set of states that a system can reach from given initial states at a specific time instant. In this paper, we study the inner-approximate reachability analysis problem based on the set-boundary reachability method for systems modelled by ordinary differential equations, in which the computed inner-approximations are represented with zonotopes. The set-boundary reachability method computes an inner-approximation by excluding states reached from the initial set's boundary. The effectiveness of this method is highly dependent on the efficient extraction of the exact boundary of the initial set. To address this, we propose methods leveraging boundary and tiling matrices that can efficiently extract and refine the exact boundary of the initial set represented by zonotopes. Additionally, we enhance the exclusion strategy by contracting the outer-approximations in a flexible way, which allows for the computation of less conservative inner-approximations. To evaluate the proposed method, we compare it with state-of-the-art methods against a series of benchmarks. The numerical results demonstrate that our method is not only efficient but also accurate in computing inner-approximations.

Keywords: Inner-approximations · Reachability Analysis · Set-boundary analysis · Zonotopal tiling · Nonlinear systems

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 307–328, 2024.
https://doi.org/10.1007/978-3-031-65633-0_14

1 Introduction

Reachability analysis involves the computation of reachable sets, which are sets of states achieved either through trajectories originating in a given initial set (i.e., forward reachable sets) or through the identification of initial states from which a system can reach a specified target set (i.e., backward reachable sets) [23]. This problem is fundamental and finds motivation in various applications such as formal verification, controller synthesis, and estimation of regions of attraction. As a result, it has garnered increasing attention from both industrial and academic communities, leading to the development of numerous theoretical results and computational approaches [2]. For many systems, exact reachability analysis is shown to be undecidable [14], particularly in the case of nonlinear systems. Hence, approximation methods are often employed. However, in order to use these approximations as a basis for formal reasoning about the system, it is crucial that they possess certain guarantees. Specifically, it is desirable for the computed approximation to either contain or be contained by the true reachable set, resulting in what are known as outer-approximations and inner-approximations.

This paper focuses on inner-approximate reachability analysis, which calculates an inner-approximation of the reachable set for systems described by ordinary differential equations (ODEs). The inner-approximate reachability analysis has various applications. For instance, it can be used to falsify a safety property by performing forward inner-approximate reachability analysis, which computes an inner-approximation of the forward reachable set [21]. If the computed inner-approximation includes states that violate the safety property, then the safety property is not satisfied. On the other hand, it can be used to find a set of initial states that satisfy a desired property by performing backward inner-approximate reachability analysis [20]. Recently, it has been applied to path-planning problems with collision avoidance [30]. Several methods have been proposed for the inner-approximate reachability analysis, such as Taylor models [7], intervals [12], and polynomial zonotopes [18].

In the computation of inner-approximations, the accumulation of computational errors, known as the wrapping effect [25], becomes pronounced with the propagation of the initial set. To overcome this, a common approach is to partition the initial set into smaller subsets, enabling independent computations on each subset. However, this widely used method often results in an excessively large number of subsets, causing burdensome computation. Consequently, in [18,33], set-boundary reachability methods were developed based on a meticulous examination of the topological structure. These methods contract a pre-computed outer-approximation by excluding the reachable set from the boundary of the initial set, resulting in an inner-approximation. Compared to the partition of the entire initial set, set-boundary methods alleviate the computational burden and enhance the tightness of results by focusing on splitting only the boundary of the initial set. Hence, the precision of extracting and refining[1] the boundary of initial set significantly influences the non-conservativeness

[1] If $\mathcal{P}$ and $\mathcal{Q}$ are partitions (or covers) of a set X, then $\mathcal{P}$ refines $\mathcal{Q}$ if for every $U \in \mathcal{P}$, there is $V \in \mathcal{Q}$ such that $U \subset V$.

of inner-approximation aimed to compute. However, existing boundary operations have limitations that impact the precision and application of set-boundary reachability methods, either restricting the initial sets to be interval-formed [18] or utilizing interval sets to outer-approximate the set boundary [33], which leads to an overly conservative inner-approximation and hinders the application of set-boundary reachability methods.

On this concern, this paper proposes a novel set-boundary reachability method focusing on efficient extraction and refinement of the initial set's boundary, along with flexible inner-approximation generations. We adopt zonotopes as the abstract representation of states due to their remarkable advantages: the facets of a zonotope remain zonotopes and can be split into non-overlapping subsets while preserving their zonotopic nature. Based on the symmetric property of zonotope's boundary, we propose an algorithm which can efficiently extract all facets of zonotopes. To further refine the extracted boundary, a fundamental algorithm is developed to partition a zonotope into smaller, non-overlapping zonotopes, termed tiling algorithm. This algorithm leverages two innovative data structures, named as boundary and tiling matrices, providing a clear and efficient implementation of the partition procedure. Complexity analysis demonstrates the superior advantages of the tiling algorithm in computational complexity compared to the existing method [17]. Finally, we contract a pre-computed outer-approximation of reachable set to obtain an inner-approximation, which is achieved by excluding the outer-approximation of the reachable set from the refined boundary of the initial set. In contrast to proportionally shrinking the shape of computed outer-approximation utilized in existing method [33], we provide a more flexible strategy that allows an adaptive modification on the configuration of zonotopic outer-approximations, leading to more non-conservative inner-approximations.

The main contributions of this paper are as follows:

- *A Non-overlapping Zonotope Splitting Algorithm.* We present a novel algorithm that efficiently splits a zonotope into non-overlapping subsets, while preserving their zonotopic properties. By utilizing boundary and tiling matrices, our algorithm offers a more straightforward implementation with improved computational complexity compared to existing methods.
- *An Adaptive Contraction Strategy.* We put forward an adaptive contraction strategy for computing a zonotopic inner-approximation of the reachable set. This strategy, compared to existing methods, provides a more flexible approach for the contraction of the pre-computed outer-approximations, generating less conservative inner-approximations.
- *A Prototype Tool - BdryReach.* We have developed a prototype tool named BdryReach to implement our proposed approach, which is available from https://github.com/ASAG-ISCAS/BdryReach. Numerous evaluations on various benchmarks demonstrate that BdryReach outperforms state-of-the-art tools in terms of efficiency and accuracy.

Related Work

Inner-Approximation Analysis. The methods for inner-approximation computation are generally categorized into two main groups: constraint solving methods and set-propagation methods. Constraint solving methods avoid the explicit computation of reachable sets, but have to address a set of quantified constraints, which are generally constructed via Lyapunov functions [6], occupation measures [19] and equations relaxation [32,34]. However, solving these quantified constraints is usually computationally intensive (except the case of polynomial constraints for which there exists advanced tools such as semi-definite programming).

The set propagation method is an extension of traditional numerical methods for solving ODEs using set arithmetic rather than point arithmetic. While this method is simple and interesting, a major challenge is the propagation and accumulation of approximation errors over time. To ease this issue efficiently, various methods employing different representations have been developed. [28] presented a Taylor model backward flowpipe method that computes inner-approximations by representing them as the intersection of polynomial inequalities. However, this approach relied on a computationally expensive interval constraint propagation technique to ensure the validity of the representation. In [12], an approach is proposed to compute interval inner-approximations of the projection of the reachable set onto the coordinate axes for autonomous nonlinear systems. This method is later extended to systems with uncertain inputs in [13]. However, they cannot compute an inner-approximation of the entire reachable set, as studied in the present work. [33] proposed a set-boundary reachability method which propagates the initial set's boundary to compute an polytopic inner-approximation of the reachable set. However, it used computationally expensive interval constraint satisfaction techniques to compute a set of intervals which outer-approximates the initial set's boundary. Recently, inspired by the computational procedure in [18,33] introduced a promising method based on polynomial zonotopes to compute inner-approximations of reachable sets for systems with an initial set in interval form. The method presented in this work is also inspired by the in [33]. However, we propose efficient and accurate algorithms for extracting and refining the boundary of the initial set represented by zonotopes and an adaptive strategy for contracting outer-approximations, facilitating the computation of non-conservative inner-approximations.

Splitting and Tiling of Zonotopes. To mitigate wrapping effect [25] and enhance computed results, it is a common way to split a zonotope into smaller zonotopes during computation. Despite zonotopes being special convex polytopes with centrally symmetric faces in all dimensions [36], traditional polytope splitting methods such as [4,15] cannot be directly applied. The results obtained through these approaches are polytopes, not necessarily zonotopes. In the works [3,31], they split a zonotope by bisecting it along one of its generators. However, the sub-zonotopes split by this way often have overlap parts, resulting in loss of precision and heavy computation burden. Hence, there is a pressing need for methods that split a zonotope into non-overlapping sub-zonotopes. The problem of zonotopal tiling, i.e., paving a zonotope by tiles (sub-zonotopes) without gaps and overlaps, is an important topic in combinatorics and topology [5,36]. In the realm of zonotopal tiling, Bohne-Dress theorem [27] plays a crucial role by proving that a tiling of a zonotope can be uniquely represented by a collection of sign vectors or oriented matroid. Inspired by this theorem, [17] developed a tiling method by enumerating the vertices of the tiles as sign vectors of the so-called hyperplane arrangement [22] corresponding to a zonotope. However, in this paper we provide a novel and more accessible method for constructing a zonotopal tiling, which has better computational complexity.

The remainder of this paper is organized as follows. The inner-approximate reachability problem of interest is presented in Sect. 2. Then, we elucidate our reachability computational approach in Sect. 3 and evaluate it in Sect. 4. Finally, we summarize the paper in Sect. 5. Due to space limitations, proofs, examples, some tables and figures are omitted and can be found in the extended version [26], the "Appendix" appeared in this paper is referred to the appendix in [26].

2 Preliminaries

2.1 Notation

The notations and operations concerning space, vectors, matrices, and sets utilized in this paper are presented in Table 1, where the symbols and descriptions for operations on vectors, matrices, and sets are mainly illustrated with specific examples of a vector $\boldsymbol{x}$, a matrix $\boldsymbol{M}$, and a set $\mathit{\Delta}$.

Table 1. Notations utilized in the paper

Symbol	Description	Symbol	Description
$\mathbb{R}^k$	k-dimensional real space	$\mathbb{R}^{m,n}$	space of $m \times n$ real matrices
$\mathbb{N}_{[m,n]}$	non-negative integers in $[m, n]$	$\boldsymbol{x}_1 \cdot \boldsymbol{x}_2$	inner product of $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$
$\boldsymbol{x}, \boldsymbol{y}, \cdots$	vectors, boldface lowercase	$\boldsymbol{M}, \boldsymbol{N}, \cdots$	matrices, boldface uppercase
$\boldsymbol{0}$	vectors with all zero entries	$\boldsymbol{1}$	vectors with all one entries
$\boldsymbol{M}(i, \cdot)$	i-th row vector of $\boldsymbol{M}$	$\boldsymbol{M}(\cdot, j)$	j-th column vector of $\boldsymbol{M}$
$\boldsymbol{x}(i)$	i-th entry of $\boldsymbol{x}$	$\boldsymbol{M}(i, j)$	j-th entry in i-th row of $\boldsymbol{M}$
rows($\boldsymbol{M}$)	number of rows of $\boldsymbol{M}$	cols($\boldsymbol{M}$)	number of columns of $\boldsymbol{M}$
$\boldsymbol{M}(-1, \cdot)$	last row of $\boldsymbol{M}$	$\boldsymbol{M}(\cdot, -1)$	last column of $\boldsymbol{M}$
$\boldsymbol{M}^{[i]}$ $[\boldsymbol{M}; \boldsymbol{x}^\intercal]$	delete i-th row of $\boldsymbol{M}$ add $\boldsymbol{x}$ to last row of $\boldsymbol{M}$	$\boldsymbol{M}^{\langle i \rangle}$ $(\boldsymbol{M}, \boldsymbol{x})$	delete i-th column of $\boldsymbol{M}$ add $\boldsymbol{x}$ to last column of $\boldsymbol{M}$
rank($\boldsymbol{M}$)	rank of $\boldsymbol{M}$	$\lVert \boldsymbol{x} \rVert$	norm of $\boldsymbol{x}$
$\mathit{\Delta}^\circ$	interior of set $\mathit{\Delta}$	$\partial \mathit{\Delta}$	boundary of set $\mathit{\Delta}$
$\lvert \mathit{\Delta} \rvert$	cardinality of set $\mathit{\Delta}$	$\mathcal{S}_1 \setminus \mathcal{S}_2$	$\{s \mid s \in \mathcal{S}_1 \wedge s \notin \mathcal{S}_2\}$

2.2 Problem Statement

This paper considers nonlinear systems which are modelled by ordinary differential equations of the following form:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) \tag{1}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{f}$ is a locally Lipschitz continuous function. Thus, given an initial state $\boldsymbol{x}_0$, there exists an unique solution $\phi(\cdot;\boldsymbol{x}_0) : [0, T_{\boldsymbol{x}_0}) \rightarrow \mathbb{R}^n$ to system (1), where $[0, T_{\boldsymbol{x}_0})$ is the maximal time interval on which $\phi(\cdot;\boldsymbol{x}_0)$ is defined.

Given a set $\mathcal{X}_0$ of initial states, the reachable set is defined as follows:

Definition 1 (Reachable Set). *Given system* (1) *and an initial set* $\mathcal{X}_0$*, the reachable set at time* $t > 0$ *is*

$$\Phi(t;\mathcal{X}_0) \triangleq \{\phi(t;\boldsymbol{x}_0) \mid \boldsymbol{x}_0 \in \mathcal{X}_0\}.$$

The exact reachable set $\Phi(t;\mathcal{X}_0)$ is usually impossible to be computed, especially for nonlinear systems. Outer-approximations and inner-approximations are often computed for formal reasoning on the system.

Definition 2. *Given an initial set* $\mathcal{X}_0$ *and a time instant* $t > 0$*, an outer-approximation* $O(t;\mathcal{X}_0)$ *of the reachable set* $\Phi(t;\mathcal{X}_0)$ *is a superset of the set* $\Phi(t;\mathcal{X}_0)$*, i.e.,*

$$\Phi(t;\mathcal{X}_0) \subseteq O(t;\mathcal{X}_0);$$

an inner-approximation $U(t;\mathcal{X}_0)$ *of the reachable set* $\Phi(t;\mathcal{X}_0)$ *is a subset of the set* $\Phi(t;\mathcal{X}_0)$*, i.e.,*

$$U(t;\mathcal{X}_0) \subseteq \Phi(t;\mathcal{X}_0).$$

In this paper, we focus on the computation of an inner-approximation represented by zonotopes. Zonotope is a special class of convex polytopes with the centrally symmetric nature. It can be viewed as a Minkowski sum of a finite set of line segments, known as *G-representation*, which is defined as the following.

Definition 3 (Zonotope). *A zonotope* Z *with* p *generators is a set*

$$\begin{aligned} Z &= \left\{\boldsymbol{x} \in \mathbb{R}^n \middle| \boldsymbol{x} = \boldsymbol{c} + \sum\nolimits_{i=1}^{p} \alpha_i \cdot \boldsymbol{g_i}, -1 \le \alpha_i \le 1\right\} \\ &= \left\{\boldsymbol{x} \in \mathbb{R}^n \middle| \boldsymbol{x} = \boldsymbol{c} + \boldsymbol{G\alpha}, -\boldsymbol{1} \le \boldsymbol{\alpha} \le \boldsymbol{1}\right\}, \end{aligned}$$

denoted by $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$*, where* $\boldsymbol{c} \in \mathbb{R}^n$ *is referred as center and* $\boldsymbol{g}_1, \cdots, \boldsymbol{g}_p \in \mathbb{R}^n$ *as generators of zonotope.* $\boldsymbol{G} = (\boldsymbol{g}_i)_{1 \le i \le p} \in \mathbb{R}^{n,p}$ *is called generator matrix.*

For a zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$ in space $\mathbb{R}^n$, it is called *k-dimensional* if $\operatorname{rank}(\boldsymbol{G}) = k, k \le n$. A k-dimensional zonotope can be reduced into space $\mathbb{R}^k$ without altering its shape. Furthermore, the facets of a k-dimensional ($k \ge 1$) zonotope are $(k-1)$-dimensional zonotopes. If an n-dimensional zonotope has n independent generators, then it's called *parallelotope*. Additionally, If there is a zonotope Z' such that $Z' \subsetneq Z$, Z' is called a *sub-zonotope* of Z.

3 Methodology

In this section we introduce our set-boundary reachability method to compute inner-approximations of reachable sets. Firstly, the framework of our method is presented in Subsect. 3.1. Then, we introduce the algorithm of extracting the exact boundary of a zonotope in Subsect. 3.2, the tiling algorithm for boundary refinement in Subsect. 3.3 and the strategy for computing an inner-approximation via contracting an outer-approximation in Subsect. 3.4.

3.1 Inner-Approximation Computation Framework

The framework of computing inner-approximations in this paper follows the one proposed in [33], but with minor modifications.

Given system (1) with an initial set $\mathcal{X}_0$, represented by a zonotope, and a time duration $T = Nh$, where $h > 0$ is the time step and N is a non-negative integer, we compute a zonotopic inner-approximation $U((k+1)h; \mathcal{X}_0)$ of the reachable set $\Phi((k+1)h; \mathcal{X}_0)$ for $k \in \{0, 1, \cdots, N\}$. The inner-approximation $U_{k+1} = U((k+1)h; \mathcal{X}_0)$ is computed based on $U_k = U(kh; \mathcal{X}_0)$ ($U_0 := \mathcal{X}_0$) with the following procedures:

1. extract and refine the boundary ∂U_k of U_k ;
2. compute a zonotopic outer-approximation $O(h; U_k)$ of reachable set $\Phi(h; U_k)$, and an outer-approximation $O(h; \partial U_k)$ of reachable set $\Phi(h; \partial U_k)$. These outer-approximations can be computed using existing zonotope-based approaches such as [3];
3. contract $O(h; U_k)$ to obtain a zonotopic inner-approximation candidate U'_{k+1} by excluding the set $O(h; \partial U_k)$, i.e., let $U'_{k+1} \cap O(h; \partial U_k) = \emptyset$;
4. compute an outer-approximation of the reachable set $O(h; \boldsymbol{c})$ of the time-inverted system $\dot{\boldsymbol{x}} = -\boldsymbol{f}(\boldsymbol{x})$ with the single initial state $\boldsymbol{c}$, where $\boldsymbol{c}$ is the center of the zonotope U'_{k+1}. If the computed outer-approximation $O(h; \boldsymbol{c})$ is included in the set U_k, then $U_{k+1} := U'_{k+1}$ is an inner-approximation of the reachable set $\Phi((k+1)h; \mathcal{X}_0)$;

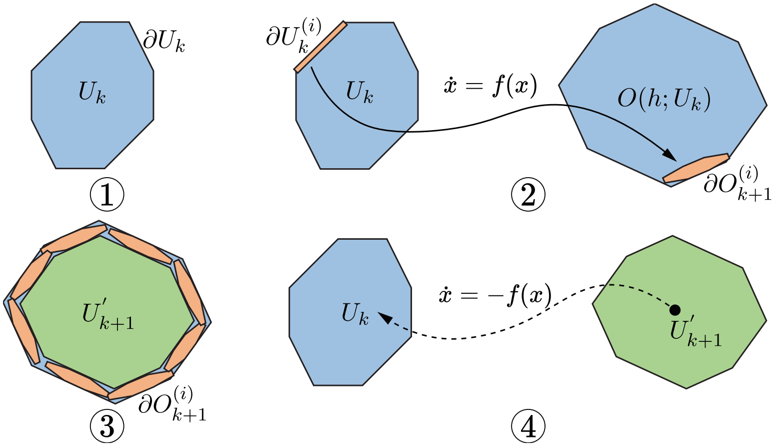


Fig. 1. Illustration of inner-approximation computation framework

The overall computational workflow is visualized in Fig. 1. There are three computational procedures that affect the efficacy (i.e., accuracy and efficiency) of inner-approximation computation in the aforementioned framework: the extraction and refinement of the boundary ∂U_k, reachability analysis for computing outer-approximations $O(h; U_k), O(h; \partial U_k)$, and contraction of $O(h; U_k)$ to obtain an inner-approximation candidate U'_{k+1}. Since there are well-developed reachability algorithms in existing literature for computing outer-approximations such as [3,11], we in the following focus on other two computational procedures. For the first one, as the outer-approximation computed $O(h; \partial U_k)$ would be excluded from $O(h; U_k)$, the accuracy of $O(h; \partial U_k)$ significantly affects the one of U_{k+1}. Additionally, the accuracy of $O(h; \partial U_k)$ strongly correlates with the size of ∂U_k. To improve the accuracy of U_{k+1}, two algorithms are proposed: one for extracting and the other for tiling the boundary of a zonotope (i.e., splitting the boundary into sub-zonotopes without overlaps). As for the third one, an adaptive strategy is developed to make the inner-approximation U_{k+1} much tighter. This is achieved by contracting $O(h; U_k)$ in a flexible way, deviating from the proportional reduction of the size of $O(h; U_k)$ in the existing methods [33].

3.2 Extraction of Zonotopes' Boundaries

In this subsection we introduce the algorithm for extracting the exact boundary of a zonotope. The concept of cross product of a matrix provided by [24] will be utilized herein, which is formulated below.

Definition 4 (Cross Product). *Given a matrix $\boldsymbol{M} \in \mathbb{R}^{n,n-1}$ in which the column vectors are linearly independent. The cross product of $\boldsymbol{M}$ is a vector of the following form:*

$$\mathtt{CP}(\boldsymbol{M}) = \left(\det\left(\boldsymbol{M}^{[1]}\right), \cdots, (-1)^{i+1}\det\left(\boldsymbol{M}^{[i]}\right), \cdots, (-1)^{n+1}\det\left(\boldsymbol{M}^{[n]}\right)\right)^{\mathsf{T}},$$

where $\det(\cdot)$ *is the determinant of a matrix.*

The cross product of $\boldsymbol{M} \in \mathbb{R}^{n,n-1}$ can be viewed as the normal vector of the hyperplane spanned by $n-1$ linearly independent column vectors in $\boldsymbol{M}$.

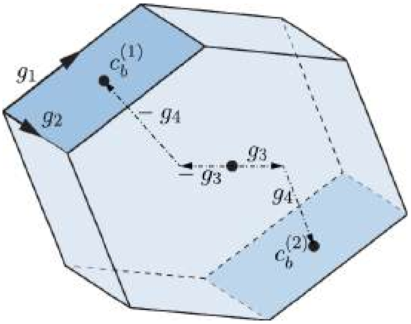


Fig. 2. Illustration of boundary extraction algorithm

The boundary extraction algorithm is established on the fact that a zonotope is centrally symmetric and each facet, which is a zonotope, has congruent facets on the opposite side of the center (e.g., two dark blue facets in Fig. 2).

Algorithm 1. Boundary Extraction Algorithm

Input: An n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle, \boldsymbol{G} = (\boldsymbol{g}_i)_{1 \le i \le p} \in \mathbb{R}^{n \times p}$.
Output: The boundary of the zonotope Z, i.e., ∂Z.

```
$\partial Z := \emptyset$
$\mathcal{B} := \{\boldsymbol{B}_b = (\boldsymbol{g}_i)_{i \in \{k_1, \cdots, k_{n-1}\}}, 1 \le k_1 < \cdots < k_{n-1} \le p \mid \mathrm{rank}(\boldsymbol{B}_b) = n - 1\}$
while $\mathcal{B} \ne \emptyset$ do
    $\boldsymbol{v} := \mathtt{CP}(\boldsymbol{B}_b)$
    $\boldsymbol{B} := \boldsymbol{B}_b = (\boldsymbol{g}_{k_1}, \boldsymbol{g}_{k_2}, \cdots, \boldsymbol{g}_{k_{n-1}}) \in \mathcal{B}$, $\boldsymbol{c}_b^{(i)} := \boldsymbol{c}, i \in \{1, 2\}$
    for all $\boldsymbol{g}_k = G(\cdot, k), k \in \{1, 2, \cdots, p\} \setminus \{k_1, k_2, \cdots, k_{n-1}\}$ do
        if $\boldsymbol{v} \cdot \boldsymbol{g}_k = 0$ then
            $\boldsymbol{B} := (\boldsymbol{B}, \boldsymbol{g}_k)$
        else if $\boldsymbol{v} \cdot \boldsymbol{g}_k > 0$ then
            $\boldsymbol{c}_b^{(i)} := \boldsymbol{c}_b^{(i)} + (-1)^i \boldsymbol{g}_k$, $i \in \{1, 2\}$
        else
            $\boldsymbol{c}_b^{(i)} := \boldsymbol{c}_b^{(i)} - (-1)^i \boldsymbol{g}_k$, $i \in \{1, 2\}$
        end if
        $Z_b^{(i)} := \left\langle \boldsymbol{c}_b^{(i)}, \boldsymbol{B} \right\rangle$, $i \in \{1, 2\}$
        $\partial Z := \partial Z \cup \{Z_b^{(1)}, Z_b^{(2)}\}$
    end for
    for all $\boldsymbol{B}_b \in \mathcal{B}$ do
        if $\boldsymbol{B}_b$ is a submatrix of $\boldsymbol{B}$ then
            $\mathcal{B} := \mathcal{B} \setminus \{\boldsymbol{B}_b\}$
        end if
    end for
end while
return $\partial Z$
```

Given an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$, where $\boldsymbol{c} \in \mathbb{R}^n$ and $\boldsymbol{G} = (\boldsymbol{g}_i)_{1 \le i \le p} \in \mathbb{R}^{n,p}$, for each two symmetric facets, they lie in parallel hyperplanes and share the same generators. The two parallel hyperplanes are spanned by a part of generators of Z, which can form a submatrix of $\boldsymbol{G}$ with rank $n - 1$. In boundary extraction algorithm, i.e., Algorithm 1, we firstly enumerate all potential $n \times (n - 1)$ submatrices of $\boldsymbol{G}$ which are able to span a hyperplane. For a certain hyperplane spanned by a submatrix $\boldsymbol{B}_b$, to confirm the center and generators of its corresponding facets, we compute its normal vector by the cross product operator $\mathtt{CP}(\cdot)$, then the center of the two symmetric facets can be respectively determined by moving the center $\boldsymbol{c}$ along the positive and negative directions of generators which are not perpendicular to $\mathtt{CP}(\boldsymbol{B}_b)$, and the generator matrix of these corresponding facets can be represented by $\boldsymbol{B}_b$ appending generators parallel to the hyperplane. The visible operations stated above are shown in Fig. 2.

The computation of a zonotope's boundary is summarized in Algorithm 1. Its soundness, i.e., the set computed by Algorithm 1 is equal to the boundary ∂Z of the zonotope Z, is justified in Theorem 1, whose proof is available in Appendix A. In order to enhance the understanding of Algorithm 1, we provide a simple example, Example 1 in Appendix B, to illustrate the computational process of Algorithm 1.

Remark 1. In space $\mathbb{R}^n$, if a zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$ isn't n-dimensional, i.e., $\text{rank}(\boldsymbol{G}) < n$, then the boundary of this zonotope is itself.

Theorem 1 (Soundness of boundary extraction algorithm). *Given an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$ with p generators, the set computed by Alg. 1 is equal to its boundary ∂Z.*

The Complexity of Boundary Extraction Algorithm. For an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle, \boldsymbol{G} \in \mathbb{R}^{n,p}$, it has M facets, where $M \le \binom{p}{n-1}$. The number of $n \times (n-1)$ submatrices of $\boldsymbol{G}$ is $\binom{p}{n-1}$, and the computation of the rank of an $n \times (n-1)$ matrix has the complexity $O(n(n-1)^2)$ (using QR decomposition), then the computation in Line 2 has the complexity $O(n(n-1)^2\binom{p}{n-1})$. In "while" Loop (Line 3–22), it has $\frac{M}{2}$ iterations. For the operation $\texttt{CP}(\cdot)$ on an $n \times (n-1)$ matrix, its complexity is $n\,\text{DET}(n-1)$, where $\text{DET}(n)$ denote the complexity of computing a determinant of an $n \times n$ square matrix. By LU-decomposition, $\text{DET}(n)$ is $O(n^3)$, however by Coppersmith-Winograd algorithm [10], it can reach $O(n^{2.373})$. For each $\boldsymbol{B}_b \in \mathcal{B}$, checking the inner product between $\boldsymbol{v}$ and remaining generators has $p - n + 1$ loops, and the inner product has complexity $O(n)$. Thus, the complexity of Algorithm 1 is $\frac{M}{2}(n\,\text{DET}(n-1) + n(p-n+1)) + O(n(n-1)^2\binom{p}{n-1}) = O\left(Mn(\text{DET}(n-1) + p) + n(n-1)^2\binom{p}{n-1}\right)$.

3.3 Zonotopal Tiling and Boundary Refinement

This subsection introduces our tiling algorithm which can split a zonotope into sub-zonotopes without overlaps and then elaborates how this tiling algorithm is employed to refine the boundaries of zonotopes.

The boundary matrix, which is constructed according to Algorithm 1, plays an important role in our tiling algorithm. Its entries are able to characterize the centers and generators for all facets of a zonotope.

Definition 5 (Boundary Matrix). *Given an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$ with M facets, where $\boldsymbol{c} \in \mathbb{R}^n$ and $\boldsymbol{G} \in \mathbb{R}^{n,p}$, its boundary matrix $\boldsymbol{B} \in \mathbb{R}^{M,p}$ is a matrix whose each entry is 0,1,or -1, where*

1. $\boldsymbol{B}(i,j) = 0$ *implies that the j-th generator $\boldsymbol{g}_j$ is a generator of the i-th facet (corresponding to Line 8 in Algorithm 1);*
2. $\boldsymbol{B}(i,j) = -1$ *implies that in order to obtain the center of the i-th facet, the MINUS operator is applied to the j-th generator $\boldsymbol{g}_j$ (corresponding to Line 10 and 12 in Algorithm 1);*
3. $\boldsymbol{B}(i,j) = 1$ *implies that in order to obtain the center of the i-th facet, the PLUS operator is applied to the j-th generator $\boldsymbol{g}_j$ (corresponding to Line 10 and 12 in Algorithm 1).*

From the boundary matrix of a zonotope, one can obtain all its facets. Appendix B provides an example (Example 2) to illustrate this claim.

Another matrix, tiling matrix, is constructed to store the outcomes of the tiling algorithm, i.e., all the non-overlapping sub-zonotopes whose union covers the original zonotope. Similar to the boundary matrix, a row of tiling matrix represents a sub-zonotope.

Definition 6 (Tiling Matrix). *Given an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$, where $\boldsymbol{c} \in \mathbb{R}^n$ and $\boldsymbol{G} \in \mathbb{R}^{n,p}$, its tiling matrix $\boldsymbol{T} \in \mathbb{R}^{s,p}$ is a matrix satisfying the following conditions:*

1. *its each entry is 0,1 or -1, which has the same meaning with the one in the boundary matrix;*
2. *each row defines a sub-zonotope Z_i such that $\bigcup_{i=1}^{s} Z_i = Z$ and $Z_i^\circ \cap Z_j^\circ = \emptyset$ for $i \neq j$.*

Our tiling algorithm is based an intuitive observation: for a zonotope, moving its one-sided facets towards to the opposite side along the direction of a generator results in a new zonotope with this generator removed, simultaneously, several sub-zonotopes are generated by adding this generator to all these facets. This process, which is visualized in Fig. 3, can be iteratively conducted, until a parallelotope remains. At this point, the tiling algorithm terminates, yielding a collection of tiles denoted as zonotopes that tile the original zonotope.

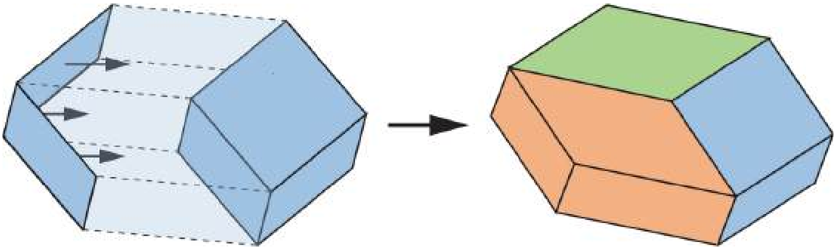


Fig. 3. Illustration of one-step tiling

The tiling algorithm leverages operations on boundary matrix $\boldsymbol{B}$ to implement the facets' movement and sub-zonotopes generation aforementioned. The results of each step, namely the sub-zonotopes after one-step tiling, are recorded in the tiling matrix $\boldsymbol{T}$.

Given an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$, where $\boldsymbol{G} = (\boldsymbol{g}_i)_{1 \leq i \leq p} \in \mathbb{R}^{n,p}$, we require that the right-most $n \times n$ submatrix of $\boldsymbol{G}$ is full rank to ensure that the sub-zonotopes with one generator removed after one-step tiling remain n-dimensional. For the specific j-th column of boundary matrix $\boldsymbol{B}$, where $1 \leq j \leq (p-n)$, we process the following operations to its entries:

1. if there exist i's such that $\boldsymbol{B}(i,j) = -1$, we add these rows in the boundary matrix $\boldsymbol{B}$ into the tiling matrix $\boldsymbol{T}$ as new rows, but change their j-th entry to 0 in the tiling matrix $\boldsymbol{T}$. Meanwhile, the j-th entries of these rows in the boundary matrix $\boldsymbol{B}$ are modified into 1, i.e., $\boldsymbol{B}(i,j) = 1$;

2. if there exist i's such that $\boldsymbol{B}(i,j)=0$, we delete these rows from the boundary matrix $\boldsymbol{B}$.

After the j-th iteration, the updated boundary matrix $\boldsymbol{B}$ characterizes the boundary of a new zonotope. This new zonotope is derived by removing the first through the j-th generators from the original zonotope Z. Simultaneously, the sub-zonotopes generated by adding the generator $\boldsymbol{g}_j$ to the facets are incorporated into the tiling matrix $\boldsymbol{T}$. Finally, after $p-n$ iterations, there remains one parallelotope, whose generator matrix is the right-most $n \times n$ submatrix of $\boldsymbol{G}$, we put this parallelotope into the tiling matrix $\boldsymbol{T}$ and then output the result.

The above computational procedures are summarized in Algorithm 2. Its soundness is justified by Theorem 2, whose proof is available in Appendix A. Moreover, Appendix B supplements an example (Example 3) to illustrate the main steps tiling a zonotope using Algorithm 2.

Remark 2. For an n-dimensional parallelotope, Algorithm 2 only return itself since there is no generator to remove while keeping it n-dimensional. However, one can use some simple methods to tile it such as parallelepiped grid.

Algorithm 2. Tiling Algorithm

Input: An n-dimensional zonotope $Z=\langle \boldsymbol{c},\boldsymbol{G}\rangle, \boldsymbol{G}=(\boldsymbol{g}_i)_{1\le i\le p}\in\mathbb{R}^{n,p}$, the right-most $n\times n$ submatrix of $\boldsymbol{G}$ is full rank, i.e., $\mathrm{rank}((\boldsymbol{g}_i)_{i\in\mathbb{N}_{[p-n+1,p]}})=n$.

Output: Tiling matrix $\boldsymbol{T}$.

```
Call Alg. 1 to get boundary matrix $\boldsymbol{B}$
$\boldsymbol{T} := [\,]$
for $j = 1$ to $p - n$ do
    for $i = 1$ to rows($\boldsymbol{B}$) do
        if $B(i,j) = 0$ then
            $\boldsymbol{B} := \boldsymbol{B}^{[i]}$
        end if
        if $B(i,j) = -1$ then
            $\boldsymbol{v}^\intercal := B(i,\cdot)$
            $\boldsymbol{v}(j) := 0$
            $\boldsymbol{T} := [\boldsymbol{T}; \boldsymbol{v}^\intercal]$
            $B(i,j) := 1$
        end if
    end for
end for
$\boldsymbol{v}^\intercal = \boldsymbol{B}(-1,\cdot)$
for $j = p - n + 1$ to $p$ do
    $\boldsymbol{v}(j) := 0$
end for
$\boldsymbol{T} := [\boldsymbol{T}; \boldsymbol{v}^\intercal]$
return $\boldsymbol{T}$
```

Remark 3. The sub-zonotopes obtained by Algorithm 2 aren't necessarily parallelotopes. To make the results of tiling are exclusively paralletopes, one can recursive applying Alg. 2 on each sub-zonotope in tiling matrix $\boldsymbol{T}$ until each sub-zonotope has n generators. Additionally, Algorithm 2 allows terminating at any iteration, and the result of each iteration can serve as a tiling of the original zonotope. This flexibility is valuable for controlling the number of partitioned sub-zonotopes. Therefore, our proposed tiling algorithm is particularly well-suited for the inner-approximation computation scenario outlined in this paper, it enables a balance between the computational burden and precision of evaluating $O(h; \partial U_k)$ by constraining the number of sub-zonotopes in the tiling.

Theorem 2 (Soundness of tiling algorithm). *Given an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle$ with p generators, the tiling matrix $\boldsymbol{T}$ obtained by Algorithm 2 satisfies the conditions in Def. 6.*

The Complexity of Tiling Algorithm. For an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle, \boldsymbol{G} = (\boldsymbol{g}_i)_{1 \le i \le p} \in \mathbb{R}^{n,p}$, where $\operatorname{rank}((\boldsymbol{g}_i)_{i \in \mathbb{N}_{[p-n+1,p]}}) = n$, assume Z has M facets. The calling of Algorithm 1 is $O\left(Mn(\mathrm{DET}(n-1) + p) + n(n-1)^2 \binom{p}{n-1}\right)$. The size of boundary matrix $\boldsymbol{B}$ is $M \times p$, the two-layer "for" Loop (Line 3–15) has iterations less than $M(p-n)$, thus the calling of Algorithm 1 is dominant in the complexity of tiling algorithm. Consequently, the complexity of Algorithm 2 is $O\left(Mn(\mathrm{DET}(n-1) + p) + n(n-1)^2 \binom{p}{n-1}\right)$.

Complexity Comparison. Here we compare the complexity of tiling algorithm proposed in [17] with ours. For an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle, \boldsymbol{G} \in \mathbb{R}^{n,p}$, assume Z has M facets and N vertexes. The main computation procure of algorithm in [17] is computing Σ (a set of sign vectors of cells of the arrangement), which is equivalent to enumerate the sign vectors of all the vertexes of Z. The computation of Σ, utilizing a reverse search algorithm [9], owns the complexity of $O(np\,\mathrm{LP}(p,n)|\Sigma|) = O(Nnp\,\mathrm{LP}(p,n))$ (the number of sign vectors in Σ is equal to N), where $\mathrm{LP}(p,n)$ is the time to solve a linear programming (LP) with p inequalities in n variables. There are various algorithms for solving LPs including simplex algorithm, interior point method and their variants. The state-of-the-art algorithms for solving LPs take complexity around $O(n^{2.37})$ [8]. As for the complexity of our algorithm, we have clarified that dominant part in complexity is the procure extracting the boundary of a zonotope, which has the complexity $O\left(Mn(\mathrm{DET}(n-1) + p) + n(n-1)^2 \binom{p}{n-1}\right)$. Additionally for zonotope Z, the number of its vertexes N is usually much larger than the one of its facets M, particularly in high dimension (for example, a hypercube in $\mathbb{R}^n$ has 2^n vertexes and $2n$ facets). According to the analysis above, we can conclude the complexity of our tiling algorithm is better than the one of algorithm ($O(Nnp\,\mathrm{LP}(p,n))$) in [17].

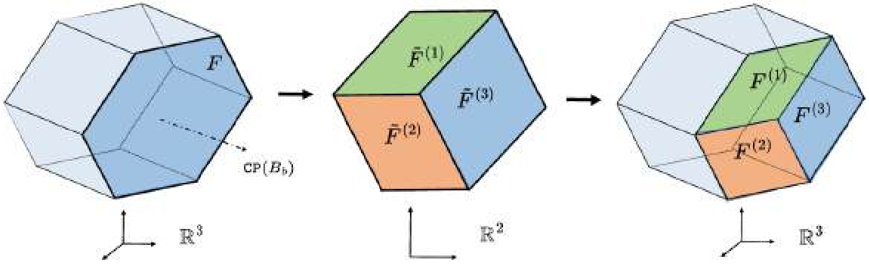


Fig. 4. Illustration of boundary refinement

Boundary Refinement via Tiling Algorithm. Given an n-dimensional zonotope $Z = \langle \boldsymbol{c}, \boldsymbol{G} \rangle, \boldsymbol{G} \in \mathbb{R}^{n,p}$, for one of its facet $F = \langle \boldsymbol{c}_b, \boldsymbol{G}_b \rangle$, we transform it into the space $\mathbb{R}^{n-1}$ with transformation matrix $\boldsymbol{B}_b^\intercal$, where $\boldsymbol{B}_b$ is the $n \times (n-1)$ submatrix of $\boldsymbol{G}_b$ with rank $n-1$, then the $(n-1)$-dimensional transformed zonotope can be denoted as $\tilde{F} = \langle \boldsymbol{B}_b^\intercal \boldsymbol{c}_b, \boldsymbol{B}_b^\intercal \boldsymbol{G}_b \rangle$. Using tiling algorithm, $\tilde{F}$ can be split into some smaller sub-zonotopes $\{\tilde{F}^{(1)}, \tilde{F}^{(2)}, \cdots, \tilde{F}^{(M)}\}$. For each of $(n-1)$-dimensional sub-zonotopes such as $\tilde{F}^{(1)} = \left\langle \tilde{\boldsymbol{c}}_{(1)}, \tilde{\boldsymbol{G}}_{(1)} \right\rangle$, an inverse transformation recovers it to the zonotope in the space $\mathbb{R}^n$, i.e., $F^{(1)} = \langle \boldsymbol{c}_{(1)}, \boldsymbol{G}_{(1)} \rangle$, where $\boldsymbol{c}_{(1)} = [\boldsymbol{B}_b^\intercal; \mathtt{CP}(\boldsymbol{B}_b)^\intercal]^{-1}[\tilde{\boldsymbol{c}}_{(1)}; \mathtt{CP}(\boldsymbol{B}_b) \cdot \boldsymbol{c}_b]$, $\boldsymbol{G}_{(1)} = [\boldsymbol{B}_b^\intercal; \mathtt{CP}(\boldsymbol{B}_b)^\intercal]^{-1}[\tilde{\boldsymbol{G}}_{(1)}; \boldsymbol{0}^\intercal]$. The main steps of boundary refinement are visualized in Fig. 4.

3.4 Contracting Computed Outer-Approximation

In this subsection we present our contraction method, yielding the inner approximation candidate U'_{k+1} by contracting $O(h; U_k)$. In contrast to the approaches in [33], which contracts $O(h; U_k)$ by reducing size proportionally, our contraction method offers a more flexible way. Specifically, the length of each generator of $O(h; U_k)$ can be adjusted and some generators can be removed. The incorporation of this adaptive contraction method enhances the tightness of the computed inner-approximation.

By extracting and refining of boundary ∂U_k of U_k, we get a collection of sub-zonotopes, i.e., $\{\partial U_k^{(i)}\}_{i \in \mathbb{N}_{[1,s]}}$, where $\bigcup_{i=1}^{s} \partial U_k^{(i)} = \partial U_k$. Then $O(h; \partial U_k)$ can be obtained by uniting all the out-approximations $\partial O_{k+1}^{(i)} := O(h; \partial U_k^{(i)}), i \in \mathbb{N}_{[1,s]}$, i.e., $O(h; \partial U_k) = \bigcup_{i=1}^{s} \partial O_{k+1}^{(i)}$.

Noticing that the shape of every outer-approximation $\partial O_{k+1}^{(i)} = \langle \boldsymbol{c}_o, \boldsymbol{G}_o \rangle$ is usually long and narrow (refer to Fig. 1), we choose the top $n-1$ independent generators by norm (such as Euclidean norm) to span a hyperplane, which can be seen as an $(n-1)$-dimensional form approximating $\partial O_{k+1}^{(i)}$. Then we compute the cross product $\mathtt{CP}(\cdot)$ of this hyperplane as its normal vector to represent the attitude of $\partial O_{k+1}^{(i)}$, denoted by $\mathtt{AT}\left(\partial O_{k+1}^{(i)}\right)$ (i.e. $\mathtt{CP}(\hat{\boldsymbol{G}}_o)$, where $\hat{\boldsymbol{G}}_o$ contains top $n-1$ indenpent generators of $\boldsymbol{G}_o$ by norm).

Initially, we set the inner-approximation candidate $U'_{k+1} := O(h; U_k)$. Subsequently, we iteratively reduce the length of generators and adjust the position (by changing the center) of U'_{k+1} until the intersections between U'_{k+1} and all outer-approximations $\partial O^{(i)}_{k+1}$ become empty sets. For each outer-approximation $\partial O^{(i)}_{k+1}$, we begin by shortening the length of generators that are most likely to yield collisions between U'_{k+1} and $\partial O^{(i)}_{k+1}$, which would prevent the unnecessary contraction of U'_{k+1} and make the result tighter. Heuristically, the generators with directions closest to $\texttt{AT}\left(\partial O^{(i)}_{k+1}\right)$, or in other words, those most likely to "perpendicular" to $\partial O^{(i)}_{k+1}$ (precisely, perpendicular to hyperplane spanned by column vectors of $\hat{\boldsymbol{G}}_o$) should be given priority considerations. When encountering a generator that dose not need to be shortened, indicating that U'_{k+1} and $\partial O^{(i)}_{k+1}$ have no overlapping parts, we turn to the next outer-approximation $\partial O^{(i+1)}_{k+1}$. The details of contraction method proposed is summarized below.

1. Initialize inner-approximation candidate $U'_{k+1} := \langle \boldsymbol{c}_u, \boldsymbol{G}_u \rangle = O(h; U_k)$.
2. For every boundary outer-approximations $\partial O^{(i)}_{k+1} = \langle \boldsymbol{c}_o, \boldsymbol{G}_o \rangle, i \in \mathbb{N}_{[1,s]}$, carry out the following processing steps.
   2a. Sort the generators $\{\boldsymbol{g}_l\}_{1 \le l \le \mathrm{cols}(\boldsymbol{G}_u)}$ of U'_{k+1} according the angle with $\texttt{AT}\left(\partial O^{(i)}_{k+1}\right)$ from small to large (i.e., $\|\mathrm{cos}\theta\| = \frac{\|\boldsymbol{g}_l \cdot \texttt{AT}\left(\partial O^{(i)}_{k+1}\right)\|}{\|\boldsymbol{g}_l\| \|\texttt{AT}\left(\partial O^{(i)}_{k+1}\right)\|}$ from large to small).
   2b. Loop all the generators according to the sorted order, for the generator $\boldsymbol{g}_l$, compute its domain $[\underline{\alpha_l}, \overline{\alpha_l}]$ which intersects with $\partial O^{(i)}_{k+1}$ by LPs (2) and (3) (using approach in [16, Chapter 4.2.5]), where $\boldsymbol{\alpha} = (\alpha_1, \cdots, \alpha_l, \cdots, \alpha_{\mathrm{cols}(\boldsymbol{G}_u)})^\intercal$, $\boldsymbol{\beta} = (\beta_1, \cdots, \beta_{\mathrm{cols}(\boldsymbol{G}_o)})^\intercal$.

$$\begin{aligned} \min \quad & \alpha_l \\ s.t. \quad & \boldsymbol{c}_u + \boldsymbol{G}_u \boldsymbol{\alpha} = \boldsymbol{c}_o + \boldsymbol{G}_o \boldsymbol{\beta} \\ & -\boldsymbol{1} \le \boldsymbol{\alpha} \le \boldsymbol{1}, -\boldsymbol{1} \le \boldsymbol{\beta} \le \boldsymbol{1} \end{aligned} \tag{2}$$

$$\begin{aligned} \max \quad & \alpha_l \\ s.t. \quad & \boldsymbol{c}_u + \boldsymbol{G}_u \boldsymbol{\alpha} = \boldsymbol{c}_o + \boldsymbol{G}_o \boldsymbol{\beta} \\ & -\boldsymbol{1} \le \boldsymbol{\alpha} \le \boldsymbol{1}, -\boldsymbol{1} \le \boldsymbol{\beta} \le \boldsymbol{1} \end{aligned} \tag{3}$$

   When the optimal value of (2) or (3) can't be found, then terminate this loop and continue for the next boundary outer-approximation $\partial O^{(i+1)}_{k+1}$.
   2c. If $[\underline{\alpha_l}, \overline{\alpha_l}] = [-1, 1]$, then delete $\boldsymbol{g}_l$ from generator matrix $\boldsymbol{G}_u$. Else, update the range of $a_l \in \max\{[-1, \underline{a_l} - \epsilon], [\overline{a_l} + \epsilon, 1]\} \triangleq [\underline{\gamma}, \overline{\gamma}]$, where the operation $\max\{\cdot, \cdot\}$ means choosing the interval with maximum length and ϵ is a user-defined small positive number.
   2d. Update $\boldsymbol{c}_u := \boldsymbol{c}_u + 0.5(\overline{\gamma} + \underline{\gamma})\boldsymbol{g}_l$ and $\boldsymbol{g}_l := 0.5(\overline{\gamma} - \underline{\gamma})\boldsymbol{g}_l$.

Remark 4. The introducing of the user-defined small positive number ϵ is to ensure $U'_{k+1} \cap \partial O^{(i)}_{k+1} = \emptyset$.

Remark 5. In practice, it is a common case that $[\underline{\alpha_l}, \overline{\alpha_l}] = [-1, 1]$, thus the number of generators of inner-approximation candidate U'_{k+1} is usually less than $O(h; U_k)$'s, which shows that this contraction method has the advantage for zonotope order reduction [35].

Appendix B provides an example (Example 4) to illustrate the procedure of the contraction method and why is necessary to sort the generators $\{\boldsymbol{g}_l\}_{1 \le l \le \text{cols}(\boldsymbol{G}_u)}$ of U'_{k+1} according to the angle with $\texttt{AT}\left(\partial O^{(i)}_{k+1}\right)$.

Verification of Inner-Approximation Candidate. According to Theorem 1 and 3 in [18], after obtaining inner-approximation candidate U'_{k+1}, it's crucial to check whether the outer-approximation $O(h; \boldsymbol{c})$ ($\boldsymbol{c}$ is the center of U'_{k+1}) of the time-inverted system $\dot{\boldsymbol{x}} = -\boldsymbol{f}(\boldsymbol{x})$ is within U_k, which confirms the correctness of computed inner-approximation U_{k+1}. Since both U'_{k+1} and $O(h; \boldsymbol{c})$ are zonotopes, this verification reduces a zonotope containment problem. In our approach, we leverage a sufficient condition outlined in [29], which can be encoded into LP to perform the inclusion verification.

4 Experiments

In this section we demonstrate the performance of our approach on various benchmarks. Our implementation utilizes the floating point linear programming solver GLPK and C++ library Eigen. We adopt the approach outlined in [3] to compute outer-approximations appeared in our method. All experiments herein are run on Ubuntu 20.04.3 LTS in virtual machine with CPU 12th Gen Intel Core i9-12900K × 8 and RAM 15.6 GB.

To evaluate the precision of the computed inner-approximations, we use the minimum width ration γ_{min} similar to [18], which is defined as

$$\begin{aligned} \gamma_{\min} &= \min_{\boldsymbol{v} \in \mathcal{V}} \frac{|\gamma_i(\boldsymbol{v})|}{|\gamma_o(\boldsymbol{v})|} \\ \text{with} \quad \gamma_i(\boldsymbol{v}) &= \max_{x \in U_k} \boldsymbol{v}^\intercal x + \max_{x \in U_k} -\boldsymbol{v}^\intercal x \\ \gamma_o(\boldsymbol{v}) &= \max_{x \in O_k} \boldsymbol{v}^\intercal x + \max_{x \in O_k} -\boldsymbol{v}^\intercal x \end{aligned} \tag{4}$$

where U_k and O_k are the inner-approximation and outer-approximation of the reachable set at k step respectively. $\boldsymbol{v} \in \mathcal{V} \subset \mathbb{R}^n$, and $\mathcal{V}$ is the set consisting of n axis-aligned unit-vectors. To ensure a fair comparison, the O_k is chosen to be the interval enclosure of 1000 random points at the final time instant simulated via ode45 in MATLAB. Intuitively, the larger this ratio, the better the approximation.

Our approach is systematically compared with the state-of-the-art method presented in [18], which is publicly available in the reachability analysis toolbox CORA [1]. Benchmarks with system's dimension from 2 to 12 are utilized to show the the comprehensive advantages of our approach. Their configurations including dimensions, initial sets and references are listed in Table 2.

Table 2. Benchmarks and their dimensions, initial sets and references

Dim	Benchmark	Initial Set	Reference
2	ElectroOsc	$\boldsymbol{c}_1 + [-0.1, 0.1]^2$	Example 3 in [33]
3	Rossler	$\boldsymbol{c}_2 + [-0.15, 0.15]^3$	Example 3.4.3 in [7]
4	Lotka-Volterra	$\boldsymbol{c}_3 + [-0.2, 0.2]^4$	Example 5.2.3 in [7]
6	Tank6	$\boldsymbol{c}_4 + [-0.2, 0.2]^6$	[3]
7	BiologicalSystemI	$\boldsymbol{c}_5 + [-0.01, 0.01]^7$	Example 5.2.4 in [7]
9	BiologicalSystemII	$\boldsymbol{c}_6 + [-0.01, 0.01]^9$	Example 5.2.4 in [7]
12	Tank12	$\boldsymbol{c}_7 + [-0.2, 0.2]^{12}$	[3]

Note: for the parameters of Tank6 and Tank12, all A_i are set to $A_i = 1$, and all k_i are set to $k_i = 0.015$, $\kappa = 0.01, v = 0, g = 9.81$, for Tank6 $n = 6$ and for Tank12 $n = 12$; for the centers of initial sets, $\boldsymbol{c}_1 = (0, 3)^\top$, $\boldsymbol{c}_2 = (0.05, -8.35, 0.05)^\top$, $\boldsymbol{c}_3 = (0.6, 0.6, 0.6, 0.6)^\top$, $\boldsymbol{c}_4 = (2, 4, 4, 2, 10, 4)^\top$, $\boldsymbol{c}_5 = (0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1)^\top$, $\boldsymbol{c}_6 = (1, 1, 1, 1, 1, 1, 1, 1, 1)^\top$, $\boldsymbol{c}_7 = (2, 4, 4, 2, 10, 4, 2, 2, 2, 2, 2, 2)^\top$.

4.1 Advantage in Efficiency and Precision

For each benchmark stated in Table 2, we compute the inner-approximations at the time instant T using our approach and the one in CORA. Table 3 demonstrates the time cost and γ_{min} for tow methods. The advantages of our approach are evident from low dimensional scenario (2-dimensional) to high dimensional scenario (12-dimensional), showcasing improved efficiency and precision, particularly in higher dimensions. Taking the benchmark Tank12 as an instance, our approach achieves nearly 38% improvement in precision while requiring only 12% of the time compared to CORA. The visualization of the inner-approximations computed by our approach and CORA is illustrated in Fig. 7 provided in Appendix C, together with the outer-approximations computed by CORA in this figure for sake of convenient comparison.

Table 3. Comparison between our approach and CORA for each benchmark

Dim	Benchmark	T	Our Approach		CORA	
			time (s)	γ_{min}	time (s)	γ_{min}
2	ElectroOsc	2.5	**23.56**	**0.88**	36.50	0.57
3	Rossler	1.5	**27.72**	0.76	36.63	**0.78**
4	Lotka-Volterra	1	**10.43**	**0.65**	335.06	0.34
6	Tank6	80	**50.83**	**0.82**	201.05	0.63
7	BiologicalSystemI	0.2	**1.74**	**0.96**	125.73	0.90
9	BiologicalSystemII	0.2	**72.47**	**0.95**	188.25	0.88
12	Tank12	60	**235.88**	**0.77**	1834.65	0.56

4.2 Advantage in Long Time Horizons

Further, we extend the time horizon in Table 3 and compare the performance of inner-approximation computation between our approach and CORA. As evidenced by the results in Table 4, our approach demonstrates the reliable capability to compute inner-approximations in relatively longer time horizons compared to CORA. It shows that our approach can consistently compute all inner-approximations while maintaining benign efficiency and precision. In contrast, the approach in CORA fails to compute inner-approximations for all benchmarks. The visualization of the inner-approximations computed by our approach and CORA is illustrated in Fig. 8 provided in Appendix C.

Table 4. Comparison between our approach and CORA for each benchmark in relatively longer time horizons

Dim	Benchmark	T	Our Approach		CORA	
			time (s)	γ_{min}	time (s)	γ_{min}
2	ElectroOsc	3	73.76	0.90	–	–
3	Rossler	2.5	63.42	0.60	–	–
4	Lotka-Volterra	1.5	81.81	0.62	–	–
6	Tank6	120	129.58	0.65	–	–
7	BiologicalSystemI	1.3	462.87	0.41	–	–
9	BiologicalSystemII	0.375	261.78	0.66	–	–
12	Tank12	100	377.85	0.49	–	–

Note: the symbol "–" means that in this experimental configuration CORA cannot compute inner-approximations.

4.3 Advantage in Big Initial Sets

We also expand the initial sets as listed in Table 2 to highlight our advantage in computing inner-approximations from larger initial sets. For each benchmark, we set both a short and a long time instant to compute inner-approximations using our approach and CORA. As shown in Table 5, our approach can accomplish all the inner-approximation computations while maintaining high levels of efficiency and precision. In contrast, for the short time instant scenario, the performance of CORA is worse than ours in both computation time and accuracy, and CORA fails to compute inner-approximations at long time instant for all benchmarks. The visualization of the inner-approximations computed by our approach and CORA is illustrated in Fig. 9 and Fig. 10 provided in Appendix C.

Table 5. Comparison between our approach and CORA for each benchmark in big initial sets.

Dim	Benchmark	Initial Set	T	Our Approach		CORA	
				time (s)	γ_{min}	time (s)	γ_{min}
2	ElectroOsc	$\boldsymbol{c}_1 + 0.5\boldsymbol{I}^2$	1.5	**4.79**	**0.92**	24.32	0.43
			2	11.29	0.84	—	—
3	Rossler	$\boldsymbol{c}_2 + 0.5\boldsymbol{I}^3$	1	**15.88**	**0.59**	36.54	0.53
			1.5	24.01	0.58	—	—
4	Lotka-Volterra	$\boldsymbol{c}_3 + 0.5\boldsymbol{I}^4$	0.4	**15.45**	**0.71**	153.57	0.38
			1	64.17	0.52	—	—
6	Tank6	$\boldsymbol{c}_4 + 0.5\boldsymbol{I}^6$	80	**66.83**	**0.60**	463.28	0.13
			100	80.91	0.53	—	—
7	BiologicalSystemI	$\boldsymbol{c}_5 + 0.05\boldsymbol{I}^7$	0.5	**118.65**	**0.62**	615.89	0.38
			0.7	281.05	0.41	—	—
9	BiologicalSystemII	$\boldsymbol{c}_6 + 0.05\boldsymbol{I}^9$	0.26	**142.17**	**0.65**	1494.38	0.32
			0.28	142.02	0.55	—	—
12	Tank12	$\boldsymbol{c}_7 + 0.5\boldsymbol{I}^{12}$	40	**162.46**	**0.73**	1693.68	0.41
			60	235.57	0.54	—	—

Note: the symbol "—" means that in this experimental configuration CORA cannot compute inner-approximations. $\boldsymbol{I}^d$ denotes the box $[-1, 1]^d$.

5 Conclusion

In this paper we propose a novel approach to compute inner-approximations of reachable sets for nonlinear systems based on zonotopic boundary analysis. To enhance the efficiency and precision of the computed inner-approximations, we introduce three innovative and efficient methods, including the algorithm of extracting boundaries of zonotopes, the algorithm of tiling zonotopes for boundary refinement, and contraction strategy for obtaining inner-approximations from pre-computed outer-approximations. In comparison to the state-of-the-art methods for inner-approximation computation, our approach demonstrates superior performance in terms of efficiency and precision, particularly within high dimensional cases. Moreover, our proposed approach exhibits a remarkable capability to compute inner-approximations for scenarios with long time horizons and large initial sets, where the inner-approximations are usually failed to be computed by existing methods.

Acknowledgement. This work is funded by the CAS Pioneer Hundred Talents Program, Basic Research Program of Institute of Software, CAS (Grant No. ISCAS-JCMS-202302) and National Key R&D Program of China (Grant No. 2022YFA1005101).

References

1. Althoff, M.: An introduction to cora 2015. In: Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems, pp. 120–151 (2015)
2. Althoff, M., Frehse, G., Girard, A.: Set propagation techniques for reachability analysis. Annual Rev. Control, Robot. Autonomous Syst. **4**, 369–395 (2021)
3. Althoff, M., Stursberg, O., Buss, M.: Reachability analysis of nonlinear systems with uncertain parameters using conservative linearization. In: 2008 47th IEEE Conference on Decision and Control, pp. 4042–4048. IEEE (2008)
4. Bajaj, C.L., Pascucci, V.: Splitting a complex of convex polytopes in any dimension. In: Proceedings of the Twelfth Annual Symposium on Computational Geometry, pp. 88–97 (1996)
5. Björner, A.: Oriented matroids. No. 46, Cambridge University Press (1999)
6. Branicky, M.S.: Multiple lyapunov functions and other analysis tools for switched and hybrid systems. IEEE Trans. Autom. Control **43**(4), 475–482 (1998)
7. Chen, X.: Reachability analysis of non-linear hybrid systems using taylor models. Ph.D. thesis, Fachgruppe Informatik, RWTH Aachen University (2015)
8. Cohen, M.B., Lee, Y.T., Song, Z.: Solving linear programs in the current matrix multiplication time. J. ACM (JACM) **68**(1), 1–39 (2021)
9. Ferrez, J.A., Fukuda, K., Liebling, T.M.: Solving the fixed rank convex quadratic maximization in binary variables by a parallel zonotope construction algorithm. Eur. J. Oper. Res. **166**(1), 35–50 (2005)
10. Fisikopoulos, V., Penaranda, L.: Faster geometric algorithms via dynamic determinant computation. Comput. Geom. **54**, 1–16 (2016)
11. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31954-2_19
12. Goubault, E., Putot, S.: Forward inner-approximated reachability of non-linear continuous systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, pp. 1–10 (2017)
13. Goubault, E., Putot, S.: Inner and outer reachability for the verification of control systems. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 11–22 (2019)
14. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, pp. 373–382 (1995)
15. Herrmann, S., Joswig, M.: Splitting polytopes. arXiv preprint arXiv:0805.0774 (2008)
16. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Interval analysis. In: Applied interval analysis, pp. 11–43. Springer (2001). https://doi.org/10.1007/978-1-4471-0249-6
17. Kabi, B.: Synthesizing invariants: a constraint programming approach based on zonotopic abstraction. Ph.D. thesis, Institut polytechnique de Paris (2020)
18. Kochdumper, N., Althoff, M.: Computing non-convex inner-approximations of reachable sets for nonlinear continuous systems. In: 2020 59th IEEE Conference on Decision and Control (CDC), pp. 2130–2137. IEEE (2020)
19. Korda, M., Henrion, D., Jones, C.N.: Inner approximations of the region of attraction for polynomial dynamical systems. IFAC Proc. Vol. **46**(23), 534–539 (2013)
20. Lakshmikantham, V., Leela, S., Martynyuk, A.A.: Practical stability of nonlinear systems. World Scientific (1990)

21. 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
22. McMullen, P.: On zonotopes. Trans. Am. Math. Soc. **159**, 91–109 (1971)
23. Mitchell, I.M.: Comparing forward and backward reachability as tools for safety analysis. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 428–443. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71493-4_34
24. Mortari, D.: n-dimensional cross product and its application to the matrix eigenanalysis. J. Guid. Control. Dyn. **20**(3), 509–515 (1997)
25. Neumaier, A.: The wrapping effect, ellipsoid arithmetic, stability and confidence regions. Springer (1993). https://doi.org/10.1007/978-3-7091-6918-6_14
26. Ren, D., Liang, Z., Wu, C., Ding, J., Wu, T., Xue, B.: Inner-approximate reachability computation via zonotopic boundary analysis. arXiv preprint arXiv:2405.11155 (2024)
27. Richter-Gebert, J., Ziegler, G.M.: Zonotopal tilings and the bohne-dress theorem. Contemp. Math. **178**, 211–211 (1994)
28. Rwth, X.C., Sankaranarayanan, S., Ábrahám, E.: Under-approximate flowpipes for non-linear continuous systems. In: 2014 Formal Methods in Computer-Aided Design (FMCAD), pp. 59–66. IEEE (2014)
29. Sadraddini, S., Tedrake, R.: Linear encodings for polytope containment problems. In: 2019 IEEE 58th Conference on Decision and Control (CDC), pp. 4367–4372. IEEE (2019)
30. Schoels, T., Palmieri, L., Arras, K.O., Diehl, M.: An nmpc approach using convex inner approximations for online motion planning with guaranteed collision avoidance. In: 2020 IEEE International Conference on Robotics and Automation (ICRA), pp. 3574–3580. IEEE (2020)
31. Wan, J., Vehi, J., Luo, N.: A numerical approach to design control invariant sets for constrained nonlinear discrete-time systems with guaranteed optimality. J. Global Optim. **44**, 395–407 (2009)
32. Xue, B., Fränzle, M., Zhan, N.: Inner-approximating reachable sets for polynomial systems with time-varying uncertainties. IEEE Trans. Autom. Control **65**(4), 1468–1483 (2019)
33. Xue, B., She, Z., Easwaran, A.: Under-approximating backward reachable sets by polytopes. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 457–476. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_25
34. Xue, B., Zhan, N., Fränzle, M., Wang, J., Liu, W.: Reach-avoid verification based on convex optimization. IEEE Trans. Autom. Control **69**(1), 598–605 (2024)
35. Yang, X., Scott, J.K.: A comparison of zonotope order reduction techniques. Automatica **95**, 378–384 (2018)
36. Ziegler, G.M.: Lectures on polytopes, vol. 152. Springer Science & Business Media (2012)

Scenario-Based Flexible Modeling and Scalable Falsification for Reconfigurable CPSs

Jiawan Wang, Wenxia Liu, Muzimiao Zhang, Jiaqi Wei, Yuhui Shi, Lei Bu(✉), and Xuandong Li

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
wangjw@smail.nju.edu.cn, bulei@nju.edu.cn

Abstract. Cyber-physical systems (CPSs) are used in many safety-critical areas, making it crucial to ensure their safety. However, with CPSs increasingly dynamically deployed and reconfigured during runtime, their safety analysis becomes challenging. For one thing, reconfigurable CPSs usually consist of multiple agents dynamically connected during runtime. Their highly dynamic system topologies are too intricate for traditional modeling languages, which, in turn, hinders formal analysis. For another, due to the growing size and uncertainty of reconfigurable CPSs, their system models can be huge and even unavailable at design time. This calls for runtime analysis approaches with better scalability and efficiency. To address these challenges, we propose a scenario-based hierarchical modeling language for reconfigurable CPS. It provides template models for agent inherent features, together with an instantiation mechanism to activate single agent's runtime behavior, communication configurations for multiple agents' connected behaviors, and scenario task configurations for their dynamic topologies. We also present a path-oriented falsification approach to falsify system requirements. It employs classification-model-based optimization to explore search space effectively and cut unnecessary system simulations and robustness calculations for efficiency. Our modeling and falsification are implemented in a tool called `SNIFF`. Experiments have shown that it can largely reduce modeling time and improve modeling accuracy, and perform scalable CPS falsification with high success rates in seconds.

1 Introduction

A cyber-physical system (CPS) consists of multiple computing devices that communicate with each other and interact with the physical world in a feedback loop.

The authors are supported in part by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001), the National Natural Science Foundation of China (No. 62232008, 62172200), the Fundamental Research Funds for the Central Universities (No. 2023300180), and the Program A for Outstanding PhD Candidates of Nanjing University.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 329–355, 2024.
https://doi.org/10.1007/978-3-031-65633-0_15

It is vital to ensure that CPSs work correctly as intended, especially in safety-critical applications like aircrafts, automobiles, and medical devices [4,41,42].

Model checking is a powerful, computer-assisted technique to analyze CPS correctness [11,19]. It involves formalizing system behavior in mathematical models, describing system requirements in logic specifications, and analyzing whether specifications are satisfied in models through complete verification algorithms or incomplete falsification algorithms. Verification algorithms prove system correctness automatically but are typically limited to CPS with few dimensions and simple dynamics [16,18,30,35,36,54,55]. Falsification algorithms, on the other hand, satisfy weaker forms of completeness but can handle complex system dynamics and disprove system correctness via counterexamples [3,8,27,33,53,63,66]. As a result, they are of practical interest and are the main focus of model checking in industrial applications [19,20,53].

However, with real-world CPSs increasingly deployed in uncertain environments and dynamically reconfigured during runtime [1,2,23,32,37,59], their formal modeling and falsification face great challenges.

Challenge 1: CPS modeling is complicated due to changing agent behaviors and dynamic system topologies. For one thing, in a reconfigurable CPS, each agent's role, function, and behavior may vary across scenarios to meet changing requirements. Therefore, it needs to be frequently modeled, especially when there are many different scenarios. For another, agents are usually dynamically wired due to changing demands or limitations. Some agents may even be part of the system in one scenario but leave in another. Traditional modeling languages, however, lack flexible mechanisms to formalize dynamic topology shifts.

Challenge 2: CPS falsification requires better scalability and efficiency due to the growing size and complexity of reconfigurable CPSs. Models of reconfigurable CPSs become huge and may even unavailable at design time. This calls for more scalable falsification approaches with improved runtime performance.

For challenge 1, we designed a hierarchical scenario-based modeling language for reconfigurable CPSs. To simplify agent modeling, we introduced a template model to formalize agent inherent features, including potential dynamics and communication capabilities. Based on template models, an instantiation mechanism is provided so that users can model and update agent runtime behaviors easily. Further, to capture the dynamically wired behaviors of multiple agents, we introduced communication configurations and scenario task configurations for instantiated agent models. Users can formalize agents' intricately connected behaviors through brief communication configurations, and abstract their dynamic topologies flexibly through intuitive scenario task configurations.

For challenge 2, we developed a path-oriented falsification approach for reconfigurable CPSs, searching for counterexample system behavior along generated paths. For the vast behavior space along each path, we designed a two-layered, classification-model-based optimization method to explore it effectively. Besides, since system simulation and robustness calculation are most time-consuming during falsification, we introduced concepts of time-context and lifespan for temporal specifications, to cut unnecessary system simulations and robustness calculations that do not affect the satisfaction of specifications.

The main contributions of this work are as follows:

- We designed a scenario-based hierarchical modeling language to formalize reconfigurable CPSs. We also extended a topology-aware temporal logic to specify their system requirements. (See Sect.3)
- We developed a path-oriented approach to falsify temporal logic specifications for CPSs. It employs classification-model-based optimization and lifespan-guided robustness calculation to improve efficiency. (See Sect.4)
- We implemented a tool called `SNIFF`, which supports scenario-based graphical modeling and falsification for reconfigurable CPSs. Studies show that it can greatly reduce modeling time and improve modeling accuracy. Experiments demonstrated its scalability and efficiency in falsification. (See Sect.5)

2 Background

2.1 Preliminaries

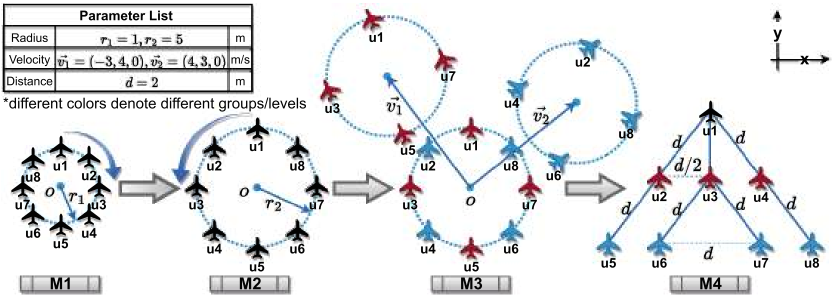
Hybrid Automaton (HA) is a popular formal language for modeling tightly coupled discrete and continuous CPS behaviors. Dense-time formalisms like Signal Temporal Logic (STL) serve as the basis for specifying CPS requirements.

Definition 1 *(***Hybrid Automaton** *[5]).* *An HA is a tuple $H = (X, U, Q, F, Inv, E, G, R)$, where X are continuous variables; U are external inputs*[1]*; Q are discrete operating modes; $F = \{F_q |\ q \in Q\}$ are flow functions, defining the flow of variables in mode q by differential equation $\dot{X} = F_q(X, U)$; $Inv = \{Inv_q |\ q \in Q\}$ are invariant conditions, constraining conditions of mode q by constraints $Inv_q(X, U)$; $E \subseteq Q \times Q$ are discrete transitions, denoting jumps of modes; $G = \{G_e |\ e \in E\}$ are guard conditions on transitions; $R = \{R_e |\ e \in E\}$ are reset functions on transitions, resetting variables by $X := R_e(X, U)$.*

Semantics: The state space of H is $S = Q \times X$. Its state, denoted by the pair $s = (q, x) \in S$, evolves in the following two ways. It can stay in the current mode q and evolve continuously based on the flow function F_q, within the space constrained by the invariant condition Inv_q. It can also jump from the mode q to q' discretely and instantaneously through transition $e = (q, q') \in E$ when its guard condition G_e is met, and be reset by the function R_e.

Note that under nondeterministic semantics, HA can choose arbitrarily between staying in the current mode and jumping to another mode, as long as conditions of current mode and target transition are both satisfied. It can also choose among multiple transitions when their guard conditions are all met [44].

[1] External inputs are often assumed to be time-varying, smooth, and parameterized.



Parameter List		
Radius	$r_1 = 1, r_2 = 5$	m
Velocity	$\vec{v_1} = (-3, 4, 0), \vec{v_2} = (4, 3, 0)$	m/s
Distance	$d = 2$	m

*different colors denote different groups/levels

Fig. 1. The Formation Target of the Multi-UAV System in Each Flight Mission

Definition 2 *(**Signal Temporal Logic** [46,47]). Let $\mathcal{X}$ denotes variables defined over domain $\overline{\mathbb{R}}^n$, where $\overline{\mathbb{R}}$=$\mathbb{R} \cup \{\top, \bot\}$ is the totally ordered set of real numbers with the smallest and greatest boolean elements $\bot$ and $\top$, $\top = -\bot$, $-\top = \bot$. Let $w : \mathbb{T} \to \overline{\mathbb{R}}^n$ denotes a multi-dimensional signal of $\mathcal{X}$, with $\mathbb{T} = [0, t) \subseteq \mathbb{R}$. The syntax of an STL formula φ interpreted over $\mathcal{X}$ is defined as:*

$\varphi := true \mid \theta(\mathcal{X}) \geq d \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U_I} \varphi_2.$

Here, $\theta : \overline{\mathbb{R}}^n \to \mathbb{R}$ denotes a function that maps $\mathcal{X}$'s valuations into a real, and d is a constant. $\mathbf{U}$ denotes the until operator and $\mathbf{I} \subseteq \mathbb{R}^+$ denotes a timing interval, which can be omitted when it is $[0, \infty)$. In a formula φ, other standard operators can be derived as follows: false $\equiv \neg$true, $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \to \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$, $\Diamond_I \varphi \equiv true\mathbf{U_I}\varphi$, and $\Box_I \varphi \equiv \neg\Diamond_I(\neg\varphi)$.

Semantics: For a given signal, an STL formula φ has both boolean and quantitative semantics [29,46]. Its boolean semantics indicates φ's validity on this signal at some given time t via a binary satisfaction relation, while its quantitative semantics indicates how well this signal satisfies φ at time t via a real-valued robustness degree. Due to space limitations, we only provide quantitative semantics here. Given an STL formula φ, a signal w of variables $\mathcal{X}$, and time t, the quantitative semantics of its robustness degree $\rho(\varphi, w, t)$ is defined by induction:

$$\begin{aligned} \rho(true, w, t) &= \top \\ \rho(\theta(\mathcal{X}) \geq d, w, t) &= \theta(w(t)) - d \\ \rho(\neg\varphi_1, w, t) &= -\rho(\varphi_1, w, t) \\ \rho(\varphi_1 \vee \varphi_2, w, t) &= \max\{\rho(\varphi_1, w, t), \rho(\varphi_2, w, t)\} \\ \rho(\varphi_1 \mathbf{U_I} \varphi_2, w, t) &= \sup_{t' \in t \oplus \mathbf{I}} \min\{\rho(\varphi_2, w, t'), \inf_{t'' \in [t, t')} \rho(\varphi_1, w, t'')\} \end{aligned} \tag{1}$$

Here, $\oplus$ denotes the Minkowski sum, $t \oplus \mathbf{I} = \{t + i | i \in \mathbf{I}\}$. This quantitative semantics is used in most STL falsification. The degree is positive only if the signal satisfies the formula, and the higher the better.

2.2 Motivating Example: A Multi-UAV System

Consider a reconfigurable CPS with eight UAV (Unmanned Aerial Vehicle) agents (u_1-u_8). Each agent's position, speed, and acceleration are continuous variables, with communication delay as an external input. The system undertakes various flight missions based on runtime demands. To execute missions, agents collaborate through different communication strategies and composition architectures (e.g., classical virtual structures, leader-follower, behavior-based, and consensus-based ones [26]). Below is a example of its runtime mission sequence (M1-M4), with formation details in Fig. 1.

M1 All UAVs communicate with each other to locate their 3-D central point O. They then calculate and move to their target positions, forming a clockwise ring around the center O with a radius of r_1, at the same height of O.

M2 It is similar to M1, but forms a counter-clockwise ring with a radius of r_2.

M3 Upon receiving commands from the ground control center, odd-numbered and even-numbered UAVs adjust their speed to v_1 and v_2, respectively.

M4 It employs a hierarchical leader-follower architecture for tree formation. Phase 1: according to the leader u_1, followers u_{2-4} change their heading angles and move to desired relative positions. Phase 2: u_{2-4} act as leaders for u_5, u_{6-7}, and u_8 respectively, forming the third row of the tree similarly.

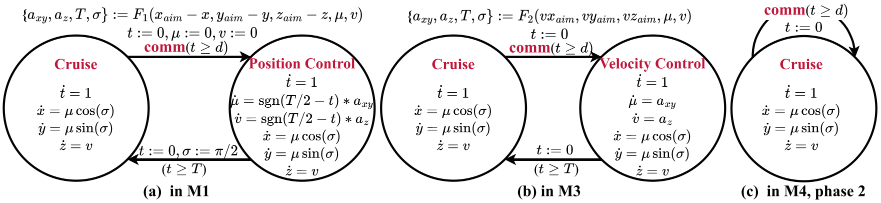

$\{a_{xy}, a_z, T, \sigma\} := F_1(x_{aim} - x, y_{aim} - y, z_{aim} - z, \mu, v)$
$t := 0, \mu := 0, v := 0$
comm$(t \geq d)$
Cruise
$\dot{t} = 1$
$\dot{x} = \mu\cos(\sigma)$
$\dot{y} = \mu\sin(\sigma)$
$\dot{z} = v$
Position Control
$\dot{t} = 1$
$\dot{\mu} = \text{sgn}(T/2 - t) * a_{xy}$
$\dot{v} = \text{sgn}(T/2 - t) * a_z$
$\dot{x} = \mu\cos(\sigma)$
$\dot{y} = \mu\sin(\sigma)$
$\dot{z} = v$
$t := 0, \sigma := \pi/2$
$(t \geq T)$
(a) in M1
$\{a_{xy}, a_z, T, \sigma\} := F_2(vx_{aim}, vy_{aim}, vz_{aim}, \mu, v)$
$t := 0$
comm$(t \geq d)$
Cruise
$\dot{t} = 1$
$\dot{x} = \mu\cos(\sigma)$
$\dot{y} = \mu\sin(\sigma)$
$\dot{z} = v$
Velocity Control
$\dot{t} = 1$
$\dot{\mu} = a_{xy}$
$\dot{v} = a_z$
$\dot{x} = \mu\cos(\sigma)$
$\dot{y} = \mu\sin(\sigma)$
$\dot{z} = v$
$t := 0$
$(t \geq T)$
(b) in M3
comm$(t \geq d)$
$t := 0$
Cruise
$\dot{t} = 1$
$\dot{x} = \mu\cos(\sigma)$
$\dot{y} = \mu\sin(\sigma)$
$\dot{z} = v$
(c) in M4, phase 2

Fig. 2. HA Models of u_3 in Scenarios with Differently Activated Functions

Although all agents are homogeneous in this case, each plays a different role. Moreover, even the same agent behaves differently across scenarios, and therefore, has different models. For example, in the models of u_3 shown in Fig. 2(a)-(c),

- in M1, it cruises while waiting for others' states. Once received, it calculates control parameters to adjust its position to the target location accordingly.
- in M3, it cruises while waiting for ground commands, and then adjusts velocity.
- in the phase 2 of M4, it cruises while broadcasting its state to followers, u_{6-7}.

These functions are embedded in UAV codes, activated dynamically at runtime.

Furthermore, the topology of the entire multi-UAV system changes dynamically, resulting in a complex and vast system model. In M1 and M2, vehicle-to-vehicle communication exists among all agents for central point calculation.

While in M3, agents are divided into two groups, equipped with ground-to-vehicle communication. In M4, system topology even shifts inside the mission and becomes hierarchical. Across scenarios, system models should be structured differently, and their shifts should also be formally modeled as a whole. Besides, specifying system requirements, especially topology-related ones, is also difficult. Many extra intermediate variables are required to describe system topology. We will address these modeling and specifying challenges in the next section.

3 Scenario-Based Formalism for Reconfigurable Systems

In this section, we present our formalism for reconfigurable CPSs with dynamic behaviors, compositions, and system topologies. It includes scenario-based system behavior modeling and topology-aware system property specifying.

3.1 Scenario-Based System Modeling

Our modeling language A) provides static template models for agents, along with an instantiation mechanism to customize each agent's actual internal behavior in each scenario; B) offers communication configurations to formalize multiple agents' interconnected behaviors in each single scenario; C) introduces scenario task configurations to describe system dynamic topologies in multiple scenarios. Finally, in D), we conclude the hierarchy of our modeling language.

A. Modeling Agent Runtime Behavior by Template Instantiation.

Definition 3 ***(Template Hybrid Automaton).*** *A template-HA is a tuple* $H^T = (X^T, U, Q, F, Inv, E^T, G, R, \alpha, \beta)$*, where*

- X^T *are variables classified by the function* α*:* $X^T \rightarrow \{X_{inp}, X_{out}, X_{local}\}$*.* X_{inp} *denotes global input variables, whose values can be updated by other agents during communication.* X_{out} *denotes global output variables that others can read during communication.* X_{local} *refers to private local variables.*
- E^T *are transitions classified by the function* β*:* $E \rightarrow \{E_{global}, E_{local}\}$*. Global transitions are public to other agents, therefore, communicating with others on global transitions is allowed. Local transitions are fixed in the agent's implementation, and are private to the agent itself.*
- U, Q, F, Inv, G, R *are parameters defined identically as HA in Definition 1.*

Template-HA formalizes the inherent features implemented in an agent. It constrains the agent's potential modes, transitions, and communication capabilities, including available communication ports, content, and channels, denoted as X_{inp}, X_{out}, and E_{global} respectively. To further constrain the agent's actual runtime behaviors in a specific scenario, we define Instance Hybrid Automaton.

Definition 4 ***(Instance Hybrid Automaton).*** *An instance-HA is a tuple* $H^I = (X, U, Q, F, Inv, E, G, R, \alpha, \beta, \gamma)$*, which can be* ***instantiated*** *from the template-HA* $H^T = (X, U, Q, F, Inv, E, G, R, \alpha, \beta)$ *through a transition activation function* γ*:* $E_{global} \rightarrow \{E_{act}, E_{deact}\}$*.*

Semantics: The state of H^I, denoted as $s = (q, x) \in Q \times X$, evolves either continuously in mode q or discretely by jumping to mode q' as defined in Definition 1. However, when it jumps via a global transition $e = (q, q')$, there is an extra constraint: the transition e must be activated by the function γ as $\gamma(e) = E_{act}$.

This instantiation mechanism improves template-HAs' flexibility and reusability, enabling adaptation to changing demands and reuse across scenarios by agents of the same type. For instance, in the multi-UAV system, agents are homogeneous and share the same template-HA, shown in brief in Fig. 3. It has three modes and five transitions. Variables are categorized by function α. For example, agent positions are global output variables, serving as permissible communication content among agents, and *inps* are global input variables as legal communication ports. Transitions are classified by function β, with global ones marked by dotted arrows, including `comm`, `commP`, and `commV`. Agents can be registered with ID and template-HA as shown in the purple box (line 2) in Fig. 4. During mission M4, when u_3 is leading u_{6-7}, the `comm` transition should be activated in u_3's instance-HA to inform its followers of its position, while u_{6-7}'s `commP` transitions should be activated for target calculation and position control. It can be concisely configured, as shown in the green box (line 15) in Fig. 4.

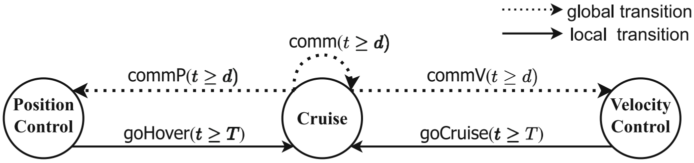
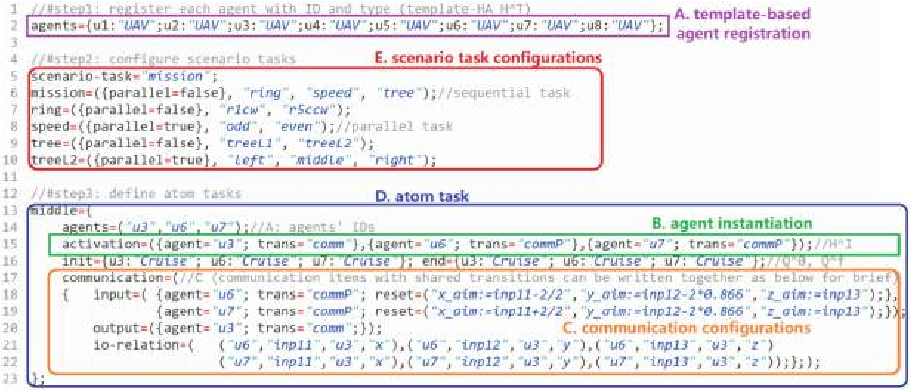


Fig. 3. An Overview of the Template-HA Model for UAV Agents


A. template-based agent registration
E. scenario task configurations
D. atom task
B. agent instantiation
C. communication configurations

```
1  //#step1: register each agent with ID and type (template-HA H^T)
2  agents={u1:"UAV";u2:"UAV";u3:"UAV";u4:"UAV";u5:"UAV";u6:"UAV";u7:"UAV";u8:"UAV"};
3
4  //#step2: configure scenario tasks
5  scenario-task="mission";
6  mission=({parallel=false}, "ring", "speed", "tree");//sequential task
7  ring=({parallel=false}, "r1cw", "r5ccw");
8  speed=({parallel=true}, "odd", "even");//parallel task
9  tree=({parallel=false}, "treeL1", "treeL2");
10 treeL2=({parallel=true}, "left", "middle", "right");
11
12 //#step3: define atom tasks
13 middle=(
14     agents=("u3","u6","u7");//A: agents' IDs
15     activation=({agent="u3"; trans="comm"},{agent="u6"; trans="commP"},{agent="u7"; trans="commP"});//H^I
16     init={u3:"Cruise"; u6:"Cruise"; u7:"Cruise"}; end={u3:"Cruise"; u6:"Cruise"; u7:"Cruise"};//Q^0, Q^f
17     communication=(//C (communication items with shared transitions can be written together as below for brief)
18     {    input=( {agent="u6"; trans="commP"; reset=("x_aim:=inp11-2/2","y_aim:=inp12-2*0.866","z_aim:=inp13");},
19                  {agent="u7"; trans="commP"; reset=("x_aim:=inp11+2/2","y_aim:=inp12-2*0.866","z_aim:=inp13");});
20          output=({agent="u3"; trans="comm";});
21          io-relation=(    ("u6","inp11","u3","x"),("u6","inp12","u3","y"),("u6","inp13","u3","z")
22                           ("u7","inp11","u3","x"),("u7","inp12","u3","y"),("u7","inp13","u3","z"));};);
23 };
```

Fig. 4. Multi-UAV System Modeling Snippet in a Structured Configuration File (Color figure online)

B. Modeling Agent Composition by Communication Configurations

Agents are usually complexly coupled rather than isolated in a CPS. To formalize agents' composition, we define communication items and configurations for them.

Definition 5 ***(Communication Item)****. Given two runtime agents modeled as $H_1^I{=}(X_1, U_1, Q_1, F_1, Inv_1, E_1, G_1, R_1, \alpha_1, \beta_1, \gamma_1)$ and $H_2^I{=}(X_2, U_2, Q_2, F_2, Inv_2, E_2, G_2, R_2, \alpha_2, \beta_2, \gamma_2)$, where $X_1{\cap}X_2 = \emptyset$, $U_1{\cap}U_2 = \emptyset$, $E_1{\cap}E_2 = \emptyset$, $Q_1{\cap}Q_2 = \emptyset$, a communication item for them is a tuple $c = (H_1^I, e_1, I, H_2^I, e_2, O, R'_{e_1})$, where*

- *$e_1 \in E_1,\ e_2 \in E_2,\ \beta_1(e_1) = \beta_2(e_2) = E_{global},\ \gamma_1(e_1) = \gamma_2(e_2) = E_{act}$;*
- *I, O are ordered sets, $|I| = |O|$,[2] elements in I must be global input variables in H_1^I, while elements in O must be global output variables in H_2^I, i.e., $\forall x \in I,\ x \in X_1 \bigwedge \alpha_1(x) = X_{inp}$; $\forall x \in O,\ x \in X_2 \bigwedge \alpha_2(x) = X_{out}$;*
- *R'_{e_1} is an additional reset function for variables in X_1.*

A communication item formalizes a composition relationship between two agents. Under the item c, global transitions e_1 and e_2 are bound together as a communication channel, expected to happen simultaneously under their guard conditions. Once they happen, communication contents in O will be received by ports in I in sequence. The additional reset function R'_{e_1} will be executed next, followed by agents' original reset functions.

Definition 6 ***(Communication Configuration)****. Given two agents modeled as H_1^I and H_2^I as above, a communication configuration $C{=}()|(c)|(c_1, \cdots, c_n)$, where $c_i{=}(H_{1\,i}^I, e_{1i}, I_i, H_{2\,i}^I, e_{2i}, O_i, R'_{e_{1\,i}})$ is a communication item for H_1^I and H_2^I.*

- *$C = ()$: H_1^I and H_2^I are independent;*
- *$C = (c)$: H_1^I and H_2^I are connected by the item c as Definition 5;*
- *$C = (c_1, \cdots, c_n)$: for any c_i and $c_j (1{\le}i{<}j{\le}n)$, if $e_{1i} \cap e_{2i} \cap e_{1j} \cap e_{2j} \neq \emptyset$, then all these transitions are expected to happen simultaneously, together with the communication between I_i and O_i, I_j and O_j. Otherwise, transitions e_{1i}, e_{2i} are bound together as Definition 5, so are transitions e_{1j} and e_{2j}.*

Semantics: Communication configuration, consisting of none, one, or multiple items, formalizes two agents' composition. Their composition under a communication configuration can be formalized as another larger instance-HA as follows.

Definition 7 ***(Instance-HA Composition)****. Given two instance-HA $H_1^I = (X_1, U_1, Q_1, F_1, Inv_1, E_1, G_1, R_1, \alpha_1, \beta_1, \gamma_1)$ and $H_2^I = (X_2, U_2, Q_2, F_2, Inv_2, E_2, G_2, R_2, \alpha_2, \beta_2, \gamma_2)$, where $X_1 \cap X_2 = \emptyset$, $U_1 \cap U_2 = \emptyset$, $E_1 \cap E_2 = \emptyset$, $Q_1 \cap Q_2 = \emptyset$, the composition of H_1^I and H_2^I under communication configuration C can be modeled as instance-HA $H^I = (X, U, Q, F, Inv, E, G, R, \alpha, \beta, \gamma)$, where*

- *$X = X_1 \cup X_2 = \{X_1, X_2\}$ are variables, classified by $\alpha(x) = \begin{cases} \alpha_1(x), x \in X_1 \\ \alpha_2(x), x \in X_2 \end{cases}$.*

[2] For any set A, |A| denotes the number of elements in A.

- $U = U_1 \cup U_2 = \{U_1, U_2\}$ *are external inputs;* $Q = Q_1 \times Q_2$ *are discrete modes;*
- $F = \{F_{(q_i,q_j)} | \ (q_i, q_j) \in Q\}$ *are flow functions, where* $\left\{ \begin{array}{l} \dot{X}_1 = F_{1q_i}(X_1, U_1) \\ \dot{X}_2 = F_{2q_j}(X_2, U_2) \end{array} \right.$.
- $Inv = \{Inv_{(q_i,q_j)} = Inv_{1(q_i)} \cap Inv_{2(q_j)} | \ (q_i, q_j) \in Q\}$ *are invariant conditions;*
- $E \subseteq Q \times Q$ *are discrete transitions constructed in the following two ways: for* $\forall e_a = (q_a, q'_a) \in E_1, \forall e_b = (q_b, q'_b) \in E_2$,
  1) if $\exists c = (H^I_1, e_1, I, H^I_2, e_2, O, R'_{e_1}) \in C$, $\{e_a, e_b\}=\{e_1, e_2\}$ *or* $\{e_a, e_b\}=\{e_2, e_1\}$*:*
  - $e_c=((q_a, q_b), (q'_a, q'_b)) \in E$*, classified by* $\beta(e_c)=E_{global}$*, activated by* $\gamma(e_c)=E_{act}$*;*
  - $G_{e_c}=G_{e_a} \cap G_{e_b}$ *is the guard condition, and* R_{e_c} *resets* $X := R_{e_c}(X, U)$ *as:*
    - * $X := \{R_{e_1}(R'_{e_1}(R_{I/O}(X_1), U_1), U_1), R_{e_2}(X_2, U_2)\}$ *when* $\{e_a, e_b\} = \{e_1, e_2\}$*;*
    - * $X := \{R_{e_2}(X_1, U_1), R_{e_1}(R'_{e_1}(R_{I/O}(X_2), U_2), U_2)\}$ *when* $\{e_a, e_b\} = \{e_2, e_1\}$*;*

    where function $R_{I/O}(x) = \left\{ \begin{array}{ll} O(i) & , I(i) = x \\ x & , x \in X - I \end{array} \right.$.

  2) otherwise:
  - $e_c = ((q_a, q_b), (q'_a, q_b)) \in E$, $e_d = ((q_a, q_b), (q_a, q'_b)) \in E$*;*
  - $\beta(e_c) = \beta(e_a)$, $\gamma(e_c) = \gamma(e_a)$, $G_{e_c} = G_{e_a}$, $R_{e_c}(X) := \{R_{e_a}(X1, U_1), X_2\}$*;*
  - $\beta(e_d) = \beta(e_b)$, $\gamma(e_d) = \gamma(e_b)$, $G_{e_d} = G_{e_b}$, $R_{e_d}(X) := \{X_1, R_{e_b}(X2, U_2)\}$.

The composition of multiple agents can be formalized by applying Definition 7 recursively. With communication configurations, agents' static compositions are efficiently configured, and always adhere to their inherent communication capabilities. For example, as shown in the orange box (line 17-22) in Fig. 4, communication configuration defines the composition of u_3 and its followers u_{6-7} in mission M4, where u_{6-7} gets u_3's position to calculate their own target positions.

C. Modeling Dynamic Typology by Scenario Task Configurations

In a reconfigurable CPS, agents usually interconnect dynamically, and the system topology shifts across scenarios during runtime. To capture this dynamic feature, we introduce atom tasks and scenario tasks below.

Definition 8 ***(Atom Task).*** *Given a system with multiple agents indexed by unique IDs, an atom task of the system is a tuple* $AT=(A, \mathbb{H}^I, Q^0, Q^f, \mathbb{C})$*, where*

- $A = \{A_i\}_{i=1}^n$ *are the IDs of all involvezd agents in the current scenario;*
- $\mathbb{H}^I=\{H^I_i\}_{i=1}^n$ *are involved agents' instance-HAs,* H^I_i *instantiated from* $H^T_{A_i}$*;*
- $Q^0=\{Q^0_i\}_{i=1}^n$, $Q^f=\{Q^f_i\}_{i=1}^n$ *are agents' initial and final modes,* $Q^0_i, Q^f_i \in Q_{A_i}$*;*
- $\mathbb{C}=\{C_i\}_{i=1}^m$ *are communication configurations defined on* $\mathbb{H}^I$*.*

An atom task describes a scenario, detailing active agents, behaviors, and compositions. It begins with agents evolving simultaneously from initial modes. It ends once all agents have reached final modes. For example, the scenario where u_3 leads u_{6-7} in mission M4 is described in the blue box (line 13-23) in Fig. 4.

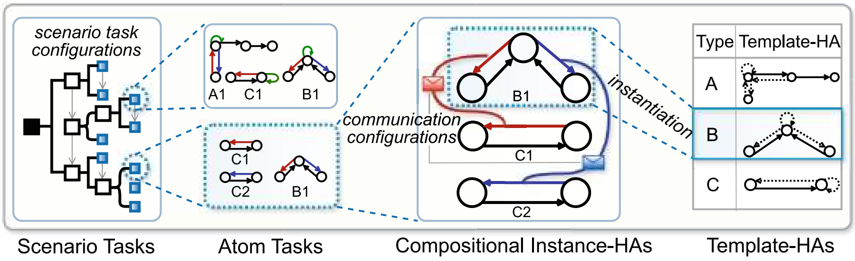


Fig. 5. The Hierarchy of Our Scenario-based Modeling Language

Besides, we offer a template mechanism for atom tasks to enhance task reusability. In a task template, involved agents' types, behaviors, and compositions are defined, leaving their IDs and some const variables for customization. For example, missions M1 and M2 can be customized by the same task template.

Definition 9 *(**Scenario Task**). A scenario task* $ST{=}AT|(ST;ST)|(ST||ST)$*, where* $(ST;ST)$ *denotes the sequential execution of two scenario tasks, while* $(ST||ST)$ *denotes the parallel execution of two scenario tasks. Therefore, it can either be a single atom task or a sequential/parallel compositional task.*

Agents involved in the two subtasks of a parallel compositional task should be disjoint. Besides, if an agent is involved first in atom task AT_1 *and then in* AT_2*, its initial mode in* AT_2 *should be consistent with its final mode in* AT_1*.*

Scenario task specifies the flow of scenarios recursively, which in turn, configures the dynamic shifts of system topologies. Therefore, it is also called **scenario task configuration** in this work. For example, as shown in the red box (line 5-10) in Fig. 4, the dynamic topology of the multi-UAV system in mission M1-4 is defined by scenario task configurations. The scenario task $T_{mission}{=}((T_{ring};T_{speed});T_{tree})$ represents the sequential execution of three scenario tasks: T_{ring} for M1-2, T_{speed} for M3, and T_{tree} for M4. See Fig. 8a for a graphical illustration.

D. Hierarchical Modeling for Reconfigurable CPSs

The hierarchy of our modeling language is summarized in Fig. 5. The design idea is to abstract agents' inherent features into template-HA models, so that reconfigurable CPSs can be modeled through simple instantiations and configurations.

Agent internal runtime behavior is modeled by an instance-HA, which is instantiated from its inherent template-HA model. Multiple agents' interconnected behaviors are modeled by the composition of their instance-HAs, which are connected by communication configurations. Then, the behavior of a reconfigurable CPS in a specific scenario is modeled by an atom task, with its involved agents' compositional instance-HAs. Finally, the CPS's runtime behaviors in changing scenarios with dynamic topologies are structured by scenario task configurations.

3.2 Specifying System Requirements in Topology-Aware STL

Topology-aware STL is a domain-specific extension of STL, designed to specify requirements for multi-agent systems with dynamic topologies. Although it shares the same expressiveness as STL, it facilitates the construction and evaluation of specifications related to system topologies.

Definition 10 ***(Topology-Aware STL).*** *A topology-aware STL formula φ interpreted over boolean, integer, and real variables $\mathcal{X}=\{\mathcal{X}_b, \mathcal{X}_i, \mathcal{X}_r\}$ is defined as:*

$\varphi := true \mid \psi \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U_I} \varphi_2$, with the atomic predicate ψ as
$\psi := End[T] \mid Mode[A] = d_q \mid \theta(\mathcal{X}_r) \geq d$, where

$End[T] \in \mathcal{X}_b$ indicates whether a scenario task T has been completed. $Mode[A] \in \mathcal{X}_i$ denotes the mode index of an agent A, while d_q is an integer mode index constant. Real variables $\mathcal{X}_r$ consist of agents' variables, external inputs, and scenario tasks' run time, denoted as $Time[T]$ for the scenario task T. Function θ: $\overline{\mathbb{R}}^n \rightarrow \mathbb{R}$ maps valuations of $\mathcal{X}_r$ into real numbers, and d is a constant.

Semantics: Given a topology-aware STL formula φ, a signal w of $\mathcal{X}$, and time t, the quantitative semantics of its robustness degree $\rho(\varphi, w, t)$ is defined as Definition 2, except the robustness of atomic predicates is defined as below:

$$\rho\ (End[T], w, t) = \begin{cases} 0 & , T \text{ is a completed atom task} \\ -1 & , T \text{ is an uncompleted atom task} \\ (\sum_{i=1}^{2} \rho(End[T_i], w, t))/2 & , T = (T_1; T_2) \text{ is a sequential task} \\ \min_{i\in[1,2]} \rho(End[T_i], w, t) & , T = (T_1 || T_2) \text{ is a parallel task} \end{cases} \tag{2}$$

$$\rho\ (Mode[A] = d_q, w, t) = -\mathbf{Dist}_A(Mode[A](t), d_q) \tag{3}$$

$$\rho\ (\theta(\mathcal{X}_r) \geq d, w, t) = \theta(w(t)) - d \tag{4}$$

The robustness of $End(T)$ indicates scenario task T's completion extent. The robustness of the predicate $q(A) = d_q$ indicates agent A's distance to the target mode d_q. The function $\mathbf{Dist}_A$: $Q_A \times Q_A \rightarrow \mathbb{Z}^+$ maps two modes in agent A to the minimum number of hops required to jump from the first mode to the second by transitions in A. The robustness degree is negative only if the signal of variables violates the formula, and it decreases as the severity of the violation increases. In this work, signals are interpreted according to system trajectories. When the trajectory is clear, we use a simpler notation $\rho(\varphi)$, instead of $\rho(\varphi, w, 0)$, to denote the robustness of formula φ on trajectory signal w at time zero.

Topology-aware STL is augmented with customized variables, predicates, and semantics. Predefined elements for task duration, completion, and mode identification help to describe topology-related requirements while avoiding introducing loads of extra variables, constraints, and functions to system models. For example, Table 1 lists some topology-aware STL specifications for the multi-UAV system modeled in the scenario task $T_{mission}$. Furthermore, topology-aware STL provides more meaningful quantitative semantics with better robustness degrees. This can facilitate CPS falsification by offering more effective guidance.

Table 1. Examples of Topology-Aware STL Specifications

Topology-Aware STL Specification	Description
$\square(\neg(Time[T_{tree}] \geq 10))$	tree formation takes less than 10 s
$\Diamond(End[T_{ring}] \wedge \frac{Time[AT_{r5ccw}]}{Time[AT_{r1cw}]} \geq 1)$	ring formations are completed eventually and the second ring takes no less time than the first
$\neg\Diamond(\square_{[0,4]}(Mode[u_2] = 1 \wedge u_2.\mu \geq 6.6))$	u_2 never stays in the Cruise mode (index 1) with dangerous horizontal speeds for 4 s

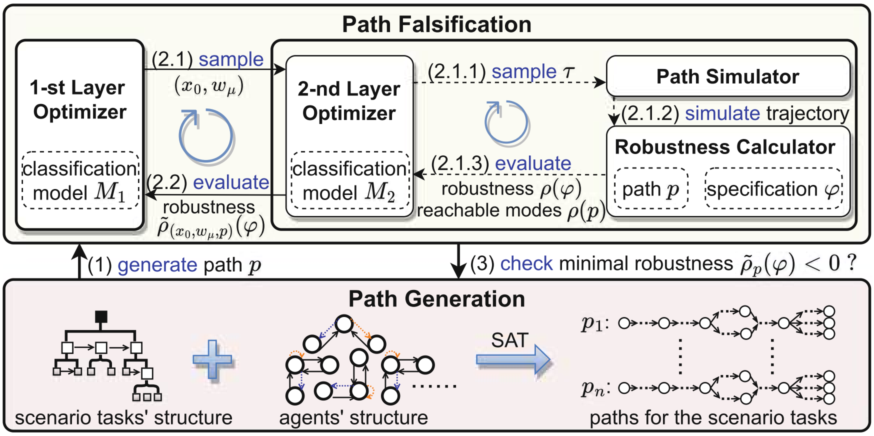

Path Falsification
1-st Layer Optimizer
classification model M_1
(2.1) sample
(x_0, w_μ)
(2.2) evaluate
robustness
$\tilde{\rho}_{(x_0,w_\mu,p)}(\varphi)$
2-nd Layer Optimizer
classification model M_2
(2.1.1) sample τ
(2.1.3) evaluate
robustness $\rho(\varphi)$
reachable modes $\rho(p)$
Path Simulator
(2.1.2) simulate trajectory
Robustness Calculator
path p
specification φ
(1) generate path p
(3) check minimal robustness $\tilde{\rho}_p(\varphi) < 0$?
Path Generation
scenario tasks' structure
agents' structure
SAT
p_1:
p_n:
paths for the scenario tasks

Fig. 6. The Path-oriented Optimization-based Falsification Framework

4 Path-Oriented Optimization-Based System Falsification

4.1 Falsification Framework

For a reconfigurable CPS modeled in a scenario task T and its requirement specified as a topology-aware STL φ, falsification aims to find a witness trajectory that violates φ, i.e. specification φ's robustness is negative on this trajectory. We denote a counterexample pair as (x_0, w_μ, p, τ), under which the trajectory is a witness. Here, x_0 represents the initial value of agents' variables, w_μ represents the signal of external inputs, p represents the path of agents' mode and transition sequence, and τ represents agents' dwelling time in modes along p.

To fully explore both discrete and continuous dimensions of the vast search space, our falsification approach uses a path-oriented strategy, as shown in Fig. 6. At the bottom, it explores discrete space, generating paths based on the discrete structure of agents and scenario tasks. On the top, it explores continuous space for each generated path p, performing path falsification to find an optimal pair (x_0, w_μ, τ) that minimizes the specification φ's robustness on trajectories along p. The falsification problem is solved when the minimal robustness is negative.

Our SAT-based path generation and optimization-based path falsification are given next. For brevity, we will only consider models with nondeterministic semantics, since deterministic models can be viewed as a simplified case.

4.2 Path Generation for Hierarchical Scenario Tasks

For a scenario task, its path determines how its involved agents change their modes and make their transitions during it. Below is the definition of path.

Definition 11 ***(Path of Scenario Task).*** *For a given scenario task ST, if*

- *$ST = AT = (A, \mathbb{H}^I, Q^0, Q^f, \mathbb{C})$ is an atom task: a path of it with length l is a tuple $p = (\{\mathcal{M}^i, \mathcal{T}^i\}_{i=0}^{l-1}, \mathcal{M}^l)$, where $\mathcal{M}^i = (\mathcal{M}_1^i, ..., \mathcal{M}_{|A|}^i)$ denotes all agents' modes in the i-th step, and $\mathcal{T}^i = (\mathcal{T}_1^i, ..., \mathcal{T}_{|A|}^i)$ denotes their next transitions after this step. For each agent $A_k \in A$, the tuple p should satisfy:*
  - ***agent discrete structure**: $\mathcal{M}_k^i$ should be a mode of agent A_k's instance-HA $H_k^I \in \mathbb{H}^I$; $\mathcal{T}_k^i$ can either be a transition that jumps from $\mathcal{M}_k^i$ to $\mathcal{M}_k^{i+1}$ in H_k^I or a stutter transition* `S` *that indicates no mode shift.*
  - ***atom task requirement**: agent A_k should start from its initial mode Q_k^0 and end in its final mode Q_k^f. If A_k is connected to another agent by a communication item in $\mathbb{C}$, their synchronized transitions should occur simultaneously.*
- *ST is a sequential task: its path is the concatenation of its subtasks' paths.*
- *ST is a parallel task: its path is the combination of its subtasks' paths.*

A composition scenario task's paths are produced by recursively generating its subtasks' paths and then concatenating or combining them. An atom task's paths are generated by SAT encoding, solving, and decoding.

We transform the length-bounded path generation for atom tasks into a SAT problem. Constraints for paths with lengths no longer than m are encoded as the propositional formula set $\mathcal{F}^m$ in Eq.(8). Specifically, constraints on each agent's discrete structure are encoded as Eq.(5)-(6), and requirements on the atom task are encoded as Eq.(7). To solve this SAT problem, advanced SAT solvers like Glucose [9] and CryptoMiniSat [60] can be used, and then the solutions can be decoded into paths. For instance, ((`Cruise`, `Cruise`, `Cruise`), (`comm`, `commP`, `commP`), (`Cruise`, `Position Control`, `Position Control`), (`S`, `goHover`, `S`), (`Cruise`, `Cruise`, `Position Control`), (`S`, `S`, `goHover`), (`Cruise`, `Cruise`,

`Cruise`)) is a path of the atom task AT_{middle} in the running example.

$$NEXT_k^i := \bigwedge_{q\in Q_{A_k}} ((\mathcal{M}_k^i = q) \rightarrow ((\bigvee_{e=(q,q')\in E_{A_k} \wedge \gamma_{A_k}(e)=E_{act}} (\mathcal{T}_k^i = e \wedge \mathcal{M}_k^{i+1} = q')) \vee (\mathcal{T}_k^i = S \wedge \mathcal{M}_k^{i+1} = q))) \quad (5)$$

$$EXCLD_k^i := (\bigwedge_{q,q'\in Q_{A_k} \wedge q\neq q'} (\mathcal{M}_k^i = q \rightarrow \mathcal{M}_k^i \neq q')) \wedge (\bigwedge_{e,e'\in (E_{A_k} \cup\ S)\wedge e\neq e'} (\mathcal{T}_k^i = e \rightarrow \mathcal{T}_k^i \neq e')) \quad (6)$$

$$INIT_k := (\mathcal{M}_k^0 = Q_k^0),\ END_k^l := (\mathcal{M}_k^l = Q_k^f),\ SYNC_k^i := \bigwedge_{(H_k^I, e_1, I, H_{k'}^I, e_2, O, R'_{e_1})\in C_j \in \mathbb{C}} (\mathcal{T}_k^i = e_1 \leftrightarrow \mathcal{T}_{k'}^i = e_2) \quad (7)$$

$$\mathcal{F}^m := \bigwedge_{A_k\in\mathbb{A}} (INIT_k \wedge (\bigvee_{0<l\leq m} (END_k^l \wedge \bigwedge_{0\leq i<l} SYNC_k^i \wedge \bigwedge_{0\leq i<l} NEXT_k^i \wedge \bigwedge_{0\leq i\leq l} EXCLD_k^i))) \quad (8)$$

4.3 Optimization-Based Falsification for Paths

Path falsification is achieved by optimizing for a trajectory that minimizes the target specification's robustness along the path. However, this optimization problem can be high-dimensional, nonlinear, nonconvex, and challenging, due to the complexity of agents, topologies, and scenarios involved. Therefore, we developed a two-layered classification-model-based optimization method for it, along with an efficient robustness calculation algorithm.

Two-Layered Classification-Model-Based Optimization. In Fig. 6, the first layer optimizes for the best pair of initial values and external inputs that minimizes robustness. For each pair (x_0, w_μ), the second layer performs multi-objective optimization for the best dwelling time τ that minimizes robustness while maximizes the number of reachable modes along the path. τ is evaluated by the two objective functions' weighted sum, i.e. $\min f = w_1\rho(\varphi) + w_2\rho(p)$ $(|w_1|+|w_2|=1, w_1 > 0, w_2 < 0)$, with weighting coefficients updated dynamically during optimization.

Both optimization layers are solved with classification models. Unlike classical heuristic optimization methods with weak theoretical guarantees or poor scalability, classification-model-based optimization has a grounded theory about complexity and convergence [39,64]. It has been theoretically proven to solve problems with local-Lipschitz continuity in polynomial time, and empirically proven to be scalable to high-dimensional problems. Its basic idea is to learn a classification model iteratively to discriminate bad solutions from good ones.

We employ a hyper-rectangle M_1 in the first layer as the classification model to represent the positive region, and a hyper-rectangle M_2 in the second layer as Fig. 6. In both layers, solutions are sampled from corresponding models, evaluated accordingly, and classified into positive/negative ones to refine models. The sampling, evaluating, and model refinement cycles iterate to enhance classification models' accuracy and optimize solutions. Our two-layered optimization enables accurate model refinement and alleviates the scalability issue.

Efficient Robustness Calculation. During falsification, system simulation and robustness calculation are compute-intensive and time-consuming. We introduce time-context and lifespan in temporal logic to reduce them.

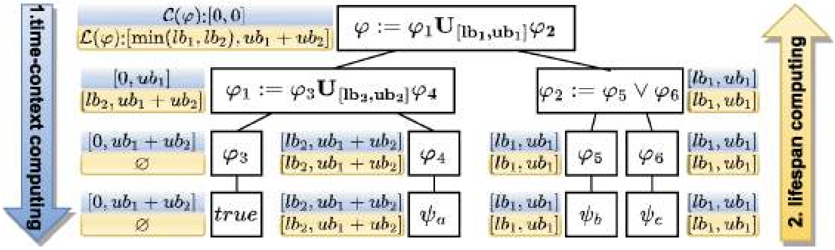


Fig. 7. An Example of Lifespan Calculation for the Target Specification φ

Definition 12 ***(Time-Context, Lifespan).*** *Given a target temporal logic specification φ, for any formula φ' in it (including φ itself), its* ***time-context*** *$\mathcal{C}(\varphi'|\varphi)$ is the period during which its satisfaction affects φ's satisfaction, its* ***lifespan*** *$\mathcal{L}(\varphi'|\varphi)$ is the period during which the signals affect its satisfaction in its time-context. We omit the target specification φ when it is clear. Relations between φ' and its subformulas, in terms of time-context and lifespan are:*

$$
\begin{aligned}
\varphi' := true &\rightarrow \mathcal{C}(true) = \mathcal{C}(\varphi') & \mathcal{L}(\varphi') &= \varnothing & (9)\\
\varphi' := \psi &\rightarrow \mathcal{C}(\psi) = \mathcal{C}(\varphi') & \mathcal{L}(\varphi') &= \mathcal{L}(\psi) = \mathcal{C}(\psi) & (10)\\
\varphi' := \neg\varphi_1 &\rightarrow \mathcal{C}(\varphi_1) = \mathcal{C}(\varphi') & \mathcal{L}(\varphi') &= \mathcal{L}(\varphi_1) &\\
\varphi' := \varphi_1 \vee \varphi_2 &\rightarrow \mathcal{C}(\varphi_1) = \mathcal{C}(\varphi_2) = \mathcal{C}(\varphi') & \mathcal{L}(\varphi') &= \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2) &\\
\varphi' := \varphi_1 \mathbf{U}_{[\mathbf{lb},\mathbf{ub}]} \varphi_2 &\rightarrow \mathcal{C}(\varphi_1) = \mathcal{C}(\varphi') \cup (ub \oplus \mathcal{C}(\varphi')) & \mathcal{L}(\varphi') &= \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2) &\\
&\mathcal{C}(\varphi_2) = (lb \oplus \mathcal{C}(\varphi')) \cup (ub \oplus \mathcal{C}(\varphi')) & & &
\end{aligned}
$$

To compute the target specification φ's robustness, we only need to simulate and record system trajectory within φ's lifespan. This helps to avoid unnecessary trajectory simulations, saving computation time and memory. The lifespan of the target specification φ is computed as defined in Definition 12. We demonstrate an example of it in Fig. 7. Note that the time-context of φ is $[0, 0]$, as our goal is to evaluate $\rho(\varphi, w, 0)$, the target specification's quantitative satisfaction on the trajectory signal starting at time zero. As shown in blue boxes in Fig. 7, starting with φ's time-context $[0, 0]$, the time-context of all subformulas is computed from top to bottom along the parse tree. Next, we get the lifespan of all atomic predicates according to Eq.(9)–(10). Finally, the lifespan of all formulas, including the root φ, is computed from bottom to top as yellow boxes.

During the robustness calculation of the target specification, its subformulas' time-context also can be utilized, avoiding unnecessary robustness calculations. Efficient dynamic programming algorithms have been developed for temporal logic robustness calculation [34,56]. Our algorithm follows the dynamic programming principle and further improves efficiency by considering the time-context of formulas in the target specification. Pseudocode appears in Algorithm 1.

This algorithm calculates the robustness of all formulas in φ's parse tree at each timestamp in the simulated trajectory, constructing a dynamic programming table $\mathcal{R}$ to store the robustness of formula $\phi[i]$ at time $\boldsymbol{t}[j]$ in $\mathcal{R}[i, j]$.

Algorithm 1: Time-Context Guided Robustness Calculation

```
   Input: $\varphi$: target specification, $w = (\boldsymbol{t}, \boldsymbol{\mathcal{X}})$: trajectory signal.
   Output: $\rho$: Robustness of the target specification.
1  Function computeRobustness($\varphi, w$)
2      $\phi$=parse($\varphi$);          ▷ formulas in $\varphi$'s parse tree in partial order (top-down)
3      for $i = |\phi|$ to 1 do          ▷ from bottom atoms up to the root $\varphi$
4          for $j = |\boldsymbol{t}|$ to 1 do          ▷ from last timestamp to time zero
5              if $\boldsymbol{t}[j] < \phi[i].\mathcal{C}[1]$ then break;     ▷ skip this robustness calculation
6              else if $\boldsymbol{t}[j] > \phi[i].\mathcal{C}[2]$ && $\phi[i] \neq \varphi_1 \mathbf{U}_{[\mathbf{c},\infty)} \varphi_2$ then continue;     ▷ skip
7              else   ▷ calculate robustness at timestamps within time-context
8                  if $\phi[i] := true$ then $\mathcal{R}[i,j] = \top$;
9                  else if $\phi[i] := \psi$ then $\mathcal{R}[i,j] = \rho(\psi, w, \boldsymbol{t}[j])$;     ▷ as Eq.(2)-(4)
10                 else if $\phi[i] := \neg\phi[k]$ then $\mathcal{R}[i,j] = -\mathcal{R}[k,j]$;     ▷ as Eq.(1)
11                 else if $\phi[i] := \phi[k_1] \vee \phi[k_2]$ then $\mathcal{R}[i,j] = \max(\mathcal{R}[k_1,j], \mathcal{R}[k_2,j])$;
12                 else if $\phi[i] := \phi[k_1]\mathbf{U}_I\phi[k_2]$ then     ▷ as Eq.(1)
13                     $m = j$;   $\mathcal{R}[i,j] = \bot$;   $r_1 = \top$;
14                     while $(t[m] - \boldsymbol{t}[j]) \notin I$ do $r_1 = \min(r_1, \mathcal{R}[k_1, m++])$;
15                     while $(\boldsymbol{t}[m] - \boldsymbol{t}[j]) \in I$ do
16                         if $I = [c,\infty)$ && $j \neq |t|$ && $(\boldsymbol{t}[m] - \boldsymbol{t}[j+1]) \in I$ then break;
17                         $\mathcal{R}[i,j] = \max(\mathcal{R}[i,j], min(r_1, \mathcal{R}[k_2, m]))$;
18                         $r_1 = \min(r_1, R[k_1, m++])$;
19                     if $I{=}[c,\infty)$&&$j{\neq}|t|$ then
                         $\mathcal{R}[i,j]{=}\max(\mathcal{R}[i,j], \min(\mathcal{R}[k_1,j], \mathcal{R}[i,j{+}1]))$;
20     return $\rho = \mathcal{R}[1,1]$          ▷ output $\varphi$'s robustness at time zero
```

It calculates robustness for atomic predicates, higher-level subformulas, and the root specification sequentially (line 3-19). Each is calculated from the last timestamp backwards to the initial timestamp (line 4-22). It skips unnecessary robustness calculations for timestamps outside the formula's time-context (line 5-6), i.e., the formula's satisfaction at that point does not affect the target specification's satisfaction. For timestamps inside the time-context, the formula's robustness is calculated by reusing its robustness at future timestamps and its subformulas' robustness (line 8-19), according to its robustness definitions in Eq.(1)–(4).

5 Implementation and Evaluation

5.1 Implementation and Research Questions

We have implemented our **S**ce**N**ar**I**o-based reconfigurable CPS modeling and **F**alsi**F**ication approach in a tool called SNIFF[3] in C++. It provides a graphical user interface (GUI) for modeling, supporting the construction of agent template models and agents' runtime instantiations, communication configurations and

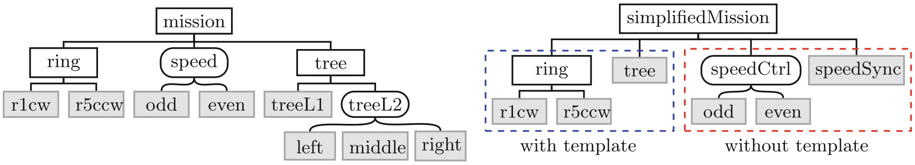
[3] SNIFF is available at https://github.com/njuwjw/SNIFF.

scenario task configurations. It also provides path-oriented CPS falsification, displaying witness paths and signals.
Below are research questions to evaluate our modeling and falsification approach:

RQ1: Can SNIFF help users reduce modeling time for reconfigurable CPS?
RQ2: Can SNIFF help users improve CPS modeling accuracy?
RQ3: Is SNIFF effective and efficient at CPS falsification? In particular, how is its scalability and runtime performance?

Next, we introduce the reconfigurable CPSs used in our evaluations in detail.

Example 1: Multi-UAV System. The motivating example system is used for our evaluations. Initially, eight UAVs are scattered in a closed space, $x, y \in [-10, 10]$, $z \in [1, 10]$. During runtime, they are dynamically connected with uncertain communication delays of $d \in [0.05, 0.2]$. Its behavior is modeled by the scenario task $T_{mission}$, as shown in Fig. 8a. Atom tasks are in grey. Sequential and parallel compositional tasks are drawn in white-filled rectangles with square and rounded corners, respectively. Besides, for fair modeling evaluations, the flight mission was slightly modified as shown in Fig. 8b. Four UAVs are involved and its scenario task includes three compositional tasks and six atom tasks.



(a) for falsification evaluations (as Sec.2.2) (b) for modeling evaluations

Fig. 8. Scenario Tasks for the Multi-UAV System

Example 2: Automatic Assembly System. It consists of four conveyors, three robots with grippers, and six robots with welding guns for car production. Conveyors transport floor, door, and roof panels along the assembly line, starting at $x \in [-2, 0]$. Grabbing robots communicate with conveyors with a delay of $t \in [0, 0.1]$, and wheel around to grasp, move, and place doors and roofs. Welding robots work together to spot weld joints in $x \in [8, 12], y \in [8, 12]$.

Its behavior is modeled by the scenario task $T_{assembly}$ as Fig. 9. Initially, in $T_{conveyGrasp}$, conveyors and grabbing robots collaborate to convey and grasp four panels in parallel. Then, in task T_{move}, grabbing robots first move doors and then slot the roof. Finally, in $T_{spotWeld}$, welding robots simultaneously weld joints while grabbing robots return home once relevant joints are welded, ready for the next assembly.

Experimental Setup. To evaluate SNIFF's modeling time and accuracy, we conducted a user study comparing it to another tool called SpaceEx [36],

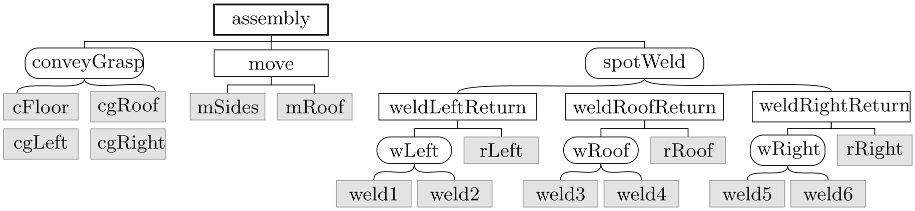


Fig. 9. Scenario Tasks for the Automatic Assembly System

which is a well-established platform for hybrid system modeling and verification. SpaceEx's modeling language, called SX [21,28], is based on compositional hybrid automata, presented as base or network components. Both tools provide a graphical modeling editor with automatic user action logging, which allows us to evaluate modeling time accurately. Also, graphical models are saved automatically in their respective modeling languages, enabling our checks for modeling accuracy.

This user study was conducted with 111 computer science department students, including undergraduates, graduates, and PhDs. They received a 50-minute training session on CPS modeling and were required to model a multi-UAV system, described as Fig. 8b, using both tools in a week. Participants were given with a UAV template model and the templates for half atom tasks (in the blue box in Fig. 8b) in SNIFF format. They are also provided with base components in SpaceEx formats, containing the same information as the SNIFF templates we provided. Specifically, it contains information on all potential dynamics of a UAV agent and agent compositional information in three atom tasks.

To evaluate SNIFF's falsification performance, we conducted a experiment comparing it with S-TaLiRo [8], an advanced optimization-based CPS falsification tool. We ran S-TaLiRo with its SOAR [48] option, which outperformed all of its other optimization options in our benchmarks. Systems are modeled in 'hautomaton' class as its input. We falsified various temporal specifications for the above two example systems on both tools. We conducted all experiments 100 times on the same PC (Intel Core i5-12500, 16GB RAM) with a time limit of 1800 s. The system simulation limit is 1000 iterations for deterministic benchmarks and 25000 for non-deterministic ones. We did not use SpaceEx for nonlinear systems' temporal logic falsification comparison since it mainly focuses on reachability verification for hybrid systems with piecewise affine dynamics.

5.2 Experimental Evaluation and Analysis

Time Cost of Modeling. According to participants' modeling logs, models were constructed in an average of 46.9 min in SNIFF, while it took 308.1 min, 6.6 times longer, in SpaceEx. To further analyze how time was saved in SNIFF, Table 2 compares their time cost in different modeling elements.

Table 2. Time Costs to Model Different System Elements in SNIFF and SpaceEx

Type of Elements	Number	Time (min)		Average Time (min)	
		SNIFF	SpaceEx	SNIFF	SpaceEx
Agent	4	2.3	31.4	0.6	7.9
Atom Task (without Template)	3	8.4 3.1 11.4	41.8 16.4 29.4	7.6	29.2
Atom Task (with Template)	3	5.1 2.4 3.0	40.0 15.4 25.0	3.5	26.8
Compositional Scenario Task	3	1.4 2.3 1.3	21.8 18.8 31.9	1.6	24.2
Others (Other GUI Actions)	—	6.3	36.4	—	—
Total		**46.9**	**308.1**		

- Agent internal behavior was modeled 13 times faster by `SNIFF` than by `SpaceEx`. `SNIFF`'s agent-instantiation mechanism allows quick updates to agent internal behavior, while `SpaceEx` requires additional components to be built.
- For both tools, modeling system behaviors in atom tasks took the longest. On average, it took 7.6 min with `SNIFF` when no task template was given, while nearly half an hour with `SpaceEx`. It indicates that communication configurations in `SNIFF` were helpful in defining agent composition in atom tasks.
- With task templates, atom tasks' modeling time was reduced by half in `SNIFF`, from 7.6 min to 3.5 min. `SNIFF`'s support for task templates makes agent composition more convenient.
- To describe topologies in compositional scenario tasks, it averagely took 1.6 min in `SNIFF`, but 24.2 min in `SpaceEx`. Dynamic topologies can be set quickly by `SNIFF`'s scenario task configurations, but challenging in `SpaceEx`.

The Answer to RQ1: `SNIFF`'s template models, together with its flexible instantiation and configuration mechanisms, simplify the modeling of reconfigurable CPS, saving considerable modeling time and effort for users.

Accuracy of Modeling. Among the models submitted by participants, 70.27% of `SNIFF` models were accurate, while only 7.21% of `SpaceEx` models were accurate. We analyzed all submitted modeling errors and classified them into four major types in Table 3. It lists their sources and probabilities, which are counted per source to eliminate the influence of the number of sources.

- Compared to `SpaceEx`, `SNIFF` reduces all types of modeling errors significantly.
- In `SpaceEx`, more than half (59.46%) of system behaviors with parallel topology shifts were incorrectly modeled, whereas `SNIFF`'s scenario task configurations simplify this modeling process and eliminate such errors completely.
- In `SNIFF`, modeling agents' compositions has the highest error rate (6.91%). Our additional statistics found that only 13% of these errors occur when task templates are used, indicating that task templates can help reduce such errors.

Table 3. Probabilities of Different Modeling Errors in SNIFF and SpaceEx

Type of Errors	Source of Errors	Probability (per Source)	
		SNIFF	SpaceEx
Agent Internal Behavior Error	Agent	1.13%	15.32%
Agents' Composition Error	Atom Task	**6.91**%	38.89%
Sequential Topology Shift Error	Sequential Compositional Task	0.90%	17.12%
Parallel Topology Shift Error	Parallel Compositional Task	0.00%	**59.46**%
Others[1]	—	1.80%	29.73%

[1] This includes misnamed, incomplete and undefined parameters, agents, tasks, etc.

The Answer to RQ2: `SNIFF` significantly improves the accuracy of CPS models constructed by users, reducing all types of modeling errors.

Falsification Performance. Two example systems are falsified under various scenarios, ranging from the simplest to the most complex. Table 4 lists the size of each scenario task, including the number of atom tasks. Scenarios within 20 s are highlighted in green, others in red. We listed the falsification performance of `SNIFF` and `S-TaLiRo` for each specification, including success rates, and costs of time and simulation iterations to find a witness. We also calculated the average falsification time for each atom task, presented in grey.

Generally, `SNIFF` supports falsification for models with richer semantics and can handle a wider range of specifications. The multi-UAV system follows deterministic semantics, but the assembly system follows nondeterministic semantics, which is not specifically supported in `S-TaLiRo`. Out of the total thirty specifications, six are unsupported in `S-TaLiRo` and are underlined in the table.

- Effectiveness: On average, `SNIFF`'s success rate is 99.7%, whereas `S-TaLiRo` is below 45%, indicating that our path-oriented falsification approach helps explore search space effectively. Especially, `SNIFF`'s success rate is at least 97% in all cases, while `S-TaLiRo` achieved this in only two cases.
- Efficiency: `SNIFF` is more efficient than `S-TaLiRo`. It takes 0.1–39.2 s to falsify these specifications, while `S-TaLiRo` takes an average of 730.9 s, more than 12 min. Also, in the 16 cases that `S-TaLiRo` supports, `SNIFF` averagely requires only one-fifth as many simulation iterations as `S-TaLiRo`.
- Scalability: As the scenario grows in size and complexity, system behaviors become more intricate, and falsification inevitably becomes challenging and time-consuming. The table shows that `S-TaLiRo`'s success rate decreases greatly in complex scenarios, whereas `SNIFF`'s success rate barely drops. The rise in `SNIFF`'s time-cost and simulation iterations is also reasonable.
- Runtime Performance: CPSs may need online modeling and analysis due to runtime reconfiguration [13]. To evaluate our performance for runtime atom task scenarios, we roughly divide the falsification time by the number of atom tasks, as highlighted in grey. It indicates that `SNIFF` can handle such scenarios in around 1.7 s on average. Since automatic modeling in our language

Table 4. Falsification Results of Example Systems in SNIFF and S-TaLiRo

System	Scenario Task				Spec[2]	SNIFF				S-TaLiRo[3]			
	name	size[1]				success rate	time (s)	time per AT (s)	sim-iter	success rate	time (s)	time per AT (s)	sim-iter
		s_1	s_2	s_3									
Multi -UAV System	treeL1	1	4	16	s1	100%	0.7	0.7	31	100%	20.1	20.1	179
					s2	100%	2.2	2.2	110	100%	33.9	33.9	179
					s3*	100%	0.5	0.5	25	91%	211.1	211.1	653
					s4*	100%	0.9	0.9	47	78%	207.1	207.1	588
	tree	4	8	32	s5	100%	1.8	0.5	42	40%	776.0	194.0	373
					s6	100%	2.3	0.6	51	62%	492.3	123.1	276
					s7*	100%	2.1	0.5	22	45%	905.7	226.4	308
					s8*	100%	2.1	0.5	22	79%	728.7	182.2	348
	mission	8	8	32	s9	99%	25.3	3.2	55	0%	#N/A	#N/A	#N/A
					s10	97%	39.2	4.9	120	12%	1330.3	166.3	341
					s11	98%	38.2	4.8	87	3%	989.1	123.6	333
					s12	100%	34.6	4.3	129	43%	1154.9	144.4	336
					s13*	100%	23.1	2.9	130	1%	1532.5	191.6	346
					s14*	100%	17.7	2.2	65	34%	1269.3	158.7	350
					s15*	100%	30.2	3.8	111	4%	1220.9	152.6	339
					s16*	100%	37.9	4.7	83	24%	91.8	11.5	202
					s17	100%	10.1	1.3	73	/	/	/	/
					s18	100%	15.3	1.9	68	/	/	/	/
					s19	100%	13.6	1.7	62	/	/	/	/
					s20	100%	28.9	3.6	108	/	/	/	/
Automatic Assyembly System	weld2	1	1	2	s21*	100%	0.1	0.1	1414	/	/	/	/
					s22*	100%	0.2	0.2	2123	/	/	/	/
	wRoof	2	2	4	s23	100%	0.8	0.4	3865	/	/	/	/
					s24*	100%	0.6	0.3	2799	/	/	/	/
	convey -Grasp	4	7	14	s25*	99%	1.9	0.5	2335	/	/	/	/
					s26*	99%	2.7	0.7	2172	/	/	/	/
	assembly	15	13	26	s27	100%	14.2	0.9	1260	/	/	/	/
					s28*	100%	13.9	0.9	1219	/	/	/	/
					s29*	100%	18.3	1.2	1635	/	/	/	/
					s30*	98%	16.3	1.1	1803	/	/	/	/
Average						99.7%	13.2	1.7	736	44.8%	730.9	143.1	343

[1] This column indicates the complexity of scenarios. s_1, s_2, and s_3 respectively denote the number of involved atom tasks, involved agents, and uncertain variables/external inputs.

[2] Asterisked specifications are topology-related and are indirectly supported by S-TaLiRo. Underlined specifications involve nonlinear arithmetic that are unsupported by S-TaLiRo.

[3] '#N/A' denotes unknown costs to falsify when a case hasn't been successfully falsified. '/' denotes unavailable falsification performance due to an unsupported model or specification.

barely costs time, we can handle the entire online configuring, modeling, and falsification process in seconds, improving system runtime safety.

The Answer to RQ3: SNIFF is highly effective, efficient, and scalable for temporal logic falsification in CPSs. It also demonstrates satisfactory runtime performance, handling runtime scenarios with a single atom task in seconds.

5.3 Threats to Validity

In this section, we discuss possible threats to the validity of our user study.

Internal Validity. is the assessment of modeling accuracy. In our user study, to evaluate modeling accuracy accurately, we restricted names of agents and tasks in `SNIFF` and components in `SpaceEx`. In this way, `SNIFF`/`SpaceEx` models could be automatically checked by examining their scripts. Another threat is that participants' familiarity with the target CPS could affect modeling. After collecting participants' modeling logs (automatically logged by both tools), we examined the order in which participants used the tools. It shows that 60% of participants used `SNIFF` first, and 84% of participants completed modeling in `SNIFF` first. This may introduce bias into our modeling evaluation.

External Validity. The main external threat arises from the representative of subjects of our user study. Formal modeling is typically conducted by engineers with relevant expertise. To conduct the user study with representative subjects, we recruited 111 participants from a course called Formal Languages and Automata, ensuring that they had a basic understanding of modeling and were close to real-world industry target users. Another threat is that our conclusions might not generalize to all reconfigurable CPS. We alleviate the threat by deriving the Multi-UAV system from a real-world case. Results from real-world CPS will help us evaluate our approach better. Additionally, we plan to conduct larger-scale experiments on more real-world reconfigurable CPSs in the future.

6 Related Work

Formalizing CPS is complex due to its hybrid and compositional nature. Formal models, such as hybrid automata [38,45] and hybrid process algebras [12,22], are used to formalize the tangled continuous and discrete behaviors in CPS. Further, compositional hybrid models formalize the behavior of multiple agents, by composing hybrid automata [6,45], processes [15], etc. Based on these formalisms, various modeling languages have been developed for multi-agent systems, including Unified Modeling Language (UML) based notations such as AUML [40], Constraint Logic Programming (CLP) notations [50,51], and Verse [43], a Python library for systems with multiple agents moving on a map, etc. However, for reconfigurable multi-agent CPS whose system topologies change dynamically at runtime [37,59,62], these models and notations do not provide sufficient mechanisms to support such flexibility. Although there have been works on modeling reconfigurable systems, they focus mainly on general architectural modeling [10,17,58]. In contrast, this work aims to provide a behavior modeling formalism that can effectively capture the reconfigurability of these systems.

CPS falsification is an effective way to detect system behaviors that violate specifications. Many falsification approaches search for witness behaviors by applying optimization algorithms to minimize robustness, including optimistic optimization [65,66], Bayesian optimization [25,61], classical heuristic-based methods [7,49,52,57], etc. Our falsification approach also falls under this category. Its most closely related work is [63], which also applies classification-model-based optimization [39,64]. However, it is limited to the safety falsification of a single agent. In contrast, we support broader temporal logic falsification for CPSs with multiple dynamically wired compositional agents. Apart from optimization-based ones, there are also other falsification approaches, such as motion-planning-based falsification [24,31,53], gradient-based falsification [14], and learning-based falsification [3].

7 Conclusion and Future Work

We present a hierarchical formal modeling language for reconfigurable CPSs. It provides templates for agent inherent features and allows for the formalization of reconfigurable CPS through agent instantiations, agents' communication configurations, and scenario task configurations. While it doesn't add expressivity beyond HA, its flexible hierarchy can simplify the complex modeling process, saving time and improving accuracy. Our future plans involve providing support for more CPS architectures and system topology variations.

We also propose a path-oriented falsification approach for reconfigurable CPSs. It employs a two-layered optimization approach to explore the search space effectively and cuts unnecessary calculations to improve efficiency. Experiments have shown that it can perform scalable falsification with high success rates. In the future, we plan to further enhance its runtime performance by leveraging offline and past online analysis results.

References

1. Acharya, S., Bharadwaj, A., Simmhan, Y., Gopalan, A., Parag, P., Tyagi, H.: Cornet: A co-simulation middleware for robot networks. In: 2020 International Conference on COMmunication Systems & NETworkS (COMSNETS), pp. 245–251. IEEE (2020)
2. Ahmadi, A., Moradi, M., Cherifi, C., Cheutet, V., Ouzrout, Y.: Wireless connectivity of cps for smart manufacturing: a survey. In: 2018 12th International Conference on Software, Knowledge, Information Management & Applications (SKIMA), pp. 1–8. IEEE (2018)
3. Akazaki, T., Liu, S., Yamagata, Y., Duan, Y., Hao, J.: Falsification of cyber-physical systems using deep reinforcement learning. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 456–465. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_27
4. Alur, R.: Principles of cyber-physical systems. MIT press (2015)
5. Alur, R., et al.: The algorithmic analysis of hybrid systems. Theoret. Comput. Sci. **138**(1), 3–34 (1995)
6. Alur, R., Grosu, R., Lee, I., Sokolsky, O.: Compositional modeling and refinement for hierarchical hybrid systems. J. Logic Algebraic Program. **68**(1–2), 105–128 (2006)
7. Annapureddy, Y.S.R., Fainekos, G.E.: Ant colonies for temporal logic falsification of hybrid systems. In: IECON 2010-36th Annual Conference on IEEE Industrial Electronics Society, pp. 91–96. IEEE (2010)

8. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21
9. Audemard, G., Simon, L.: Glucose: a solver that predicts learnt clauses quality. SAT Competition pp. 7–8 (2009)
10. Bazydło, G.: Designing reconfigurable cyber-physical systems using unified modeling language. Energies **16**(3), 1273 (2023)
11. Bérard, B., et al.: Systems and software verification: model-checking techniques and tools. Springer Science & Business Media (2013)
12. Bergstra, J.A., Middelburg, C.A.: Process algebra for hybrid systems. Theoret. Comput. Sci. **335**(2-3), 215–280 (2005)
13. Bersani, M.M., García-Valls, M.: Online verification in cyber-physical systems: Practical bounds for meaningful temporal costs. J. Softw. Evolution Proc. **30**(3) (2018)
14. Bogomolov, S., Frehse, G., Gurung, A., Li, D., Martius, G., Ray, R.: Falsification of hybrid systems using symbolic reachability and trajectory splicing. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 1–10 (2019)
15. Brinksma, E., Krilaviĉius, T., Usenko, Y.S.: Process algebraic approach to hybrid systems. IFAC Proc. Vol. **38**(1), 325–330 (2005)
16. Bu, L., Li, Y., Wang, L., Li, X.: Bach: bounded reachability checker for linear hybrid automata. In: 2008 Formal Methods in Computer-Aided Design, pp. 1–4. IEEE (2008)
17. Cavalcante, E., Batista, T., Oquendo, F.: Supporting dynamic software architectures: From architectural description to implementation. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture, pp. 31–40. IEEE (2015)
18. 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
19. 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
20. Copty, F., et al.: Benefits of bounded model checking at an industrial setting. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 436–453. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_43
21. Cotton, S., Frehse, G., Lebeltel, O.: The spaceex modeling language. SpaceEx tool (2010)
22. Cuijpers, P.J.L., Reniers, M.A.: Hybrid process algebra. J. Logic Algebraic Program. **62**(2), 191–245 (2005)
23. Dafflon, B., Moalla, N., Ouzrout, Y.: The challenges, approaches, and used techniques of cps for manufacturing in industry 4.0: a literature review. Inter. J. Adv. Manufact. Technol. **113**, 2395–2412 (2021)
24. Dang, T., Nahhal, T.: Coverage-guided test generation for continuous and hybrid systems. Formal Methods Syst. Design **34**, 183–213 (2009)
25. Deshmukh, J., Horvat, M., Jin, X., Majumdar, R., Prabhu, V.S.: Testing cyber-physical systems through bayesian optimization. ACM Trans. Embedded Comput. Syst. (TECS) **16**(5s), 1–18 (2017)
26. Do, H.T., et al.: Formation control algorithms for multiple-uavs: a comprehensive survey. EAI Endorsed Trans. Indus. Netw. Intell. Syst. **8**(27), e3–e3 (2021)

27. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
28. Donzé, A., Frehse, G.: Modular, hierarchical models of control systems in spaceex. In: 2013 European Control Conference (ECC), pp. 4244–4251. IEEE (2013)
29. 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
30. Doyen, L., Frehse, G., Pappas, G.J., Platzer, A.: Verification of hybrid systems. In: Handbook of Model Checking, pp. 1047–1110 (2018)
31. Dreossi, T., Dang, T., Donzé, A., Kapinski, J., Jin, X., Deshmukh, J.V.: Efficient guiding strategies for testing of temporal properties of hybrid systems. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 127–142. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_10
32. Du, Y., Lu, X., Wang, J., Chen, B., Tu, H., Lukic, S.: Dynamic microgrids in resilient distribution systems with reconfigurable cyber-physical networks. IEEE J. Emerging Selected Topics Power Electr. **9**(5), 5192–5205 (2020)
33. Ernst, G., Sedwards, S., Zhang, Z., Hasuo, I.: Fast falsification of hybrid systems using probabilistically adaptive input. In: Parker, D., Wolf, V. (eds.) QEST 2019. LNCS, vol. 11785, pp. 165–181. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30281-8_10
34. Fainekos, G.E., Sankaranarayanan, S., Ueda, K., Yazarel, H.: Verification of automotive control applications using s-taliro. In: 2012 American Control Conference (ACC), pp. 3567–3572. IEEE (2012)
35. Frehse, G.: Phaver: algorithmic verification of hybrid systems past hytech. Int. J. Softw. Tools Technol. Transfer **10**, 263–279 (2008)
36. 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
37. Gray, J., Rumpe, B.: Modeling dynamic structures (2020)
38. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science, pp. 278–292. IEEE (1996)
39. Hu, Y.Q., Qian, H., Yu, Y.: Sequential classification-based optimization for direct policy search. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 31 (2017)
40. Huget, M.P.: Agent uml notation for multiagent system design. IEEE Internet Comput. **8**(4), 63–71 (2004)
41. Lee, E.A.: The past, present and future of cyber-physical systems: a focus on models. Sensors **15**(3), 4837–4869 (2015)
42. Lee, E.A., Seshia, S.A.: Introduction to embedded systems: A cyber-physical systems approach. MIT press (2016)
43. Li, Y., Zhu, H., Braught, K., Shen, K., Mitra, S.: Verse: a python library for reasoning about multi-agent hybrid system scenarios. In: International Conference on Computer Aided Verification, pp. 351–364. Springer (2023). https://doi.org/10.1007/978-3-031-37706-8_18
44. Lygeros, J., Johansson, K.H., Simic, S.N., Zhang, J., Sastry, S.S.: Dynamical properties of hybrid automata. IEEE Trans. Autom. Control **48**(1), 2–17 (2003)
45. Lynch, N., Segala, R., Vaandrager, F.: Hybrid i/o automata. Inform. Comput. **185**(1), 105–157 (2003)

46. 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
47. Maler, O., Nickovic, D., Pnueli, A.: Checking temporal properties of discrete, timed and continuous behaviors. Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, pp. 475–505 (2008)
48. Mathesen, L., Pedrielli, G., Ng, S.H., Zabinsky, Z.B.: Stochastic optimization with adaptive restart: a framework for integrated local and global learning. J. Global Optim. **79**, 87–110 (2021)
49. Mathesen, L., Yaghoubi, S., Pedrielli, G., Fainekos, G.: Falsification of cyber-physical systems with robustness uncertainty quantification through stochastic optimization with adaptive restart. In: 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE), pp. 991–997. IEEE (2019)
50. Mohammed, A., Furbach, U.: Multi-agent systems: modeling and verification using hybrid automata. In: Braubach, L., Briot, J.-P., Thangarajah, J. (eds.) ProMAS 2009. LNCS (LNAI), vol. 5919, pp. 49–66. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14843-9_4
51. Mohammed, A., Stolzenburg, F.: Implementing hierarchical hybrid automata using constraint logic programming. In: Proceedings of 22nd Workshop on (Constraint) Logic Programming, Dresden, pp. 60–71 (2008)
52. Nghiem, T., Sankaranarayanan, S., Fainekos, G., Ivancić, F., Gupta, A., Pappas, G.J.: Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, pp. 211–220 (2010)
53. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: from verification to falsification by combining motion planning and discrete search. Formal Methods Syst. Design **34**(2), 157–182 (2009)
54. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_32
55. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Trans. Autom. Control **52**(8), 1415–1428 (2007)
56. Rosu, G., Havelund, K.: Synthesizing dynamic programming algorithms from linear temporal logic formulae (2001)
57. Sankaranarayanan, S., Fainekos, G.: Falsification of temporal properties of hybrid systems using the cross-entropy method. In: Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control, pp. 125–134 (2012)
58. Selić, B.: Specifying dynamic software system architectures. Softw. Syst. Model. **20**(3), 595–605 (2021)
59. Sha, L., Gopalakrishnan, S., Liu, X., Wang, Q.: Cyber-physical systems: A new frontier. In: 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (Sutc 2008), pp. 1–9. IEEE (2008)
60. 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
61. Waga, M.: Falsification of cyber-physical systems with robustness-guided black-box checking. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control, pp. 1–13 (2020)

62. Wan, K., Hughes, D., Man, K.L., Krilavicius, T., Zou, S.: Investigation on composition mechanisms for cyber physical systems. Inter. J. Design, Anal. Tools Integrated Circ. Syst. **2**(1), 30 (2011)
63. Wang, J., Bu, L., Xing, S., Li, X.: Path-oriented, derivative-free approach for safety falsification of nonlinear and nondeterministic cps. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(2), 238–251 (2021)
64. Yu, Y., Qian, H., Hu, Y.Q.: Derivative-free optimization via classification. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 30 (2016)
65. Zhang, Z., Ernst, G., Sedwards, S., Arcaini, P., Hasuo, I.: Two-layered falsification of hybrid systems guided by monte carlo tree search. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2894–2905 (2018)
66. Zhang, Z., Lyu, D., Arcaini, P., Ma, L., Hasuo, I., Zhao, J.: Effective hybrid system falsification using monte carlo tree search guided by QB-robustness. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 595–618. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_29

Probabilistic Systems

Playing Games with Your PET: Extending the Partial Exploration Tool to Stochastic Games

Tobias Meggendorfer[1] and Maximilian Weininger[2](✉)

[1] Lancaster University Leipzig, Leipzig, Germany
tobias@meggendorfer.de
[2] Institute of Science and Technology Austria, Klosterneuburg, Austria
mweining@ista.ac.at

Abstract. We present version 2.0 of the *Partial Exploration Tool* (PET), a tool for verification of probabilistic systems. We extend the previous version by adding support for *stochastic games*, based on a recent unified framework for sound value iteration algorithms. Thereby, PET2 is the first tool implementing a sound and efficient approach for solving stochastic games with objectives of the type reachability/safety and mean payoff. We complement this approach by developing and implementing a partial-exploration based variant for all three objectives. Our experimental evaluation shows that PET2 offers the most efficient partial-exploration based algorithm and is the most viable tool on SGs, even outperforming unsound tools.

Keywords: Probabilistic verification · Stochastic games · Partial exploration · Model checker

1 Introduction

Stochastic games (SGs) [12] are a foundational model for sequential decision making in the presence of uncertainty and two antagonistic agents. They are practically relevant, with applications ranging from economics [1] over IT security [35] to medicine [7]; and they are theoretically fundamental, in particular because many associated classical decision problems are representative of the important complexity class NP ∩ co-NP, e.g. deciding whether the value of a reachability or mean payoff objective is greater than a given threshold [2,22] (see [10] for recent advances). See [36, Chp. 1.3] for further motivation.

However, even mature tools either do not support SGs at all (STORM [21]) or employ approaches without formal guarantees, i.e. their results can be wrong

M. Weininger has received funding from the EU's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101034413.

Data availability: We refer to the artefact with the exact code used for the submission and all logs [31], and the gitlab with the continually developed source code [30].

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 359–372, 2024.
https://doi.org/10.1007/978-3-031-65633-0_16

(Prism-games [27] and Tempest [33]), which is unacceptable in the context of safety-critical applications. This is because *value iteration* (VI), the de-facto standard approach to solving stochastic systems, lacks a sound and efficient *stopping criterion*, i.e. a "rule" to check whether the current iterates are sufficiently close to the correct value. For Markov decision processes (MDPs) (SGs with only one player) such a sound variant of VI (often called *interval iteration*) was developed a decade ago [8,17] and subsequently implemented in practically all major model checkers. However, extending the underlying reasoning to SGs proved to be surprisingly tricky, with sound variants even for special cases only developed quite recently [15]. Just a year ago, [24] presented a unified way of ensuring the soundness of VI for solving SGs with various quantitative objectives, which forms the theoretical basis for this work.

Note that the classical approaches strategy iteration and quadratic programming are sound in theory, but (i) the available implementations of [26] are prototypical and unsound [26, Sec. 5], and (ii) these approaches usually either are not practically efficient or use heuristics that actually make them unsound [18].

Contributions. We present version 2.0 of the *Partial Exploration Tool* (Pet), the first tool implementing a sound and efficient approach for solving SGs with objectives of the type reachability/safety and mean payoff (a.k.a. long-run average reward). In the following, we write Pet1 and Pet2 to refer to the previous and now presented version of Pet, respectively.

Theoretically, Pet2 is based on the results of [24]. We provide two flavours of their approach: Firstly, we implement the basic complete-exploration (CE) algorithm, enhanced with several theoretical improvements, both new and suggested in the literature. Secondly, we develop a *partial-exploration (PE)* approach, the focus of Pet2, by combining the ideas of [24] with those in [8,15,29].

Practically, Pet2 is an extension of Pet1 [29] (only applicable to MDPs). Apart from adding support for dealing with SGs and completely replacing the approach of Pet1 with the ideas of [24], we implemented many engineering improvements. Concretely, despite employing a more general algorithm, our experimental evaluation shows that Pet2's performance is on par with Pet1. Moreover, Pet2 outperforms the existing SG solvers Prism-games and Tempest, despite those not providing guarantees (and indeed returning wrong results).

2 Preliminaries

Here, we very briefly recall turn-based stochastic games as far as they are necessary to understand this paper, with more details in [32, App. A] and [24].

A *(turn-based) stochastic game (SG)* (e.g. [12]) consists of a set of states S that belong to either the *Maximizer* or *Minimizer* player; a set of available actions for every state, denoted $\mathsf{A}(s)$; and a probabilistic transition function δ that for a state-action pair gives a probability distribution over successor states. An SG where all states belong to one player is called *Markov decision process (MDP)*, see [34]; without nondeterministic choices, it is a *Markov chain (MC)*.

SGs are played in turns as follows: Starting in an *initial state* s_0, the player to whom this state belongs chooses an action $a_0 \in \mathsf{A}(s)$. Then, the play advances to the next state s_1, which is sampled according to the probability distribution given by $\delta(s, a)$. Repeating this process indefinitely yields an infinite path $\rho = s_0 a_0 s_1 a_1 \ldots$. We write $\mathsf{Paths}_\mathsf{G}$ for the set of all such infinite paths in a game G.

A *memoryless deterministic (MD) strategy* σ of Maximizer assigns an action to every Maximizer state s, i.e. $\sigma(s) \in \mathsf{A}(s)$. Minimizer strategies τ are defined analogously. By fixing a pair of strategies (σ, τ) and thereby resolving all nondeterministic choices, we obtain a Markov chain that together with an initial state $\hat{s}$ induces a unique probability distribution over the set of all infinite paths $\mathsf{Paths}_\mathsf{G}$ [6, Sec. 10.1]. For a random variable over paths $\varPhi : \mathsf{Paths}_\mathsf{G} \to \mathbb{R}$ we write $\mathbb{E}^{\sigma,\tau}_{\mathsf{G},\hat{s}}[\varPhi]$ for its expected value under this probability measure.

An *objective* $\varPhi : \mathsf{Paths}_\mathsf{G} \to \mathbb{R}$ formalizes the "goal" of both players by assigning a value to each path. In this paper, we focus on *mean payoff* (also called long-run average reward) [16], which assign to every path the average reward that is obtained in the limit. The presented algorithms and tools can also explicitly handle reachability/safety objectives, which compute the probability of reaching a given set of states while avoiding another. Such objectives are special cases of mean payoff, see e.g. [4]. Another prominent objective is *total reward* [11], but this is practically incompatible with our approach and goals (see [32, App. B]).

Given an SG and an objective, we want to compute the *value of the game*, i.e. the optimal value the players can ensure by choosing optimal strategies. Formally, the value of state s is defined as $\mathsf{V}_{\mathsf{G},\varPhi}(s) := \sup_\sigma \inf_\tau \mathbb{E}^{\sigma,\tau}_{\mathsf{G},s}[\varPhi] (= \inf_\tau \sup_\sigma \mathbb{E}^{\sigma,\tau}_{\mathsf{G},s}[\varPhi])$. We are interested in *approximate solutions*, i.e. given a concrete state $\hat{s}$ and precision requirement ε, our goal is to determine a number v such that $|\mathsf{V}_{\mathsf{G},\varPhi}(\hat{s}) - v| < \varepsilon$.

An *end component (EC)* intuitively is a set of states in which the system *can* remain forever, given suitable strategies. Inclusion-maximal ECs are *maximal end components (MECs)*. The play of an SG eventually remains inside a single MEC with probability one [14]. In other words, MECs capture all relevant long-run behaviour of the system. The set of MECs can be identified in PTIME [13].

3 Complete-Exploration Algorithm for Solving SGs

In this section, we very briefly recall Alg. 2 of [24], a generic value-iteration based approach for SGs, which in particular is the first such algorithm that provides guarantees on the precision for mean payoff. This recapitulation is the basis for the following descriptions of both (i) the main practical improvements over [24, Alg. 2] added in our implementation (see the end of this section); and (ii) the new partial-exploration approach described in Sec.4.

Intuition. The key insight of [24] is to "split" the analysis of SGs into infinite and transient behaviour. Infinite behaviour occurs in ECs where both players want to remain under optimal strategies; this is where the mean payoff is actually "obtained". Transient behaviour occurs in states that are not part of such an

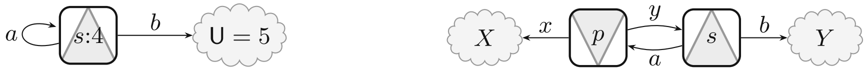


Fig. 1. Example SGs to explain deflating and inflating. States with upward and downward triangles denote Maximizer and Minimizer states, respectively.

EC, i.e. that are almost surely only visited finitely often; such states in turn achieve their value by trying to reach ECs that give them the best mean payoff.

Algorithm. Based on this intuition, we can summarize the overall structure of the algorithm. It maintains two functions L and U, which map every state to a lower and upper bound on its true value, respectively. Our aim is to improve these bounds until they are sufficiently close to each other; then we can derive the correct value up to precision ε. After initializing the bounds to safe under- and over-approximations (e.g. the minimum and maximum reward occurring in the SG), we repeatedly perform two operations, further described below: Firstly, we use so-called *Bellman updates* to back-propagate bounds through the SG, which also is the classical "value iteration step". This corresponds to the transient behaviour, intuitively computing the optimal choice of actions to reach the ECs with the best value. Secondly, we use the operations of *deflating* and *inflating*. These essentially inform states about the value obtainable by staying.

Bellman Updates. Bellman updates are at the core of all value iteration style algorithms (see e.g. [9]). Intuitively, they update the current estimates by "taking one step", i.e. computing the expectation of following the action that is optimal according to the current estimates. Formally, given a function $x \colon \mathsf{S} \to \mathbb{R}$, the Bellman update $\mathcal{B}$ computes a new estimate function as $\mathcal{B}(x)(s) := \mathsf{opt}^s_{a \in \mathsf{A}(s)} \sum_{s' \in \mathsf{S}} \delta(s,a)(s') \cdot x(s')$, where $\mathsf{opt}^s = \max$ if s is a Maximizer state and min otherwise. Importantly, if x is a correct lower or upper bound on the true value, then $\mathcal{B}(x)(s)$ is, too. However, only applying $\mathcal{B}(x)$ may not converge!

Deflating and Inflating. We briefly describe why the convergence problem arises and how deflating and inflating solve the issue, summarizing [24, Sec. V-C].

Recall the intuition that a state can get its value either from infinite behaviour (i.e. "staying" in the current area of the game) or from transient behaviour ("leaving" the current area). As a simple example, consider the SG (even MDP) in Fig. 1 (left). Assume we have $\mathsf{U} = 5$ in the grey area, i.e. by taking action b we obtain at most 5, but for s the current upper bound is the conservative over-approximation $\mathsf{U}(s) = 10$. The Bellman update prefers a over b and keeps $\mathsf{U}(s)$ at 10, even though following a forever will only yield a mean payoff of 4. This is due to a cyclic dependency: s "believes" it can (eventually) achieve 10 because it "promises" to achieve 10 by having $\mathsf{U}(s) = 10$. Deflating now identifies that Maximizer wants to "stay" in s using a, computes the actual value that is obtained by staying, i.e. 4, and compares it to the best possible exit from this region, namely leaving with b to obtain at most 5. Maximizer has to either stay forever or eventually leave, thus we can decrease the upper bound to the maximum of these two actions, i.e. 5. Dually, inflating raises the value of Minimizer states to the minimum of leaving and staying. In other words, de-/inflating complement Bellman updates by informing players about the consequences of staying forever, forcing them to choose between this staying value and the best exit.

While this reasoning is simple for single-state, single-player cycles, it gets much more involved in the general case of stochastic games. In particular, and in contrast to MDPs, in two-player SGs the opponent can restrict which cycles and exits are reachable from certain states. For example, consider the SG in Fig. 1 (right): States p and s can form a cycle. However, if s goes to p, it depends on the choice of Minimizer in p whether the play stays in the cycle $\{p, s\}$ or leaves towards X. To tackle this issue, [24, Alg. 2] repeatedly identifies regions where players want to remain *based on their current estimates*, called *simple end component (SEC)*-candidates. It does so by fixing one player's choices (pretending that this is the strategy they "commit" to), and the regions where the other player could remain are the SEC-candidates. These are then de-/inflated. We highlight that as the player's estimates change, the SEC-candidates do, too.

Improvements. We provide several practical improvements to this approach. We briefly describe their ideas here and refer to [32, App. C] for further information.

- Instead of searching SEC-candidates in the complete SG, we once identify all MECs and then search for SEC-candidates in each MEC independently.
- SEC-candidate search is not performed every iteration, but only heuristically (improving a suggestion from [15, Sec. 6.2]).
- Instead of computing staying values precisely, which is often unnecessary, we successively approximate them (as suggested in [25, App. E-B]).
- If possible, we locally employ MDP solution approaches, *collapsing* ECs that are completely "controlled" by a single player into one state that then is handled by the normal Bellman updates (as suggested in [15, Sec. 6.2]). This transparently generalizes the existing algorithms for MDPs [4,8,17].

4 Partial-Exploration Algorithm for Solving SGs

Here, we present the novel *partial-exploration (PE)* algorithm obtained by combining the *complete-exploration (CE)* algorithm of Sect. 3 with the ideas of partial exploration [8,15,29]. Intuitively, for particular models and objectives, some states are hardly relevant, and computing their exact value is unnecessary for an ε-precise result. For example, in the `zeroconf` protocol (choosing an IP address in a nearly empty network), it is hardly interesting what we should do when we run into a collision 10 times: Since this is so unlikely to happen, the exact outcome in this case barely influences the true result. Thus, we avoid exploring what exactly is possible in this case and just assume the worst.

More generally, we want to avoid working on the complete model when executing Bellman updates or de-/inflating SEC-candidates. Instead, we use simulations to partially explore the model, finding the states that are likely to be reached under optimal strategies, and focus computations on these. If the "relevant" part of the state space (see [23]) is small in comparison to the whole model, we can save a large amount of time and memory. We refer to the [8,15,29] for a comprehensive discussion of the (dis-)advantages of this approach.

Algorithm. The algorithm follows the established structure of [8,15]: We sample a path through the model, at every state picking an action and a successor according to a guidance heuristic that prefers "relevant" states. We terminate the simulation when it has reached a state where continuing the path does not generate new information (e.g. an EC that cannot be exited). Then, we perform Bellman updates, but only on the states of the sampled path. Additionally, we repeatedly identify both collapsible areas and SEC-candidates in the partial model and de-/inflate if necessary. This final step is the main technical difference to the previous algorithms [8,15], which only employed collapsing and deflating, respectively. It also is one of the major engineering difficulties, see [32, App. D.1].

Soundness and Correctness. In [32, App. D.2], we extend the proof of correctness and termination from reachability [15, Thm. 3] to mean payoff. In the process of proving correctness, we found and fixed an error in the guidance heuristic of [15].

Practical Improvements. As before, we applied several optimizations and heuristics to this algorithm. Broadly speaking, the overall ideas are the same as for the complete exploration approach, however with several intricacies. In particular, observe that the partial model constantly changes as new states are explored. Thus, efficiently tracking and updating SEC-candidates or collapsible parts of the game is much more involved and intertwined with the rest of the algorithm.

5 Tool Description

In this section, we briefly describe relevant aspects of PET2, focussing on new features and changes compared to its predecessor PET1. Like PET1, PET2 is implemented in Java, reads models and objectives specified in the PRISM modelling language, and outputs the computed value in JSON format.

Design Choices. Firstly, PET2 exclusively employs (sound) VI-based approaches, as opposed to, e.g., SI or LP. It offers two variants, based on complete exploration (CE) and partial exploration (PE), as presented above.

Secondly, PET2 deliberately comes without any configuration flags. In contrast, model checkers such as PRISM [27] and STORM [21] implement numerous different approaches and variants, each of which offers several hyper-parameters. While our choice eliminates the potential for fine-tuning, we experienced that even expert users often do not know how to best choose such parameters. Thus, we tried to select internal parameters such that the tool works reasonably well out-of-the-box on all models. This comes with the additional benefit of greatly reducing the number of code paths, which in turn makes testing much easier.

Thirdly, PET2 fully commits to solving a single combination of model and objective. This allows us to exploit information about the objective already when building the model; for example, in reachability/safety objectives, we can directly re-map goal and unsafe states to dedicated absorbing states. Arguably, this might become a drawback when solving multiple queries on the same model, since the model would need to be explored several times. However, by caching the parts of the model constructed so far, we can effectively eliminate this problem. We discuss further ways to exploit this design choice in [32, App. E].

Finally, PET2 does not differentiate between Markov chains, MDPs, and SGs. The algorithms are written for SGs, and, whenever possible, apply specialized solutions locally. For example, if a part of an SG "looks like" an MDP (because only one player has meaningful choices), PET locally applies MDP reasoning where applicable. Aside from transparently gaining the performance of these specialized solutions in most cases, this also eliminates code duplication, resulting in a code base that is more understandable, maintainable, and extendable.

Differences to PET1. PET2 effectively constitutes a complete re-write of nearly all aspects of PET, with roughly 9k lines of code added and 5k deleted compared to PET1 (according to `git`); for comparison, the overall source code of PET2 has roughly 14k lines. Most importantly, PET2 now also parses SGs, fully focusses on solving one model-objective combination, and also provides efficient CE variants of the algorithms. These CE variants also come with numerous optimizations, such as graph analysis to identify states with value 0 and 1 for reachability and safety, collapsing end components, etc. Moreover, each PE variant is now specialized to the concrete objective. PET1 tried to unify as many aspects of the sampling approach as possible, which however proved to be a major design obstacle and performance penalty when incorporating all the different specialized solutions and practical optimizations for stochastic games. Additionally, we also found and fixed a bug in the mean payoff computation of PET1. In terms of data structures, we improved several smaller aspects of working with probabilistic models. For example, the "standard" internal model representation of PET2 offers dedicated support for merging / collapsing sets of states while simultaneously tracking predecessors of each state. We note that the model representation etc. is still provided by our separately available, generic purpose library, making many of these improvements available independently of PET.

Engineering Improvements. We evaluated several practical improvements, which sometimes led to quite surprising effects. We highlight some insights which we deem relevant for other developers.

1) "Unrolling" loops in hot zones of the code can lead to significant performance improvements. For example, to find the optimal action for a Bellman update, we need to use a for-loop to iterate over all available actions. This process is

performed millions of times during a normal execution. Unrolling and specializing these loops for small action sets (≤ 3 actions) led to noticeable performance improvements. Similarly, switching from for-each loops (which allocate an iterator) to index-based for loops also yielded notable improvements.

2) We trade memory for time by adding additional data structures to optimize certain access patterns. For example, maintaining the set of predecessors for each state speeds up graph algorithms such as attractor computations. Moreover, in several cases it proved beneficial to store information in multiple formats. For example, on top of a sorted array-based set, we explicitly store the set of successors of a distribution object as a (roaring) bitmap [28], offering fast "bulk" operations, such as intersection or subset checks.
3) We also investigated ahead-of-time compilation through GraalVM. While this improved start-up time, it did not result in significant speed-ups but rather even increased the runtime on some of the larger models (even with profile-guided optimization). We conjecture that this is mainly due to Java's just-in-time compiler being able to apply better fine-tuning.

6 Experimental Evaluation

We now discuss our evaluation. The first goal of our experiments is to validate that Pet2 can indeed solve SGs with reachability and mean payoff objectives in a sound way. Secondly, we assess the impact of our performance improvements and design choices, in particular having only the general algorithm for SG, by comparing Pet1 and Pet2. Finally, we investigate whether our implementation is competitive with other tools. We report some further insights in [32, App. F.3]. Our artefact, including all data, tools, scripts, logs, etc., is available at [31].

6.1 Experimental Setup

Technical Setup. We ran each experiment in a separate Docker container and, as usual, restricted it to a single CPU core (of an AMD Ryzen 5 3600) and 8 GB RAM. The timeout is 60 s (including the startup time of the Docker container). We ran every instance three times to even out potential fluctuations in execution times. While the PE approach is randomized by design, even "deterministic" algorithms may behave differently due to, e.g., non-deterministic iteration order of hash sets. We observed that the variance is negligible (the geometric standard deviation usually was ≤ 1.05). We thus only report the geometric average of the three runs in seconds. We require an absolute precision of $\varepsilon = 10^{-6}$ for all experiments.

Metrics. To summarize relative performance of Pet2 compared to tool X, we introduce a four-figure score, written $t[m{+}k/l]$, computed as follows: Let M the set of instances *where both tools terminated in time*. Then, t equals the geometric mean of $\text{time}_{\texttt{X}}(I)/\text{time}_{\text{Pet2}}(I)$ over all instances $I \in M$, with $\text{time}_{\mathcal{T}}$ referring to the overall runtime of tool $\mathcal{T}$ on an instance, $m = |M|$ refers to the number of such instances, while k describes how often `X` timed out where PET2 did not and l vice versa. Note that $t > 1$ indicates that PET2 is faster on average (on M). When $t < 1$ but $k \gg l$, we see that on instances that both tools solved, PET2 was slower, but overall, PET2 solved much more models, which one may still consider advantageous.

Tools. Aside from both versions of PET, for Markov chains and MDPs we consider PRISM-GAMES[1] [27], and STORM [21]. On SGs, we compare to the unsound algorithms in PRISM-GAMES and TEMPEST [33] (an extension of STORM), as well as PRISM-EXT, an extension of PRISM-GAMES with sound algorithms described in [5,26]. Note that this selection includes all tools that participated in the SG performance comparison in QComp 2023 [3]. For all tools, we provide the exact version, ways to obtain them, and invocations in [32, App. F.1] and the artefact.

Performance Considerations. Restricting to a single CPU is commonly done to ensure that no tool accidentally exploits parallelism. However, we observed a significant decrease in performance for PET, even though all algorithms are sequential. This turned out to be due to garbage collection. Using `jhsdb`, we verified that in the single CPU case Java by default selects the Serial GC (instead of the overall default G1GC). On some instances, we consistently observed improvements of **up to 33%** (!), nearly on par with the performance without any CPU restriction, by simply changing to the parallel GC (`-XX:+UseParallelGC`), even though the parallel GC uses only one thread. Concretely, comparing CE and PE with Serial and (single-thread) parallel GC, we get scores of 1.06 and 1.04, respectively, meaning that even *on average* this change leads to a significant difference. Interestingly, for PRISM-GAMES the Serial GC performed better. We configured PET2 to always use the Parallel GC by default. In a similar manner, the hybrid engine of STORM experienced a slowdown of more than 30x due to being restricted to a single CPU, which we addressed by adding appropriate switches (see [32, App. F.1] for details). We are working with the authors of STORM to automatically detect this case.

While these differences would not invalidate our conclusions in particular, we still want to highlight these observations and emphasize the importance of both careful evaluation and choosing good default parameters.

Benchmarks. We consider benchmarks from multiple sources. Firstly, we include applicable models from the quantitative verification benchmark set (QVBS) [20], which however does not provide SGs. Secondly, we consider the SGs used in QComp 2023 [3]. Finally, we also gather several models from literature, provide variations of existing models, and create completely new models. For details on the models, we refer to [32, App. F.2]. All models are included in the artefact.

To ease the evaluation, we remove instances of QVBS that are very simple (CE- and PE-approach of STORM and PET2 taking less than one second) or

[1] Personal communication with the lead developer confirmed that on Markov chains and MDPs PRISM-GAMES uses the same approach as PRISM.

very time-consuming (all four approaches taking more than 30 s). With such a timespan, differences and trends are clearly visible, but models remain small enough for the experiments to be reproducible within reasonable time. This filtering reduces the number of executions from nearly 10000 to about 1800. Even then, the overall evaluation still takes about 24 h (with timeout of 60 s).

6.2 Results

We present central results in Fig. 2 and discuss each of our research questions.

Soundness and Scalability. We empirically validate the correctness of Pet2 by (i) comparing against the reference results in QVBS (this only affects the specialized MDP reasoning of our algorithm), (ii) ensuring that both algorithms inside Pet2 yield the same results, and (iii) comparing against manually computed values, both for existing SG benchmarks as well as handcrafted ones exhibiting various graph structures (some of which arose as test or corner cases). In all cases, Pet2's results are sound, i.e. within the allowed precision of $\varepsilon = 10^{-6}$. In contrast, throughout the whole SG benchmark set, Prism-games and Tempest return several wrong answers, see [32, App. F.3] for details. In particular, Tempest returns wrong answers in 6 out of 13 cases where we have known reference results, often by a significant margin, e.g. returning 0.0003 instead of 0.481.

Additionally, we see that Pet2 can solve models with millions of states and various difficult graph structures within a minute. Thus, we conclude that both our algorithm and implementation scale well.

Comparison to Pet1. When solving Markov chains or MDPs, Pet2 still uses algorithms that can handle SGs. This generality comes with some overhead, for example because data structures for tracking ownership of states are not necessary in MDPs. However, a score of 1.07[85+8/1] and Fig. 2 (left) show that PE in both versions of Pet performs remarkably similar, with Pet2 even slightly

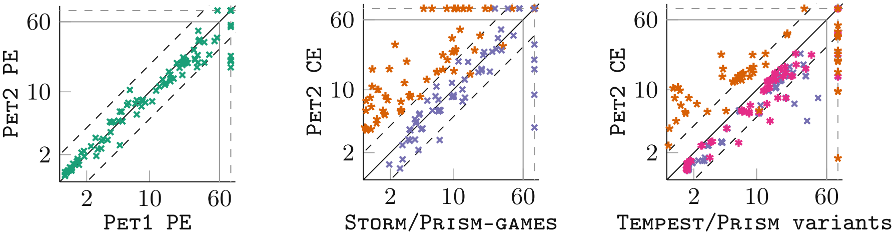


Fig. 2. Comparison of Pet2 to other tools. From left to right we compare Pet2-PE and Pet1-PE, Pet2-CE on MDP and MC with Storm (✱) and Prism-games (✖), and finally Pet2-CE on SG with Tempest (✱), Prism-games (✖), and Prism-ext (✱). A point (x, y) denotes that tool X and Pet2 needed x and y seconds, respectively. If a point is above/below the diagonal, tool X is faster/slower. Plots are on logarithmic scale, dashed diagonals indicate that one tool is twice as fast. Timeouts are pushed to the orthogonal dashed line.

faster. We conclude that the improvements in the implementation make up for the algorithmic overhead.

Comparison to other Tools. It is well known that the *structure* of a model is *the* determining factor for the relative performance of different algorithmic approaches, see e.g. [5,18,29], in particular far more than the number of states or transitions. (This is also supported by our comparison of CE and PE in [32, App. F.3], sometimes showing order of magnitude advantages in either direction.) Thus, instead of comparing tools as a whole, we compare matching algorithmic approaches (CE and PE based value/interval iteration) to assess only the impact of different implementations. For similar reasons, here we only compare the explicit engines and not symbolic or hybrid approaches (results on the latter are provided in [32, App. F.3]).

Comparing PE approaches, PET2 outperforms the only competitor STORM with a score of 0.3[27+29/1] (PET2 solves more than twice the number of models); see [32, App. F.3] for details. For CE algorithms, across all instances where both tools are applicable, the score of PET2 against the Java-based tools PRISM-GAMES and PRISM-EXT is 1.3[104+10/4] and 1.3[53+4/0], respectively, while against the C++ tools STORM and TEMPEST it achieves 0.3[69+0/12] and 0.4[54+13/1], respectively. For a more detailed comparison, Fig. 2 (middle) compares the CE algorithms of PET2 with those in STORM and PRISM-GAMES on Markov chains and MDPs. Here, as expected, STORM outperforms other tools, at least partially due to performance differences of C++ and Java. However, PET2 performs favourably against the state-of-the-art tool PRISM-GAMES. This shows that our first, generic implementation of the CE algorithm for SG is comparable to established tools even on Markov chains and MDPs. Finally, Fig. 2 (right) compares PET2 with the other tools on SGs, namely PRISM-GAMES, TEMPEST, and PRISM-EXT. Recall that only PRISM-EXT uses a sound algorithm, while the other tools use an unsound stopping criterion and thus require less work. Nonetheless, PET2 often outperforms the other tools (even TEMPEST, which builds on the highly optimized STORM), making it the most viable tool on SGs not only because of soundness, but also because of performance.

Finally, to (superficially) evaluate how much objective-specific optimizations yield, we implemented viewing reachability objectives as "trivial" mean payoff objective, i.e. goal states are set to be absorbing with reward 1, and all others with reward zero. This modified query is then passed to our generic mean payoff algorithm. Notably, even then PET2 slightly outperforms PRISM-GAMES solving the reachability objective directly (1.1[98+8/10]), and in turn the dedicated reachability approach of PET2 "only" scores 1.2[106+8/0] against this variant.

7 Conclusion

We presented PET2, the first tool implementing a sound and efficient approach for solving SGs with objectives of the type reachability/safety and mean payoff. Our experimental evaluation shows that (i) it is sound, while other tools indeed return wrong answers in practice, (ii) it offers the most efficient partial-exploration based algorithm, and (iii) it is the most viable tool on SGs.

For future work, there is still a lot of room for heuristics and engineering improvements, for example adaptively choosing internal parameters, more efficient tracking and handling of SEC-candidates, using topological order of updates in VI, improved pre-computation for mean payoff, etc. Additionally, support for total reward is planned; however, as described in [32, App. B], this requires using ideas such as optimistic value iteration [5,19] in order to be reasonably efficient.

References

1. Amir, R.: Stochastic games in economics and related fields: an overview. In: Neyman, A., Sorin, S. (eds.) Stochastic Games and Applications. NATO Science Series, vol. 570, pp. 455–470. Springer, Dordrecht (2003). https://doi.org/10.1007/978-94-010-0189-2_30
2. Andersson, D., Miltersen, P.B.: The complexity of solving stochastic games on graphs. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 112–121. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10631-6_13
3. Andriushchenko, R., et al.: Tools at the frontiers of quantitative verification: QComp 2023 competition report. TOOLympics (to appear)
4. Ashok, P., Chatterjee, K., Daca, P., Křetínský, J., Meggendorfer, T.: Value iteration for long-run average reward in Markov decision processes. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 201–221. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_10
5. Azeem, M., Evangelidis, A., Kretínský, J., Slivinskiy, A., Weininger, M.: Optimistic and topological value iteration for simple stochastic games. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) ATVA 2022. LNCS, vol. 13505, pp. 285–302. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19992-9_18
6. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
7. Bellomo, N., Delitala, M.: From the mathematical kinetic, and stochastic game theory to modelling mutations, onset, progression and immune competition of cancer cells. Phys. Life Rev. **5**(4), 183–206 (2008). https://doi.org/10.1016/j.plrev.2008.07.001
8. Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
9. Chatterjee, K., Henzinger, T.A.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_7
10. Chatterjee, K., Meggendorfer, T., Saona, R., Svoboda, J.: Faster algorithm for turn-based stochastic games with bounded treewidth. In: SODA, pp. 4590–4605. SIAM (2023). https://doi.org/10.1137/1.9781611977554.CH173
11. Chen, T., Forejt, V., Kwiatkowska, M.Z., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Formal Methods Syst. Des. **43**(1), 61–92 (2013). https://doi.org/10.1007/s10703-013-0183-7

12. Condon, A.: On algorithms for simple stochastic games. In: Advances In Computational Complexity Theory. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 13, pp. 51–71. DIMACS/AMS (1990)
13. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995)
14. De Alfaro, L.: Formal verification of probabilistic systems. Ph.D. thesis, Stanford University (1997)
15. Eisentraut, J., Kelmendi, E., Kretínský, J., Weininger, M.: Value iteration for simple stochastic games: stopping criterion and learning algorithm. Inf. Comput. **285**(Part), 104886 (2022). https://doi.org/10.1016/j.ic.2022.104886
16. Gillette, D.: Stochastic games with zero stop probabilities. Contrib. Theory Games **3**, 179–187 (1957)
17. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)
18. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13993, pp. 469–488. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_24
19. 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
20. 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
21. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022). https://doi.org/10.1007/S10009-021-00633-Z
22. Johnson, D.S.: The NP-completeness column: finding needles in haystacks. ACM Trans. Algorithms **3**(2), 24 (2007). https://doi.org/10.1145/1240233.1240247
23. Kretínský, J., Meggendorfer, T.: Of cores: a partial-exploration framework for Markov decision processes. Log. Methods Comput. Sci. **16**(4) (2020). https://lmcs.episciences.org/6833
24. Kretínský, J., Meggendorfer, T., Weininger, M.: Stopping criteria for value iteration on stochastic games with quantitative objectives. In: LICS, pp. 1–14 (2023). https://doi.org/10.1109/LICS56636.2023.10175771
25. Kretínský, J., Meggendorfer, T., Weininger, M.: Stopping criteria for value iteration on stochastic games with quantitative objectives. CoRR abs/2304.09930 (2023)
26. Kretínský, J., Ramneantu, E., Slivinskiy, A., Weininger, M.: Comparison of algorithms for simple stochastic games. Inf. Comput. **289**(Part), 104885 (2022). https://doi.org/10.1016/j.ic.2022.104885
27. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: PRISM-games 3.0: stochastic game verification with concurrency, equilibria and time. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 475–487. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_25
28. Lemire, D., Kai, G.S.Y., Kaser, O.: Consistently faster and smaller compressed bitmaps with roaring. SPE **46**(11), 1547–1569 (2016)
29. Meggendorfer, T.: PET - a partial exploration tool for probabilistic verification. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) ATVA 2022. LNCS, vol. 13505, pp. 320–326. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19992-9_20

30. Meggendorfer, T., Weininger, M.: Partial exploration tool gitlab. https://gitlab.lrz.de/i7/partial-exploration
31. Meggendorfer, T., Weininger, M.: Artifact for "Partial Exploration Tool 2.0" (2024). https://doi.org/10.5281/zenodo.10927672
32. Meggendorfer, T., Weininger, M.: Playing games with your pet: extending the partial exploration tool to stochastic games. CoRR abs/2405.03885 (2024). https://doi.org/10.48550/ARXIV.2405.03885
33. Pranger, S., Könighofer, B., Posch, L., Bloem, R.: TEMPEST - synthesis tool for reactive systems and shields in probabilistic environments. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 222–228. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_15
34. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887
35. Roy, S., Ellis, C., Shiva, S.G., Dasgupta, D., Shandilya, V., Wu, Q.: A survey of game theory as applied to network security. In: HICSS, pp. 1–10. IEEE Computer Society (2010). https://doi.org/10.1109/HICSS.2010.35
36. Weininger, M.: Solving Stochastic Games Reliably. Ph.D. thesis, Technical University of Munich, Germany (2022). https://mediatum.ub.tum.de/node?id=1661588

What Should Be Observed for Optimal Reward in POMDPs?

Alyzia-Maria Konsta(✉), Alberto Lluch Lafuente, and Christoph Matheja

Technical University of Denmark, Kongens Lyngby,
Denmark
{akon,albl,chmat}@dtu.dk

Abstract. Partially observable Markov Decision Processes (POMDPs) are a standard model for agents making decisions in uncertain environments. Most work on POMDPs focuses on synthesizing strategies based on the available capabilities. However, system designers can often control an agent's observation capabilities, e.g. by placing or selecting sensors. This raises the question of how one should select an agent's sensors cost-effectively such that it achieves the desired goals. In this paper, we study the novel *optimal observability problem* (OOP): Given a POMDP $\mathscr{M}$, how should one change $\mathscr{M}$'s observation capabilities within a fixed budget such that its (minimal) expected reward remains below a given threshold? We show that the problem is undecidable in general and decidable when considering positional strategies only. We present two algorithms for a decidable fragment of the OOP: one based on optimal strategies of $\mathscr{M}$'s underlying Markov decision process and one based on parameter synthesis with SMT. We report promising results for variants of typical examples from the POMDP literature.

Keywords: POMDPs · Partial Observability · Probabilistic Model Checking

1 Introduction

Partially observable Markov Decision Processes (POMDPs) [1,15,28] are the reference model for agents making decisions in uncertain environments. They appear naturally in various application domains, including software verification [4], planning [5,15,16], computer security [24], and cyber-physical systems [13].

Most work on POMDPs focuses on synthesizing strategies for making decisions based on available observations. A less explored, but relevant, question for system designers is how to place or select sensors cost-effectively such that they suffice to achieve the desired goals.

To illustrate said question, consider a classical grid(world) POMDP [21], (cf. Fig. 1), where an agent is placed randomly on one of the states s_0 - s_7. The agent's goal is to reach the *goal state* s_8 (indicated with green color). The agent is free to move

This work has been supported by Innovation Fund Denmark and the Digital Research Centre Denmark, through the bridge project "SIOT – Secure Internet of Things – Risk analysis in design and operation".

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 373–394, 2024.
https://doi.org/10.1007/978-3-031-65633-0_17

through the grid using the actions $\{left, right, up, down\}$. For simplicity, self-loops are omitted and the actions are only shown for state s_4. We assume that every time the agent picks an action, (s)he takes one step. With an unlimited budget, one can achieve full observability by attaching one sensor to each state. In this case, the minimum expected number of steps the agent should take to reach the goal is 2.25.

However, the number of available sensors might be limited. Can we achieve the same optimal reward with fewer sensors? What is the minimal number of sensors needed? Where should they be located? It turns out that, in this example, 2 sensors (one in s_2 and one in s_5) suffice to achieve the minimal expected number of steps, i.e., 2.25. Intuitively, the agent just needs a simple positional (aka memory-less), deterministic strategy: if no sensor is present, go right; otherwise, go down. A "symmetric" solution would be to place the sensors in s_6 and s_7. Any other choice for placing 2 sensors yields a higher expected number of steps. For example, placing the sensors in s_1 and s_2 yields a minimal expected number of steps of 2.75. The problem easily becomes more complex. Indeed, we show that this class of problems (our main focus of study) is undecidable.

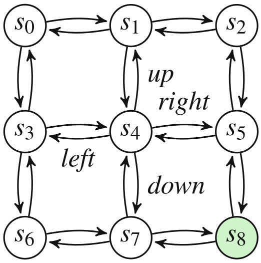


Fig. 1. 3×3 grid

The Problem. We introduce the *optimal observability problem* which is concerned with turning an MDP M into a POMDP $\mathcal{M}$ such that $\mathcal{M}$'s expected reward remains below a given threshold and, at the same time, the number of available observations (i.e. classes of observationally-equivalent states) is limited by a given budget. We show that the problem is undecidable in the general case, by reduction to the (undecidable) policy-existence problem for POMDPs [22]. Consequently, we focus on decidable variants of the problem, where POMDPs can use positional strategies only, for which we provide complexity results, decision procedures, and experiments.

Contributions. Our main contributions can be summarized as follows:

1. We introduce the novel *optimal observability problem* (**OOP**) and show that it is undecidable in general (Sect. 3) by reduction to the (undecidable) policy-existence problem for POMDPs [22]. Consequently, we study four decidable **OOP** variants by restricting the considered strategies and observation capabilities.
2. We show in Sect. 4.1 that, when restricted to *positional and deterministic* strategies, the **OOP** becomes NP-complete. Moreover, we present an algorithm that uses optimal MDP strategies to determine the minimal number of observations required to solve the **OOP** for the optimal threshold.
3. We show in Sect. 4.2 that the **OOP** becomes decidable in PSPACE if restricted to positional, but *randomized*, strategies. The proofs are by a reduction to the feasibility problem for a typed extension of parametric Markov chains [12,14].
4. We provide in Sect. 5 an experimental evaluation of approaches for the decidable **OOP** variants on common POMDP benchmarks.

Missing proofs and additional experiments are found in an extended version of this paper, which is available online [17].

Related Work. To the best of our knowledge, this is the first work considering the *optimal observability problem* and its variants. The closest problem we have found in the literature is the sensor synthesis problem for POMDPs with reachability objectives presented in [6]. Said problem departs from a POMDP with a partially defined observation function and consists of finding a completion of the function by adding additional observations subject to a budget on the number of observations (as in our case) and the size of the (memory-bounded) strategies. The main difference w.r.t. our problem is in the class of POMDP objectives considered. The problem in [6] focuses on *qualitative* almost-sure reachability properties (which is decidable for POMDPs [7]), while we focus on *quantitative* optimal expected reward properties, which are generally undecidable for POMDPs [22]. This leads to different complexity results for the decision problem studied (NP-complete for [6], undecidable in our general case) and their solution methods (SAT-based in [6], SMT-based for the decidable variants we study).

Optimal placement or selection of sensors has been studied before (e.g. [18,26,30]). However, the only work we are aware of in this area that uses POMDPs is [30]. The problem studied in [30] is concerned with having a POMDP where the selection of k out of n sensors is part of the set of states of the POMDP together with the location of some entities in a 2D environment. At each state, the agent controlling the sensors has one of the $\binom{n}{k}$ choices to select the next k active sensors. The goal is to synthesize and find strategies that dynamically select those sensors that optimize some objective function (e.g. increasing certainty of specific state properties). The observation function in the POMDPs used in [30] is fixed whereas we aim at synthesizing said function. The same holds for security applications of POMDPs, such as [29].

We discuss further related work, particularly about decidability results for POMDPs, parametric Markov models (cf. [12]), and related tools in the relevant subsections.

2 Preliminaries

We briefly introduce the formal models underlying our problem statements and their solution: Markov decision processes (MDPs) in Sect. 2.1 and partially observable MDPs (POMDPs) in Sect. 2.2. A comprehensive treatment is found in [2,28].

Notation. A *probability distribution* over a countable set X is a function $\mu : X \to [0,1] \subseteq \mathbb{R}$ such that the (real-valued) probabilities assigned to all elements of X sum up to one, i.e. $\sum_{x \in X} .\mu(x) = 1$. For example, the *Dirac distribution* δ_x assigns probability 1 to an a-priori selected element $x \in X$ and probability 0 to all other elements. We denote by $\mathit{Dist}(X)$ the set of all probability distributions over X.

2.1 Markov Decision Processes (MDPs)

We first recap Markov decision processes with rewards and dedicated goal states.

Definition 1 (MDPs). *A* Markov decision process *is a tuple* $M = (S, I, G, \mathit{Act}, P, \mathit{rew})$ *where* S *is a finite set of* states, $I \subseteq S$ *is a set of (uniformly distributed)* initial states, $G \subseteq S$ *is a set of* goal states*,* Act *is a finite set of* actions*,* $P\colon S \times \mathit{Act} \to \mathit{Dist}(S)$ *is a* transition probability function*, and* $\mathit{rew}\colon S \to \mathbb{R}_{\geq 0}$ *is a* reward function.

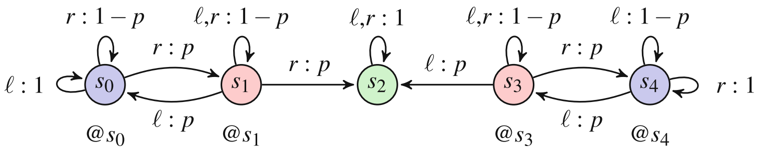


Fig. 2. MDP M_{line} for some fixed constant $p \in [0,1]$; the initial states are s_0, s_1, s_3, s_4.

Example 1. As a running example, consider an agent that is placed at one of four random locations on a line. The agent needs to reach a goal by moving to the ℓ(eft) or r(ight). Whenever (s)he decides to move, (s)he successfully does so with some fixed probability $p \in [0,1]$; with probability $1-p$, (s)he stays at the current location due to a failure. Figure 2 depicts[1] an MDP M_{line} modeling the above scenario using five states s_0-s_4. Here, s_2 is the single goal state. All other states are initial. An edge $s_i \xrightarrow{\alpha:q} s_j$ indicates that $P(s_i,\alpha)(s_j) = q$. The reward (omitted in Fig. 2) is 0 for s_2, and 1 for all other states.

We often write S_M, P_M, and so on, to refer to the components of an MDP M. An MDP M is a *Markov chain* if there are no choices between actions, i.e. $|Act_M| = 1$. We omit the set of actions when considering Markov chains. If there is more than one action, we use *strategies* to resolve all non-determinism.

Definition 2 (Strategy). *A* strategy *for MDP M is a function $\sigma \colon S_M^+ \to Dist(Act_M)$ that maps non-empty finite sequences of states to distributions over actions. We denote by $\mathfrak{S}(M)$ the set of all strategies for MDP M.*

Expected rewards. We will formalize the problems studied in this paper in terms of the (minimal) expected reward $\mathsf{MinExpRew}(M)$ accumulated by an MDP M over all paths that start in one of its initial states and end in one of its goal states. Towards a formal definition, we first define paths. A *path fragment* of an MDP M is a finite sequence $\pi = s_0\, \alpha_0\, s_1\, \alpha_1\, s_2 \ldots \alpha_{n-1}\, s_n$ for some natural number n such that every transition from one state to the next one can be taken for the given action with non-zero probability, i.e. $P_M(s_i,\alpha_i)(s_{i+1}) > 0$ holds for all $i \in \{0,\ldots,n\}$. We denote by *first*$(\pi) = s_0$ (resp. *last*$(\pi) = s_n$) the first (resp. last) state in π. Moreover, we call π a *path* if s_0 is an initial state, i.e. $s_0 \in I_M$, and $s_n \in G_M$ is the first encountered goal state, i.e. $s_1,\ldots,s_{n-1} \in S_M \setminus G_M$ and $s_n \in G_M$. We denote by *Paths*(M) the set of all paths of M.

The *cumulative reward* of a path fragment $\pi = s_0\, \alpha_0 \ldots \alpha_{n-1}\, s_n$ of M is the sum of all rewards along the path, that is,

$$rew_M(\pi) \;=\; \sum_{i=0}^{n} rew_M(s_i).$$

Furthermore, for a given strategy σ, the *probability* of the above path fragment π is[2]

$$P_M^{\sigma}(\pi) \;=\; \prod_{i=0}^{n-1} P_M(s_i,\alpha_i)(s_{i+1}) \cdot \sigma(s_0 \ldots s_i)(\alpha_i).$$

[1] The red and blue colors as well as the @s-labels will become relevant later.

[2] If π is a path, notice that our definition does *not* include the probability of starting in *first*(π).

Put together, the expected reward of M for strategy σ is the sum of rewards of all paths weighted by their probabilities and divided by the number of initial states (since we assume a uniform initial distribution) – at least as long as the goal states are reachable from the initial states with probability one; otherwise, the expected reward is infinite (cf. [2]). Formally, $\mathsf{ExpRew}^{\sigma}(M) = \infty$ if $\frac{1}{|I_M|} \cdot \sum_{\pi \in Paths(M)} P_M^{\sigma}(\pi) < 1$. Otherwise,

$$\mathsf{ExpRew}^{\sigma}(M) \;=\; \frac{1}{|I_M|} \cdot \sum_{\pi \in Paths(M)} P_M^{\sigma}(\pi) \cdot rew_M(\pi).$$

The *minimal expected reward* of M (over an infinite horizon) is then the infimum among the expected rewards for all possible strategies, that is,

$$\mathsf{MinExpRew}(M) \;=\; \inf_{\sigma \in \mathfrak{S}(M)} \mathsf{ExpRew}^{\sigma}(M).$$

The (maximal) expected reward is defined analogously by taking the supremum instead of the infimum. Throughout this paper, we focus on minimal expected rewards.

Optimal, positional, and deterministic strategies. In general, strategies may choose actions randomly and based on the history of previously encountered states. We will frequently consider three subsets of strategies. First, a strategy σ for M is *optimal* if it yields the minimal expected reward, i.e. $\mathsf{ExpRew}^{\sigma}(M) = \mathsf{MinExpRew}(M)$. Second, a strategy is *positional* if actions depend on the current state only, i.e. $\sigma(ws) = \sigma(s)$ for all $w \in S_M^*$ and $s \in S_M$. Third, a strategy is *deterministic* if the strategy always chooses exactly one action, i.e. for all $w \in S_M^+$ there is an $a \in Act_M$ such that $\sigma(w) = \delta_a$.

Example 2 (cntd.). An optimal, positional, and deterministic strategy σ for the MDP M_{line} (Fig. 2) chooses action r(ight) for states s_0, s_1 and ℓ(eft) for s_3, s_4. For $p = 2/3$, the (minimal) expected number of steps until reaching s_2 is $\mathsf{ExpRew}^{\sigma}(M_{\text{line}}) = 3$.

Every positional strategy for an MDP M induces a Markov chain over the same states.

Definition 3 (Induced Markov Chain). *The* induced Markov chain *of an MDP M and a positional strategy σ of M is given by $M[\sigma] = (S_M, I_M, G_M, P, rew_M)$, where the transition probability function P is given by*

$$P(s,s') \;=\; \sum_{\alpha \in Act_M} P_M(s,\alpha)(s') \cdot \sigma(s)(a).$$

For MDPs, there always exists an optimal strategy that is also positional and deterministic (cf. [2]). Hence, the minimal expected reward of such an MDP M can alternatively be defined in terms of the expected rewards of its induced Markov chains:

$$\mathsf{MinExpRew}(M) \;=\; \min\{\, \mathsf{ExpRew}(M[\sigma]) \mid \sigma \in \mathfrak{S}(M) \text{ positional}\}$$

2.2 Partially Observable Markov Decision Processes

A partially observable Markov decision process (POMDP) is an MDP with imperfect information about the current state, that is, certain states are indistinguishable.

Definition 4 (POMDPs). *A* partially observable Markov decision process *is a tuple* $\mathscr{M} = (M, O, obs)$*, where* $M = (S, I, G, Act, P, rew)$ *is an MDP,* O *is a finite set of* observations*, and* $obs\colon S \to O \uplus \{\checkmark\}$ *is an* observation function *such that* $obs(s) = \checkmark$ *iff* $s \in G$*.*[3]

For simplicity, we use a dedicated observation $\checkmark$ for goal states and only consider observation functions of the above kind. We write $O_{\checkmark}$ as a shortcut for $O \uplus \{\checkmark\}$.

Example 3 (cntd.). The colors in Fig. 2 indicate a POMDP obtained from M_{line} by assigning observations o_1 to s_0 and s_4, o_2 to s_1 and s_4, and $\checkmark$ to s_2. Hence, we know how far away from the goal state s_2 we are but not which action leads to the goal.

In a POMDP $\mathscr{M}$, we assume that we cannot directly see a state, say s, but only its assigned observation $obs_{\mathscr{M}}(s)$ – all states in $obs^{-1}_{\mathscr{M}}(o) = \{s \mid obs_{\mathscr{M}}(s) = o\}$ thus become indistinguishable. Consequently, multiple path fragments in the underlying MDP M might also become indistinguishable. More formally, the *observation path fragment* $obs_{\mathscr{M}}(\pi)$ of a path fragment $\pi = s_0\alpha_0 s_1 \ldots s_n \in Paths(M)$ is defined as

$$obs_{\mathscr{M}}(\pi) \;=\; obs_{\mathscr{M}}(s_0)\,\alpha_0\, obs_{\mathscr{M}}(s_1)\,\ldots\, obs_{\mathscr{M}}(s_n).$$

We denote by $OPaths(\mathscr{M})$ the set of all observation paths obtained from the paths of $\mathscr{M}$'s underlying MDP M, i.e. $OPaths(\mathscr{M}) = \{obs_{\mathscr{M}}(\pi) \mid \pi \in Paths(M)\}$. Strategies for POMDPs are defined as for their underlying MDPs. However, POMDP strategies must be *observation-based*, that is, they have to make the same decisions for path fragments that have the same observation path fragment.

Definition 5 (Observation-Based Strategies). *An* observation-based strategy σ *for a POMDP* $\mathscr{M} = (M, O, obs)$ *is a function* $\sigma : OPaths(\mathscr{M}) \to Dist(Act_M)$ *such that:*

- *σ is a strategy for the MDP M, i.e. $\sigma \in \mathfrak{S}(M)$ and*
- *for all path fragments $\pi = s_0\alpha_0 s_1 ... s_n$ and $\pi' = s'_0\alpha'_0 s'_1 ... s'_n$, if $obs(\pi) = obs(\pi')$, then $\sigma(s_0 s_1 \ldots s_n) = \sigma(s'_0 s'_1 \ldots s'_n)$.*

We denote by $\mathfrak{O}(\mathscr{M})$ the set of all observation-based strategies of $\mathscr{M}$. The *minimal expected reward* of a POMDP $\mathscr{M} = (M, O, obs)$ is defined analogously to the expected reward of the MDP M when considering only observation-based strategies:

$$\mathsf{MinExpRew}(\mathscr{M}) \;=\; \inf_{\sigma \in \mathfrak{O}(\mathscr{M})} \mathsf{ExpRew}^{\sigma}(\mathscr{M}).$$

Strategies for POMDPs. Optimal, positional, and deterministic observation-based strategies for POMDPs are defined analogously to their counterparts for MDPs. Furthermore, given a positional strategy σ, we denote by $\mathscr{M}[\sigma] = (M_{\mathscr{M}}[\sigma], O_{\mathscr{M}}, obs_{\mathscr{M}})$ the POMDP in which the underlying MDP M is changed to the Markov chain induced by M and σ.

When computing expected rewards, we can view a POMDP as an MDP whose strategies are restricted to observation-based ones. Hence, the minimal expected reward of a POMDP is always greater than or equal to the minimal expected reward of its underlying MDP. In particular, if there is one observation-based strategy that is also optimal for the MDP, then the POMDP and the MDP have the same expected reward.

[3] Here, $A \uplus B$ denotes the union $A \cup B$ of sets A and B if $A \cap B = \emptyset$; otherwise, it is undefined.

Example 4 (cntd.). Consider the POMDP $\mathscr{M}$ in Fig. 2 for $p = 1/2$. For the underlying MDP M_{line}, we have $\mathsf{MinExpRew}(M_{\text{line}}) = 4$. Since we cannot reach s_2 from s_1 and s_3 by choosing the same action, every positional and deterministic observation-based strategy σ yields $\mathsf{ExpRew}^{\sigma}(\mathscr{M}) = \infty$. An observation-based positional strategy σ' can choose each action with probability $1/2$, which yields $\mathsf{ExpRew}^{\sigma'}(\mathscr{M}) = 10$. Moreover, for deterministic, but not necessarily positional, strategies, $\mathsf{MinExpRew}(\mathscr{M}) \approx 4.74$.[4]

Notation for (PO)MDPs. Given a POMDP $\mathscr{M} = (M, O, obs)$ and an observation function $obs' \colon S_M \to O'_{\checkmark}$, we denote by $\mathscr{M}\langle obs' \rangle$ the POMDP obtained from $\mathscr{M}$ by setting the observation function to obs', i.e. $\mathscr{M}\langle obs' \rangle = (M, O', obs')$. We call $\mathscr{M}$ *fully observable* if all states can be distinguished from one another, i.e. $s_1 \neq s_2$ implies $obs(s_1) \neq obs(s_2)$ for all $s_1, s_2 \in S_M$. Throughout this paper, we do not distinguish between a fully-observable POMDP $\mathscr{M}$ and its underlying MDP M. Hence, we use notation introduced for POMDPs, such as $\mathscr{M}\langle obs' \rangle$, also for MDPs.

3 The Optimal Observability Problem

We now introduce and discuss observability problems of the form *"what should be observable for a POMDP such that a property of interest can still be guaranteed?"*.

As a simple example, assume we want to turn an MDP M into a POMDP $\mathscr{M} = (M, O, obs)$ by selecting an observation function $obs \colon S_M \to O_{\checkmark}$ such that M and $\mathscr{M}$ have the same expected reward, that is, $\mathsf{MinExpRew}(M) = \mathsf{MinExpRew}(\mathscr{M})$. Since every MDP is also a POMDP, this problem has a trivial solution: We can introduce one observation for every non-goal state, i.e. $O = (S_M \setminus G_M)$, and encode full observability, i.e. $obs(s) = s$ if $s \in S_M \setminus G_M$ and $obs(s) = \checkmark$ if $s \in G_M$. However, we will see that the above problem becomes significantly more involved if we add objectives or restrict the space of admissible observation functions obs.

In particular, we will define in Sect. 3.1 the *optimal observability problem* which is concerned with turning an MDP M into a POMDP $\mathscr{M}$ such that $\mathscr{M}$'s expected reward remains below a given threshold and, at the same time, the number of available observations, i.e. how many non-goal states can be distinguished with certainty, is limited by a budget. In Sect. 3.2, we show that the problem is undecidable.

3.1 Problem Statement

Formally, the optimal observability problem is the following decision problem:

Definition 6 (Optimal Observability Problem (OOP)). *Given an MDP M, a budget $B \in \mathbb{N}_{\geq 1}$, and a (rational) threshold $\tau \in \mathbb{Q}_{\geq 0}$, is there an observation function $obs \colon S_M \to O_{\checkmark}$ with $|O| \leq B$ such that $\mathsf{MinExpRew}(M\langle obs \rangle) \leq \tau$?*

Example 5 (cntd.). Recall from Fig. 2 the MDP M_{line} and consider the OOP-instance $(M_{\text{line}}, B, \tau)$ for $p = 1/2$, $B = 2$, and $\tau = \mathsf{MinExpRew}(M_{\text{line}}) = 4$. As discussed in Example 4, the observation function given by the colors in Fig. 2 is *not* a solution. However,

[4] Approximate solution provided by PRISM's POMDP solver.

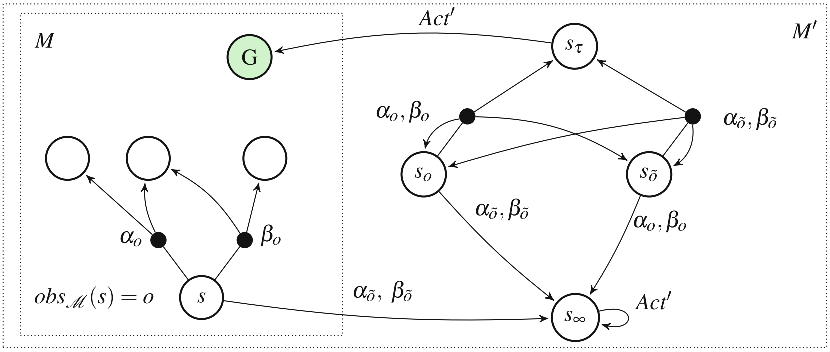

M
M′
G
Act′
s_τ
α_o, β_o
$\alpha_{\tilde{o}}, \beta_{\tilde{o}}$
s_o
$s_{\tilde{o}}$
$\alpha_{\tilde{o}}, \beta_{\tilde{o}}$
α_o, β_o
α_o
β_o
$obs_{\mathscr{M}}(s) = o$
s
$\alpha_{\tilde{o}},\ \beta_{\tilde{o}}$
s_∞
Act′

Fig. 3. Sketch of the MDP M' constructed in the undecidability proof (Theorem 1). We assume the original POMDP $\mathscr{M}$ uses actions $Act = \{\alpha, \beta\}$ and observations $O = \{o, \tilde{o}\}$. Edges with a black dot indicate the probability distribution selected for the action(s) next to it. Edges without a black dot are taken with probability one. Concrete probabilities, rewards, and transitions of unnamed states have been omitted for simplicity.

there is a solution: For $obs(s_0) = obs(s_1) = o_1$, $obs(s_2) = \checkmark$, and $obs(s_2) = obs(s_3) = o_2$, we have $\mathsf{MinExpRew}(M_{\text{line}}\langle obs\rangle) = 4$, because the optimal strategy for M_{line} discussed in Example 2 is also observation-based for $M_{\text{line}}\langle obs\rangle$.

3.2 Undecidability

We now show that the optimal observability problem (Definition 6) is undecidable.

Theorem 1 (Undecidability). *The optimal observability problem is undecidable.*

The proof is by reduction to the policy-existence problem for POMDPs [22].

Definition 7 (Policy-Existence Problem). *Given a POMDP $\mathscr{M}$ and a rational threshold $\tau \in \mathbb{Q}_{\geq 0}$, does $\mathsf{MinExpRew}(\mathscr{M}) \leq \tau$ hold?*

Proposition 1 (Madani et al. [22]). *The policy-existence problem is undecidable.*

Proof (of Theorem 1). By reduction to the policy-existence problem. Let $(\mathscr{M}, \tau)$ be an instance of said problem, where $\mathscr{M} = (M, O, obs)$ is a POMDP, $M = (S, I, G, Act, P, rew)$ is the underlying MDP, and $\tau \in \mathbb{Q}_{\geq 0}$ is a threshold. Without loss of generality, we assume that G is non-empty and that $|range(obs)| = |O| + 1$, where $range(obs) = \{obs(s) \mid s \in S\}$. We construct an **OOP**-instance (M', B, τ), where $B = |O|$, to decide whether $\mathsf{MinExpRew}(\mathscr{M}) \leq \tau$ holds.

Construction of M'. Figure 3 illustrates the construction of M'; a formal definition is found in Fig. 4. Our construction extends M in three ways: First, we add a sink state s_∞ such that reaching s_∞ with some positive probability immediately leads to an infinite total expected reward. Second, we add a new initial state s_o for every observation $o \in O$. Those new initial states can only reach each other, the sink state s_∞, or the goal

$$\begin{aligned} M' &= (S', I', G, Act', P', rew') \\ S' &= S \uplus S_O \uplus \{s_\infty, s_\tau\} \\ S_O &= \{s_o \mid o \in O\} \\ I' &= I \uplus S_O \\ Act' &= \biguplus_{o \in O} Act_o \\ Act_o &= \{\alpha_o \mid \alpha \in Act\} \end{aligned} \qquad P'(s, \alpha_o) = \begin{cases} P(s,\alpha), & \text{if } s \in (S \setminus G)\text{ , } obs(s) = o \\ P(s,\alpha), & \text{if } s \in G \\ unif(S_O \cup \{s_\tau\}), & \text{if } s = s_o \in S_O \\ unif(G), & \text{if } s = s_\tau \\ \delta_{s_\infty}, & \text{otherwise} \end{cases}$$

$$rew'(s) = \begin{cases} rew(s), & \text{if } s \in S \\ 0, & \text{if } s \in S_O \\ 1, & \text{if } s = s_\infty \\ \tau & \text{if } s = s_\tau \end{cases}$$

Fig. 4. Formal construction of the MDP M' in the proof of Theorem 1. Here, M' is derived from the POMDP $\mathscr{M} = (M, O, obs)$, where $M = (S, I, G, Act, P, rew)$. Moreover, *uniform*$(S'')$ assigns probability $1/|S''|$ to states in S'' and probability 0, otherwise.

states via the new state s_τ. Third, we *tag* every action $\alpha \in Act$ with an observation from O, i.e. for all $\alpha \in Act$ and $o \in O$, we introduce an action α_o. For every state $s \in S \setminus G$, taking actions tagged with $obs(s)$ behaves as in the original POMDP $\mathscr{M}$. Taking an action with any other tag leads to the sink state. Intuitively, strategies for M' thus have to pick actions with the same tags for states with the same observation in $\mathscr{M}$. However, it could be possible that a different observation function than the original obs could be chosen. To prevent this, every newly introduced initial state s_o (for each observation $o \in O$) leads to s_∞ if we take an action that is not tagged with o. Each s_o thus represents one observable, namely o. To rule out observation functions with less than $|O|$ observations, our transition probability function moves from every new initial state $s_o \in S_O$ to every $s'_o \in S_O$ and to s_τ with some positive probability (uniformly distributed for convenience). If we would assign the same observation to two states in $s_o, s'_o \in S_O$, then there would be two identical observation-based paths to s_o and s'_o. Hence, any observation-based strategy inevitably has to pick an action with a tag that leads to s_∞. In summary, the additional initial states enforce that – up to a potential renaming of observations – we have to use the *same* observation function as in the original POMDP.

Clearly, the MDP M' is computable (even in polynomial time). Our construction also yields a correct reduction (see [17, Appendix A.1] for details), i.e. we have

$$\underbrace{\mathsf{MinExpRew}(\mathscr{M}) \leq \tau}_{\text{policy-existence problem}} \iff \underbrace{\exists obs' \colon \mathsf{MinExpRew}(M'\langle obs' \rangle) \leq \tau.}_{\text{optimal observability problem, where } |range(obs')| \leq |O_{\checkmark}|} \qquad \square$$

4 Optimal Observability for Positional Strategies

Since the optimal observability problem is undecidable in general (cf. Theorem 1), we consider restricted versions. In particular, we focus on *positional* strategies throughout the remainder of this paper. We show in Sect. 4.1 that the optimal observability problem becomes NP-complete when restricted to positional *and* deterministic strategies. Furthermore, one can determine the minimal required budget that still yields the exact minimal expected reward by analyzing the underlying MDP (Sect. 4.1). In Sect. 4.2, we explore variants of the optimal observability problem, where the budget is lower than the minimal required one. We show that an extension of parameter synthesis techniques can be used to solve those variants.

4.1 Positional and Deterministic Strategies

We now consider a version of the optimal observability problem in which only positional and deterministic strategies are taken into account. Recall that a positional and deterministic strategy for $\mathscr{M}$ assigns one action to every state, i.e. it is of the form $\sigma\colon S_{\mathscr{M}} \to Act_{\mathscr{M}}$. Formally, let $\mathfrak{S}_{pd}(\mathscr{M})$ denote the set of all positional and deterministic strategies for $\mathscr{M}$. The minimal expected reward over strategies in $\mathfrak{S}_{pd}(\mathscr{M})$ is

$$\mathsf{MinExpRew}_{pd}(\mathscr{M}) \;=\; \inf_{\sigma \in \mathfrak{S}_{pd}(\mathscr{M})} \mathsf{ExpRew}^{\sigma}(M).$$

The *optimal observability problem for positional and deterministic strategies* is then defined as in Definition 6, but using $\mathsf{MinExpRew}_{pd}(\mathscr{M})$ instead of $\mathsf{MinExpRew}(\mathscr{M})$:

Definition 8 (Positional Deterministic Optimal Observability Problem (PDOOP)). *Given an MDP M, $B \in \mathbb{N}_{\geq 1}$, and $\tau \in \mathbb{Q}_{\geq 0}$, does there exist an observation function $obs\colon S_M \to O_{\checkmark}$ with $|O| \leq B$ such that $\mathsf{MinExpRew}_{pd}(\mathscr{M}\langle obs\rangle) \leq \tau$?*

Example 6 (ctnd.). Consider the **PDOOP**-instance $(M_{\text{line}}, 2, 4)$, where M_{line} is the MDP in Fig. 2 for $p = 1/2$. Then there is a solution by assigning the observation o_1 to s_0 and s_1 (and moving r(ight) for o_1), and o_2 to s_3 and s_4 (and moving ℓ(eft) for o_2).

Analogously, we restrict the policy-existence problem (cf. Definition 7) to positional and deterministic strategies.

Definition 9 (Positional Deterministic Policy-Existence Problem (PDPEP)). *Given a POMDP $\mathscr{M}$ and $\tau \in \mathbb{Q}_{\geq 0}$, does $\mathsf{MinExpRew}_{pd}(\mathscr{M}) \leq \tau$ hold?*

Proposition 2 (Sec. 3 from [20]). PDPEP *is* NP*-complete.*

NP-hardness of **PDOOP** then follows by a reduction from **PDPEP**, which is similar to the reduction in our undecidability proof for arbitrary strategies (cf. Theorem 1). In fact, **PDOOP** is not only NP-hard but also in NP.

Theorem 2 (NP-completeness). PDOOP *is* NP*-complete.*

Proof (Sketch). To see that **PDOOP** is in NP, consider a **PDOOP**-instance (M, B, τ). We guess an observation function $obs\colon S_M \to \{1, \ldots, |B|\} \uplus \{\checkmark\}$ and a positional and deterministic strategy $\sigma\colon S_M \to Act_M$. Both are clearly polynomial in the size of M and B. Then obs is a solution for the **PDOOP**-instance (M, B, τ) iff (a) σ is an observation-based strategy and (b) $\mathsf{ExpRew}^{\sigma}(M\langle obs\rangle) \leq \tau$. Since σ is positional and deterministic, property (a) amounts to checking whether $obs(s) = obs(t)$ implies $\sigma(s) = \sigma(t)$ for all states $s, t \in S_M$, which can be solved in time quadratic in the size of M. To check property (b), we construct the induced Markov chain $M\langle obs\rangle[\sigma]$, which is linear in the size of M (see Definition 3). Using linear programming (cf. [2]), we can determine the Markov chain's expected reward in polynomial time, i.e. we can check that

$$\mathsf{ExpRew}(M\langle obs\rangle[\sigma]) \;=\; \mathsf{ExpRew}^{\sigma}(M\langle obs\rangle) \;\leq\; \tau.$$

We show NP-hardness by polynomial-time reduction from **PDPEP** to **PDOOP**. The reduction is similar to the proof of Theorem 1 but uses Proposition 2 instead of Proposition 1. We refer to [17, Appendix A.3] for details. In particular, notice that the construction in Fig. 4 is polynomial in the size of the input $\mathscr{M}$, because the constructed MDP M' has $|S| + |O_{\mathscr{M}}| + 2$ states and $|Act_{\mathscr{M}}| \cdot |O_{\mathscr{M}}|$ actions. □

Before we turn to the optimal observability problem for possibly randomized strategies, we remark that, for positional and deterministic strategies, we can also solve a stronger problem than optimal observability: how many observables are needed to turn an MDP into POMDP with the same minimal expected reward?

Definition 10 (Minimal Positional Budget Problem (MPBP)). *Given an MDP M, determine an observation function $obs\colon S_M \to O_{\checkmark}$ such that*

- $\mathsf{MinExpRew}_{pd}(M\langle obs\rangle) = \mathsf{MinExpRew}_{pd}(M)$ *and*
- $\mathsf{MinExpRew}_{pd}(M\langle obs\rangle) < \mathsf{MinExpRew}_{pd}(M\langle obs'\rangle)$ *for all observation functions $obs'\colon S_M \to O'_{\checkmark}$ with $|O'| < |O|$.*

The main idea for solving the problem **MPBP** is that every optimal, positional, and deterministic (OPD, for short) strategy $\sigma\colon S_M \to Act_M$ for an MDP M also solves **PDOOP** for M with threshold $\tau = \mathsf{MinExpRew}_p(M)$ and budget $B = |range(\sigma)|$: A suitable observation function $obs\colon S_M \to range(\sigma)$ assigns action α to every state $s \in S_M$ with $\sigma(s) = \alpha$. It thus suffices to find an OPD strategy for M that uses a minimal set of actions. A brute-force approach to finding such a strategy iterates over all subsets of actions $A \subseteq Act_M$: For each A, we construct an MDP M_A from M that keeps only the actions in A, and determine an OPD strategy σ_A for M_A. The desired strategy is then given by the strategy for the smallest set A such that $\mathsf{ExpRew}^{\sigma_A}(M_A) = \mathsf{MinExpRew}_p(M)$. Since finding an OPD strategy for a MDP is possible in polynomial time (cf. [2]), the problem **MPBP** can be solved in $O(2^{|Act_M|} \cdot \mathit{poly(size(M))})$.

Example 7 (ctnd.). An OPD strategy σ for the MDP M_{line} in Fig. 2 with $p = 1$ is given by $\sigma(s_0) = \sigma(s_1) = r$ and $\sigma(s_3) = \sigma(s_4) = \ell$. Since this strategy maps to two different actions, two observations suffice for selecting an observation function obs such that $\mathsf{MinExpRew}_{pd}(M_{\text{line}}\langle obs\rangle) = \mathsf{MinExpRew}_{pd}(M_{\text{line}}) = 3/2$.

4.2 Positional Randomized Strategies

In the remainder of this section, we will remove the restriction to deterministic strategies, i.e. we will study the optimal observability problem for positional and *possibly randomized* strategies. Our approach builds upon a typed extension of parameter synthesis techniques for Markov chains, which we briefly introduce first. For a comprehensive overview of parameter synthesis techniques, we refer to [12, 14].

Typed Parametric Markov Chains. A typed parametric Markov chain (tpMC) admits expressions instead of constants as transition probabilities. We admit variables (also called *parameters*) of different types in expressions. The types $\mathbb{R}$ and $\mathbb{B}$ represent real-valued and $\{0,1\}$-valued variables, respectively. We denote by $\mathbb{R}_{=C}$ (resp. $\mathbb{B}_{=C}$) a type for real-valued (resp. $\{0,1\}$-valued) variables such that the values of all variables of this type sum up to some fixed constant C.[5] Furthermore, we denote by $V(T)$ the subset of V consisting of all variables of type T. Moreover, $\mathbb{Q}[V]$ is the set of multivariate polynomials with rational coefficients over variables taken from V.

Definition 11 (Typed Parametric Markov Chains). *A* typed parametric Markov chain *is a tuple $\mathscr{D} = (S, I, G, V, P, rew)$, where S is a finite set of* states*, $I \subseteq S$ is a set of* initial states*, $G \subseteq S$ is a set of* goal states*, V is a finite set of typed variables, $P\colon S \times S \to \mathbb{Q}[V]$ is a* parametric transition probability function*, and $rew\colon S \to \mathbb{R}_{\geq 0}$ is a* reward function.

An *instantiation* of a tpMC $\mathscr{D}$ is a function $\iota\colon V_{\mathscr{D}} \to \mathbb{R}$ such that

- for all $x \in V_{\mathscr{D}}(\mathbb{B}) \cup V_{\mathscr{D}}(\mathbb{B}_{=C})$, we have $\iota(x) \in \{0,1\}$;
- for all $V_{\mathscr{D}}(\mathbb{D}_{=C}) = \{x_1, \ldots, x_n\} \neq \emptyset$ with $\mathbb{D} \in \{\mathbb{B}, \mathbb{R}\}$, we have $\sum_{i=1}^{n} \iota(x_i) = C$.

Given a polynomial $q \in \mathbb{Q}[V_{\mathscr{D}}]$, we denote by $q[\iota]$ the real value obtained from replacing in q every variable $x \in V_{\mathscr{D}}$ by $\iota(x)$. We lift this notation to transition probability functions by setting $P_{\mathscr{D}}[\iota](s,s') = P_{\mathscr{D}}(s,s')[\iota]$ for all states $s, s' \in S_{\mathscr{D}}$. An instantiation ι is *well-defined* if it yields a well-defined transition probability function, i.e. if $\sum_{s' \in S_{\mathscr{D}}} P_{\mathscr{D}}[\iota](s,s') = 1$ for all $s \in S_{\mathscr{D}}$. Every well-defined instantiation ι induces a Markov chain $\mathscr{D}[\iota] = (S_{\mathscr{D}}, I_{\mathscr{D}}, G_{\mathscr{D}}, P[\iota], rew_{\mathscr{D}})$.

We focus on the feasibility problem – is there a well-defined instantiation satisfying a given property? – for tpMCs, because of a closed connection to POMDPs.

Definition 12 (Feasibility Problem for tpMCs). *Given a tpMC $\mathscr{D}$ and a threshold $\tau \in \mathbb{Q}_{\geq 0}$, does there exist a well-defined instantiation ι such that $\mathsf{ExpRew}(\mathscr{D}[\iota]) \leq \tau$.*

Junges [14] studied decision problems for parametric Markov chains (pMCs) over real-typed variables. In particular, he showed that the feasibility problem for pMCs over real-typed variables is ETR-complete. Here, ETR refers to the *Existential Theory of Reals*, i.e. all true sentences of the form $\exists x_1 \ldots \exists x_n . P(x_1, ..., x_n)$, where P is a quantifier-free first-order formula over (in)equalities between polynomials with real coefficients and free variables $x_1, \ldots, x_n$. The complexity class ETR consists of all problems that can be reduced to the ETR in polynomial time. We extend this result to tpMCs.

Lemma 1. *The feasibility problem for tpMCs is* ETR*-complete.*

A proof is found in [17, Appendix A.5]. Since ETR lies between NP and PSPACE (cf. [3]), decidability immediately follows:

Theorem 3. *The feasibility problem for tpMCs is decidable in* PSPACE.

[5] We allow using multiple types with different names of this form. For example, $\mathbb{R}^1_{=1}$ and $\mathbb{R}^2_{=1}$ are types for two different sets of variables whose values must sum up to one.

Positional Optimal Observability via Parameter Synthesis. We are now ready to show that the optimal observability problem over positional strategies is decidable. Formally, let $\mathfrak{S}_p(\mathscr{M})$ denote the set of all positional strategies for $\mathscr{M}$. The minimal expected reward over strategies in $\mathfrak{S}_p(\mathscr{M})$ is then given by

$$\mathsf{MinExpRew}_p(\mathscr{M}) \;=\; \inf_{\sigma \in \mathfrak{S}_p(\mathscr{M})} \mathsf{ExpRew}^{\sigma}(\mathscr{M}).$$

Definition 13 (Positional Observability Problem (POP)). *Given an MDP M, a budget $B \in \mathbb{N}_{\geq 1}$, and a threshold $\tau \in \mathbb{Q}_{\geq 0}$, is there a function $obs\colon S_M \to O_{\checkmark}$ with $|O| \leq B$ such that $\mathsf{MinExpRew}_p(M\langle obs \rangle) \leq \tau$?*

To solve a **POP**-instance (M, B, τ), we construct a tpMC $\mathscr{D}$ such that every well-defined instantiation corresponds to an induced Markov chain $M\langle obs \rangle[\sigma]$ obtained by selecting an observation function $obs\colon S_M \to \{1, \ldots, B\} \uplus \{\checkmark\}$ and a positional strategy σ. Then the **POP**-instance (M, B, τ) has a solution iff the feasibility problem for $(\mathscr{D}, \tau)$ has a solution, which is decidable by Theorem 3.

Our construction of $\mathscr{D}$ is inspired by [14]. The main idea is that a positional randomized POMDP strategy takes every action with some probability depending on the given observation. Since the precise probabilities are unknown, we represent the probability of selecting action α given observation o by a parameter $x_{o,\alpha}$. Those parameters must form a probability distribution for every observation o, i.e. they will be of type $\mathbb{R}^{o}_{=1}$. In the transition probability function, we then pick each action with the probability given by the parameter for the action and the current observation. To encode observation function obs, we introduce a Boolean variable $y_{s,o}$ for every state s and observation o that evaluates to 1 iff $obs(s) = o$. Formally, the tpMC $\mathscr{D}$ is constructed as follows:

Definition 14 (Observation tpMC of an MDP). *For an MDP M and a budget $B \in \mathbb{N}_{\geq 1}$, the corresponding* observation tpMC $\mathscr{D}_M = (S_M, I_M, G_M, V, P, rew_M)$ *is given by*

$$O \;=\; \{1, \ldots, B\} \qquad\qquad V \;=\; \biguplus_{s \in S_M \setminus G_M} V(\mathbb{B}^{s}_{=1}) \;\uplus\; \biguplus_{o \in O} V(\mathbb{R}^{o}_{=1})$$

$$V(\mathbb{B}^{s}_{=1}) \;=\; \{y_{s,o} \mid o \in O\} \qquad\qquad V(\mathbb{R}^{o}_{=1}) \;=\; \{x_{o,\alpha} \mid \alpha \in Act_M\}$$

$$P(s, s') \;=\; \sum_{\alpha \in Act_M} \sum_{o \in O} y_{s,o} \cdot x_{o,\alpha} \cdot P_M(s, \alpha)(s'),$$

where, to avoid case distinctions, we define $y_{s,o}$ as the constant 1 *for all $s \in G_M$.*

Our construction is sound in the sense that every Markov chain obtained from an MDP M by selecting an observation function and an observation-based positional strategy corresponds to a well-defined instantiation of the observation tpMC of M.

Lemma 2. *Let M be an MDP and $\mathscr{D}$ the observation tpMC of M for budget $B \in \mathbb{N}_{\geq 1}$. Moreover, let $O = \{1, \ldots, B\}$. Then, the following sets are identical:*

$$\{M\langle obs \rangle[\sigma] \mid obs\colon S_M \to O_{\checkmark}, \sigma \in \mathfrak{S}_p(M\langle obs \rangle)\} \;=\; \{\mathscr{D}[\iota] \mid \iota\colon V_{\mathscr{D}_M} \to \mathbb{R} \textit{ well-defined}\}$$

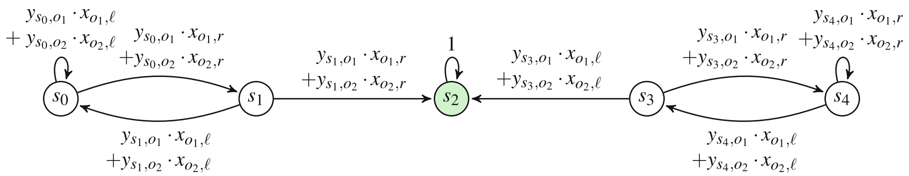

$y_{s_0,o_1} \cdot x_{o_1,\ell}$
$+ y_{s_0,o_2} \cdot x_{o_2,\ell}$
$y_{s_0,o_1} \cdot x_{o_1,r}$
$+y_{s_0,o_2} \cdot x_{o_2,r}$
$y_{s_1,o_1} \cdot x_{o_1,r}$
$+y_{s_1,o_2} \cdot x_{o_2,r}$
1
$y_{s_3,o_1} \cdot x_{o_1,\ell}$
$+y_{s_3,o_2} \cdot x_{o_2,\ell}$
$y_{s_3,o_1} \cdot x_{o_1,r}$
$+y_{s_3,o_2} \cdot x_{o_2,r}$
$y_{s_4,o_1} \cdot x_{o_1,r}$
$+y_{s_4,o_2} \cdot x_{o_2,r}$
s_0
s_1
s_2
s_3
s_4
$y_{s_1,o_1} \cdot x_{o_1,\ell}$
$+y_{s_1,o_2} \cdot x_{o_2,\ell}$
$y_{s_4,o_1} \cdot x_{o_1,\ell}$
$+y_{s_4,o_2} \cdot x_{o_2,\ell}$

Fig. 5. Observation tpMC for the MDP M_{line} in Fig. 2 with $p = 1$ and budget 2.

Proof. Intuitively, the values of $y_{s,o}$ determine the observation function *obs*, and the values of $x_{s,\alpha}$ determine the positional strategy. See [17, Appendix A.6] for details. □

Put together, Lemma 2 and Theorem 3 yield a decision procedure for the positional observability problem: Given a **POP**-instance (M, B, τ), construct the observation tpMC $\mathscr{D}$ of M for budget B. By Theorem 3, it is decidable in ETR whether there exists a well-defined instantiation ι such that $\mathsf{ExpRew}(\mathscr{D}[\iota]) \leq \tau$, which, by Lemma 2, holds iff there exists an observation function $obs\colon S \to \{1, \ldots B\} \uplus \{\checkmark\}$ and a positional strategy $\sigma \in \mathfrak{S}_p(M\langle obs\rangle)$ such that $\mathsf{MinExpRew}_p(M\langle obs\rangle) \leq \mathsf{ExpRew}^{\sigma}(M\langle obs\rangle) \leq \tau$. Hence,

Theorem 4. *The positional observability problem **POP** is decidable in* ETR.

In fact, **POP** is ETR-complete because the policy-existence problem for POMDPs is ETR-complete when restricted to positional strategies [14, Theorem 7.7]. The hardness proof is similar to the reduction in Sect. 3.2. Details are found in [17, Appendix A.3].

Example 8 (ctnd.). Figure 5 depicts the observation tpMC of the MDP M_{line} in Fig. 2 for $p = 1$ and budget $B = 2$. The Boolean variable $y_{s,o}$ is true if we observe o for state s. Moreover, $x_{o,\alpha}$ represents the rate of choosing action α when o is been observed. As is standard for Markov models [27], including parametric ones [14], the expected reward can be expressed as a set of recursive Bellman equations (parametric in our case). For the present example those equations yield the following ETR constraints:

$$\begin{aligned}
r_0 &= 1 + (y_{s_0,o_1} \cdot x_{o_1,\ell} + y_{s_0,o_2} \cdot x_{o_2,\ell}) \cdot r_0 + (y_{s_0,o_1} \cdot x_{o_1,r} + y_{s_0,o_2} \cdot x_{o_2,r}) \cdot r_1 \\
r_1 &= 1 + (y_{s_1,o_1} \cdot x_{o_1,\ell} + y_{s_1,o_2} \cdot x_{o_2,\ell}) \cdot r_0 + (y_{s_1,o_1} \cdot x_{o_1,r} + y_{s_1,o_2} \cdot x_{o_2,r}) \cdot r_2 \\
r_2 &= 0 \\
r_3 &= 1 + (y_{s_3,o_1} \cdot x_{o_1,\ell} + y_{s_3,o_2} \cdot x_{o_2,\ell}) \cdot r_2 + (y_{s_3,o_1} \cdot x_{o_1,r} + y_{s_3,o_2} \cdot x_{o_2,r}) \cdot r_4 \\
r_4 &= 1 + (y_{s_4,o_1} \cdot x_{o_1,\ell} + y_{s_4,o_2} \cdot x_{o_2,\ell}) \cdot r_3 + (y_{s_4,o_1} \cdot x_{o_1,r} + y_{s_4,o_2} \cdot x_{o_2,r}) \cdot r_4 \\
\tau &\geq \tfrac{1}{4} \cdot (r_0 + r_1 + r_3 + r_4)
\end{aligned}$$

where r_i is the expected reward for paths starting at s_i, i.e. $r_i = \sum_{\pi \in Paths(M_{\text{line}}) | \pi[0]=s_i} P^{\sigma}_{M_{\text{line}}}(\pi) \cdot rew_{M_{\text{line}}}(\pi)$. Note that $\mathsf{ExpRew}^{\sigma}(M_{\text{line}}) = \frac{1}{4} \cdot (r_0 + r_1 + r_3 + r_4)$ for the strategy σ defined by the parameters $x_{o,\alpha}$.

Sensor Selection Problem. We finally consider a variant of the positional observability problem in which observations can only be made through a fixed set of location sensors that can be turned on or off for every state. In this scenario, a POMDP can either observe its position (i.e. the current state) or nothing at all (represented by $\bot$).[6] Formally, we consider *location POMDPs* $\mathscr{M}$ with observations $O_{\mathscr{M}} = D \uplus \{\bot\}$, where $D \subseteq \{@s \mid s \in (S_{\mathscr{M}} \setminus G_{\mathscr{M}})\}$ are the observable locations and the observation function is

$$obs_{\mathscr{M}}(s) \;=\; \begin{cases} @s, & \text{if } @s \in D \\ \checkmark, & \text{if } s \in G_{\mathscr{M}} \\ \bot, & \text{if } @s \notin D \text{ and } s \notin G_{\mathscr{M}}. \end{cases}$$

Example 9 (ctnd.). Consider the MDP M_{line} with $p = 1$ and location sensors assigned as in Fig. 2. With a budget of 2 we can only select 2 of the 4 location sensors. For example, we can turn on the sensors on one side, say $@s_0$, $@s_1$. The observation function is then given by $obs(s_0) = @s_1$, $obs(s_1) = @s_2$, and $obs(s_3) = obs(s_4) = \bot$. This is an optimal sensor selection as it reveals whether one is located left or right of the goal.

The *sensor selection problem* aims at turning an MDP into a location POMDP with a limited number of observations such that the expected reward stays below a threshold.

Definition 15 (Sensor Selection Problem (SSP)). *Given an MDP M, a budget $B \in \mathbb{N}_{\geq 1}$, and $\tau \in \mathbb{Q}_{\geq 0}$, is there an observation function $obs\colon S_M \to O_{\checkmark}$ with $|O| \leq B$ such that $\mathscr{M} = (M, O, obs)$ is a location POMDP and $\mathsf{MinExpRew}_p(\mathscr{M}) \leq \tau$?*

To solve the **SSP**, we construct a tpMC similar to Definition 14. The main difference is that we use a Boolean variable y_i to model whether the location sensor $@s_i$ is on (1) or off (0). Moreover, we require that at most B sensors are turned on.

Definition 16 (Location tpMC of an MDP). *For an MDP M and a budget $B \in \mathbb{N}_{\geq 1}$, the corresponding* location tpMC $\mathscr{D}_M = (S_M, I_M, G_M, V, P, rew_M)$ *is given by*

$$V = V(\mathbb{B}_{=B}) \uplus \biguplus_{o \in O} V(\mathbb{R}^{o}_{=1}) \quad V(\mathbb{B}_{=B}) = \{y_s \mid s \in S_M \setminus G_M\} \quad V(\mathbb{R}^{o}_{=1}) = \{x_{s,\alpha} \mid \alpha \in Act_M\}$$

$$P(s, s') \;=\; \sum_{\alpha \in Act} y_s \cdot x_{s,\alpha} \cdot P(s,\alpha)(s') + (1 - y_s) \cdot x_{\bot,\alpha} \cdot P(s,\alpha)(s'),$$

where, to avoid case distinctions, we define y_s as the constant 1 *for all $s \in G_M$.*

Analogously, to Lemma 2 and Theorem 5, soundness of the above construction then yields a decision procedure in PSPACE for the sensor selection problem (see [17, Appendix A.7]).

Lemma 3. *Let M be an MDP and $\mathscr{D}$ the location tpMC of M for budget $B \in \mathbb{N}_{\geq 1}$. Moreover, let LocObs be the set of observation functions $obs\colon S_M \to O_{\checkmark}$ such that $M\langle obs\rangle$ is a location MDP. Then, the following sets are identical:*

$$\{M\langle obs\rangle[\sigma] \mid obs \in LocObs,\ \sigma \in \mathfrak{S}_p(M\langle obs\rangle)\} \;=\; \{\mathscr{D}[\iota] \mid \iota\colon V_{\mathscr{D}_M} \to \mathbb{R} \text{ well-defined}\}$$

Theorem 5. *The sensor selection problem **SSP** is decidable in* ETR, *and thus in* PSPACE.

[6] We provide a generalized version for multiple sensors per state in [17, Appendix A.8].

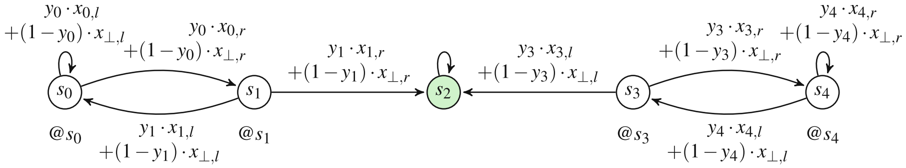

$y_0 \cdot x_{0,l}$
$+(1-y_0)\cdot x_{\perp,l}$
$y_0 \cdot x_{0,r}$
$+(1-y_0)\cdot x_{\perp,r}$
$y_1 \cdot x_{1,r}$
$+(1-y_1)\cdot x_{\perp,r}$
$y_3 \cdot x_{3,l}$
$+(1-y_3)\cdot x_{\perp,l}$
$y_3 \cdot x_{3,r}$
$+(1-y_3)\cdot x_{\perp,r}$
$y_4 \cdot x_{4,r}$
$+(1-y_4)\cdot x_{\perp,r}$
s_0
s_1
s_2
s_3
s_4
$@s_0$
$@s_1$
$@s_3$
$@s_4$
$y_1 \cdot x_{1,l}$
$+(1-y_1)\cdot x_{\perp,l}$
$y_4 \cdot x_{4,l}$
$+(1-y_4)\cdot x_{\perp,l}$

Fig. 6. Location tpMC for the location POMDP in Fig. 2 with $p = 1$ and budget 2.

Example 10. Figure 6 shows the location tpMC of the location POMDP in Fig. 2 for $p = 1$ and budget 2. The Boolean variable y_s indicates if the sensor $@s$ is be turned on, while the variables $x_{s,\alpha}$ indicates the rate of choosing action α if sensor $@s$ is turned on; otherwise, i.e. if sensor $@s$ is turned off, $x_{\perp,\alpha}$ is used, which is the rate of choosing action α for unknown locations.

5 Implementation and Experimental Evaluation

Our approaches for solving the optimal observability problem and its variants fall into two categories: (a) parameter synthesis (cf. Section 4.2) and (b) brute-force enumeration of observation functions combined with probabilistic model checking (cf. Theorem 2). In this section, we evaluate the feasibility of both approaches. Regarding approach (a), we argue in Sect. 5.1 why existing parameter synthesis tools cannot be applied out-of-the-box to the optimal observability problem. Instead, we performed SMT-backed experiments based on direct ETR-encodings (see Theorems 4 and 5); the implementation and experimental setup is described in Sect. 5.2. Section 5.3 presents experimental results using our ETR-encodings for approach (a) and, for comparison, an implementation of approach (b) using the probabilistic model checker PRISM [19].

5.1 Solving Optimal Observability Problems with Parameter Synthesis Tools

Existing tools that can solve parameter synthesis problems for Markov models, such as PARAM [10], PROPHESY [8,9,12], and STORM [11], are, to the best of our knowledge, restricted to (1) *real-valued parameters* and (2) *graph-preserving* models. Restriction (1) means that they do not support *typed* parametric Markov chains, which are needed to model the search for an observation function and budget constraints. Restriction (2) means that the choice of synthesized parameter values may not affect the graph structure of the considered Markov chain. For example, it is not allowed to set the probability of a transition to zero, which effectively means that the transition is removed. While the restriction to graph-preserving models is sensible for performance reasons, it rules out Boolean-typed variables, which we require in our tpMC-encodings of the positional observability problem (Definition 13) and the sensor selection problem (Definition 15). For example, the tpMCs in Fig. 5 and Fig. 6, which resulted from our running example, are *not* graph-preserving models. It remains an open problem whether the same efficient techniques developed for parameter synthesis of graph-preserving models can be applied to typed parametric Markov chains. It is also worth mentioning that for both **POP** and **SSP** the typed extension for pMCs is not strictly necessary. However, the types simplify the presentation and are straightforward to encode into ETR. Alternatively, one can encode Boolean variables in ordinary pMCs as in [14, Fig. 5.23 on page 144]. We opted for the typed version of pMCs to highlight what is challenging for existing parameter synthesis tools.

5.2 Implementation and Setup

As outlined above, parameter synthesis tools are currently unsuited for solving the positional observability (**POP**, Definition 13) and the sensor selection problem (**SSP**, Definition 15). We thus implemented direct ETR-encodings of **POP** and **SSP** instances for positional, but randomized, strategies based on the approach described in Sect. 4.2. We also consider the positional-deterministic observability problem (**PDOOP**, Definition 8) by adding constraints to our implementation for the **POP** to rule out randomized strategies. Our code is written in Python with Z3 [25] as a backend. More precisely, for every tpMC parameter in Definition 14 and Definition 16 there is a corresponding variable in the Z3 encoding. For example, if a Z3 model assigns 1 to the Z3 variable `ys01`, which corresponds to the tpMC parameter y_{s_0,o_1} (Definition 14), we have $obs(s_0) = o_1$. Thus, we can directly construct the observation function. Similarly, we can map the results for the **SSP**. Furthermore, the expected reward for each state is computed using standard techniques based on Bellmann equations as explained in Example 8.

For comparison, we also implemented a brute-force approach for positional and deterministic strategies described in Sect. 4.1, which enumerates all observation functions and corresponding observation-based strategies, hence analyzing the resulting induced DTMCs with PRISM [19].[7] Our code and all examples are available online.[8]

Benchmark Selection. To evaluate our approaches for **P(D)OP** and **SSP**, we created variants (with different state space sizes, probabilities, and thresholds) of two standard benchmarks from the POMDP literature, grid(world) [21] and maze [23], and our running example (cf. Example 1). Overall, we considered 26 variants for each problem.

Setup. All experiments were performed on an HP EliteBook 840 G8 with an 11th Gen Intel(R) Core(TM) i7@3.00GHz and 32GB RAM. We use Ubuntu 20.04.6 LTS, Python 3.8.10, Z3 version 4.12.4, and PRISM 4.8 with default parameters (except for using exact model checking). We use a timeout of 15 min for each individual execution.

5.3 Experimental Results

Tables 1 and 2 show an excerpt of our experiments for selected variants of the three benchmarks, including the largest variant of each benchmark that can be solved for randomized and deterministic strategies, respectively. The full tables containing the results of all experiments are found in [17, Appendix A.10]. The left-hand side of Tables 1 and

[7] Brute-force enumeration of POMDPs has not been considered as PRISM's POMDP solver uses approximation techniques and does not allow to restrict to positional strategies.

[8] https://github.com/alyziakonsta/Optimal-Observability-Problem.

Table 1. Excerpt of experimental results for randomised strategies.

POP - Randomised Strategies

Problem Instance			Z3	
Model	Threshold	Budget	Time(s)	Reward
L(249)	$\leq \frac{250}{2}$	2	t.o.	N/A
	$\leq \frac{125}{2}$	2	19.051	$\frac{125}{2}$
	$< \frac{125}{2}$	2	15.375	N/A
G(20)	$\leq \frac{15200}{399}$	2	t.o.	N/A
	$\leq \frac{7600}{399}$	2	19.164	$\frac{7600}{399}$
	$< \frac{7600}{399}$	2	15.759	N/A
M(7)	$\leq \frac{168}{15}$	4	t.o.	N/A
	$\leq \frac{84}{15}$	4	15.598	$\frac{84}{15}$
	$< \frac{84}{15}$	4	31.986	N/A

SSP - Randomised Strategies

Problem Instance			Z3	
Model	Threshold	Budget	Time(s)	Reward
L(61)	≤ 31	30	t.o.	N/A
	$\leq \frac{31}{2}$	30	17.894	$\frac{31}{2}$
	$< \frac{31}{2}$	30	30.198	N/A
G(6)	$\leq \frac{360}{35}$	5	t.o.	N/A
	$\leq \frac{180}{35}$	5	16.671	$\frac{180}{35}$
	$< \frac{180}{35}$	5	30.204	N/A
M(15)	$\leq \frac{868}{35}$	21	t.o.	N/A
	$\leq \frac{434}{35}$	21	19.067	$\frac{434}{35}$
	$< \frac{434}{35}$	21	30.463	N/A

2 show our results for the **P(D)OP**, whereas the right-hand side shows our results for the **SSP**. We briefly go over the columns used in both tables.

The considered variant is given by the columns model, threshold and budget. There are three kinds of models. We denote by $L(k)$ a variant of our running example MDP M_{line} scaled up to k states. We choose k as an odd number such that the goal is always in the middle of the line. Likewise, we write $G(k)$ to refer to an $k \times k$ grid model, where the goal state is in the bottom right corner. Finally, $M(k)$ refers to the maze model, where k is the (odd) number of states; an example is found in [17, Appendix A.4].

The column Z3 represents the runtime for our direct ETR-encoding with Z3 as a backend. The column PRISM shows the runtime for the brute-force approach. All runtimes are in seconds. We write t.o. if a variant exceeds the timeout. If the (expected) reward is not available due to a timeout, we write N/A in the respective column. In both cases, we color the corresponding cells grey. If our implementation manages to prove that there is no solution, we also write N/A, but leave the cell white.

We choose three different threshold constraints in each problem, if the optimal cumulative expected reward is τ we use the threshold constraints $\leq 2\tau$, $\leq \tau$, and $< \tau$. The last one should yield no solution. The budget is always the minimal optimal one.

Randomised Strategies. Table 1 shows that our implementation can solve several nontrivial **POP/SSP**-instances for randomized strategies. Performance is better when the given thresholds are closer to the optimal one (namely $\leq \tau$ and $< \tau$). For large thresholds ($\leq 2\tau$) the implementation times out earlier (see [17, Appendix A.10] for details). We comment on this phenomenon in more detail later.

Deterministic Strategies. Table 2 shows our results for deterministic strategies. We observe that we can solve larger instances for deterministic strategies than for randomized ones. Considering the performances of both tools, the SMT-backed approach outperforms the brute-force PRISM-based one. For the **PDOOP**, we observe that Z3 can solve some of the problems for the $L(377)$ states, whereas PRISM times our for instances larger than $L(9)$. Also, Z3 is capable of solving problems for grid instances $G(y)$ up to $k = 24$ and maze instances $M(k)$ up to $k = 39$, while PRISM cannot solve any problem instance of these models. For the **SSP**, we can see that Z3 manages to solve $L(y)$ instances up to $k = 193$, whereas PRISM gives up after $k = 7$.

On The Impact of Thresholds. For both randomized and deterministic strategies we observe that larger thresholds yield considerably longer solver runtimes and often lead to a time-out. At first, this behavior appears peculiar because larger thresholds allow for more possible solutions. To investigate this peculiar further, we studied the benchmark $L(7)$ considering the **PDOOP** with thresholds $\leq \tau$, for τ ranging from 1 to 1000. An excerpt of the considered thresholds and verification times is provided in Table 3. For the optimal threshold 2, Z3 finds a solution in 0.079 s. Increasing the threshold (step size 0.25) until 4.5 leads to a steady increase in verification time up to 15.027 s. Verification requires more than 10min for thresholds in [4.75, 5.5]. For larger thresholds, verification time drops to less than 0.1 s. Hence, increasing the threshold first decreases performance, but at some point, performance becomes better again. We have no definitive answers on the threshold's impact, but we conjecture that a larger threshold increases the search space, which might decrease performance. At the same time, a larger threshold can also admit more models, which might increase performance.

Table 2. Excerpt of experimental results for deterministic strategies.

PDOOP - Deterministic Strategies

Problem Instance			Z3		PRISM	
Model	Thresh.	Budg.	Time(s)	Rew.	Time(s)	Rew.
L(9)	≤ 5	2	0.081	$\frac{5}{2}$	205.615	$\frac{5}{2}$
	$\leq \frac{5}{2}$	2	0.082	$\frac{5}{2}$		
	$< \frac{5}{2}$	2	0.086	N/A		
L(377)	≤ 189	2	55.735	$\frac{189}{2}$	t.o.	N/A
	$\leq \frac{189}{2}$	2	19.148	$\frac{189}{2}$		
	$< \frac{189}{2}$	2	353.311	N/A		
G(24)	$\leq \frac{26496}{575}$	2	t.o.	N/A	t.o.	N/A
	$\leq \frac{13248}{575}$	2	19.751	$\frac{13248}{575}$		
	$< \frac{13248}{575}$	2	30.843	N/A		
M(39)	$\leq \frac{6232}{95}$	4	t.o.	N/A	t.o.	N/A
	$\leq \frac{3116}{95}$	4	20.424	$\frac{3116}{95}$		
	$< \frac{3116}{95}$	4	30.149	N/A		

SSP - Deterministic Strategies

Problem Instance			Z3		PRISM	
Model	Thresh.	Budg.	Time(s)	Rew.	Time(s)	Rew.
L(7)	≤ 4	3	0.086	2	186.257	2
	≤ 2	3	0.087	2		
	< 2	3	0.123	N/A		
L(193)	≤ 97	96	t.o.	N/A	t.o.	N/A
	$\leq \frac{97}{2}$	96	20.530	$\frac{97}{2}$		
	$< \frac{97}{2}$	96	30.412	N/A		
G(15)	$\leq \frac{3150}{112}$	14	t.o.	N/A	t.o.	N/A
	$\leq \frac{3150}{224}$	14	20.204	$\frac{3150}{224}$		
	$< \frac{3150}{224}$	14	30.804	N/A		
M(49)	$\leq \frac{9912}{120}$	72	t.o.	N/A	t.o.	N/A
	$\leq \frac{4956}{120}$	72	20.35	$\frac{4956}{120}$		
	$< \frac{4956}{120}$	72	30.333	N/A		

Table 3. PDOOP $L(7)$ with deterministic strategies.

Thresh.	1	1.5	2	3	4	4.5	4.75	5.5	5.75	50	100	500	1000
Time (s)	30.125	30.444	0.079	1.498	8.803	15.027	t.o.	t.o.	0.083	0.089	0.083	0.079	0.083

Discussion. Our experiments demonstrate that SMT solvers, specifically Z3, can be used out-of-the-box to solve small-to-medium sized **POP**- and **SSP**-instances that have been derived from standard examples in the POMDP literature. In particular, for deterministic strategies, the SMT-backed approach clearly outperforms a brute-force approach based on (exact) probabilistic model checking.

Although the considered problem instances are, admittedly, small-to-medium sized[9], they are promising for several reasons: First, our SMT-backed approach is a

[9] At least for notoriously-hard POMDP problems; some instances have ca. 600 states.

faithful, yet naive, ETR-encoding of the **POP**, and leaves plenty of room for optimization. Second, Z3 does, to the best of our knowledge, not use a decision procedure specifically for ETR, which might further hurt performance. Finally, we showed in Sect. 4.2 that **POP** can be encoded as a feasibility problem for (typed) parametric Markov chains. Recent advances in parameter synthesis techniques (cf. [8,12]) demonstrate that those techniques can scale to parametric Markov chains with tens of thousands of states. While the available tools cannot be used out-of-the-box for solving observability problems because of the graph-preservation assumption, it might be possible to extend them in future work.

It is also worth mentioning that our implementation not only provides an answer to the decidability problems **P(D)OP** and **SSP**, but it also synthesizes the corresponding observation function and the strategy if they exist. However, the decision problem and the problem of synthesising such observation function have the same complexities.

6 Conclusion and Future Work

We have introduced the novel *optimal observability problem* (**OOP**). The problem is undecidable in general, NP-complete when restricted to positional and deterministic strategies. We have also shown that the **OOP** becomes decidable in PSPACE if restricted to positional, but *randomized*, strategies, and that it can be reduced to parameter synthesis on a novel typed extension of parametric Markov chains [12,14], which we exploit in our SMT-based implementation. Our experiments show that SMT solvers can be used out-of-the-box to solve small-to-medium-sized instances of observability problems derived from POMDP examples found in the literature. Although we have focused on proving upper bounds on minimal expected rewards, our techniques also apply to other observability problems on POMDPs that can be encoded as a query on tpMCs, based on our faithful encoding of POMDPs as tpMCs with the observation function as a parameter. For example, the sensor synthesis for almost-sure reachability properties [6] can be encoded. Moreover, one obtains dual results for proving lower bounds on maximal expected rewards. For future work, we believe that scalability could be significantly improved by extending parameter synthesis tools such that they can deal with typed and non-graph-preserving parametric Markov chains.

References

1. Åström, K.J.: Optimal control of Markov processes with incomplete state information I. J. Math. Anal. Appl. **10**, 174–205 (1965)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT press (2008)
3. Canny, J.F.: Some algebraic and geometric computations in PSPACE. In: STOC, pp. 460–467. ACM (1988)
4. Černý, P., Chatterjee, K., Henzinger, T.A., Radhakrishna, A., Singh, R.: Quantitative synthesis for concurrent programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 243–259. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_20

5. Chades, I., Carwardine, J., Martin, T.G., Nicol, S., Sabbadin, R., Buffet, O.: MOMDPs: a solution for modelling adaptive management problems. In: AAAI, pp. 267–273. AAAI Press (2012)
6. Chatterjee, K., Chmelik, M., Topcu, U.: Sensor synthesis for POMDPs with reachability objectives. In: Proceedings of the International Conference on Automated Planning and Scheduling, vol. 28, pp. 47–55 (2018)
7. Chatterjee, K., Doyen, L., Henzinger, T.A.: Qualitative analysis of partially-observable Markov decision processes. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 258–269. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15155-2_24
8. Cubuktepe, M., Jansen, N., Junges, S., Katoen, J.-P., Topcu, U.: Synthesis in pMDPs: a tale of 1001 parameters. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 160–176. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_10
9. Dehnert, C., et al.: PROPhESY: a PRObabilistic ParamEter SYnthesis tool. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 214–231. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_13
10. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. Int. J. Softw. Tools Technol. Transf. **13**(1), 3–19 (2011)
11. Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T., Volk, M.: The probabilistic model checker storm. Int. J. Softw. Tools Technol. Transfer **24**(4), 589–610 (2022). https://doi.org/10.1007/s10009-021-00633-z, https://doi.org/10.1007/s10009-021-00633-z
12. Jansen, N., Junges, S., Katoen, J.-P.: Parameter synthesis in Markov models: a gentle survey. In: Raskin, JF., Chatterjee, K., Doyen, L., Majumdar, R. (eds.) Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday, vol. 13660, pp. 407–437 (2022). https://doi.org/10.1007/978-3-031-22337-2_20
13. Jdeed, M., et al.: The CPSwarm technology for designing swarms of cyber-physical systems. In: STAF (Co-Located Events). CEUR Workshop Proceedings, vol. 2405, pp. 85–90. CEUR-WS.org (2019)
14. Junges, S.: Parameter synthesis in Markov models. Ph.D. thesis, Dissertation, RWTH Aachen University, 2020 (2020)
15. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1), 99–134 (1998). https://doi.org/10.1016/S0004-3702(98)00023-X, https://www.sciencedirect.com/science/article/pii/S000437029800023X
16. Kochenderfer, M.J.: Decision Making Under Uncertainty: Theory and Application. MIT press (2015)
17. Konsta, A.M., Lafuente, A.L., Matheja, C.: What should be observed for optimal reward in pomdps? arXiv preprint arXiv:2405.10768 (2024)
18. Krause, A., Singh, A.P., Guestrin, C.: Near-optimal sensor placements in gaussian processes: theory, efficient algorithms and empirical studies. J. Mach. Learn. Res. **9**, 235–284 (2008). https://doi.org/10.5555/1390681.1390689, https://dl.acm.org/doi/10.5555/1390681.1390689
19. 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
20. Littman, M.L.: Memoryless policies: theoretical limitations and practical results. In: From Animals to Animats 3: Proceedings of the third International Conference on Simulation of Adaptive Behavior, vol. 3, p. 238. MIT Press Cambridge, MA, USA (1994)
21. Littman, M.L., Cassandra, A.R., Kaelbling, L.P.: Learning policies for partially observable environments: scaling up. In: Prieditis, A., Russell, S. (eds.) Machine Learning Proceedings 1995, pp. 362–370. Morgan Kaufmann, San Francisco (CA) (1995). https://doi.org/10.1016/B978-1-55860-377-6.50052-9, https://www.sciencedirect.com/science/article/pii/B9781558603776500529

22. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In: AAAI/IAAI, pp. 541–548 (1999)
23. McCallum, R.A.: Overcoming incomplete perception with utile distinction memory. In: Machine Learning Proceedings 1993, pp. 190–196. Morgan Kaufmann, San Francisco (CA) (1993). https://doi.org/10.1016/B978-1-55860-307-3.50031-9, https://www.sciencedirect.com/science/article/pii/B9781558603073500319
24. Miehling, E., Rasouli, M., Teneketzis, D.: A POMDP approach to the dynamic defense of large-scale cyber networks. IEEE Trans. Inf. Forensics Secur. **13**(10), 2490–2505 (2018)
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. Pahalawatta, P., Pappas, T., Katsaggelos, A.: Optimal sensor selection for video-based target tracking in a wireless sensor network. In: 2004 International Conference on Image Processing, 2004. ICIP 2004, vol. 5, pp. 3073–3076 (2004). https://doi.org/10.1109/ICIP.2004.1421762
27. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887, https://doi.org/10.1002/9780470316887
28. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 4th edn. Pearson (2020)
29. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.: Automated generation and analysis of attack graphs. In: Proceedings 2002 IEEE Symposium on Security and Privacy, pp. 273–284 (2002). https://doi.org/10.1109/SECPRI.2002.1004377
30. Spaan, M., Lima, P.: A decision-theoretic approach to dynamic sensor selection in camera networks. In: Proceedings of the International Conference on Automated Planning and Scheduling, vol. 19, pp. 297–304 (2009)

Stochastic Omega-Regular Verification and Control with Supermartingales

Alessandro Abate[1](✉), Mirco Giacobbe[2](✉), and Diptarko Roy[1](✉)

[1] University of Oxford, Oxford, UK
{alessandro.abate,diptarko.roy}@cs.ox.ac.uk
[2] University of Birmingham, Birmingham, UK
m.giacobbe@bham.ac.uk

Abstract. We present for the first time a supermartingale certificate for ω-regular specifications. We leverage the Robbins & Siegmund convergence theorem to characterize supermartingale certificates for the almost-sure acceptance of Streett conditions on general stochastic processes, which we call Streett supermartingales. This enables effective verification and control of discrete-time stochastic dynamical models with infinite state space under ω-regular and linear temporal logic specifications. Our result generalises reachability, safety, reach-avoid, persistence and recurrence specifications; our contribution applies to discrete-time stochastic dynamical models and probabilistic programs with discrete and continuous state spaces and distributions, and carries over to deterministic models and programs. We provide a synthesis algorithm for control policies and Streett supermartingales as proof certificates for ω-regular objectives, which is sound and complete for supermartingales and control policies with polynomial templates and any stochastic dynamical model whose post-expectation is expressible as a polynomial. We additionally provide an optimisation of our algorithm that reduces the problem to satisfiability modulo theories, under the assumption that templates and post-expectation are in piecewise linear form. We have built a prototype and have demonstrated the efficacy of our approach on several exemplar ω-regular verification and control synthesis problems.

Keywords: Probabilistic model checking · Stochastic control synthesis · ω-regular verification · Linear temporal logic · Martingale theory

1 Introduction

Stochastic processes describe phenomena, systems and computations whose behaviour is probabilistic. They are ubiquitous in science and engineering and, in particular, are employed in artificial intelligence and control theory to characterize dynamical models subject to stochastic disturbances, whose correctness is crucial when modelling systems that are deployed to safety-critical environments. Ensuring their correctness with mathematical certainty is an important

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 395–419, 2024.
https://doi.org/10.1007/978-3-031-65633-0_18

yet challenging question, in particular for processes with infinite and possibly continuous state spaces. Systems of this kind include sequential decision and planning problems in stochastic environments, auto-regressive time series as well as probabilistic programs, cryptographic protocols, randomised algorithms and much more. Specifications of correctness for complex systems entail complex temporal behaviour, which can be described using linear temporal logic (LTL) or, more generally, ω-regular properties.

Probabilistic verification algorithms for finite state systems based on explicit-state techniques or symbolic algorithms based on multi-terminal decision diagrams are inapplicable to systems with enumerably infinite or continuous (i.e. uncountably infinite) state spaces [37,41]. For stochastic processes with infinite state space, existing methods usually build upon finite abstractions or proof rules based on martingale theory. Finite abstractions first partition the state space into a grid that forms an equivalent (or an approximately equivalent) finite state process, which is then checked using a finite-state verification algorithm [4,5,68]. Instead, proof rules directly reduce the verification problem to that of computing proof certificates—known as supermartingale certificates—which are synthesised using constraint solving, guess-and-check procedures, or are learned from data [3,15,19]. Proof rules based on supermartingale certificates enable effective verification for infinite-state systems without the intermediate step of computing an abstraction, and have been employed with success in the termination and correctness analysis of probabilistic programs as well as the verification of stochastic dynamical models.

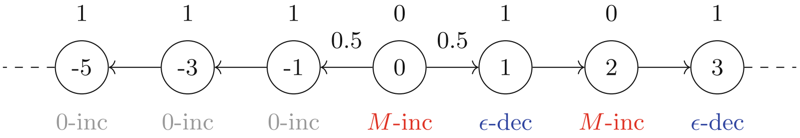


Fig. 1. A simple infinite-state stochastic process over variable $x \in \mathbb{Z}$. Above, the value of a Streett supermartingale for the reactivity property $\mathsf{GF}(x \text{ is even}) \vee \mathsf{FG}(x < 0)$.

Supermartingale certificates for stochastic models have been developed in the past for specific classes of properties. Previous results introduced proof rules for the almost-sure and the quantitative questions of whether a process eventually hits a target condition (guarantee) [6,15,21,23], always avoids an undesirable condition (safety) [16,20], and for Boolean combinations of them (obligation), such as reach-avoid specifications [22]. Supermartingale certificates were further generalised to the properties for which the system eventually satisfies a condition permanently (persistence) [8,17], or hits it infinitely often (recurrence) [17]. Yet, arbitrary Boolean combinations of the latter two, which define the ω-regular properties (reactivity) and include LTL [45], are beyond reach for existing techniques. This includes the example of Fig. 1, which exhibits a process over one integer variable x that, when initialised at $x = 0$, chooses with 0.5 probability to either enumerate the positive numbers or the odd negative numbers. This process satisfies almost surely the property requiring that either x is even infinitely often or that x stays strictly negative from some time onwards; however, as we illustrate in Sect. 6, previous proof rules cannot verify this property.

Notably, reducing ω-regular verification to Büchi acceptance does not easily apply to stochastic processes [28,51,63]. This is because, to express ω-regular as well as LTL properties, this introduces nondeterminism for which standard martingale theory falls short. To reason about ω-regular specifications while preserving the probabilistic nature of the system, it is necessary to reason about Rabin, Streett, Muller or Parity acceptance conditions, as the respective automata express ω-regular languages in their deterministic form [10,56].

We introduce a proof rule for the probabilistic verification of Streett acceptance conditions. Our proof rule leverages the Robbins & Siegmund convergence theorem for nonnegative almost supermartingales [9,55], which we show to characterise the almost-sure acceptance of Streett pairs. A Streett pair (A, B) is satisfied when either A is visited finitely many times or B is visited infinitely often. To conclude that a stochastic process satisfies a Streett pair (A, B) almost surely, we show that it is sufficient to present a nonnegative real function of the state space that strictly decreases in expectation when visiting $A \setminus B$, possibly increases in expectation when visiting B, and never increases in expectation in any other case. Such functions—which we call *Streett supermartingales*—constitute formal proof certificates that stochastic processes satisfy Streett pairs almost surely. For example, consider the reactivity property in Fig. 1, which corresponds to the Streett pair where $A = \{x \mid x \geq 0\}$ and $B = \{x \mid x \text{ is even}\}$. A Streett supermartingale for (A, B) is a function $V \colon \mathbb{Z} \to \mathbb{R}_{\geq 0}$ that strictly decreases in expectation when visiting positive odd numbers, possibly increases in expectation when visiting nonnegative even numbers, and does not increase in expectation when visiting negative numbers: a valid Streett supermartingale is the function $V(x)$ that takes value 1 if x is odd and takes value 0 if x is even. Notably, for general Streett acceptance conditions with multiple pairs, it suffices to compute one Streett supermartingale for each pair.

Our result enables effective and automated ω-regular and LTL verification and control of discrete-time stochastic dynamical models. We leverage our novel proof rule together with the standard result that deterministic Streett automata (DSA) recognise ω-regular languages. Our proof rule readily applies to the synchronous product between a stochastic process and a DSA, where it suffices to compute one Streett supermartingale for each Streett pair together with a supporting invariant, essential to exclude unreachable states for which the specification fails to hold. We provide an automated synthesis algorithm to compute a (1) Streett supermartingale for each pair, (2) a supporting invariant and (3) a control policy simultaneously, with one call to a decision procedure.

We show that for time-homogeneous Markov processes with real-valued state space and piecewise polynomial post-expectation, synthesising Streett supermartingales, supporting invariants and policies with piecewise polynomial template of known degree reduces to quantifier elimination over the first-order theory of the reals with one quantifier alternation. Moreover, we show that synthesising piecewise linear Streett supermartingales and policies, and polyhedral supporting invariants for processes with piecewise linear post-expectation reduces to the first-order existential theory of reals. Finally, we show that when a polyhedral inductive invariant is externally provided, then the synthesis of piecewise linear controllers and Streett supermartingales reduces to quadratically-constrained programming (QCP); furthermore, when the system is autonomous, the sole synthesis of Streett supermartingales reduces to linear programming (LP).

We showcase the practical efficacy of our method on continuous-state probabilistic systems with piecewise affine dynamics, with a prototype implementation. Our implementation is fully automated and capable of synthesizing Streett supermartingale certificates, supporting invariants and control policies simultaneously with a single invocation of a satisfiability modulo theory (SMT) solver. As an experimental benchmark, we consider a collection of ω-regular properties ranging over safety, guarantee, recurrence, persistence and reactivity. This demonstrates that our approach is computationally feasible in practice and that it effectively unifies and generalises prior work on supermartingale certificates.

Our contribution is threefold: we present theory, methods, and experiments for a novel approach to automated stochastic ω-regular verification and control.

Theory We introduce the first supermartingale certificate for full ω-regular specifications: the Streett supermartingale. By preserving the probabilistic nature of the model, our proof rule enables effective ω-regular verification of infinite state models by reasoning about their post-expectation.

Methods We provide sound and complete algorithms for ω-regular verification and control based on our proof rule. Our algorithms compute Streett supermartingales, supporting invariants and control policies with known templates, and are complete relative to provided templates and post-expectations.

Experiments We have built a prototype showcasing the efficacy of our algorithms on a set of continuous-state probabilistic systems and ω-regular properties that include and extend beyond the scope of existing approaches.

Our theoretical contribution applies to any discrete-time deterministic and stochastic dynamical model as well as deterministic and probabilistic programs with discrete and continuous distributions, whose semantics are all special cases of general stochastic processes. Our synthesis algorithm applies to any model whose post-expectation is expressible in piecewise polynomial closed form.

2 Streett Supermartingales

We define stochastic processes on a filtered probability space whose space of outcomes Ω defines an $\mathcal{F}$-measurable space of infinite runs, and $\{\mathcal{F}_t\}$ is the associated filtration $\mathcal{F}_t \subseteq \mathcal{F}_{t+1} \subseteq \mathcal{F}$ for all $t \geq 0$. A discrete-time stochastic process over a Σ-measurable state space S is a sequence $\{X_t\}$ with $X_t \colon \Omega \to S$ that maps every outcome to the state of a trajectory at time t. We say that $\{X_t\}$ is adapted to $\{\mathcal{F}_t\}$ if every X_t is $\mathcal{F}_t$-measurable, namely for all $A \in \Sigma$ it holds that

$X_t^{-1}[A] \in \mathcal{F}_t$. A trajectory τ is an infinite sequence of states $\tau = \tau_0, \tau_1, \tau_2, \ldots$ such that $\tau_t = X_t(\omega)$ for all $t \geq 0$, for some $\omega \in \Omega$. Stochastic processes provide a general characterisation for the semantics of stochastic dynamical models described as stochastic difference equations as well as reactive probabilistic programs that run over infinite time.

Our supermartingale certificate for almost-sure ω-regular verification and control of stochastic processes is underpinned by the Robbins & Siegmund theorem for the convergence of *nonnegative almost supermartingales.*

Theorem 1 (Robbins & Siegmund Convergence Theorem [55]). *Let $\{\mathcal{F}_t\}$ be a filtration and let $\{V_t\}$, $\{U_t\}$, and $\{W_t\}$ be three real-valued nonnegative stochastic processes adapted to $\{\mathcal{F}_t\}$. Suppose that, for all $t \in \mathbb{N}$, the following statement holds almost surely:*

$$E(V_{t+1} \mid \mathcal{F}_t) \leq V_t - U_t + W_t. \tag{1}$$

Then,

$$\Pr\left(\sum_{t=0}^{\infty} U_t < \infty \vee \sum_{t=0}^{\infty} W_t = \infty\right) = 1. \tag{2}$$

This result generalises the classic convergence theorem for nonnegative supermartingales [54, Theorem 22, p.148], allowing the real-valued process $\{V_t\}$ to satisfy the weaker almost-supermartingale condition of Eq. (1) with respect to the two other real-valued processes $\{U_t\}$ and $\{W_t\}$ [9,55]. The statement establishes that the event that either series $\sum_{t=0}^{\infty} U_t$ converges or series $\sum_{t=0}^{\infty} W_t$ diverges has probability 1. As we show below, this naturally characterises almost-sure Streett acceptance for general stochastic processes.

A Streett pair (A, B) consists of two measurable regions of the state space $A, B \in \Sigma$. A trajectory $\tau = \tau_0, \tau_1, \tau_2, \ldots$ satisfies (A, B) if either it visits all states in A finitely many times or it visits any states in B infinitely many times; more formally, τ satisfies (A, B) if $\sum_{t=0}^{\infty} \mathbf{1}_{A_i}(\tau_t) < \infty \vee \sum_{t=0}^{\infty} \mathbf{1}_{B_i}(\tau_t) = \infty$, where $\mathbf{1}_{\mathcal{S}}(\cdot)$ denotes the indicator function of set $\mathcal{S}$, which takes value 1 when its argument is a member of $\mathcal{S}$ and takes value 0 otherwise. Our result establishes that, to conclude that a stochastic process $\{X_t\}$ satisfies (A, B) almost surely, it suffices to present a function V that maps $\{X_t\}$ to a nonnegative almost-supermartingale whose expected value decreases strictly when visiting $A \setminus B$, possibly increases when visiting B, and never increases anywhere else almost surely. We call function V a Streett supermartingale for (A, B).

Theorem 2 (Streett Supermartingales). *Let $\{X_t\}$ be a stochastic process over state space S and (A, B) be a Streett pair. Suppose that there exists a nonnegative function $V : S \to \mathbb{R}_{\geq 0}$ and positive constants $\epsilon, M > 0$ such that, for all $t \in \mathbb{N}$, the following condition holds almost surely:*

$$E[V(X_{t+1}) \mid \mathcal{F}_t] \leq V(X_t) - \epsilon \cdot \mathbf{1}_{A \setminus B}(X_t) + M \cdot \mathbf{1}_B(X_t). \tag{3}$$

Then, $\{X_t\}$ satisfies (A, B) almost surely, i.e.,

$$\Pr\left(\sum_{t=0}^{\infty} \mathbf{1}_A(X_t) < \infty \vee \sum_{t=0}^{\infty} \mathbf{1}_B(X_t) = \infty\right) = 1. \tag{4}$$

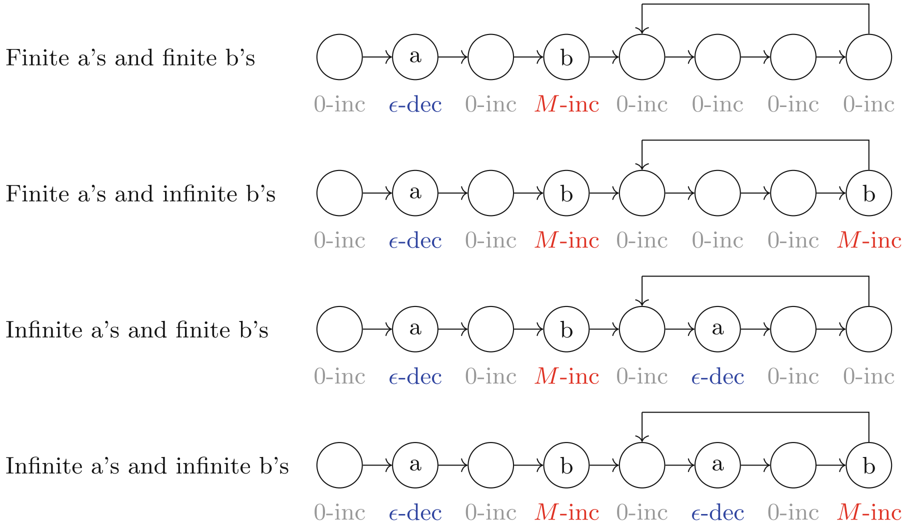


Fig. 2. Intuition for Theorem 2 on exemplar trajectories.

Example 1. Figure 2 illustrates Theorem 2 over four exemplar trajectories, with respect to the Streett pair $(\{s \mid s \text{ has label a}\}, \{s \mid s \text{ has label b}\})$. In this example, we illustrate that a Streett supermartingale V—which must be nonnegative—cannot be constructed for the third trajectory, as Eq. (3) requires V to strictly decrease by ϵ infinitely many times in the tail behaviour of the trajectory while being never allowed to increase. For all other trajectories instead, a Streett supermartingale V and suitable constants $\epsilon, M > 0$ exist. In particular, in the first and second trajectories any V is only required to strictly decrease finitely many times. In the fourth trajectory, V is permitted to compensate its requirement to decrease infinitely many times by increasing infinitely many times in the tail behaviour. Notably, the first, the second, and the fourth trajectory are precisely those trajectories that satisfy the specification. □

We provide a specialisation of Theorem 2 (which applies to general stochastic processes) to time-homogeneous Markov processes, whose dynamics only depend on their transition kernel. A *transition kernel* $T : S \times \Sigma \to [0, 1]$ gives the probability that the process makes a transition from state $s \in S$ into the set $S' \in \Sigma$, independently of time, i.e., for all $t \in \mathbb{N}$, $T(X_t, S') = \Pr(X_{t+1} \in S' \mid \mathcal{F}_t)$.

The transition kernel in turn determines the *post-expectation* $(\text{Post}\, h)\colon S \to \mathbb{R}$ of any real-valued measurable function $h\colon S \to \mathbb{R}$, defined as the conditional expectation of h after one time step (regardless of absolute time t) as follows:

$$\text{Post}\, h(X_t) = \int_S h(s)\; T(X_t, \mathrm{d}s) = E(h(X_{t+1}) \mid \mathcal{F}_t). \tag{5}$$

This denotes the expected value of h when evaluated in the subsequent state, given the current state being X_t. For time-homogeneous Markov processes, we establish that to obtain a valid Streett supermartingale it suffices to enforce the requirement of Eq. (3) over Post V of a Streett supermartingale V whose domain is restricted to a sufficiently strong supporting invariant I.

Theorem 3 (Supporting Invariants). *Let $\{X_t\}$ be a time-homogeneous Markov process with initial state $s_0 \in S$ and transition kernel $T\colon S \times \Sigma \to [0,1]$. Let (A, B) be a Streett pair. Suppose there exists a measurable set $I \in \Sigma$, a nonnegative function $V\colon I \to \mathbb{R}_{\geq 0}$ and positive constants $\epsilon, M > 0$ that satisfy the following five conditions:*

$$s_0 \in I \tag{6}$$

$$\forall s \in I\colon T(s, I) = 1 \tag{7}$$

$$\forall s \in (A \setminus B) \cap I\colon\ \text{Post}\, V(s) \leq V(s) - \epsilon \tag{8}$$

$$\forall s \in B \cap I\colon\ \text{Post}\, V(s) \leq V(s) + M \tag{9}$$

$$\forall s \in I \setminus (A \cup B)\colon\ \text{Post}\, V(s) \leq V(s) \tag{10}$$

Then, V is a Streett supermartingale for (A, B).

Example 2. Consider the time-homogeneous Markov process in Fig. 1 and the LTL property $\mathsf{GF}(x \text{ is even})$, corresponding to the Streett pair $(\mathbb{Z}, \{x \mid x \text{ is even}\})$. Provided the supporting invariant $\{x \in \mathbb{Z} \mid x > 0\}$, the function that maps the positive even numbers to 0 and the positive odd numbers to 1 is a valid Streett supermartingale if the process is initialised on a positive number. Without a supporting invariant, function V would be required to strictly decrease along all negative numbers, necessarily violating nonnegativity. Notably, the process satisfies $\mathsf{GF}(x \text{ is even})$ almost surely only on the positive numbers. □

Finally, a general Streett acceptance condition consists of a finite set of Streett pairs, and a trajectory satisfies the acceptance condition if it satisfies all pairs. To establish that a stochastic process satisfies a general Streett acceptance condition, it suffices to present one Streett supermartingale for each pair.

Theorem 4. *Let $\{X_t\}$ be a stochastic process and $\{(A_i, B_i)\colon i = 1, \ldots, k\}$ be a Streett acceptance condition. If every Streett pair admits a Streett supermartingale, then $\{X_i\}$ satisfies the acceptance condition almost surely:*

$$\Pr\left(\bigwedge_{i=1}^{k} \left(\sum_{t=0}^{\infty} \mathbf{1}_{A_i}(X_t) < \infty \vee \sum_{t=0}^{\infty} \mathbf{1}_{B_i}(X_t) = \infty\right)\right) = 1. \tag{11}$$

3 Stochastic Omega-Regular Verification and Control

A stochastic dynamical model $\mathcal{M}$ over $\mathbb{R}^n$ consists of an initial state vector $x_0 \in \mathbb{R}^n$ and a parameterised update function $f\colon \mathbb{R}^n \times \mathcal{W} \times K \to \mathbb{R}^n$ with a space $\mathcal{W}$ of input disturbances and a space K of control parameters. This defines a time-homogeneous Markov process over the $\mathcal{B}(\mathbb{R}^n)$-measurable state space $\mathbb{R}^n$ given by the following equation:

$$X^{\mathcal{M}}_{t+1} = f(X^{\mathcal{M}}_t, W_t; \kappa), \quad X^{\mathcal{M}}_0 = x_0, \tag{12}$$

where $\{W_t\}$ is a sequence of i.i.d. stochastic input disturbances, each of which draws from the sample space $\mathcal{W}$. This assumption restricts our model to time-homogeneous Markov processes, for which Theorem 3 applies. This model subsumes autonomous systems as well as control systems with parameterised policies. For example, a stochastic dynamical model $f'\colon \mathbb{R}^n \times \mathcal{U} \times \mathcal{W} \to \mathbb{R}^n$ with finite or infinite space of control inputs $\mathcal{U}$ and a parameterised (memoryless deterministic) policy $\pi\colon \mathbb{R}^n \times K \to \mathcal{U}$ results in the special case $f(x, w; \kappa) = f'(x, \pi(x; \kappa), w)$. Notably, our model also encompasses finite memory policies with known template and known memory size, for which it is sufficient to add extra state variables and extra input disturbances.

We associate our model with a finite set of observable propositions Π and an observation function $\langle\!\langle \cdot \rangle\!\rangle\colon \mathbb{R}^n \to 2^{\Pi}$ that maps every state to the set of propositions that hold true in that state. This defines a (measurable) set of traces—the trace language of $\mathcal{M}$—where a trace $\hat{\tau}$ is an infinite sequence $\hat{\tau} = \hat{\tau}_0, \hat{\tau}_1, \hat{\tau}_2, \dots$ where $\hat{\tau}_i = \langle\!\langle \tau_i \rangle\!\rangle$ for all $i \geq 0$, with $\tau = \tau_0, \tau_1, \tau_2, \dots$ being some trajectory of $\{X^{\mathcal{M}}_t\}$. We treat the question of synthesizing a controller for which $\mathcal{M}$ satisfies an LTL formula over atomic propositions in Π or, more generally, satisfies an ω-regular property over alphabet 2^{Π} almost surely. For this purpose, we leverage the standard result that deterministic Streett automata (DSA) recognise the ω-regular languages. The control synthesis problem amounts to computing a control parameter $\kappa \in K$ for which the event that the trace language of $\mathcal{M}$ is accepted by DSA $\mathcal{A}$ has probability 1. The verification problem for autonomous systems or systems with fixed control policy can be simply seen as the special case where K is a singleton.

A deterministic Streett automaton $\mathcal{A}$ over alphabet 2^{Π} consists of a finite set of states Q, an initial state q_0, a transition function $\delta\colon Q \times 2^{\Pi} \to Q$, and a finite set of Streett pairs $\mathrm{Acc} = \{(A_1, B_1), \dots, (A_k, B_k)\}$ where $A_i \subseteq Q$ and $B_i \subseteq Q$ for all $i = 1, \dots, k$. The run ρ of $\mathcal{A}$ on input trace $\hat{\tau} = \hat{\tau}_0, \hat{\tau}_1, \hat{\tau}_2, \dots$ is the infinite sequence of states $\rho = \rho_0, \rho_1, \rho_2, \dots$ such that $\rho_0 = q_0$ and $\rho_{t+1} = \delta(\rho_t, \hat{\tau}_t)$ for all $t \geq 0$. The automaton accepts $\hat{\tau}$ if either ρ visits A_i finitely many times or B_i infinitely many times for all $i = 1, \dots k$, i.e., $\bigwedge_{i=1}^{k} \sum_{t=1}^{\infty} \mathbf{1}_{A_i}(\rho_t) < \infty \vee \sum_{t=1}^{\infty} \mathbf{1}_{B_i}(\rho_t) = \infty$. Our approach to probabilistic ω-regular verification leverages the fact that a DSA (indeed, any deterministic automaton) recognising the traces of a stochastic process forms in its turn a stochastic process:

$$X^{\mathcal{A}}_{t+1} = \delta(X^{\mathcal{A}}_t, \langle\!\langle X^{\mathcal{M}}_t \rangle\!\rangle), \quad X^{\mathcal{A}}_0 = q_0. \tag{13}$$

Our approach determines whether $\{X_t^{\mathcal{A}}\}$ satisfies the Streett acceptance condition of $\mathcal{A}$ with probability 1. We note that Streett automata are dual to Rabin automata, thus any tool to translate an LTL formula φ to a Rabin automaton, equivalently produces a Streett automaton for $\neg\varphi$ [30,39]. Our output can thus be equivalently cast as Rabin acceptance with probability 0.

To determine whether $\{X_t^{\mathcal{A}}\}$ satisfies the acceptance condition of $\mathcal{A}$, we leverage Theorems 3 and 4 to synthesize a Streett supermartingale for each Streett pair and a supporting invariant over the *synchronous product* of $\mathcal{M}$ and $\mathcal{A}$. This is because the process $\{X_t^{\mathcal{A}}\}$ is not time-homogeneous when considered in isolation, as the distribution of next states in the automaton requires information about $\{X_t^{\mathcal{M}}\}$ to be determined. Therefore, we define the product process $\{X_t^{\mathcal{M}\otimes\mathcal{A}}\}$ as $X_t^{\mathcal{M}\otimes\mathcal{A}} = (X_t^{\mathcal{M}}, X_t^{\mathcal{A}})$ for all $t \in \mathbb{N}$, where $X_{t+1}^{\mathcal{M}\otimes\mathcal{A}} = (f(X_t^{\mathcal{M}}, W_t; \kappa)$, $\delta(X_t^{\mathcal{A}}, \langle\!\langle X_t^{\mathcal{M}} \rangle\!\rangle))$ and $X_0^{\mathcal{M}\otimes\mathcal{A}} = (x_0, q_0)$. We then extend the acceptance condition of $\mathcal{A}$ to the product state space $\mathbb{R}^n \times Q$. Concretely, we define $\overline{A}_i = \mathbb{R}^n \times A_i$ and $\overline{B}_i = \mathbb{R}^n \times B_i$ for $i = 1, \ldots, k$, and we define the acceptance condition of the product process as $\{(\overline{A}_1, \overline{B}_1), \ldots, (\overline{A}_k, \overline{B}_k)\}$. Finally, we establish that $\{X_t^{\mathcal{M}\otimes\mathcal{A}}\}$ satisfies the given acceptance condition almost surely by computing k Streett supermartingales and one supporting invariant over $\mathbb{R}^n \times Q$.

We assume a known parameterised form for Streett supermartingales and invariant as well as for the control policy (as described above) and, by using Theorems 3 and 4, we express the verification and control problem as the problem of deciding a quantified first-order logic formula. Let $V \colon \mathbb{R}^n \times Q \times \Theta \to \mathbb{R}_{\geq 0}$ be a parameterised non-negative function of $\mathbb{R}^n \times Q$ (the Streett supermartingale certificate), with parameter space Θ. The post-expectation of V results in the parameterised function $(\operatorname{Post} V) \colon \mathbb{R}^n \times Q \times \Theta \times K \to \mathbb{R}_{\geq 0}$ over the certificate parameters Θ of V and the control parameters K defined as

$$\operatorname{Post} V(x, q; \theta, \kappa) = \int_{\mathcal{W}} V(f(x, w; \kappa), \delta(q, \langle\!\langle x \rangle\!\rangle); \theta) \Pr(\mathrm{d}w) \tag{14}$$

To construct our first-order logic decision problem, it is essential to express $\operatorname{Post} V$ in a symbolic closed-form representation. Notably, computing symbolic closed-form representations for the post-expectation is a general problem in probabilistic verification, for which automated tools exist [35]. Provided that $\operatorname{Post} V$ is computable, we template k parameterised Streett supermartingale certificates $V_1, \ldots, V_k$ with parameter spaces $\Theta_1, \ldots, \Theta_k$ respectively, and template one parameterised invariant predicate $I \colon \mathbb{R}^n \times Q \times H \to \{\text{true}, \text{false}\}$ with parameter space H. Then, solving the ω-regular control problem with our method amounts to searching for certificate parameters $\theta_1 \in \Theta_1, \ldots, \theta_k \in \Theta_k$, invariant parameter $\eta \in H$, control parameter $\kappa \in K$ and coefficients $\epsilon, M > 0$ such that, for every $i = 1, \ldots, k$, the following universally quantified sentences hold:

$$I(x_0, q_0; \eta) \tag{15}$$

$$\forall x \in \mathbb{R}^n, q \in Q, w \in \mathcal{W}\colon I(x, q; \eta) \implies I(f(x, w; \kappa), \delta(q, \langle\!\langle x \rangle\!\rangle); \eta) \tag{16}$$

$$\forall x \in \mathbb{R}^n, q \in (A_i \setminus B_i)\colon I(x, q; \eta) \implies \operatorname{Post} V_i(x, q; \theta_i, \kappa) \leq V_i(x, q; \theta_i) - \epsilon \tag{17}$$

$$\forall x \in \mathbb{R}^n, q \in B_i\colon I(x, q; \eta) \implies \operatorname{Post} V_i(x, q; \theta_i, \kappa) \leq V_i(x, q; \theta_i) + M \tag{18}$$

$$\forall x \in \mathbb{R}^n, q \in Q \setminus (A_i \cup B_i)\colon I(x, q; \eta) \Rightarrow \operatorname{Post} V_i(x, q; \theta_i, \kappa) \leq V_i(x, q; \theta_i) \tag{19}$$

$$\forall x \in \mathbb{R}^n, q \in Q\colon I(x, q; \eta) \implies V_i(x, q; \theta_i) \geq 0 \tag{20}$$

In particular, Eqs. (15) and (16) respectively indicate the conditions of *initiation* and *consecution* for the supporting invariant, yielding a subset of the product space satisfying Eqs. (6) and (7). Equations (17) to (19) indicate the *drift conditions*, which ensure that $V_1, \ldots, V_k$ satisfy Eqs. (8) to (10) w.r.t. the acceptance conditions extended to the product space. Equation (20) enforces the premise of Theorem 3 that requires V to be non-negative over its domain I.

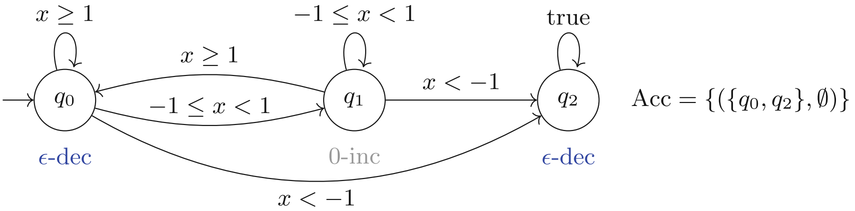


Fig. 3. Deterministic Streett Automaton for $(x \geq -1)\mathsf{U}\mathsf{G}(-1 \leq x \leq 1)$

Example 3. Consider a simple Markov process over one real-valued variable x and control parameter κ, described by the following stochastic difference equation:

$$x_{t+1} = \kappa \cdot x_t + w_t, \qquad x_0 = 100, \qquad w_t \sim \text{Uniform}(-0.1, 0.1) \tag{21}$$

We wish to synthesise a control parameter κ for which the process satisfies the *stabilize-while-avoid* property $\Phi = (x \geq -1)\mathsf{U}\mathsf{G}(-1 \leq x \leq 1)$, which requires the system to avoid $x < -1$ until it stabilizes within $(-1 \leq x \leq 1)$. This corresponds to the DSA in Fig. 3, whose states define the necessary drift conditions.

Applying Theorem 3 to the DSA of Fig. 3, we make two observations. Firstly, recalling the intuition of the first trajectory in Fig. 2, we note that the specification is satisfied only if q_0 is visited finitely many times by the product process, and this must be established by a Streett supermartingale that strictly decreases when $x \geq 1$ and does not increase otherwise. Secondly, recalling the intuition of the third trajectory, we note that such a Streett supermartingale exists only if q_2 is never reached, and this must be established by a supporting invariant. A control parameter for which Φ is satisfied is $\kappa = 0.5$, and this is established by the following Streett supermartingale and supporting invariant:

$$V(x,q) = \begin{cases} x+1 & \text{if } q = q_0 \\ 0 & \text{otherwise} \end{cases} \qquad I(x,q) = \begin{cases} x \geq -0.2 & \text{if } q = q_0 \\ -0.2 \leq x \leq 0.9 & \text{if } q = q_1 \\ \text{false} & \text{if } q = q_2 \end{cases} \tag{22}$$

Here, the post-expectation of V results in the function below; note that no term for the stochastic disturbance appears because, in this case, the expected value of w is 0, and so is its contribution to the post-expectation of V:

$$\text{Post}\, V(x,q) = \begin{cases} 0.5 \cdot x + 1 & \text{if } x \geq 1 \text{ and } q \in \{q_0, q_1\} \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

Altogether, we obtain the following (satisfied) system of universally quantified sentences. The initiation condition (cf. Eq. (15)) results in the following sentence:

$$100 \geq -0.2 \equiv I(x_0, q_0) \tag{24}$$

The consecution condition (cf. Eq. (16)) expands into the following three implications, each of which corresponds to a case of I, a feasible transition in the automaton, and a feasible transition of the dynamical model according to the sample space $\mathcal{W} = [-0.1, 0.1]$ of stochastic disturbances:

$$\forall x \in \mathbb{R}, w \in \mathcal{W}\colon \underbrace{x \geq -0.2}_{I(x,q_0)} \wedge \underbrace{x \geq 1}_{\delta(q_0,\cdot)=q_0} \implies \underbrace{0.5x + w \geq -0.2}_{I(0.5x+w,q_0)} \tag{25}$$

$$\forall x \in \mathbb{R}, w \in \mathcal{W}\colon \underbrace{x \geq -0.2}_{I(x,q_0)} \wedge \underbrace{-1 \leq x < 1}_{\delta(q_0,\cdot)=q_1} \implies \underbrace{-0.2 \leq 0.5x + w \leq 0.9}_{I(0.5x+w,q_1)} \tag{26}$$

$$\forall x \in \mathbb{R}, w \in \mathcal{W}\colon \underbrace{x \geq -0.2}_{I(x,q_0)} \wedge \underbrace{x < -1}_{\delta(q_0,\cdot)=q_2} \implies \underbrace{\text{false}}_{I(0.5x+w,q_2)} \tag{27}$$

$$\forall x \in \mathbb{R}, w \in \mathcal{W}\colon \underbrace{-0.2 \leq x \leq 0.9}_{I(x,q_1)} \wedge \underbrace{x \geq 1}_{\delta(q_1,\cdot)=q_0} \implies \underbrace{0.5x + w \geq -0.2}_{I(0.5x+w,q_0)} \tag{28}$$

$$\forall x \in \mathbb{R}, w \in \mathcal{W}\colon \underbrace{-0.2 \leq x \leq 0.9}_{I(x,q_1)} \wedge \underbrace{-1 \leq x < 1}_{\delta(q_1,\cdot)=q_1} \Rightarrow \underbrace{-0.2 \leq 0.5x + w \leq 0.9}_{I(0.5x+w,q_1)} \tag{29}$$

$$\forall x \in \mathbb{R}, w \in \mathcal{W}\colon \underbrace{-0.2 \leq x \leq 0.9}_{I(x,q_1)} \wedge \underbrace{x < -1}_{\delta(q_1,\cdot)=q_2} \implies \underbrace{\text{false}}_{I(0.5x+w,q_2)} \tag{30}$$

$$\forall x \in \mathbb{R}, w \in \mathcal{W}\colon \underbrace{\text{false}}_{I(x,q_2)} \wedge \underbrace{\text{true}}_{\delta(q_2,\cdot)=q_2} \implies \underbrace{\text{false}}_{I(0.5x+w,q_2)} \tag{31}$$

The strict decrease drift condition associated with q_0 (cf. Eq. (17)) results, with $\epsilon = 0.5$, in the following two sentences associated with each case of $\text{Post}\, V$:

$$\forall x \in \mathbb{R}\colon \underbrace{x \geq -0.2}_{I(x,q_0)} \wedge (x \geq 1) \implies \underbrace{0.5 \cdot x + 1}_{\text{Post}\, V(x,q_0)} \leq \underbrace{x+1}_{V(x,q_0)} - \underbrace{0.5}_{\epsilon} \tag{32}$$

$$\forall x \in \mathbb{R}\colon \underbrace{x \geq -0.2}_{I(x,q_0)} \wedge (x < 1) \implies \underbrace{0}_{\text{Post}\, V(x,q_0)} \leq \underbrace{x+1}_{V(x,q_0)} - \underbrace{0.5}_{\epsilon} \tag{33}$$

Similarly, the non-increase drift condition associated with q_1 (cf. Eq. (19)) results in the following two implications:

$$\forall x \in \mathbb{R}\colon\ \underbrace{-0.2 \leq x \leq 0.9}_{I(x,q_1)} \wedge (x \geq 1) \implies \underbrace{0.5 \cdot x + 1}_{\text{Post}\, V(x,q_1)} \leq \underbrace{0}_{V(x,q_1)} \tag{34}$$

$$\forall x \in \mathbb{R}\colon\ \underbrace{-0.2 \leq x \leq 0.9}_{I(x,q_1)} \wedge (x < 1) \implies \underbrace{0}_{\text{Post}\, V(x,q_1)} \leq \underbrace{0}_{V(x,q_1)} \tag{35}$$

We note that the invariant in state q_1 is sufficiently strong to exclude the possibility of a transition back to q_0 from q_1 (which is associated with $x \geq 1$), making the premise of the implication in Eq. (34) false. The drift condition of q_2 is also trivially satisfied, as the premise of the respective implication is false:

$$\forall x \in \mathbb{R}\colon\ \underbrace{\text{false}}_{I(x,q_2)} \implies \underbrace{0}_{\text{Post}\, V(x,q_2)} \leq \underbrace{0}_{V(x,q_2)} - \underbrace{0.5}_{\epsilon} \tag{36}$$

Finally, the non-negativity condition (cf. Eq. (20)) is trivially satisfied on q_1 and q_2 as $V(x, q_1) = V(x, q_2) = 0$. For q_0 instead, the condition is the following:

$$\forall x \in \mathbb{R}\colon\ \underbrace{x \geq -0.2}_{I(x,q_0)} \implies \underbrace{x + 1}_{V(x,q_0)} \geq 0 \tag{37}$$

Notably, every sentence consists of a conjunction of inequalities implying an inequality. As we show in Sect. 4, difference equations, Streett supermartingales and supporting invariants that are piecewise-defined according to a template as in Eq. (22) always result in systems of constraints in this form. This enables effective algorithmic synthesis of Streett supermartingales, supporting invariants and control parameters using symbolic or numerical decision procedures. □

4 Algorithmic Synthesis of Streett Supermartingales

Exhibiting Streett supermartingales and supporting invariants constitutes a witness that the stochastic dynamical model and its control parameter comply with the ω-regular property at hand. Under the assumption that these three objects are constrained to be in the form of a template, the verification and control problem is reducible to a decision procedure for quantified first-order formulae. In this section, we define templates that allow effective synthesis using standard symbolic and numerical decision procedures.

We show that under different assumptions and problem settings, the verification and control problem reduces to the following decision procedures:

General Control This refers to the general synthesis of a Streett supermartingale, supporting invariant, and control parameters. When these and the associated post-expectation are in piecewise polynomial form (Sect. 4.1), then the synthesis problem is reducible to a quantified formula (with one quantifier alternation) in non-linear real arithmetic (NRA). When they are in piecewise linear form (Sect. 4.2) then the synthesis problem reduces to the existential theory of non-linear real arithmetic (QF_NRA).

Shielded Control This refers to the synthesis of a Streett supermartingale and control parameter, given an externally provided inductive invariant. Externally provided invariants are relevant when a shield that ensures the safety of the policy (but not necessarily its liveness) is computed beforehand [7]. This reduces to quadratically constrained programming (QCP) with piecewise linear templates (Sect. 4.3 and Example 5).

Verification This refers to the sole synthesis of Streett supermartingales, when the system has a known invariant that is provided a priori. This reduces to linear programming (LP) when templates and post-expectation are piecewise linear (Sect. 4.3 and Example 6).

We introduce a functional template $F\colon \mathbb{R}^N \times Q \times \Lambda \to \mathbb{R}$ that maps an N-dimensional real-valued vector, a state of the automaton $q \in Q$, and a generic template parameter $\lambda \in \Lambda$ to a real-valued output according to a number of cases, guarded by logical predicates:

$$F(x, q; \lambda) = \begin{cases} g_{1,l+1}(x; \lambda) & \text{if } \bigwedge_{i=1}^{l} g_{1,i}(x; \lambda) \lesssim_{1,i} 0, \text{ and } q \in Q_1 \\ & \vdots \\ g_{m,l+1}(x; \lambda) & \text{if } \bigwedge_{i=1}^{l} g_{m,i}(x) \lesssim_{m,i} 0, \text{ and } q \in Q_m, \end{cases} \tag{38}$$

The value N is a placeholder for either the dimensionality of the state space $\mathbb{R}^n$, or the joint dimensionality of the system and the stochastic disturbance inputs $\mathbb{R}^n \times \mathcal{W}$, according to context. The sets $Q_1, \ldots, Q_m \subseteq Q$ denote constraints on the automaton states and $\lesssim$ denotes either a strict- or non-strict inequality. This makes the form of Eq. (38) suitable as a template for expressing Streett supermartingales $V(x, q; \theta) \equiv F(x, q; \theta)$, supporting invariants $I(x, q; \eta) \equiv \bigwedge[F(x, q; \eta) \leq 0]$, dynamical models $f(x, w; \kappa) \equiv F((x, w), -; \kappa)$ as well as the symbolic post-expectation $\operatorname{Post} V(x, q; \theta, \kappa) \equiv F(x, q; \theta, \kappa)$.

Assuming, without loss of generality, that each observable proposition in Π (cf. Sect. 3) corresponds to a single inequality over the state space $\mathbb{R}^n$, the transition function $\delta(q, \langle\!\langle x \rangle\!\rangle)$ of the automaton takes the form of template $D : \mathbb{R}^n \times Q \to Q$:

$$D(x, q) = \begin{cases} q'_1 & \text{if } \bigwedge_{i=1}^{l} g_{1,i}(x) \lesssim_{1,i} 0, \text{ and } q = q_1 \\ & \vdots \\ q'_m & \text{if } \bigwedge_{i=1}^{l} g_{m,i}(x) \lesssim_{m,i} 0, \text{ and } q = q_m. \end{cases} \tag{39}$$

where each of the automaton's transitions corresponds to a case of Eq. (39).

The requirements of Eqs. (15) to (20) reduce to a conjunction of sentences of the form Eq. (40), namely, a universally quantified implication over N-dimensional real-valued variables, where each implication has a premise that is a finite conjunction of inequalities (where L is a placeholder for the number of conjuncts), and a consequent that is a single non-strict inequality:

$$\forall y \in \mathbb{R}^N \colon \bigwedge_{i=1}^{L} g_i(y; \lambda) \lesssim_i 0 \implies g_{L+1}(y; \lambda) \leq 0, \tag{40}$$

This is because our construction only invokes compositions of the templates F and D that produce results that are representable in the form of template F, namely, a piecewise function over $\mathbb{R}^N \times Q$ with parameters $\lambda \in \Lambda$. In combination with rewriting at the level of propositional logic, we establish Eq. (40).

Finally, we note that the conjunction of sentences of the form Eq. (40) is existentially quantified over the certificate, invariant and control parameters, as well as the parameters ϵ and M, all of which we notationally subsume within λ. We now discuss algorithms for finding a satisfying assignment to these existentially quantified parameters under the problem scenarios outlined earlier.

4.1 Piecewise Polynomial Systems and Templates

Under the assumption that all functions g in the templates Eqs. (38) and (39) are polynomials in $x \in \mathbb{R}^N$ and $\lambda \in \Lambda$, the synthesis problem reduces to an existentially quantified conjunction of statements in the form of Eq. (40), which are in turn universally quantified implications over polynomial inequalities. This synthesis problem belongs to the first-order theory of nonlinear real arithmetic (NRA) and is decidable using quantifier elimination [24].

4.2 Piecewise Linear Systems and Templates with Parametric Guards

Despite its decidability, the decision procedures for NRA are computationally feasible only for small problems. By making additional assumptions about the system dynamics and templates, we improve the feasibility of the synthesis problem using Farkas' Lemma. The Farkas' Lemma [44, p.32 & Table 2.4.1, p.34] states that the following two sentences are equivalent:

$$\forall y \in \mathbb{R}^N : Ay \leq b \implies c^{\mathsf{T}} y \leq d \tag{41}$$

$$\exists z \in \mathbb{R}^L_{\geq 0} : \begin{pmatrix} A^{\mathsf{T}} z = c \\ \wedge\ b^{\mathsf{T}} z \leq d \end{pmatrix} \vee \begin{pmatrix} A^{\mathsf{T}} z = 0 \\ \wedge\ b^{\mathsf{T}} z < 0 \end{pmatrix} \tag{42}$$

with z constituting a freshly introduced set of variables. This rewrite eliminates the quantifier alternation and yields a decision problem in the first-order existential theory of non-linear real arithmetic (QF_NRA). In the case where the functions g in Eq. (40) are linear in the variables $y \in \mathbb{R}^N$, and with the help of a technical result that allows strict inequalities in Eq. (40) to be replaced by non-strict inequalities (cf. [20, Lemma 1]), we find that Eq. (40) takes the form of Eq. (41), allowing Farkas' Lemma to be applied.

Example 4 (General Control). Considering Example 3, suppose we want to synthesise a value for the control parameter κ such that the specification Φ is satisfied almost surely, along with a Streett supermartingale and supporting invariant. For this purpose, we introduce template parameters $\theta = (\alpha_0, \beta_0, \alpha_1, \beta_1, \alpha_2, \beta_2)$, $\eta = (\eta_1, \eta_2, \eta_3, \eta_4)$ and template the Streett supermartingale and supporting invariant using the following form:

$$\begin{aligned} V(x, q_1; \theta) &= \alpha_1 \cdot x + \beta_1 \\ I(x, q_1; \eta) &= fv(\eta_1 \cdot x \leq \eta_2) \wedge (\eta_3 \cdot x \leq \eta_4) \end{aligned} \tag{43}$$

proceeding analogously for states other than q_1, which yields for $q \in \{q_0, q_1, q_2\}$ the following expression for $\operatorname{Post} V$ in terms of the control parameter κ:

$$\operatorname{Post} V(x, q; \theta, \kappa) = \begin{cases} \alpha_0 \kappa \cdot x + \beta_0 & \text{if } x \geq 1 \\ \alpha_1 \kappa \cdot x + \beta_1 & \text{if } -1 \leq x < 1 \\ \alpha_2 \kappa \cdot x + \beta_2 & \text{if } x < -1 \end{cases} \tag{44}$$

Substituting these expressions into Eqs. (15) to (20) results in a conjunction of implications of the form Eq. (40) over inequalities that are linear in the variable $x \in \mathbb{R}$, but polynomial over the existentially quantified parameters. For example, the non-increasing drift condition associated with q_1 (cf. Eqs. (19), (34) and (35)) corresponds to a number of implications, one for each case of the piecewise-defined $\operatorname{Post} V(x, q_1; \theta, \kappa)$. Considering the case $x \geq 1$, we see that the templated implication analogous to Eq. (34) is:

$$\forall x \in \mathbb{R} \colon \underbrace{\begin{bmatrix} \eta_1 \\ \eta_3 \\ -1 \end{bmatrix} \begin{bmatrix} x \end{bmatrix} \leq \begin{bmatrix} \eta_2 \\ \eta_4 \\ -1 \end{bmatrix}}_{I(x, q_1; \eta) \wedge (x \geq 1)} \implies \underbrace{\begin{bmatrix} \alpha_0 \kappa - \alpha_1 \end{bmatrix} \begin{bmatrix} x \end{bmatrix} \leq \begin{bmatrix} \beta_1 - \beta_0 \end{bmatrix}}_{\operatorname{Post} V(x, q_1; \theta, \kappa) \leq V(x, q_1; \theta)} \tag{45}$$

which is in the form of Eq. (41) and yields an existentially quantified disjunction of polynomial inequalities (over the existentially quantified variables, which include the template and control parameters) once rewritten into form Eq. (42), namely a problem in the existential first-order theory of non-linear real arithmetic. □

4.3 Piecewise Linear Systems and Templates with Known Guards

Supposing additionally that an inductive invariant is externally provided, we further improve the computational feasibility of the synthesis problem by reducing it to a quadratically-constrained programming (QCP) problem. In this setting, all inequalities in the premise of Eq. (40) are known linear inequalities of the vector y, and the matrix A and vector b in Eq. (41) are constant (i.e. contain no existentially quantified variables). Therefore, the satisfiability of the premise of Eq. (41) is decidable using linear programming to check whether $Ay \leq b$ admits any solution for y. After removing any implications of the form Eq. (41) which possess an unsatisfiable premise, we may exploit a special case of Farkas' Lemma that assumes a satisfiable premise [20, Theorem 3]. This version states that if there exists a solution to the system $Ay \leq b$, then the formula Eq. (41) is equivalent to

$$\exists z \in \mathbb{R}^L_{\geq 0} \colon \begin{pmatrix} A^\mathsf{T} z = c \\ \wedge \; b^\mathsf{T} z \leq d \end{pmatrix}. \tag{46}$$

This formula is an existentially quantified conjunction of inequalities, thus transforming the synthesis problem into deciding the satisfiability of a conjunction of polynomial constraints. Such a system of polynomial constraints is reducible to QCP, since higher degree polynomial expressions may be constructed from quadratic constraints by introducing fresh variables. This establishes the reduction to QCP for *shielded control* when applied to piecewise linear systems and templates, with known invariant. Furthermore, as illustrated in Example 6, if additionally the system is autonomous, the synthesis problem reduces to an LP.

Example 5 (Shielded Control). Continuing from Example 4, we note that if a sufficiently strong invariant is provided a priori (such as that of Eq. (22)), then the synthesis problem reduces to implications of the form Eq. (40) with the property that the linear inequalities occurring within the premise of an implication have constant coefficients. Instead of Eq. (45), for example, we obtain:

$$\forall x \in \mathbb{R}\colon \quad \underbrace{\begin{bmatrix}1\\-1\\-1\end{bmatrix}\begin{bmatrix}x\end{bmatrix} \leq \begin{bmatrix}0.9\\0.2\\-1\end{bmatrix}}_{I(x,q_1;1,0.9,-1,0.2)\wedge(x\geq 1)} \quad \Longrightarrow \quad \underbrace{\begin{bmatrix}\alpha_0\kappa - \alpha_1\end{bmatrix}\begin{bmatrix}x\end{bmatrix} \leq \begin{bmatrix}\beta_1 - \beta_0\end{bmatrix}}_{\text{Post } V(x,q_1;\theta,\kappa)\leq V(x,q_1;\theta)} \tag{47}$$

The premise of Eq. (47) is a known system of linear inequalities, so if its premise is satisfiable (decidable via linear programming) an application of Eq. (46) transforms the synthesis problem into an existentially quantified conjunction of polynomial constraints. The particular constraint Eq. (47) has an unsatisfiable premise, however, and is thus vacuously true. □

Example 6 (Verification). Assuming that $\kappa = 0.5$, the dynamical model results in an autonomous system, and if a sufficiently strong supporting invariant is provided a priori (as is precisely the case in Example 3), then the implication Eq. (47) becomes:

$$\forall x \in \mathbb{R}\colon \quad \underbrace{\overbrace{\begin{bmatrix}1\\-1\\-1\end{bmatrix}}^{A}\begin{bmatrix}x\end{bmatrix} \leq \overbrace{\begin{bmatrix}0.9\\0.2\\-1\end{bmatrix}}^{b}}_{I(x,q_1;1,0.9,-1,0.2)\wedge(x\geq 1)} \quad \Longrightarrow \quad \underbrace{\overbrace{\begin{bmatrix}0.5\cdot\alpha_0 - \alpha_1\end{bmatrix}}^{c^{\mathsf{T}}}\begin{bmatrix}x\end{bmatrix} \leq \overbrace{\begin{bmatrix}\beta_1 - \beta_0\end{bmatrix}}^{d}}_{\text{Post } V(x,q_1;\theta,0.5)\leq V(x,q_1;\theta)} \tag{48}$$

In this case, for an implication with satisfiable premise, we may apply Eq. (46) to obtain an existentially quantified conjunction of inequalities that are linear in x, but further note that matrix A and vector b have constant entries, whereas the vector c and scalar d are linear expressions over template variables. Thus, an application of Eq. (46) generates an existentially quantified conjunction of linear constraints, which is decidable using a linear program. □

5 Experimental Evaluation

We implement our algorithmic technique for the synthesis of Streett supermartingales, supporting invariants, and control policies. Our implementation does not require externally provided invariants, and assumes a template for the Streett supermartingale that is linear in the state variables, and thereby of the form Eq. (38) with a single case for each automaton state. We assume a convex polyhedral template for the supporting invariant, and apply Farkas' Lemma to produce a decision problem in QF_NRA (Sect. 4.2). In Table 1, we demonstrate examples of ω-regular properties and of infinite-state probabilistic systems, with piecewise linear dynamics, certificates and supporting invariants. The *Output* column of Table 1 describes the synthesis problem (cf. Sect. 4): VIC indicates *general control*; VC indicates *shielded control*; VI indicates synthesis of Streett supermartingales and supporting invariants for an autonomous stochastic dynamical model; V indicates *verification* (namely, synthesis of only Streett supermartingales, using an externally provided invariant).

We use the Spot library [30] to translate the LTL formulae shown in Table 1 into deterministic Streett automata with state-based acceptance conditions. We use SymPy [47] to perform symbolic manipulations and generate the corresponding decision problem, which we solve using an off-the-shelf SMT solver, Z3 [38,50]. The systems in Table 1 are all infinite-state, namely continuous-state models, with the exception of `evenOrNegative` that has a countably-infinite state space. The benchmarks make use of discrete random variables, which allows for the post-expectation to be calculated by weighted enumeration of probabilistic choices in the product process (with the exception of `evenOrNegative`, for which the post-expectation is provided manually). Since our implementation entails deterministic algorithms, we provide the time associated with a single execution owing to negligible variance in these timings.

Table 1. Output of our experiments for a range of infinite-state probabilistic systems and ω-regular properties.

Benchmark	ω-Regular Specification	Output	Time [s]
`evenOrNegative` (Fig. 1)	$\mathsf{GF}(x \text{ even}) \vee \mathsf{FG}(x < 0)$	V	0.09
`SafeRWalk1`	$\mathsf{G}(x < 100)$	VIC	1.09
`PersistRW`	$\mathsf{FG}(x \leq 10)$	VI	1.16
`RecurRW`	$\mathsf{GF}(x > 100)$	VI	1.49
`SafeRWalk2`	$\mathsf{G}(x \geq 10)$	VIC	1.09
`GuaranteeRW`	$\mathsf{G}(x \geq -10) \rightarrow \mathsf{F}(x \geq 10^3)$	VI	5.61
`Temperature1`	$\mathsf{FG}(\neg\text{Hot} \wedge \neg\text{Cold})$	VIC	4.11
`Temperature2`	$\mathsf{GF}(t \leq 30) \wedge \mathsf{G}(t \leq 60)$	VI	28.93
`Temperature3`	$\mathsf{G}(\text{Safe}) \wedge [\mathsf{GF}(\text{Cold}) \rightarrow \mathsf{GF}(\text{Hot})]$	VIC	28.58
`Temperature4`	$\mathsf{G}(\text{Safe}) \wedge [\mathsf{GF}(\text{Cold}) \rightarrow \mathsf{GF}(\text{Hot})]$	VC	4.64
`FinMemoryControl`	$\mathsf{GF}(x \leq 0) \rightarrow \mathsf{GF}(x \geq 100)$	VIC	16.73

The benchmark `Temperature1` (Table 1) is an instance of a *general control* problem (Sect. 4) that models an air-conditioned room that dissipates heat to its surroundings (at temperature x_{ext}), with stochastic fluctuations of the room temperature. The state $x_t \in \mathbb{R}$ is the temperature of the room, $x_{\text{ext}} = 280\text{K}$ is the external temperature, and the desired temperature is $x_{\text{set}} = 295\text{K}$, with $x_0 = x_{\text{ext}}$. The system has the following dynamics:

$$x_{t+1} = x_t - \frac{1}{100}(x_t - x_{\text{ext}}) + (\alpha x_t + \beta) + \frac{1}{10}(2w_t - 1), \tag{49}$$

with $w_t \sim \text{Bernoulli}(0.5)$. The dynamics depend upon the parameters α, β of the controller, restricted to $\alpha, \beta \in [-10, 10]$, to reflect the capabilities of the controller. We define two observations, $\Pi = \{\text{Hot}, \text{Cold}\}$ with $\langle\!\langle x \rangle\!\rangle = \{\text{Hot}\}$ when the temperature x exceeds $x_{\text{set}} + 3$, $\langle\!\langle x \rangle\!\rangle = \{\text{Cold}\}$ when the temperature falls below $x_{\text{set}} - 3$, and $\langle\!\langle x \rangle\!\rangle = \emptyset$ otherwise. The specification is $\mathsf{FG}(\neg\text{Hot} \wedge \neg\text{Cold})$, namely, that the temperature eventually persists within the interval $(292, 298)$ around $x_{\text{set}} = 295$. Our method synthesises a certificate, supporting invariant, and controller with $\alpha = -1/32, \beta = 4787/512$. The supporting invariant is a conjunction of two linear inequalities at each automaton state.

We next illustrate how shielding improves the efficiency of our synthesis algorithm. `Temperature3` involves the same controlled dynamics as Eq. (49), but we add a new observation $\{\text{Safe}\}$ associated with the temperature being under 310 K, and aim to satisfy the property $\mathsf{G}(\text{Safe}) \wedge (\mathsf{GF}(\text{Cold}) \rightarrow \mathsf{GF}(\text{Hot}))$. To synthesise a memoryless controller $\alpha = -1/64, \beta = 9/2$ for `Temperature3` along with a suitable inductive invariant requires a total of 28.58 s (of which the QF_NRA solver requires 23.65 s). In `Temperature4` we consider a shielded memoryless controller that ensures the temperature always stays under 310 K:

$$x_{t+1} = x_t - \frac{1}{100}(x_t - x_{\text{ext}}) + \frac{1}{10}(2w_t - 1) + \begin{cases} \alpha x_t + \beta & x_t < 305 \\ -3 & x_t \geq 305 \end{cases} \tag{50}$$

and we desire a certified controller for the same reactivity property as in benchmark `Temperature3`. We constrain $\alpha, \beta \in [-5, 5]$ (as modelling assumptions), and we impose $305 \cdot \alpha + \beta < 5.24$ to ensure that the temperature never exceeds 310K. We provide an invariant a priori, and the resulting QCP is solvable in 0.03 s to obtain $\alpha = -43/3200, \beta = 4$.

To illustrate how our framework is applicable to finite memory controllers, we consider in `FinMemoryControl` (Table 1) a controller that has one bit of memory (denoted by $b \in \{0, 1\}$), which is updated according to the current state x. That is, the system has state space $\mathbb{R} \times \{0, 1\}$, with update function (cf. Eq. (12)):

$$f(x, b, w; \kappa) = \begin{cases} (x + \alpha + w, 1) & \text{if } b = 1 \wedge l \cdot x \leq m \\ (x + \alpha + w, 0) & \text{if } b = 1 \wedge l \cdot x > m \\ (x + w - 1, 1) & \text{if } b = 0 \wedge p \cdot x \leq q \\ (x + w - 1, 0) & \text{if } b = 0 \wedge p \cdot x > q \end{cases} \tag{51}$$

where $\kappa = (l, m, p, q, \alpha)$ is the set of control parameters. That is, we wish to synthesise the output of the controller, α, but also the logic that determines how the controller's memory is to be updated, given the reactivity specification $\mathsf{GF}(x \leq 0) \rightarrow \mathsf{GF}(x \geq 100)$. This is a decision problem in QF_NRA given that the guards of the template for $f(x, b, w; \kappa)$ contain template parameters (Sect. 4.2). Our method finds $\alpha = 56, l = 1/8, m = 14, p = 1/2, q = 51$.

In summary, we find that our synthesis procedure based on Farkas' Lemma (Sect. 4.2) allows for the practical synthesis of Streett supermartingales, supporting invariants and control parameters for a range of ω-regular properties for infinite (countable/continuous) state piecewise linear probabilistic systems, with our tool terminating in under 30 s for the examples considered. We further illustrate that stronger assumptions (e.g. the external provision of shields or supporting invariants) improve the computational efficiency of control synthesis (cf. `Temperature3` vs. `Temperature4`).

6 Related Work

The verification of finite Markov chains is a classic topic for which automated and scalable algorithmic tools exist [41], which combine graph algorithms and linear algebra to directly compute the probability of satisfying an ω-regular specification [10]. This approach exploits the limit behaviour of finite Markov chains, reducing the problem to computing the reachability probabilities of bottom strongly connected components by leveraging the finite graph structure. These techniques do not, however, apply to probabilistic processes over countably infinite or continuous state spaces, which are the focus of this work.

Verification of continuous-state Markov processes has been addressed via two main strategies [42]. The first approximates the continuous-state process with an abstract finite state process (e.g., through state space discretisation) and performs finite-state model checking on the abstraction [29,58,62,69]. The second strategy certifies the property of interest by providing a suitable certificate, using supermartingale theory to analyse Markov chains over general state spaces [48, Chapter 8.4]. This includes supermartingale certificates for specific linear-time properties including almost-sure reachability [15], probabilistic safety [21,23,59], reach-avoidance [22], persistence, and recurrence [17,43]. These rules are justified using martingale theory, including concentration inequalities (e.g. Azuma's inequality [23]) and the supermartingale convergence theorem [31, Theorem 5.2.9, p.236], with recent order-theoretic justifications [59].

Although prior work introduced supermartingale proof rules for almost-sure persistence and recurrence [17], these are too conservative for general reactivity properties. For instance, in Fig. 1, a reactivity property (a disjunction of persistence and recurrence requirements) holds almost-surely, even though neither disjunct holds almost-surely. Recent work [8] addressed proving ω-regular properties with deterministic Streett automata by synthesising a control policy and barrier certificates for the persistence component of each Streett pair. However, as the authors mention, this approach disregards the recurrence component and is thus conservative and mainly suited to safety specifications [8, Section 8.1].

Our approach, by contrast, applies to general Streett pairs, using the Robbins & Siegmund convergence theorem (Theorem 1) to establish that the disjunction of persistence and recurrence properties in each Streett pair is satisfied with probability one, without requiring either disjunct to hold almost surely. While the Robbins & Siegmund convergence theorem has applications in statistics [9], stochastic optimisation [49, Theorem 17.15], and reinforcement learning [12, Proposition 4.2], we are the first to apply it to derive a supermartingale certificate for Streett conditions and, as a result, ω-regular properties.

The algorithmic synthesis of supermartingale certificates for reachability, probabilistic safety, persistence, and recurrence has been addressed for affine programs and certificates using Farkas' Lemma [6,15,20,23] and for polynomial programs and certificates using Putinar's Positivstellensatz [19,21], producing linear- or quadratically-constrained programs, assuming suitably strong inductive invariants are provided a priori. These techniques were first introduced for the synthesis of ranking certificates and invariants for deterministic programs [26,27]. We apply these techniques to synthesise Streett supermartingales, supporting invariants, and control policies by deciding the satisfiability of a single query in the existential first-order theory of reals (Sect. 5), or by solving a QCP or LP when suitable invariants are externally provided (Sect. 4.3), which may be derived from a shield associated with the controller [7], or when a polyhedral enclosure for the reachable states is known or computed a priori with other methods [25,32,46,53,57]. Our approach to the joint synthesis of certificates and supporting invariants is in principle applicable to other supermartingale notions studied by prior work [6,17,21,59].

The problem of certified control synthesis in infinite state Markov decision processes (MDPs) has been addressed for specific objectives, such as reachability-reward objectives in countable-state MDPs [11] and reach-avoid specifications for continuous-state MDPs [22,64,66], as well as specific infinite-horizon properties [60,61]. Here, we provide an automated synthesis approach (Sect. 4) applicable to general reactivity properties over continuous-state stochastic processes.

Further automata-theoretic approaches such as recursive Markov chains (RMC) [33,67] and probabilistic pushdown automata (pPDA) [13,40,65] provide means for specifying stochastic processes over countably infinite state spaces. The ω-regular model checking problem for these has been studied which, under some restrictions on the model, is decidable [14]. By contrast, our Streett supermartingale theorems (Sect. 2) apply to general stochastic processes (including over uncountably infinite state spaces), though identifying the class of ω-regular properties and stochastic processes for which Streett supermartingales are complete (analogous to the notion of positive almost-sure termination [34]) remains an open problem. However, the algorithms in Sect. 4 are relatively complete: if a Streett supermartingale in linear or polynomial form with a known degree exists, our algorithm will compute it.

7 Conclusion

We have introduced the first supermartingale certificate for ω-regular properties, by exploiting the Robbins & Siegmund convergence theorem applied to deterministic Streett automata. Our result is the most expressive supermartingale certificate to date, enabling effective almost-sure verification of reactivity properties without requiring each persistence and recurrence component to hold with probability one, as in previous work. We have provided an algorithm to reduce the problem of synthesising Streett supermartingales along with supporting inductive invariants and control policies to symbolic (SMT) and numerical (QCP, LP) decision procedures, and have demonstrated the practical efficacy of our method on several verification and control examples.

Our approach lends itself to extension towards quantitative verification [21, 59], and towards effective algorithmic synthesis of supermartingale certificates via Positivstellensatz [19]. Furthermore, it is open to data-driven techniques along the lines of recent work on neural certificate learning [1–3, 18, 22, 36, 52].

Acknowledgments. This research was supported in part by the EPSRC Doctoral Training Partnership, and the Department of Computer Science Scholarship at the University of Oxford.

References

1. Abate, A., Ahmed, D., Giacobbe, M., Peruffo, A.: Formal synthesis of Lyapunov Neural Networks. IEEE Control. Syst. Lett. **5**(3), 773–778 (2021)
2. Abate, A., Edwards, A., Giacobbe, M., Punchihewa, H., Roy, D.: Quantitative verification with neural networks. In: CONCUR. LIPIcs, vol. 279, pp. 22:1–22:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023)
3. Abate, A., Giacobbe, M., Roy, D.: Learning probabilistic termination proofs. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 3–26. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_1
4. Abate, A., Giacobbe, M., Schnitzer, Y.: Bisimulation learning. In: Ganesh, V., Gurfinkel, A. (eds.) CAV 2024. LNCS, vol. 14683, pp. 161–183. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-65633-0_8
5. Abate, A., Katoen, J., Lygeros, J., Prandini, M.: Approximate model checking of stochastic hybrid systems. Eur. J. Control. **16**(6), 624–641 (2010)
6. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. Proc. ACM Program. Lang. **2**(POPL), 34:1–34:32 (2018)
7. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI, pp. 2669–2678. AAAI Press (2018)
8. Anand, M., Lavaei, A., Zamani, M.: Compositional synthesis of control barrier certificates for networks of stochastic systems against ω-regular specifications. Nonlinear Anal. Hybrid Syst **51**, 101427 (2024)
9. Anbar, D.: An application of a theorem of Robbins and Siegmund. Ann. Stat. **4**(5), 1018–1021 (1976)
10. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)

11. Batz, K., Biskup, T.J., Katoen, J., Winkler, T.: Programmatic strategy synthesis: resolving nondeterminism in probabilistic programs. Proc. ACM Program. Lang. **8**(POPL), 2792–2820 (2024)
12. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Optimization and Neural Computation Series, vol. 3. Athena Scientific (1996)
13. Brázdil, T., Esparza, J., Kiefer, S., Kucera, A.: Analyzing probabilistic pushdown automata. Formal Methods Syst. Des. **43**(2), 124–163 (2013)
14. Brázdil, T., Kučera, A., Stražovský, O.: On the decidability of temporal properties of probabilistic pushdown automata. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 145–157. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31856-9_12
15. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
16. Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: Müller-Olm, M., Seidl, H. (eds.) SAS 2014. LNCS, vol. 8723, pp. 85–100. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10936-7_6
17. Chakarov, A., Voronin, Y.-L., Sankaranarayanan, S.: Deductive proofs of almost sure persistence and recurrence properties. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 260–279. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_15
18. Chang, Y., Roohi, N., Gao, S.: Neural Lyapunov control. In: NeurIPS, pp. 3240–3249 (2019)
19. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_1
20. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. ACM Trans. Program. Lang. Syst. **40**(2), 7:1–7:45 (2018)
21. Chatterjee, K., Goharshady, A.K., Meggendorfer, T., Žikelić, D.: Sound and complete certificates for quantitative termination analysis of probabilistic programs. In: Shoham, S., Vizel, Y. (eds.) CAV (2022. LNCS, vol. 13371, pp. 55–78. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_4
22. Chatterjee, K., Henzinger, T.A., Lechner, M., Žikelić, D.: A learner-verifier framework for neural network controllers and certificates of stochastic systems. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13993, pp. 3–25. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_1
23. Chatterjee, K., Novotný, P., Žikelić, D.: Stochastic invariants for probabilistic termination. In: POPL, pp. 145–160. ACM (2017)
24. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition-preliminary report. SIGSAM Bull. **8**(3), 80–90 (1974)
25. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_39
26. Colóon, M.A., Sipma, H.B.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_6

27. Colón, M.A., Sipma, H.B.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_36
28. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: POPL, pp. 265–276. ACM (2007)
29. Desharnais, J., Laviolette, F., Tracol, M.: Approximate analysis of probabilistic processes: logic, simulation and games. In: QEST, pp. 264–273. IEEE Computer Society (2008)
30. Duret-Lutz, A., et al.: From spot 2.0 to spot 2.10: what's new? In: Shoham, S., Vizel, Y. (eds.) CAV 2022. LNCS, vol. 13372, pp. 174–187. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_9
31. Durrett, R.: Probability: Theory and Examples, 4th edn. Cambridge University Press, Cambridge (2010)
32. Ernst, M.D., et al.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
33. Etessami, K., Yannakakis, M.: Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. J. ACM **56**(1), 1:1–1:66 (2009)
34. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: soundness, completeness, and compositionality. In: POPL, pp. 489–501. ACM (2015)
35. Gehr, T., Misailovic, S., Vechev, M.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 62–83. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_4
36. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. In: ESEC/SIGSOFT FSE, pp. 633–645. ACM (2022)
37. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022)
38. Jovanovic, D., de Moura, L.: Solving non-linear arithmetic. ACM Commun. Comput. Algebra **46**(3/4), 104–105 (2012)
39. Křetínský, J., Meggendorfer, T., Sickert, S., Ziegler, C.: Rabinizer 4: from LTL to your favourite deterministic automaton. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 567–577. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_30
40. Kucera, A., Esparza, J., Mayr, R.: Model checking probabilistic pushdown automata. Log. Methods Comput. Sci. **2**(1) (2006)
41. 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
42. Lavaei, A., Soudjani, S., Abate, A., Zamani, M.: Automated verification and synthesis of stochastic hybrid systems: a survey. Automatica **146**(12) (2022)
43. Lechner, M., Žikelić, D., Chatterjee, K., Henzinger, T.A.: Stability verification in stochastic control systems via neural network supermartingales. In: AAAI, pp. 7326–7336. AAAI Press (2022)
44. Mangasarian, O.L.: Nonlinear Programming. Society for Industrial and Applied Mathematics (1994)
45. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: PODC, pp. 377–410. ACM (1990)
46. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems - Safety. Springer, New York (1995). https://doi.org/10.1007/978-1-4612-4222-2
47. Meurer, A., et al.: SymPy: symbolic computing in Python. PeerJ Prepr. **4**, e2083 (2016)

48. Meyn, S., Tweedie, R.L., Glynn, P.W.: Markov Chains and Stochastic Stability, 2nd edn. Cambridge Mathematical Library. Cambridge University Press, New York (2009)
49. Mohri, M., Rostamizadeh, A., Talwalkar, A.: Foundations of Machine Learning. Adaptive computation and Machine Learning. MIT Press, Cambridge (2012)
50. 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
51. Murali, V., Trivedi, A., Zamani, M.: Closure certificates. In: HSCC, pp. 10:1–10:11. ACM (2024)
52. Nadali, A., Murali, V., Trivedi, A., Zamani, M.: Neural closure certificates. In: AAAI, pp. 21446–21453. AAAI Press (2024)
53. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: DIG: a dynamic invariant generator for polynomial and array invariants. ACM Trans. Softw. Eng. Methodol. **23**(4), 30:1–30:30 (2014)
54. Pollard, D.: A User's Guide to Measure Theoretic Probability. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, New York (2001)
55. Robbins, H., Siegmund, D.: A convergence theorem for non negative almost supermartingales and some applications. Optim. Methods Stat. **1971**, 233–257 (1971)
56. Safra, S.: On the complexity of omega-automata. In: FOCS, pp. 319–327. IEEE Computer Society (1988)
57. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27864-1_7
58. Soudjani, S.E.Z., Abate, A.: Adaptive and sequential gridding for abstraction and verification of stochastic processes. SIAM J. Appl. Dyn. Syst. **12**(2), 921–956 (2012)
59. Takisaka, T., Oyabu, Y., Urabe, N., Hasuo, I.: Ranking and repulsing supermartingales for reachability in randomized programs. ACM Trans. Program. Lang. Syst. **43**(2), 5:1–5:46 (2021)
60. Tkachev, I., Abate, A.: Characterization and computation of infinite horizon specifications over markov processes. Theoret. Comput. Sci. **515**, 1–18 (2014)
61. Tkachev, I., Mereacre, A., Katoen, J.P., Abate, A.: Quantitative model checking of controlled discrete-time markov processes. Inf. Comput. **253**(1), 1–35 (2017)
62. Tkachev, I., Abate, A.: Formula-free finite abstractions for linear temporal verification of stochastic hybrid systems. In: HSCC, pp. 283–292. ACM (2013)
63. Vardi, M.Y.: Verification of concurrent programs: the automata-theoretic framework. Ann. Pure Appl. Log. **51**(1–2), 79–98 (1991)
64. Wang, Y., Zhu, H.: Verification-guided programmatic controller synthesis. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13994, pp. 229–250. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_16
65. Winkler, T., Gehnen, C., Katoen, J.-P.: Model checking temporal properties of recursive probabilistic programs. In: FoSSaCS 2022. LNCS, vol. 13242, pp. 449–469. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99253-8_23
66. Yang, Z., Zhang, L., Zeng, X., Tang, X., Peng, C., Zeng, Z.: Hybrid controller synthesis for nonlinear systems subject to reach-avoid constraints. In: Enea, C., Lal, A. (eds.) CAV 2023. LNCS, vol. 13964, pp. 304–325. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37706-8_16
67. Yannakakis, M., Etessami, K.: Checking LTL properties of recursive markov chains. In: QEST, pp. 155–165. IEEE Computer Society (2005)

68. Zamani, M., Esfahani, P.M., Majumdar, R., Abate, A., Lygeros, J.: Symbolic control of stochastic systems via approximately bisimilar finite abstractions. IEEE Trans. Autom. Control **59**(12), 3135–3150 (2014)
69. Zhang, L., She, Z., Ratschan, S., Hermanns, H., Hahn, E.M.: Safety verification for probabilistic hybrid systems. Eur. J. Control. **18**(6), 572–587 (2012)

Lexicographic Ranking Supermartingales with Lazy Lower Bounds

Toru Takisaka[1](✉), Libo Zhang[2], Changjiang Wang[1], and Jiamou Liu[2]

[1] University of Electronic Science and Technology of China, Chengdu, China
takisaka@uestc.edu.cn, 202222080938@std.uestc.edu.cn

[2] The University of Auckland, Auckland, New Zealand
lzha797@aucklanduni.ac.nz, jiamou.liu@auckland.ac.nz

Abstract. *Lexicographic Ranking SuperMartingale* (LexRSM) is a probabilistic extension of *Lexicographic Ranking Function* (LexRF), which is a widely accepted technique for verifying program termination. In this paper, we are the first to propose sound probabilistic extensions of LexRF with a weaker non-negativity condition, called *single-component* (SC) non-negativity. It is known that such an extension, if it exists, will be nontrivial due to the intricacies of the probabilistic circumstances.

Toward the goal, we first devise the notion of *fixability*, which offers a systematic approach for analyzing the soundness of possibly negative LexRSM. This notion yields a desired extension of LexRF that is sound for general stochastic processes. We next propose another extension, called *Lazy LexRSM*, toward the application to automated verification; it is sound over probabilistic programs with linear arithmetics, while its subclass is amenable to automated synthesis via linear programming. We finally propose a LexRSM synthesis algorithm for this subclass, and perform experiments.

1 Introduction

Background 1: Lexicographic RFs with Different Non-negativity Conditions. *Ranking function* (RF) is one of the most well-studied tools for verifying program termination. An RF is typically a real-valued function over program states that satisfies: (a) the *ranking condition*, which requires an RF to decrease its value by a constant through each transition; and (b) the *non-negativity condition*, which imposes a lower bound on the value of the RF so that its infinite descent through transitions is prohibited. The existence of such a function implies termination of the underlying program, and therefore, one can automate verification of program termination by RF synthesis algorithms.

Improving the *applicability* of RF synthesis algorithms, i.e., making them able to prove termination of a wider variety of programs, is one of the core interests in the study of RF. A *lexicographic extension* of RF (LexRF) [8,10] is known as a simple but effective approach to the problem. Here, a LexRF is a function to real-valued *vectors* instead of the reals, and its ranking condition is imposed

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 420–442, 2024.
https://doi.org/10.1007/978-3-031-65633-0_19

$$\begin{array}{ll} \ell_1 : \textbf{skip}\,; & //\boldsymbol{\eta} = (a_1, \underline{b_1}, c_1) \\ \ell_2 : x := 1\,; & //\boldsymbol{\eta} = (\underline{a_2}, b_2, c_2) \\ \quad \cdots & \end{array}$$

Non-negativity condition	$\boldsymbol{\eta}$ should be non-neg. at
Strong (ST) non-neg.	$a_1, b_1, c_1, a_2, b_2, c_2$
Leftward (LW) non-neg.	a_1, b_1, a_2
Single-component (SC) non-neg.	b_1, a_2

Fig. 1. A demo of different non-negativity conditions for LexRFs. There, the ranking dimensions of the LexRF $\boldsymbol{\eta}$ are indicated by underlines, and the last column of the table shows where each condition requires $\boldsymbol{\eta}$ to be non-negative.

with respect to the lexicographic order. For example, the value of a LexRF may change from $(1, 1, 1)$ to $(1, 0, 2)$ through a state transition; here, the value "lexicographically decreases by 1" through the transition, that is, it decreases by 1 in some dimension while it is non-increasing on the left to that dimension. LexRF is particularly good at handling nested structures of programs, as vectors can measure the progress of different "phases" of programs separately. LexRF is also used in top-performing termination provers (e.g., [1]).

There are several known ways to impose non-negativity on LexRFs (see also Fig. 1): (a) *Strong non-negativity*, which requires non-negativity in every dimension of the LexRF; (b) *leftward non-negativity*, which requires non-negativity on the left of the *ranking dimension* of each transition, i.e., the dimension where the value of the LexRF should strictly decrease through the transition; and (c) *single-component non-negativity*, which requires non-negativity only in the ranking dimensions. It is known that any of these non-negativity conditions makes the resulting LexRF *sound* [8,10], i.e., a program indeed terminates whenever it admits a LexRF with either of these non-negativity conditions. For better applicability, single-component non-negativity is the most preferred, as it is the weakest constraint among the three.

Background 2: Probabilistic Programs and Lexicographic RSMs. One can naturally think of a probabilistic counterpart of the above argument. One can consider *probabilistic programs* that admit randomization in conditional branching and variable updates. The notion of RF is then generalized to *Ranking Super-Martingale* (RSM), a function similar to RFs except that the ranking condition requires an RSM to decrease its value *in expectation*. The existence of an RSM typically implies *almost-sure termination* of the underlying program, i.e., termination of the program with probability 1.

Such a probabilistic extension has been actively studied, in fact: probabilistic programs are used in e.g., stochastic network protocols [33], randomized algorithms [18,26], security [6,7,27], and planning [11]; and there is a rich body of studies in RSM as a tool for automated verification of probabilistic programs (see Sect. 8). Similar to the RF case, a lexicographic extension of RSM (*LexRSM*, [2,15]) is an effective approach to improve its applicability. In addition to its advantages over nested structures, LexRSM can also witness almost-sure termination of certain probabilistic programs with infinite expected runtime [2, Fig. 2]; certifying such programs is known as a major challenge for RSMs.

Problem: Sound Probabilistic Extension of LexRF with Weaker Non-negativity. Strongly non-negative LexRF soundly extends to LexRSM in a canonical way [2], i.e., basically by changing the clause "decrease by a constant" in the ranking condition of LexRF to "decrease by a constant *in expectation*". In contrast, the similar extension of leftward or single-component non-negative LexRF yields an *unsound* LexRSM notion [15,20]. To date, a sound LexRSM with the weakest non-negativity in the literature is *Generalized LexRSM* (GLexRSM) [15], which demands leftward non-negativity *and* an additional one, so-called *expected leftward non-negativity.* Roughly speaking, the latter requires LexRSMs to be non-negative in each dimension (in expectation) upon "exiting" the left of the ranking dimension. For example, in Fig. 1, it requires b_2 to be non-negative, as the second dimension of $\boldsymbol{\eta}$ "exits" the left of the ranking dimension upon the transition $\ell_1 \to \ell_2$. GLexRSM does not generalize either leftward or single-component non-negative LexRF, in the sense that the former is strictly more restrictive than the latter two when it is considered over non-probabilistic programs.

These results do not mean that leftward or single-component non-negative LexRF can never be extended to LexRSM, however. More concretely, the following problem is valid (see the last paragraph of Sect. 3 for a formal argument):

> *KEY PROBLEM: Find a sound LexRSM notion that* instantiates[1] *single-component non-negative LexRF, i.e., a LexRSM notion whose condition is no stronger than that of single-component non-negative LexRF in non-probabilistic settings.*

We are motivated to study this problem for a couple of reasons. First, it is a paraphrase of the following fundamental question: *when do negative values of (Lex)RSM cause trouble*, say, to its soundness? This question is a typical example in the study of RSM where the question becomes challenging due to its probabilistic nature. The question also appears in other topics in RSM; for example, it is known that the classical variant rule of Floyd-Hoare logic does not extend to almost-sure termination of probabilistic programs in a canonical way [24], due to the complicated treatment of negativity in RSMs. To our knowledge, this question has only been considered in an ad-hoc manner through counterexamples (e.g., [15,20,24]), and we do not yet have a systematic approach to answering it.

Second, relaxing the non-negativity condition of LexRSM is highly desirable if we wish to fully unlock the benefit of the lexicographic extension in automated verification. A motivating example is given in Fig. 2. The probabilistic program in Fig. 2 terminates almost-surely, but it does not admit any linear GLexRSM (and hence, the GLexRSM synthesis algorithms in [15] cannot witness its almost-sure termination); for example, the function $\boldsymbol{\eta}$ ranks every transition of the program, but violates both leftward and expected leftward non-negativity at the transition $\ell_1 \to \ell_2$ (note $\boldsymbol{\eta}$ ranks this transition in the third dimension; to check the violation of expected leftward non-negativity, also note $\boldsymbol{\eta}$ ranks $\ell_2 \to \ell_4$ in the first dimension). Here, the source of the problem is that the program has two

[1] We use the term "instantiate" to emphasize that we compare LexRSM and LexRF.

```
      $x := 0$;
$\ell_1$ : while $x < 5$ do                $\boldsymbol{\eta} = (15 - 2x, 12 - y, 1)$     $[x < 7]$
$\ell_2$ :     if $y < 10$ then            $\boldsymbol{\eta} = (15 - 2x, 12 - y, 0)$     $[x < 5]$
$\ell_3$ :         $y := y + Unif[1, 2]$   $\boldsymbol{\eta} = (15 - 2x, 11 - y, 2)$     $[y < 10, x < 5]$
           else
$\ell_4$ :         $x := x + Unif[1, 2]$   $\boldsymbol{\eta} = (14 - 2x,      0, 1)$     $[y \geq 10, x < 5]$
           fi
      od
$\ell_5$ :                                 $\boldsymbol{\eta} = (      0,      0, 0)$     $[x \geq 5]$
```

Fig. 2. A probabilistic modification of speedDis1 [4], where *Unif*$[a, b]$ is a uniform sampling from the (continuous) interval $[a, b]$. Inequalities on the right represent invariants. While $\boldsymbol{\eta}$ is not a GLexRSM, it is an LLexRSM we propose; thus it witnesses almost-sure termination of the program.

variables whose progress must be measured (i.e., increment y to 10 in ℓ_3; and increment x to 5 in ℓ_4), but one of their progress measures can be arbitrarily small during the program execution (y can be initialized with any value). Not only that this structure is rather fundamental, it is also expected that our desired LexRSM could handle it, if it exists. Indeed, modify the probabilistic program in Fig. 2 into a non-probabilistic one by changing "*Unif*$[1, 2]$" to "1"; then the program admits $\boldsymbol{\eta}$ as a single-component non-negative LexRF.

Contributions. In this paper, we are the first to introduce sound LexRSM notions that instantiate single-component non-negative LexRF. Our contributions are threefold, as we state below.

- First, in response to the first motivation we stated above, we devise a novel notion of *fixability* as a theoretical tool to analyze if negative values of a LexRSM "cause trouble". Roughly speaking, we identify the source of the trouble as "ill" exploitation of unbounded negativity of LexRSM; our ε-*fixing* operation prohibits such exploitation by basically setting all the negative values of a LexRSM into the same negative value $-\varepsilon$, and we say a LexRSM is ε-*fixable* if it retains the ranking condition through such a transformation. We give more details about its concept and key ideas in Sect. 2.
  The soundness of ε-fixable LexRSM immediately follows from that of strongly non-negative one [2] because any LexRSM becomes strongly non-negative through the ε-fixing operation (after globally adding ε). Fixable LexRSM instantiates single-component non-negative LexRF for general stochastic processes (Theorem 4.3), while also serving as a technical basis for proving the soundness of other LexRSMs. Meanwhile, fixable LexRSM cannot be directly applied to automated verification algorithms due to the inherent non-linearity of ε-fixing; this observation leads us to our second contribution.
- Second, in response to the second motivation we stated above, we introduce *Lazy LexRSM* (LLexRSM) as another LexRSM notion that instantiates single-component non-negative LexRF. LLexRSM does not involve the ε-fixing operation in its definition; thanks to this property, we have a subclass of LLexRSM that is amenable to automated synthesis via linear programming

(see Sect. 6). The LLexRSM condition consists of the single-component non-negative LexRSM condition and *stability at negativity* we propose (Definition 5.1), which roughly requires the following: Once the value of a LexRSM gets negative in some dimension, it must stay negative until that dimension exits the left of the ranking one. For example, $\boldsymbol{\eta}$ in Fig. 2 is an LLexRSM; indeed, $\ell_2 \to \ell_4$ and $\ell_1 \to \ell_5$ are the only transitions where $\boldsymbol{\eta}$ possibly changes its value from negative to non-negative in some dimension (namely, the second one), which is although the right to the ranking dimension (the first one).

We prove linear LLexRSM is sound for probabilistic programs over linear arithmetics (see Theorem 5.4 for the exact assumption). The proof is highly nontrivial, which is realized by subtle use of a refined variant of fixability; we explain its core idea in Sect. 2. Furthermore, Theorem 5.4 shows that expected leftward non-negativity in GLexRSM [15] is actually redundant under the assumption in Theorem 5.4. This is surprising, as expected leftward non-negativity has been invented to restore the soundness of leftward non-negative LexRSM, which is generally unsound.

- Third, we present a synthesis algorithm for the subclass of LLexRSM we mentioned above, and do experiments; there, our algorithms verified almost-sure termination of various programs that could not be handled by (a better proxy of) the GLexRSM-based one. The details can be found in Sect. 7.

2 Key Observations with Examples

Here we demonstrate by examples how intricate the treatment of negative values of LexRSM is, and how we handle it by our proposed notion of fixability.

Blocking "Ill" Exploitation of Unbounded Negativity. Figure 3 is a counterexample that shows leftward non-negative LexRSM is generally unsound (conceptually the same as [15, Ex. 1]). The probabilistic program in Fig. 3 does not terminate almost-surely because the chance of entering ℓ_4 from ℓ_3 quickly decreases as t increases. Meanwhile, $\boldsymbol{\eta} = (\eta_1, \eta_2, \eta_3)$ in Fig. 3 is a leftward non-negative LexRSM over a global invariant $[0 \leq x \leq 1]$; in particular, observe η_2 decreases by 1 in expectation from ℓ_3, whose successor location is either ℓ_4 or ℓ_1.

This example reveals an inconsistency between the ways how the single-component non-negativity and ranking condition evaluate the value of a LexRSM, say $\boldsymbol{\eta} = (\eta_1, \dots, \eta_n)$. The single-component non-negativity claims $\boldsymbol{\eta}$ cannot rank a transition in a given dimension k whenever η_k is negative; intuitively, this means that *any* negative value in the ranking domain $\mathbb{R}$ should be understood as the same state, namely the "bottom" of the domain. Meanwhile, the ranking condition evaluates different negative values differently;

```
        $x := 0; t := 1;$
$\ell_1$ : while $x = 0$ do        $\boldsymbol{\eta} = (2 - x, 0, 2)$
$\ell_2$ :     $t := t + 1$;          $\boldsymbol{\eta} = (2, 0, 1)$
$\ell_3$ :     if prob($2^{-t}$)      $\boldsymbol{\eta} = (2, 0, 0)$
$\ell_4$ :         then $x := 1$      $\boldsymbol{\eta} = (2, -2^t, 0)$
               fi
           od
$\ell_5$ :                          $\boldsymbol{\eta} = (0, 0, 0)$
```

Fig. 3. An example of "ill" exploitation.

a smaller negative value of η_k can contribute more to satisfy the ranking condition, as one can see from the behavior of η_2 in Fig. 3 at ℓ_3. The function $\boldsymbol{\eta}$ in Fig. 3 satisfies the ranking condition over a possibly non-terminating program through "ill" exploitation of this inconsistency; as t becomes larger, the value of η_2 potentially drops more significantly through the transition from ℓ_3, but with a smaller probability.

The first variant of our fixability notion, called *ε-fixability*, enables us to ensure that such exploitation is not happening. We simply set every negative value in a LexRSM $\boldsymbol{\eta}$ to a negative constant $-\varepsilon$, and say $\boldsymbol{\eta}$ is ε-fixable if it retains the ranking condition through the modification[2]. For example, the ε-fixing operation changes the value of η_2 in Fig. 3 at ℓ_4 from -2^t to $-\varepsilon$, and $\boldsymbol{\eta}$ does not satisfy the ranking condition after that. Therefore, $\boldsymbol{\eta}$ in Fig. 3 is not ε-fixable for any $\varepsilon > 0$ (i.e., we successfully reject this $\boldsymbol{\eta}$ through the fixability check). Meanwhile, an ε-fixable LexRSM witnesses almost-sure termination of the underlying program; indeed, the fixed LexRSM is a strongly non-negative LexRSM (by globally adding ε to the fixed $\boldsymbol{\eta}$), which is known to be sound [2].

The notion of ε-fixability is operationally so simple that one might even feel it is a boring idea; nevertheless, its contribution to revealing the nature of possibly negative LexRSM is already significant in our paper. Indeed, (a) ε-fixable LexRSM instantiates single-component non-negative LexRF with an appropriate ε (Theorem 4.3); (b) ε-fixable LexRSM generalizes GLexRSM [15], and the proof offers an alternative proof of soundness of GLexRSM that is significantly simpler than the original one (Theorem 4.4); and (c) its refined variant takes the crucial role in proving soundness of our second LexRSM variant, lazy LexRSM.

Allowing "Harmless" Unbounded Negativity. While ε-fixable LexRSM already instantiates single-component non-negative LexRF, we go one step further to obtain a LexRSM notion that is amenable to automated synthesis, in particular via *Linear Programming* (LP). The major obstacle to this end is the case distinction introduced by ε-fixability, which makes the fixed LexRSM nonlinear. *Lazy LexRSM* (LLexRSM), our second proposed LexRSM, resolves this problem while it also instantiates single-component non-negative LexRF.

Linear LLexRSM is sound over probabilistic programs with linear arithmetics (Theorem 5.4). The key to the proof is, informally, the following observation: *Restrict our attention to probabilistic programs and functions $\boldsymbol{\eta}$ that are allowed in the LP-based synthesis. Then "ill" exploitation in Fig. 3 never occurs*, and therefore, a weaker condition than ε-fixability (namely, the LLexRSM one) suffices for witnessing program termination. In fact, Fig. 3 involves (a) non-linear arithmetics in the program, (b) parametrized if-branch in the program (i.e., the grammar "**if prob**(p) **then** P **else** Q **fi**" with p being a variable), and (c) nonlinearity of $\boldsymbol{\eta}$. None of them are allowed in the LP-based synthesis (at least, in the standard LP-based synthesis via Farkas' Lemma [2,12,15]). Our informal statement above is formalized as Theorem 5.3, which roughly says: Under such

[2] To give the key ideas in a simpler way, the description here slightly differs from the actual definition in Sect. 4; referred results in Sect. 2 are derived from the latter. See Remark 4.1.

a restriction to probabilistic programs and $\boldsymbol{\eta}$, any LLexRSM is (ε, γ)-*fixable*. Here, (ε, γ)-fixability is a refined version of ε-fixability; while it also ensures that "ill" exploitation is not happening in $\boldsymbol{\eta}$, it is less restrictive than ε-fixability by allowing "harmless" unbounded negative values of $\boldsymbol{\eta}$.

Figure 4 gives an example of such a harmless behavior of $\boldsymbol{\eta}$ rejected by ε-fixability. It also shows why we cannot simply use ε-fixability to check an LLexRSM does not do "ill" exploitation. The function $\boldsymbol{\eta} = (\eta_1, \eta_2)$ in Fig. 4 is leftward non-negative over the global invariant $[0 \leq x \leq 1 \wedge t \geq 1]$, so it is an LLexRSM for the probabilistic program there; the program and $\boldsymbol{\eta}$ are also in the scope of LP-based synthesis; but $\boldsymbol{\eta}$ is not ε-fixable for any $\varepsilon > 0$. Indeed, the ε-fixing operation changes the value of η_2 at ℓ_4 from $-2t - 4$ to $-\varepsilon$, and $\boldsymbol{\eta}$ does not satisfy the ranking condition at ℓ_2 after the change. Here we notice that, however, the unbounded negative values of η_2 are "harmless"; that is, the "ill-gotten gains" by the unbounded negative values of η_2 at ℓ_4 are only "wasted" to unnecessarily increase η_2 at ℓ_3. In fact, $\boldsymbol{\eta}$ still satisfies the ranking condition if we change the value of η_2 at ℓ_1, ℓ_2, ℓ_3 to $2, 1$, and 0, respectively.

```
       $x := 0; t := 1;$
$\ell_1$ : while $x = 0$ do          $\boldsymbol{\eta} = (2 - x, \quad t + 1)$
$\ell_2$ :     if prob(0.5)          $\boldsymbol{\eta} = (\quad 2, \quad t)$
$\ell_3$ :         then $t := 4t$    $\boldsymbol{\eta} = (\quad 2, \quad 4t + 2)$
$\ell_4$ :     else $x := 1$         $\boldsymbol{\eta} = (\quad 2, -2t - 4)$
           fi
       od
$\ell_5$ :                           $\boldsymbol{\eta} = (\quad 0, \quad 0)$
```

Fig. 4. An example of "harmless" unbounded negativity.

We resolve this issue by partially waiving the ranking condition of $\boldsymbol{\eta}$ after the ε-fixing operation. It is intuitively clear that the program in Fig. 4 almost-surely terminates, and the intuition here is that the program essentially repeats an unbiased coin tossing until the tail is observed (here, "observe the tail" corresponds to "observe **prob**$(0.5) =$ **false** at ℓ_2"). This example tells us that, to witness the almost-sure termination of this program, we only need to guarantee the program (almost-surely) visits either the terminal location ℓ_5 *or the "coin-tossing location"* ℓ_2 from anywhere else. The ε-fixed $\boldsymbol{\eta}$ in Fig. 4 does witness such a property of the program, as it ranks every transition *except those that are from a coin-tossing location*, namely ℓ_2.

We generalize this idea as follows: Fix $\gamma \in (0, 1)$, and say a program state is a "coin-tossing state" for $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_n)$ in the k-th dimension if η_k drops from non-negative to negative (i.e., the ranking is "done" in the k-th dimension) with the probability γ or higher. Then we say $\boldsymbol{\eta}$ is (ε, γ)-*fixable* (Definition 4.6) if the ε-fixed $\boldsymbol{\eta}$ is a strongly non-negative LexRSM (after adding ε) *except that*, at each coin-tossing state, we waive the ranking condition of $\boldsymbol{\eta}$ in the corresponding dimension. For example, $\boldsymbol{\eta}$ in Fig. 4 is (ε, γ)-fixable for any $\gamma \in (0, 0.5]$. As expected, (ε, γ)-fixable LexRSM is sound for any $\varepsilon > 0$ and $\gamma \in (0, 1)$ (Corollary 4.7).

3 Preliminaries

We recall the technical preliminaries. Omitted details are in [36, Appendix A].

Notations. We assume the readers are familiar with the basic notions of measure theory, see e.g. [5,9]. The sets of non-negative integers and reals are denoted by $\mathbb{N}$ and $\mathbb{R}$, respectively. The collection of all Borel sets of a topological space $\mathcal{X}$ is denoted by $\mathcal{B}(\mathcal{X})$. The set of all probability distributions over the measurable space $(\Omega, \mathcal{B}(\Omega))$ is denoted by $\mathcal{D}(\Omega)$. The value of a vector $\boldsymbol{x}$ at the i-th index is denoted by $\boldsymbol{x}[i]$ or x_i. A subset $D \subseteq \mathbb{R}$ of the reals is *bounded* if $D \subseteq [-x, x]$ for some $x > 0$.

For a finite variable set V and the set val^V of its valuations, we form *predicates* as first-order formulas with atomic predicates of the form $f \leq g$, where $f, g \colon val^V \to R$ and R is linearly ordered. Often, we are only interested in the value of a predicate φ over a certain subset $\mathcal{X} \subseteq val^V$, in which case, we call φ a predicate *over* $\mathcal{X}$. We identify a predicate φ over $\mathcal{X}$ with a function $\tilde{\varphi} \colon \mathcal{X} \to \{0, 1\}$ such that $\tilde{\varphi}(x) = 1$ if and only if $\varphi(x)$ is true. The *semantics* of φ, i.e., the set $\{x \in \mathcal{X} \mid \varphi(x) \text{ is true}\}$, is denoted by $[\![\varphi]\!]$. The *characteristic function* $\mathbf{1}_A : \mathcal{X} \to \{0, 1\}$ of a subset A of $\mathcal{X}$ is a function such that $[\![\mathbf{1}_A = 1]\!] = A$. For a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, we say φ over Ω is *($\mathcal{F}$-)measurable* when $[\![\varphi]\!] \in \mathcal{F}$. For such a φ, the *satisfaction probability* of φ w.r.t. $\mathbb{P}$, i.e., the value $\mathbb{P}([\![\varphi]\!])$, is also denoted by $\mathbb{P}(\varphi)$; we say φ *holds* $\mathbb{P}$*-almost surely* ($\mathbb{P}$-a.s.) if $\mathbb{P}(\varphi) = 1$.

3.1 Syntax and Semantics of Probabilistic Programs

Syntax. We define the syntax of *Probabilistic Programs* (PPs) similarly to e.g., [2,35]. More concretely, PPs have the standard control structure in imperative languages such as if-branches and while-loops, while the if-branching and variable assignments can also be done in either nondeterministic or probabilistic ways. Namely, '**if** $\star$' describes a nondeterministic branching; '**ndet**(D)' describes a nondeterministic assignment chosen from a bounded[3] domain $D \subseteq \mathcal{B}(\mathbb{R})$; '**if prob**$(p)$' with a constant $p \in [0, 1]$ describes a probabilistic branching that executes the '**then**' branch with probability p, or the '**else**' branch with probability $1 - p$; and '**sample**(d)' describes a probabilistic assignment sampled from a distribution $d \in \mathcal{D}(\mathbb{R})$. We consider PPs without *conditioning*, which are also called *randomized programs* [35]; PPs with conditioning are considered in e.g. [32]. The exact grammar is given in [36, Appendix A].

In this paper, we focus our attention on PPs with linear arithmetics; we say a PP is *linear* if each arithmetic expression in it is linear, i.e., of the form $b + \sum_{i=1}^{n} a_i \cdot v_i$ for constants $a_1, \ldots, a_n, b$ and program variables $v_1, \ldots, v_n$.

Semantics. We adopt *probabilistic control flow graph* (pCFG) as the semantics of PPs, which is standard in existing RSM works (e.g., [12,15,35]). Informally, it is a labeled directed graph whose vertices are program locations, and whose edges represent possible one-step executions in the program. Edges are labeled with the necessary information so that one can reconstruct the PP represented by the

[3] This is also assumed in [15] to avoid a complication in possibly negative LexRSMs.

pCFG; for example, an edge e can be labeled with the assignment commands executed through e (e.g., '$x := x + 1$'), the probability $p \in [0, 1]$ that e will be chosen (through '**if prob**(p)'), the guard condition, and so on. Below we give its formal definition for completeness; see [36, Appendix A] for how to translate PPs into pCFGs.

Definition 3.1 (pCFG). A *pCFG* is a tuple $(L, V, \mathit{\Delta}, \mathit{Up}, G)$, where

1. L is a finite set of locations.
2. $V = \{x_1, \ldots, x_{|V|}\}$ is a finite set of program variables.
3. $\mathit{\Delta}$ is a finite set of *(generalized) transitions*[4], i.e., tuples $\tau = (\ell, \delta)$ of a location $\ell \in L$ and a distribution $\delta \in \mathcal{D}(L)$ over successor locations.
4. Up is a function that receives a transition $\tau \in \mathit{\Delta}$ and returns a tuple (i, u) of a target variable index $i \in \{1, \ldots, |V|\}$ and an update element u. Here, u is either (a) a Borel measurable function $u : \mathbb{R}^{|V|} \to \mathbb{R}$, (b) a distribution $d \in \mathcal{D}(\mathbb{R})$, or (c) a bounded measurable set $R \in \mathcal{B}(\mathbb{R})$. In each case, we say τ is *deterministic*, *probabilistic*, and *non-deterministic*, respectively; the collections of these transitions are denoted by $\mathit{\Delta}_d$, $\mathit{\Delta}_p$, and $\mathit{\Delta}_n$, respectively.
5. G is a guard function that assigns a $G(\tau) : \mathbb{R}^{|V|} \to \{0, 1\}$ to each $\tau \in \mathit{\Delta}$.

Below we fix a pCFG $\mathcal{C} = (L, V, \mathit{\Delta}, \mathit{Up}, G)$. A *state* of $\mathcal{C}$ is a tuple $s = (\ell, \boldsymbol{x})$ of location $\ell \in L$ and variable assignment vector $\boldsymbol{x} \in \mathbb{R}^{|V|}$. We write $\mathcal{S}$ to denote the state set $L \times \mathbb{R}^{|V|}$. Slightly abusing the notation, for $\tau = (\ell, \delta)$, we identify the set $[\![G(\tau)]\!] \subseteq \mathbb{R}^{|V|}$ and the set $\{\ell\} \times [\![G(\tau)]\!] \subseteq \mathcal{S}$; in particular, we write $s \in [\![G(\tau)]\!]$ when τ is *enabled at* s, i.e., $s = (\ell, \boldsymbol{x})$, $\tau = (\ell, \delta)$ and $\boldsymbol{x} \in [\![G(\tau)]\!]$.

A pCFG $\mathcal{C}$ with its state set $\mathcal{S}$ can be understood as a transition system over $\mathcal{S}$ with probabilistic transitions and nondeterminism (or, more specifically, a Markov decision process with its states $\mathcal{S}$). Standard notions such as *successors* of a state $s \in \mathcal{S}$, *finite paths*, and *(infinite) runs* of $\mathcal{C}$ are defined as the ones over such a transition system. The set of all successors of $s \in [\![G(\tau)]\!]$ via τ is denoted by $\mathrm{succ}_\tau(s)$. The set of runs of $\mathcal{C}$ is denoted by $\mathit{\Pi}_{\mathcal{C}}$.

Schedulers resolve nondeterminism in pCFGs. Observe there are two types of nondeterminism: (a) nondeterministic choice of $\tau \in \mathit{\Delta}$ at a given state (corresponds to '**if** $\star$'), and (b) nondeterministic variable update in a nondeterministic transition $\tau \in \mathit{\Delta}_n$ (corresponds to '$x_i :=$ **ndet**(D)'). We say a scheduler is $\mathit{\Delta}$-*deterministic* if its choice is non-probabilistic in Case (a).

We assume pCFGs are deadlock-free; we also assume that there are designated locations ℓ_{in} and ℓ_{out} that represent program initiation and termination, respectively. An *initial state* is a state of the form $(\ell_{\mathrm{in}}, \boldsymbol{x})$. We assume a transition from ℓ_{out} is unique, denoted by τ_{out}; this transition does not update anything.

By fixing a scheduler σ and an initial state s_I, the infinite-horizon behavior of $\mathcal{C}$ is determined as a distribution $\mathbb{P}^{\sigma}_{s_I}$ over $\mathit{\Pi}_{\mathcal{C}}$; that is, for a measurable $A \subseteq \mathit{\Pi}_{\mathcal{C}}$, the value $\mathbb{P}^{\sigma}_{s_I}(A)$ is the probability that a run of $\mathcal{C}$ from s_I is in A under σ. We call the probability space $(\mathit{\Pi}_{\mathcal{C}}, \mathcal{B}(\mathit{\Pi}_{\mathcal{C}}), \mathbb{P}^{\sigma}_{s_I})$ the *dynamics of* $\mathcal{C}$ *under* σ *and* s_I. See [9] for the formal construction; a brief explanation is in [36, Appendix A].

[4] Defining these as edges might be more typical, as in our informal explanation. We adopt the style of [2,15] for convenience; it can handle '**if prob**(p)' by a single τ.

We define the *termination time* of a pCFG $\mathcal{C}$ as the function $T^{\mathcal{C}}_{\text{term}} : \Pi_{\mathcal{C}} \to \mathbb{N} \cup \{+\infty\}$ such that $T^{\mathcal{C}}_{\text{term}}(s_0 s_1 \dots) = \inf\{t \in \mathbb{N} \mid \exists \boldsymbol{x}.s_t = (\ell_{\text{out}}, \boldsymbol{x})\}$. Now we formalize our objective, i.e., almost-sure termination of pCFG, as follows.

Definition 3.2 (AST of pCFG). A run $\omega \in \Pi_{\mathcal{C}}$ *terminates* if $T^{\mathcal{C}}_{\text{term}}(\omega) < \infty$. A pCFG $\mathcal{C}$ is *a.s. terminating* (AST) under a scheduler σ and an initial state s_I if a run of $\mathcal{C}$ terminates $\mathbb{P}^{\sigma}_{s_I}$-a.s. We say $\mathcal{C}$ is AST if it is AST for any σ and s_I.

3.2 Lexicographic Ranking Supermartingales

Here we recall mathematical preliminaries of the LexRSM theory. A (Lex)RSM typically comes in two different forms: one is a vector-valued function $\boldsymbol{\eta} : \mathcal{S} \to \mathbb{R}^n$ over states $\mathcal{S}$ of a pCFG $\mathcal{C}$, and another is a stochastic process over the runs $\Pi_{\mathcal{C}}$ of $\mathcal{C}$. We recall relevant notions in these formulations, which are frequently used in existing RSM works [12,15]. We also recall the formal definition of LexRSMs with three different non-negativity conditions in Fig. 1.

LexRSM as a Quantitative Predicate. Fix a pCFG $\mathcal{C}$. An *(n-dimensional) measurable map* (MM) is a Borel measurable function $\boldsymbol{\eta} : \mathcal{S} \to \mathbb{R}^n$. For a given 1-dimensional MM η and a transition τ, The (maximal) *pre-expectation* of η under τ is a function that formalizes "the value of η after the transition τ". More concretely, it is a function $\overline{\mathbb{X}}_{\tau}\eta : [\![G(\tau)]\!] \to \mathbb{R}$ that returns, for a given state s, the maximal expected value of η at the successor state of s via τ. Here, the maximality refers to the set of all possible nondeterministic choices at s.

A *level map* $\mathsf{Lv} : \Delta \to \{0, \dots, n\}$ designates the ranking dimension of the associated LexRSM $\boldsymbol{\eta} : \mathcal{S} \to \mathbb{R}^n$. We require $\mathsf{Lv}(\tau) = 0$ if and only if $\tau = \tau_{\text{out}}$. We say an MM $\boldsymbol{\eta}$ *ranks* a transition τ in the dimension k (under Lv) when $k = \mathsf{Lv}(\tau)$. An *invariant* is a measurable predicate $I : \mathcal{S} \to \{0, 1\}$ such that $[\![I]\!]$ is closed under transitions and $\ell_{\text{in}} \times \mathbb{R}^{|V|} \subseteq [\![I]\!]$. The set $[\![I]\!]$ over-approximates the reachable states in $\mathcal{C}$.

Suppose an n-dimensional MM $\boldsymbol{\eta}$ and an associated level map Lv are given. We say $\boldsymbol{\eta}$ satisfies *the ranking condition* (under Lv and I) if the following holds for each $\tau \neq \tau_{\text{out}}$, $s \in [\![I \wedge G(\tau)]\!]$, and $k \in \{1, \dots, \mathsf{Lv}(\tau)\}$:

$$\overline{\mathbb{X}}_{\tau}\boldsymbol{\eta}[k](s) \leq \begin{cases} \boldsymbol{\eta}[k](s) & \text{if } k < \mathsf{Lv}(\tau), \\ \boldsymbol{\eta}[k](s) - 1 & \text{if } k = \mathsf{Lv}(\tau). \end{cases}$$

We also define the three different non-negativity conditions in Fig. 1, i.e., *STrong (ST)*, *LeftWard (LW)*, and *Single-Component (SC)* non-negativity, as follows:

$$\begin{aligned} &\text{(ST non-neg.) } \forall s \in [\![I]\!].\forall k \in \{1, \dots, n\}. && \boldsymbol{\eta}[k](s) \geq 0, \\ &\text{(LW non-neg.) } \forall \tau \neq \tau_{\text{out}}.\forall s \in [\![I \wedge G(\tau)]\!].\forall k \in \{1, \dots, \mathsf{Lv}(\tau)\}. && \boldsymbol{\eta}[k](s) \geq 0, \\ &\text{(SC non-neg.) } \forall \tau \neq \tau_{\text{out}}.\forall s \in [\![I \wedge G(\tau)]\!]. && \boldsymbol{\eta}[\mathsf{Lv}(\tau)](s) \geq 0. \end{aligned}$$

All the materials above are wrapped up in the following definition.

Definition 3.3 ((ST/LW/SC)-LexRSM map). Fix a pCFG $\mathcal{C}$ with an invariant I. Let $\boldsymbol{\eta}$ be an MM associated with a level map Lv. The MM $\boldsymbol{\eta}$ is called a *STrongly non-negative LexRSM map (ST-LexRSM map)* over $\mathcal{C}$ supported by I if it satisfies the ranking condition and the strong non-negativity under Lv and I. If it satisfies the leftward or single-component non-negativity instead of the strong one, then we call it *LW-LexRSM map* or *SC-LexRSM map*, respectively.

LexRSM as a Stochastic Process. When it comes to automated synthesis, a (Lex)RSM is usually a function $\boldsymbol{\eta}$ over program states, as defined in Definition 3.3. Meanwhile, when we prove the properties of (Lex)RSMs themselves (e.g., soundness), it is often necessary to inspect the behavior of $\boldsymbol{\eta}$ upon the program execution under given scheduler σ and initial state s_I. Such a behavior of $\boldsymbol{\eta}$ is formalized as a sequence $(\mathbf{X}_t)_{t=0}^{\infty}$ of random variables over the dynamics of the underlying pCFG, which forms a *stochastic process*.

A (discrete-time) *stochastic process* in a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is a sequence $(\mathbf{X}_t)_{t=0}^{\infty}$ of $\mathcal{F}$-measurable random variables $\mathbf{X}_t : \Omega \to \mathbb{R}^n$ for $t \in \mathbb{N}$. In our context, it is typically associated with another random variable $T : \Omega \to \mathbb{N} \cup \{+\infty\}$ that describes the termination time of $\omega \in \Omega$. We say T is *AST (w.r.t.* $\mathbb{P}$*)* if $\mathbb{P}(T < \infty) = 1$; observe that, if $(\Omega, \mathcal{F}, \mathbb{P})$ is the dynamics of a pCFG $\mathcal{C}$ under σ and s_I, then $\mathcal{C}$ is AST under σ and s_I if and only if $T_{\mathrm{term}}^{\mathcal{C}}$ is AST w.r.t. $\mathbb{P}$. As standard technical requirements, we assume there is a *filtration* $(\mathcal{F}_t)_{t=0}^{\infty}$ in $(\Omega, \mathcal{F}, \mathbb{P})$ such that $(\mathbf{X}_t)_{t=0}^{\infty}$ is *adapted* to $(\mathcal{F}_t)_{t=0}^{\infty}$, T is a *stopping time* w.r.t. $(\mathcal{F}_t)_{t=0}^{\infty}$, and $(\mathbf{X}_t)_{t=0}^{\infty}$ is *stopped* at T; see [36, Appendix A] for their definitions.

For a stopping time T w.r.t. $(\mathcal{F}_t)_{t=0}^{\infty}$, we define a *level map* $(\mathsf{Lv}_t)_{t=0}^{\infty}$ as a sequence of $\mathcal{F}_t$-measurable functions $\mathsf{Lv}_t : \Omega \to \{0, \dots n\}$ such that $[\![\mathsf{Lv}_t = 0]\!] = [\![T \leq t]\!]$ for each t. We call a pair of a stochastic process and a level map an *instance for* T; just like we construct an MM $\boldsymbol{\eta}$ and a level map Lv as an AST certificate of a pCFG $\mathcal{C}$, we construct an instance for a stopping time T as its AST certificate. We say an instance $((\mathbf{X}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ for T *ranks* $\omega \in \Omega$ in the dimension k at time t when $T(\omega) > t$ and $k = \mathsf{Lv}_t(\omega)$.

For $c > 0$, we say an instance $((\mathbf{X}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ satisfies the *c-ranking condition* if, for each $t \in \mathbb{N}$, $\omega \in [\![\mathsf{Lv}_t \neq 0]\!]$, and $k \in \{1, \dots, \mathsf{Lv}_t(\omega)\}$, we have:

$$\mathbb{E}[\mathbf{X}_{t+1}[k] \mid \mathcal{F}_t](\omega) \leq \mathbf{X}_t[k](\omega) - c \cdot \mathbf{1}_{[\![k = \mathsf{Lv}_t]\!]}(\omega) \quad (\mathbb{P}\text{-a.s.}) \tag{1}$$

Here, the function $\mathbb{E}[\mathbf{X}_{t+1}[k] \mid \mathcal{F}_t]$ denotes the *conditional expectation* of $\mathbf{X}_{t+1}[k]$ given $\mathcal{F}_t$, which takes the role of pre-expectation. We mostly let $c = 1$ and simply call it the ranking condition; the only result sensitive to c is Theorem 4.3.

We also define the three different non-negativity conditions for an instance as follows. Here we adopt a slightly general (but essentially the same) variant of strong non-negativity instead, calling it *uniform well-foundedness*; we simply allow the uniform lower bound to be any constant $\perp \in \mathbb{R}$ instead of fixing it to be zero. This makes the later argument simpler.

$$\begin{array}{lr}
\text{(UN well-fnd.) } \exists \bot \in \mathbb{R}. \forall t \in \mathbb{N}. \forall \omega \in \Omega. \forall k \in \{1, \ldots, n\}. & \mathbf{X}_t[k](\omega) \geq \bot, \\
\text{(LW non-neg.) } \forall t \in \mathbb{N}. \forall \omega \in \llbracket \mathsf{Lv}_t \neq 0 \rrbracket. \forall k \in \{1, \ldots, \mathsf{Lv}_t(\omega)\}. & \mathbf{X}_t[k](\omega) \geq 0, \\
\text{(SC non-neg.) } \forall t \in \mathbb{N}. \forall \omega \in \llbracket \mathsf{Lv}_t \neq 0 \rrbracket. & \mathbf{X}_t[\mathsf{Lv}_t(\omega)](\omega) \geq 0.
\end{array}$$

Definition 3.4 ((UN/LW/SC)-LexRSM). Suppose the following are given: a probability space $(\Omega, \mathcal{F}, \mathbb{P})$; a filtration $(\mathcal{F}_t)_{t=0}^{\infty}$ on $\mathcal{F}$; and a stopping time T w.r.t. $(\mathcal{F}_t)_{t=0}^{\infty}$. An instance $\mathcal{I} = ((\mathbf{X}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ is called a *UNiformly well-founded LexRSM (UN-LexRSM)* for T with the bottom $\bot \in \mathbb{R}$ and a constant $c \in \mathbb{R}$ if (a) $(\mathbf{X}_t)_{t=0}^{\infty}$ is adapted to $(\mathcal{F}_t)_{t=0}^{\infty}$; (b) for each $t \in \mathbb{N}$ and $1 \leq k \leq n$, the expectation of $\mathbf{X}_t[k]$ exists; (c) $\mathcal{I}$ satisfies the c-ranking condition; and (d) $\mathcal{I}$ is uniformly well-founded with the bottom $\bot$. We define *LW-LexRSM* and *SC-LexRSM* by changing (*d*) with LW and SC non-negativity, respectively.

We mostly assume $c = 1$ and omit to mention the constant. UN-LexRSM is known to be sound [2]; meanwhile, LW and SC-LexRSM are generally unsound [15, 20]. We still mention the latter two as parts of sound LexRSMs.

From RSM Maps to RSMs. Let $\boldsymbol{\eta}$ be an MM over a pCFG $\mathcal{C}$ with a level map Lv. Together with a Δ-deterministic scheduler σ and initial state s_I, it *induces* an instance $((\mathbf{X}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ over the dynamics of $\mathcal{C}$, by letting $\mathbf{X}_t(s_0 s_1 \ldots) = \boldsymbol{\eta}(s_t)$; it describes the behavior of $\boldsymbol{\eta}$ and Lv through executing $\mathcal{C}$ from s_I under σ. Properties of $\boldsymbol{\eta}$ such as ranking condition or non-negativity are inherited to the induced instance (if the expectation of $\mathbf{X}_t[k]$ exists for each t, k). For example, an instance induced by an ST-LexRSM map is an UN-LexRSM with $\bot = 0$.

Non-probabilistic Settings, and Instantiation of SC-LexRF. The key question in this paper is to find a LexRSM notion that instantiates SC non-negative LexRF (or SC-LexRF for short); that is, we would like to find a LexRSM notion whose conditions are satisfied by SC-LexRSM[5] in the *non-probabilistic setting*, which we formalize as follows. We say a pCFG is a *(non-probabilistic) CFG* if (a) δ is Dirac for each $(\ell, \delta) \in \Delta$, and (b) $\Delta_p = \emptyset$; this roughly means that a CFG is a model of a PP without '$\mathbf{if}\,\mathbf{prob}(p)$' and '$\mathbf{sample}(d)$'. We say a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is *trivial* if Ω is a singleton, say $\{\omega\}$.

4 Fixable LexRSMs

In Sect. 4–6 we give our novel technical notions and results. In this section, we will introduce the notion of fixability and related results. Here we focus on technical rigorousness and conciseness, see Sect. 2 for the underlying intuition. Proofs are given in appendices of [36]. We begin with the formal definition of ε-fixability.

[5] One would perhaps expect to see "SC-Lex*RF*" here; such a change does not make a difference under a canonical definition of SC-LexRF, so we define the notion of instantiation in this way to save space. See also [36, Appendix A].

Remark 4.1. As in Footnote 2, our formal definitions of fixability in this section slightly differ from an informal explanation in Sect. 2. One difference is that the ε-fixing in Definition 4.2 changes the value of a LexRSM at dimension k whenever it is negative *or* k is strictly on the right to the ranking dimension. This modification is necessary to prove Theorem 4.4. Another is that we define fixability as the notion for an instance $\mathcal{I}$, rather than for an MM $\boldsymbol{\eta}$. While the latter can be also done in an obvious way (as informally done in Sect. 2), we do not formally do that because it is not necessary for our technical development. One can "fix" the argument in Sect. 2 into the one over instances by translating "fixability of $\boldsymbol{\eta}$" to "fixability of an instance induced by $\boldsymbol{\eta}$".

Definition 4.2 (ε-fixing of an instance). Let $\mathcal{I} = ((\mathbf{X}_t)_{t=0}^{\infty}), (\mathsf{Lv}_t)_{t=0}^{\infty})$ be an instance for a stopping time T, and let $\varepsilon > 0$. The *ε-fixing* of $\mathcal{I}$ is another instance $\tilde{\mathcal{I}} = ((\tilde{\mathbf{X}}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ for T, where

$$\tilde{\mathbf{X}}_t[k](\omega) = \begin{cases} -\varepsilon & \text{if } \mathbf{X}_t[k](\omega) < 0 \text{ or } k > \mathsf{Lv}_t(\omega), \\ \mathbf{X}_t[k](\omega) & \text{otherwise.} \end{cases}$$

We say an SC-LexRSM $\mathcal{I}$ is *ε-fixable*, or call it an *ε-fixable LexRSM*, if its ε-fixing $\tilde{\mathcal{I}}$ is an UN-LexRSM with the bottom $\bot = -\varepsilon$.

Observe that the ε-fixing of any instance is uniformly well-founded with the bottom $\bot = -\varepsilon$, so the ε-fixability only asks if the ranking condition is preserved through ε-fixing. Also, observe that the soundness of ε-fixable LexRSM immediately follows from that of UN-LexRSM [2].

While we do not directly use ε-fixability as a technical tool, the two theorems below show its conceptual value. The first one answers our key problem: ε-fixable LexRSM instantiates SC-LexRF with sufficiently large ε.

Theorem 4.3 (fixable LexRSM instantiates SC-LexRF). *Suppose $\mathcal{I} = ((\boldsymbol{x}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ is an SC-LexRSM for a stopping time T over the trivial probability space with a constant c, and let $\varepsilon \geq c$. Then $\mathcal{I}$ is ε-fixable.* □

The second theorem offers a formal comparison between ε-fixable LexRSM and the state-of-the-art LexRSM variant in the literature, namely GLexRSM [15]. We show the former subsumes the latter. In our terminology, GLexRSM is LW-LexRSM that also satisfies the following *expected leftward non-negativity*:

$$\forall t \in \mathbb{N}. \forall \omega \in [\![\mathsf{Lv}_t \neq 0]\!]. \forall k \in \{1, \ldots, \mathsf{Lv}_t(\omega)\}.\ \mathbb{E}[\mathbf{1}_{[\![k > \mathsf{Lv}_{t+1}]\!]} \cdot \mathbf{X}_{t+1}[k] \mid \mathcal{F}_t](\omega) \geq 0.$$

We note that our result can be also seen as an alternative proof of the soundness of GLexRSM [15, Thm. 1]. Our proof is also significantly simpler than the original one, as the former utilizes the soundness of UN-LexRSM as a lemma, while the latter does the proof "from scratch".

Theorem 4.4 (fixable LexRSM generalizes GLexRSM). *Suppose $\mathcal{I}$ is a GLexRSM for a stopping time T. Then $\mathcal{I}$ is ε-fixable for any $\varepsilon > 0$.* □

Now we move on to a refined variant, (ε, γ)-fixability. Before its formal definition, we give a theorem that justifies the partial waiving of the ranking condition described in Sect. 2. Below, $\overset{\infty}{\exists} t.\varphi_t$ stands for $\forall k \in \mathbb{N}.\exists t \in \mathbb{N}.[t > k \wedge \varphi_t]$.

Theorem 4.5 (relaxation of the UN-LexRSM condition). *Suppose the following are given: a probability space $(\Omega, \mathcal{F}, \mathbb{P})$; a filtration $(\mathcal{F}_t)_{t=0}^{\infty}$ on $\mathcal{F}$; and a stopping time T w.r.t. $(\mathcal{F}_t)_{t=0}^{\infty}$. Let $\mathcal{I} = ((\mathbf{X}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ be an instance for T, and let $\bot \in \mathbb{R}$. For each $k \in \{1, \dots, n\}$, let $(\varphi_{t,k})_{t=0}^{\infty}$ be a sequence of predicates over Ω such that*

$$\overset{\infty}{\exists} t.\varphi_{t,k}(\omega) \Rightarrow \overset{\infty}{\exists} t.[\mathbf{X}_t[k](\omega) = \bot \vee k > \mathsf{Lv}_t(\omega)] \quad (\mathbb{P}\text{-}a.s.) \tag{2}$$

Suppose $\mathcal{I}$ is an UN-LexRSM with the bottom $\bot$ except that, instead of the ranking condition, $\mathcal{I}$ satisfies the inequality (1) only for $t \in \mathbb{N}$, $k \in \{1, \dots, n\}$, and $\omega \in [\![k \le \mathsf{Lv}_t \wedge \neg(\mathbf{X}_t[k] > \bot \wedge \varphi_{t,k})]\!]$ (with $c = 1$). Then T is AST w.r.t. $\mathbb{P}$. □

The correspondence between the argument in Sect. 2 and Theorem 4.5 is as follows. The predicate $\varphi_{t,k}$ is an abstraction of the situation "we are at a coin-tossing state at time t in the k-th dimension"; and the condition (2) corresponds to the infinite coin-tossing argument (for a given k, if $\varphi_{t,k}$ is satisfied at infinitely many t, then the ranking in the k-th dimension is "done" infinitely often, with probability 1). Given these, Theorem 4.5 says that the ranking condition of UN-LexRSM can be waived over $[\![\mathbf{X}_t[k] > \bot \wedge \varphi_{t,k}]\!]$. In particular, the theorem amounts to the soundness of UN-LexRSM when $\varphi_{t,k} \equiv$ *false* for each t and k.

Based on Theorem 4.5, we introduce (ε, γ)-fixability as follows. There, $\mathbb{P}[\varphi \mid \mathcal{F}'] := \mathbb{E}[\mathbf{1}_{[\![\varphi]\!]} \mid \mathcal{F}']$ is the *conditional probability* of satisfying φ given $\mathcal{F}'$.

Definition 4.6 ((ε, γ)-fixability). Let $\mathcal{I} = ((\mathbf{X}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ be an instance for T, and let $\gamma \in (0, 1)$. We call $\mathcal{I}$ a *γ-relaxed UN-LexRSM* for T if $\mathcal{I}$ satisfies the properties in Theorem 4.5, where $\varphi_{t,k}$ is as follows:

$$\varphi_{t,k}(\omega) \equiv \mathbb{P}[\mathbf{X}_{t+1}[k] = \bot \mid \mathcal{F}_t](\omega) \ge \gamma. \tag{3}$$

We say $\mathcal{I}$ is (ε, γ)-*fixable* if its ε-fixing $\tilde{\mathcal{I}}$ is a γ-relaxed UN-LexRSM.

The predicate $\varphi_{t,k}(\omega)$ in (3) is roughly read "the ranking by $(\mathbf{X}_t)_{t=0}^{\infty}$ is done at time $t+1$ in dimension k with probability γ or higher, given the information about ω at t". This predicate satisfies Condition (2); hence we have the following corollary, which is the key to the soundness of *lazy LexRSM* in Sect. 5.

Corollary 4.7 (soundness of (ε, γ)-fixable instances). *Suppose there exists an instance $\mathcal{I}$ over $(\Omega, \mathcal{F}, \mathbb{P})$ for a stopping time T that is (ε, γ)-fixable for any $\varepsilon > 0$ and $\gamma \in (0, 1)$. Then T is AST w.r.t. $\mathbb{P}$.* □

5 Lazy LexRSM and Its Soundness

Here we introduce another LexRSM variant, *Lazy LexRSM* (LLexRSM). We need this variant for our LexRSM synthesis algorithm; while ε-fixable LexRSM theoretically answers our key question, it is not amenable to LP-based synthesis algorithms because its case distinction makes the resulting constraint nonlinear.

We define LLexRSM map as follows; see *Contributions* in Sect. 1 for its intuitive meaning with an example. The definition for an instance is in [36, Appendix C].

Definition 5.1 (LLexRSM map). Fix a pCFG $\mathcal{C}$ with an invariant I. Let $\boldsymbol{\eta}$ be an MM associated with a level map Lv. The MM $\boldsymbol{\eta}$ is called a *Lazy LexRSM map (LLexRSM map)* over $\mathcal{C}$ supported by I if it is an SC-LexRSM map over $\mathcal{C}$ supported by I, and satisfies *stability at negativity* defined as follows:

$$\forall \tau \neq \tau_{\mathrm{out}}.\forall s \in [\![I \wedge G(\tau)]\!].\forall k \in \{1, \dots, \mathsf{Lv}(\tau)-1\}.$$
$$\boldsymbol{\eta}[k](s) < 0 \Rightarrow \forall s' \in \mathrm{succ}_\tau(s).\left[\boldsymbol{\eta}[k](s') < 0 \vee k > \max_{\tau' : s' \in [\![G(\tau')]\!]} \mathsf{Lv}(\tau')\right].$$

We first observe LLexRSM also answers our key question.

Theorem 5.2 (LLexRSM instantiates SC-LexRF). *Suppose $\boldsymbol{\eta}$ is an SC-LexRSM over a non-probabilistic CFG $\mathcal{C}$ supported by an invariant I, with a level map Lv. Then $\boldsymbol{\eta}$ is stable at negativity under I and Lv, and hence, $\boldsymbol{\eta}$ is an LLexRSM map over $\mathcal{C}$ supported by I, with Lv.* □

Below we give the soundness result of LLexRSM map. We first give the necessary assumptions on pCFGs and MMs, namely *linearity* and *well-behavedness*. we say a pCFG is *linear* if the update element of each $\tau \in \Delta_d$ is a linear function (this corresponds to the restriction on PPs to the linear ones); and an MM $\boldsymbol{\eta}$ is *linear* if $\lambda \boldsymbol{x}.\boldsymbol{\eta}(\ell, \boldsymbol{x})$ is linear for each $\ell \in L$. We say a pCFG is *well-behaved* if its variable samplings are done via *well-behaved distributions*, which roughly means that their tail probabilities vanish to zero toward infinity quickly enough. Its formal definition is given in [36, Def. C.4], which is somewhat complex; an important fact from the application perspective is that the class of such distributions covers all distributions with bounded supports *and* some distributions with unbounded supports such as the normal distributions [36, Prop. C.6]. Possibly negative (Lex)RSM typically requires some restriction on variable samplings of pCFG (e.g., the *integrability* in [15]) so that the pre-expectation is well-defined.

The crucial part of the soundness proof is the following theorem, where (ε, γ)-fixability takes the key role. Its full proof is given in [36, Appendix C].

Theorem 5.3. *Let $\boldsymbol{\eta} : \mathcal{S} \to \mathbb{R}^n$ be a linear LLexRSM map for a linear, well-behaved pCFG $\mathcal{C}$. Then for any Δ-deterministic scheduler σ and initial state s_I of $\mathcal{C}$, the induced instance is (ε, γ)-fixable for some $\varepsilon > 0$ and $\gamma \in (0, 1)$.*

Proof (sketch). We can show that the ε-fixing $\tilde{\mathcal{I}} = ((\tilde{\mathbf{X}}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ of an induced instance $\mathcal{I} = ((\mathbf{X}_t)_{t=0}^{\infty}, (\mathsf{Lv}_t)_{t=0}^{\infty})$ almost-surely satisfies the inequality (1) of the ranking condition for each t, ω, and k such that $\tilde{\mathbf{X}}_t[k](\omega) = -\varepsilon$ and $1 \leq k \leq \mathsf{Lv}_t(\omega)$ [36, Prop. C.2]. Thus it suffices to show, for each ω, k, and t such that $\tilde{\mathbf{X}}_t[k](\omega) \geq 0$ and $1 \leq k \leq \mathsf{Lv}_t(\omega)$, either $\tilde{\mathcal{I}}$ satisfies the inequality (1) or (1) as a requirement on $\tilde{\mathcal{I}}$ is waived due to the γ-relaxation.

Now take any such t, ω, and k, and suppose the run ω reads the program line *prog* at time t. Then we can show the desired property by a case distinction over *prog* as follows. Here, recall ω is a sequence $s_0 s_1 \ldots s_t s_{t+1} \ldots$ of program states; we defined $\mathbf{X}_t$ by $\mathbf{X}_t[k](\omega) = \boldsymbol{\eta}[k](s_t)$; and $\mathbb{E}[\mathbf{X}_{t+1}[k] \mid \mathcal{F}_t](\omega)$ is the expectation of $\boldsymbol{\eta}[k](s')$, where s' is the successor state of $s_0 \ldots s_t$ under σ (which is not necessarily s_{t+1}). Also observe the requirement (1) on $\tilde{\mathcal{I}}$ is waived for given t, ω, and k when the value of $\boldsymbol{\eta}[k](s')$ is negative with the probability γ or higher.

1. Suppose *prog* is a non-probabilistic program line, e.g., '$x_i := f(\boldsymbol{x})$' or '**while** φ **do**'. Then the successor state s' of s_t is unique. If $\boldsymbol{\eta}[k](s')$ is non-negative, then we have $\mathbb{E}[\tilde{\mathbf{X}}_{t+1}[k] \mid \mathcal{F}_t](\omega) = \mathbb{E}[\mathbf{X}_{t+1}[k] \mid \mathcal{F}_t](\omega)$, so the inequality (1) is inherited from $\mathcal{I}$ to $\tilde{\mathcal{I}}$; if negative, then the requirement (1) on $\tilde{\mathcal{I}}$ is waived. The same argument applies to '**if** $\star$ **then**' (recall $\mathcal{I}$ is induced from a Δ-deterministic scheduler).
2. Suppose *prog* $\equiv$ '**ifprob**(p)**then**'. By letting γ strictly smaller than p, we see either $\boldsymbol{\eta}[k](s')$ is never negative, or it is negative with a probability more than γ. Thus we have the desired property for a similar reason to Case 1 (we note this argument requires p to be a constant).
3. Suppose *prog* $\equiv$ '$x_i :=$ **sample**(d)'. We can show the desired property by taking a sufficiently small γ; roughly speaking, the requirement (1) on $\tilde{\mathcal{I}}$ is waived unless the chance of $\boldsymbol{\eta}[k](s')$ being negative is very small, in which case the room for "ill" exploitation is so small that the inequality (1) is inherited from $\mathcal{I}$ to $\tilde{\mathcal{I}}$. Almost the same argument applies to '$x_i :=$ **ndet**(D)'.

We note, by the finiteness of program locations L and transitions Δ, we can take $\gamma \in (0, 1)$ that satisfies all requirements above simultaneously. □

Now we have soundness of LLexRSM as the following theorem, which is almost an immediate consequence of Theorem 5.3 and Corollary 4.7.

Theorem 5.4 (soundness of linear LLexRSM map over linear, well-behaved pCFG). *Let $\mathcal{C}$ be a linear, well-behaved pCFG, and suppose there is a linear LLexRSM map over $\mathcal{C}$ (supported by any invariant). Then $\mathcal{C}$ is AST.* □

6 Automated Synthesis Algorithm of LexRSM

In this section, we introduce a synthesis algorithm of LLexRSM for automated AST verification of linear PPs. It synthesizes a linear MM in a certain subclass of LLexRSMs. We first define the subclass, and then introduce our algorithm.

Our algorithm is a variant of *linear template-based synthesis*. There, we fix a linear MM $\boldsymbol{\eta}$ with unknown coefficients (i.e., *the linear template*), and consider an assertion "$\boldsymbol{\eta}$ is a certificate of AST"; for example, in the standard 1-dimensional RSM synthesis, the assertion is "η is an RSM map". We then reduce this assertion into a set of linear constraints via *Farkas' Lemma* [34]. These constraints constitute an LP problem with an appropriate objective function. A certificate is synthesized, if feasible, by solving this LP problem. The reduction is standard, so we omit the details; see e.g. [35].

Subclass of LLexRSM for Automated Synthesis. While LLexRSM resolves the major issue that fixable LexRSM confronts toward its automated synthesis, we still need to tweak the notion a bit more, as the stability at negativity condition involves the value of an MM $\boldsymbol{\eta}$ in its antecedent part (i.e., it says "whenever $\boldsymbol{\eta}[k]$ is negative for some k..."); this makes the reduced constraints via Farkas' Lemma nonlinear. Therefore, we augment the condition as follows.

Definition 6.1 (MCLC). Let $\boldsymbol{\eta} : \mathcal{S} \to \mathbb{R}^n$ be an MM supported by an invariant I, with a level map Lv. We say $\boldsymbol{\eta}$ satisfies the *multiple-choice leftward condition* (MCLC) if, for each $k \in \{1, \ldots, n\}$, it satisfies either (4) or (5) below:

$$\forall \tau \in [\![k < \mathsf{Lv}]\!]. \forall s \in [\![I \wedge G(\tau)]\!]. \qquad \boldsymbol{\eta}[k](s) \geq 0, \tag{4}$$

$$\forall \tau \in [\![k < \mathsf{Lv}]\!]. \forall s \in [\![I \wedge G(\tau)]\!]. \forall s' \in \mathrm{succ}_\tau(s). \qquad \boldsymbol{\eta}[k](s') \leq \boldsymbol{\eta}[k](s). \tag{5}$$

Condition (4) is nothing but the non-negativity condition in dimension k. Condition (5) augments the ranking condition in the strict leftward of the ranking dimension (a.k.a. the *unaffecting* condition) so that the value of $\boldsymbol{\eta}[k]$ is non-increasing in the worst-case. MCLC implies stability at negativity; hence, by Theorem 5.4, linear SC-LexRSM maps with MCLC certify AST of linear, well-behaved pCFGs. They also instantiate SC-LexRFs as follows.

Theorem 6.2 (SC-LexRSM maps with MCLC instantiate SC-LexRFs). *Suppose $\boldsymbol{\eta}$ is an SC-LexRSM map over a non-probabilistic CFG $\mathcal{C}$ supported by I, with* Lv. *Then $\boldsymbol{\eta}$ satisfies MCLC under I and* Lv. □

The Algorithm. Our LexRSM synthesis algorithm mostly resembles the existing ones [2,15], so we are brief here; a line-to-line explanation with a pseudocode is in [36, Appendix D]. The algorithm receives a pCFG $\mathcal{C}$ and an invariant I, and attempts to construct a SC-LexRSM with MCLC over $\mathcal{C}$ supported by I. The construction is iterative; at the k-th iteration, the algorithm attempts to construct a one-dimensional MM η_k that ranks transitions of $\mathcal{C}$ that are not ranked by the current construction $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_{k-1})$, while respecting MCLC. If the algorithm finds η_k that ranks at least one new transition, then it appends η_k to $\boldsymbol{\eta}$ and goes to the next iteration; otherwise, it reports a failure. Once $\boldsymbol{\eta}$ ranks all transitions, the algorithm reports a success, returning $\boldsymbol{\eta}$ as an AST certificate of $\mathcal{C}$.

Our algorithm attempts to construct η_k in two ways, by adopting either (4) or (5) as the leftward condition at the dimension k. The attempt with the condition (4) is done in the same manner as existing algorithms [2,15]; we require η_k to rank the unranked transitions *as many as possible*. The attempt with the condition (5) is slightly nontrivial; the algorithm demands a user-defined parameter $\mathrm{Class}(U) \subseteq 2^U$ for each $U \subseteq \Delta \setminus \{\tau_{\mathrm{out}}\}$. The parameter $\mathrm{Class}(U)$ specifies which set of transitions the algorithm should try to rank, given the set of current unranked transitions U; that is, for each $\mathcal{T} \in \mathrm{Class}(U)$, the algorithm attempts to find η_k that *exactly* ranks transitions in $\mathcal{T}$.

There are two canonical choices of $\mathrm{Class}(U)$. One is $2^U \setminus \{\emptyset\}$, the brute-force trial; the resulting algorithm does not terminate in polynomial time, but ranks the maximal number of transitions (by trying each $\mathcal{T}$ in the descending order w.r.t. $|\mathcal{T}|$). This property makes the algorithm complete. Another choice is the singletons of U, i.e., $\{\{\tau\} \mid \tau \in U\}$; while the resulting algorithm terminates in polynomial time, it lacks the maximality property. It is our future work to verify if there is a polynomial complete instance of our proposed algorithm. Still, any instance of it is complete over yet another class of LLexRSMs, namely linear LW-LexRSMs. For a formal statement and its proof, see [36, Thm. D.1].

7 Experiments

We performed experiments to evaluate the performance of our proposed algorithm. The implementation is publicly available[6].

Our evaluation criteria are twofold: one is how the relaxed non-negativity condition of our LexRSM—SC non-negativity and MCLC—improves the applicability of the algorithm, compared to other existing non-negativity conditions. To this end, we consider two baseline algorithms.

(a) The algorithm *STR*: This is the one proposed in [2], which synthesizes an ST-LexRSM. We use the implementation provided by the authors [3].
(b) The algorithm *LWN*: This synthesizes an LW-LexRSM. LWN is realized as an instance of our algorithm with $\mathrm{Class}(U) = \emptyset$. We use LWN as a proxy of the synthesis algorithm of *GLexRSM* [16, Alg. 2], whose implementation does not seem to exist. We note [16, Alg. 2] synthesizes an LW-LexRSM with some additional conditions; therefore, it is no less restrictive than LWN.

Another criterion is how the choice of $\mathrm{Class}(U)$ affects the performance of our algorithm. To this end, we consider two instances of it: (a) *Singleton Multiple Choice* (SMC), given by $\mathrm{Class}(U) = \{\{\tau\} \mid \tau \in U\}$; and (b) *Exhaustive Multiple Choice* (EMC), given by $\mathrm{Class}(U) = 2^U \setminus \emptyset$. SMC runs in PTIME, but we do not know if it is complete; EMC does not run in PTIME, but is complete.

We use benchmarks from [2], which consist of non-probabilistic programs collected in [4] and their probabilistic modifications. The modification is done in two different ways: (a) while loops "**while** φ **do** P **od**" are replaced with probabilistic ones "**while** φ **do** (**if prob**(0.5) **then** P **else skip fi**) **od**"; (b)

[6] https://doi.org/10.5281/zenodo.10937558.

in addition to (a), variable assignments "$x := f(\boldsymbol{x}) + a$" are replaced with "$x := f(\boldsymbol{x}) + \mathit{Unif}[a-1, a+1]$". We include non-probabilistic programs in our benchmark set because the "problematic program structure" that hinders automated LexRSM synthesis already exists in non-probabilistic programs (cf. our explanation to Fig. 2). We also tried two PPs from [15, Fig. 1], which we call *counterexStr1* and *counterexStr2*.

We implemented our algorithm upon [2], which is available at [3]. Similar to [2], our implementation works as follows: (1) it receives a linear PP as an input, and translates it into a pCFG $\mathcal{C}$; (2) it generates an invariant for $\mathcal{C}$; (3) via our algorithm, it synthesizes an SC-LexRSM map with MCLC. Invariants are generated by ASPIC [19], and all LP problems are solved by CPLEX [25].

Table 1. The list of benchmarks in which a feasibility difference is observed between baselines and proposed algorithms. Ticks in "p.l." and "p.a." indicate the benchmark has a probabilistic loop and assignment, respectively. Numbers in the result indicate that the algorithm found a LexRSM with that dimension; the crosses indicate failures; "N/A" means we did not run the experiment.

Benchmark spec.			Synthesis result				Benchmark spec.			Synthesis result			
			Baselines		Our algs.					Baselines		Our algs.	
Model	p.l.	p.a	STR	LWN	SMC	EMC	Model	p.l.	p.a.	STR	LWN	SMC	EMC
complex	–	–	×	×	7	5	serpent	–	–	×	×	3	3
complex	√	–	×	×	7	5	speedDis1	–	–	×	×	4	4
complex	√	√	×	×	3	3	speedDis2	–	–	×	×	4	4
cousot9	–	–	×	3	3	3	spdSimMul	–	–	×	×	4	4
cousot9	√	–	×	×	4	4	spdSimMulDep	–	–	×	×	4	4
loops	–	–	×	×	4	3	spdSglSgl2	√	√	×	×	5	5
nestedLoop	√	√	×	×	4	3	speedpldi3	–	–	×	3	3	3
realheapsort	–	–	×	3	3	3	speedpldi3	√	–	×	×	4	4
RHS_step1	–	–	×	3	3	3	counterexStr1	–	√	N/A	3	3	3
RHS_step1	√	√	×	3	3	3	counterexStr2	–	√	×	×	4	4
realshellsort	√	√	×	2	2	2							

Results. In 135 benchmarks from 55 models, STR succeeds in 98 cases, LWN succeeds in 105 cases while SMC and EMC succeed in 119 cases (we did not run STR for *counterexStr1* because it involves a sampling from an unbounded support distribution, which is not supported by STR). Table 1 summarizes the cases where we observe differences in the feasibility of algorithms. As theoretically anticipated, LWN always succeeds in finding a LexRSM whenever STR does; the same relation is observed between SMC vs. LWN and EMC vs. SMC. In most cases, STR, LWN, and SMC return an output within a second[7], while EMC suffers from an exponential blowup when it attempts to rank transitions with Condition (5) in Definition 6.1. The full results are in [36, Appendix E].

On the first evaluation criterion, the advantage of the relaxed non-negativity is evident: SMC/EMC have unique successes vs. STR on 21 programs (21/135

[7] There was a single example for which more time was spent, due to a larger size.

= 15.6% higher success rate) from 16 different models; SMC/EMC also have unique successes vs. LWN in 14 programs (14/135 = 10.4% higher success rate) from 12 models. This result shows that the program structure we observed in Fig. 2 appears in various programs in the real world.

On the second criterion, EMC does not have any unique success compared to SMC. This result suggests that SMC can be the first choice as a concrete instance of our proposed algorithm. Indeed, we suspect that SMC is actually complete—verifying its (in)completeness is a future work. For some programs, EMC found a LexRSM with a smaller dimension than SMC.

Interestingly, LWN fails to find a LexRSM for *counterexStr2*, despite it being given in [15] as a PP for which a GLexRSM (and hence, an LW non-negative LexRSM) exists. This happens because the implementation in [3] translates the PP into a pCFG with a different shape than the one in [15] (for the latter, a GLexRSM indeed exists); the former possesses a similar structure as in Fig. 2 because different locations are assigned for the while loop and if branch. This demonstrates the advantage of our algorithm from another point of view, i.e., robustness against different translations of PPs.

8 Related Work

There is a rich body of studies in 1-dimensional RSM [12–14,17,20–23,28–30], while lexicographic RSM is relatively new [2,15]. Our paper generalizes the latest work [15] on LexRSM as follows: (a) *Soundness of LexRSM as a stochastic process:* soundness of ε-fixable LexRSMs (Definition 4.2) generalizes [15, Thm. 1] in the sense that every GLexRSM is ε-fixable for any $\varepsilon > 0$ (Theorem 4.4); (b) *Soundness of LexRSM as a function on program states:* our result (Theorem 5.4) generalizes [15, Thm. 2] under the linearity and well-behavedness assumptions; (c) *Soundness and completeness of LexRSM synthesis algorithms:* our result generalizes the results for one of two algorithms in [15] that assumes boundedness assumption on assignment distribution [15, Thm. 3].

The work [24] also considers a relaxed non-negativity of RSMs. Their *descent supermartingale*, which acts on while loops, requires well-foundedness only at every entry into the loop body. A major difference from our LexRSM is that they only consider 1-dimensional RSMs; therefore, the problem of relaxing the LW non-negativity does not appear in their setting. Compared with their RSM, our LexRSM has an advantage in verifying PPs with a structure shown in Fig. 2, where the value of our LexRSM can be arbitrarily small upon the loop entrance (at some dimension; see η_2 at ℓ_1 in Fig. 2).

The work [29] extends the applicability of standard RSM on a different aspect from LexRSM. The main feature of their RSM is that it can verify AST of the symmetric random walk. While our LexRSM cannot verify AST of this process, the RSM by [29] is a 1-dimensional one, which typically struggles on PPs with nested structures. Such a difference can be observed from the experiment result in [31] (compare [31, Table 2] and *nested_loops*, *sequential_loops* in [31, Table 1]).

9 Conclusion

We proposed the first variants of LexRSM that instantiate SC-LexRF. An algorithm was proposed to synthesize such a LexRSM, and experiments have shown that the relaxation of non-negativity contributes applicability of the resulting LexRSM. We have two open problems: one is if the class of well-behaved distributions matches with the one of integrable ones; and another is if the SMC variant of our algorithm (see Sect. 7) is complete.

Acknowledgment. We thank anonymous reviewers for their constructive comments on the previous versions of the paper. The term "ill exploitation" is taken from one of the reviews that we found very helpful. We also thank Shin-ya Katsumata, Takeshi Tsukada, and Hiroshi Unno for their comments on the paper.

This work is partially supported by National Natural Science Foundation of China No. 62172077 and 62350710215.

References

1. Ultimate automizer. https://www.ultimate-pa.org/?ui=tool&tool=automizer
2. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. Proc. ACM Program. Lang. **2**(POPL), 34:1–34:32 (2018). https://doi.org/10.1145/3158122
3. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs: implementation (2018). https://github.com/Sheshansh/prob_termination
4. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_8
5. Ash, R., Doléans-Dade, C.: Probability and Measure Theory. Harcourt/Academic Press, San Diego (2000)
6. Barthe, G., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.Y.: Proving differential privacy via probabilistic couplings. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 749–758 (2016)
7. Barthe, G., Gaboardi, M., Hsu, J., Pierce, B.: Programming language techniques for differential privacy. ACM SIGLOG News **3**(1), 34–53 (2016)
8. Ben-Amram, A.M., Genaim, S.: Complexity of Bradley-Manna-Sipma lexicographic ranking functions. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part II. LNCS, vol. 9207, pp. 304–321. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_18
9. Bertsekas, D.P., Shreve, S.E.: Stochastic Optimal Control: The Discrete-Time Case. Athena Scientific, Belmont (2007)
10. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_48
11. Canal, G., Cashmore, M., Krivić, S., Alenyà, G., Magazzeni, D., Torras, C.: Probabilistic planning for robotics with ROSPlan. In: Althoefer, K., Konstantinova, J., Zhang, K. (eds.) TAROS 2019, Part I. LNCS (LNAI), vol. 11649, pp. 236–250. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23807-0_20

12. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
13. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part I. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_1
14. Chatterjee, K., Fu, H., Novotnỳ, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 327–342 (2016)
15. Chatterjee, K., Goharshady, E.K., Novotný, P., Zárevúcky, J., Žikelić, Đ: On lexicographic proof rules for probabilistic termination. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 619–639. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_33
16. Chatterjee, K., Goharshady, E.K., Novotný, P., Zárevúcky, J., Zikelic, D.: On lexicographic proof rules for probabilistic termination. CoRR **abs/2108.02188** (2021). https://arxiv.org/abs/2108.02188
17. Chatterjee, K., Novotnỳ, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 145–160 (2017)
18. Dubhashi, D.P., Panconesi, A.: Concentration of Measure for the Analysis of Randomized Algorithms. Cambridge University Press, New York (2009)
19. Feautrier, P., Gonnord, L.: Accelerated invariant generation for C programs with aspic and C2Fsm. Electron. Notes Theoret. Comput. Sci. **267**(2), 3–13 (2010)
20. Ferrer Fioriti, L.M., Hermanns, H.: Probabilistic termination: soundness, completeness, and compositionality. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 489–501 (2015)
21. Fu, H., Chatterjee, K.: Termination of nondeterministic probabilistic programs. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 468–490. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_22
22. Giesl, J., Giesl, P., Hark, M.: Computing expected runtimes for constant probability programs. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 269–286. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_16
23. Huang, M., Fu, H., Chatterjee, K.: New approaches for almost-sure termination of probabilistic programs. In: Ryu, S. (ed.) APLAS 2018. LNCS, vol. 11275, pp. 181–201. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02768-1_11
24. Huang, M., Fu, H., Chatterjee, K., Goharshady, A.K.: Modular verification for almost-sure termination of probabilistic programs. Proc. ACM Program. Lang. **3**(OOPSLA), 129:1–129:29 (2019). https://doi.org/10.1145/3360555
25. IBM: IBM ILOG CPLEX 12.7 user's manual (IBM ILOG CPLEX division, incline village, NV) (2017)
26. Karp, R.M.: An introduction to randomized algorithms. Discret. Appl. Math. **34**(1–3), 165–201 (1991)
27. Lobo-Vesga, E., Russo, A., Gaboardi, M.: A programming language for data privacy with accuracy estimations. ACM Trans. Program. Lang. Syst. (TOPLAS) **43**(2), 1–42 (2021)
28. McIver, A., Morgan, C.: A new rule for almost-certain termination of probabilistic- and demonic programs. arXiv preprint arXiv:1612.01091 (2016)
29. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.P.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2**(POPL), 1–28 (2017)

30. Moosbrugger, M., Bartocci, E., Katoen, J.-P., Kovács, L.: Automated termination analysis of polynomial probabilistic programs. In: ESOP 2021. LNCS, vol. 12648, pp. 491–518. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_18
31. Moosbrugger, M., Bartocci, E., Katoen, J.-P., Kovács, L.: The probabilistic termination tool amber. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 667–675. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_36
32. Olmedo, F., Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J., McIver, A.: Conditioning in probabilistic programming. ACM Trans. Program. Lang. Syst. **40**(1), 4:1–4:50 (2018). https://doi.org/10.1145/3156018
33. Parker, D.: Verification of probabilistic real-time systems. In: Proceedings of the 2013 Real-time Systems Summer School (ETR 2013) (2013)
34. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, New York (1998)
35. Takisaka, T., Oyabu, Y., Urabe, N., Hasuo, I.: Ranking and repulsing supermartingales for reachability in randomized programs. ACM Trans. Program. Lang. Syst. **43**(2), 5:1–5:46 (2021). https://doi.org/10.1145/3450967
36. Takisaka, T., Zhang, L., Wang, C., Liu, J.: Lexicographic ranking supermartingales with lazy lower bounds. CoRR **abs/2304.11363** (2024). https://doi.org/10.48550/arXiv.2304.11363

Probabilistic Access Policies with Automated Reasoning Support

Shaowei Zhu[1](✉) and Yunbo Zhang[2]

[1] Princeton University, Princeton, NJ 08540, USA
shaoweiz@cs.princeton.edu

[2] Georgia Institute of Technology, Atlanta, GA 30332, USA
ybzhang3027@gatech.edu

Abstract. Existing access policy languages like Cedar equipped with SMT-based automated reasoning capabilities are effective in providing formal guarantees about the policies. However, this scheme only supports access control based on deterministic information. Observing that certain information useful for access control can be described by random variables, we are motivated to develop a new paradigm of access control in which access policies contain rules about uncertainty, or more precisely, probabilities of random events. To compute these probabilities, we rely on probabilistic programming languages. Additionally, we show that the probabilistic part of these policies can be encoded in linear real arithmetic, which enables practical automated reasoning tasks such as proving relative permissiveness between policies. We demonstrate the advantages of the proposed probabilistic policies over the existing paradigm through two case studies on real-world datasets with a prototype implementation.

Keywords: access policy · access control · domain-specific language · probability theory · uncertainty · automated reasoning · SMT

1 Introduction

Policy based access control is used by major cloud service providers such as Amazon Web Services, necessitating customers to write correct access policies to secure their services and data. Incorrectly specified policies have led to major issues like exposure of private data [2,4], which motivate research on automated reasoning techniques that can automatically identify issues with the policies [9,10,22]. It has been shown that SMT-based techniques are effective in detecting issues in real policies and formally verifying properties of access policies. In the current access control and SMT-based policy verification paradigm, access decisions are made based on *deterministic* information in access requests, such as user identity, role, IP address, or other attributes. For example, we could implement an access policy that only allows an access request if it comes from a certain IP range.

Emerging application domains call for new schemes of access control that consider *uncertainty*, e.g., augmented reality (AR) applications have raised security

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 443–466, 2024.
https://doi.org/10.1007/978-3-031-65633-0_20

and privacy concerns under a unique context [5,15,19,46]. In addition to deterministic attributes, prior work has explored access control based on information such as user location [6,7,17,30], or the type of environment the user is currently in [47]. For example, one might want to disable video recording features when entering private property, or when the user seems to be in a sensitive or private space such as a restroom. Information like the precise indoor location of the user, or the type of room where user currently resides is usually not readily available as a deterministic attribute to the access control system. This information could be supplied by annotating the rooms through posted QR code or wireless signaling [47], yet a more natural and general method would be to infer essential information from observations such as video streams or other sensor readings based on techniques like simultaneous location and mapping (SLAM) [41] and scene classification [47]. Uncertainty is inherent to these inference processes, and should be taken into consideration when evaluating information for the purpose of access control. In this work, we consider such uncertain attributes to be supplied as a random variable drawn from a probability distribution. Methods that provide such distributions include Bayesian filter, which estimates a probability density function over time based on observations and a process model, and probabilistic (Bayesian) machine learning, which is a machine learning (ML) framework that combines ML techniques and Bayesian methods that can reason about uncertainty. In particular, Bayesian neural networks (BNNs) [27,40] could be viewed as a generalization of ordinary neural networks to have stochastic weights that are learned using a Bayesian paradigm, which yield predictions as well as uncertainty associated with the predictions.

Enriching current policy languages with the notion of uncertainty represented by probabilities is also beneficial when incorporating machine learning based access control (MLBAC) [14,18,33,35,37,43,44] into the current policy-based access control paradigm. MLBAC uses ML models to implement access control systems that learn from data such as access logs, which offers an alternative to the potentially costly and error-prone process of manual policy engineering [11,23,49]. However, MLBAC poses new challenges in interpretability, policy adaptability [43], and formal guarantees of correctness. Based on the Bayesian interpretation of probability as *degree of belief* that an ML prediction is correct, we can think of the following scenarios where MLBAC could benefit from our proposed policy language equipped with probabilities:

1. We can consider only trusting the ML predictions for access rights if the amount of *uncertainty* in their predictions is low, which would be useful to guard against out-of-distribution access requests.
2. ML could be used *along with* traditional policy-based approaches for various decision problems. In certain applications such as spam filtering or firewalls, we can maintain deterministic and explicit rules that encode the allow-lists and deny-lists created by the user, and let an ML model decide what to do with the rest. Using our proposed policy language, one can encode all these rules into one policy so that we could automatically reason about their combined effects.

3. We might want to write policies that reflect our prior knowledge of the problem and *combine* existing models. For example, consider a scenario where access rights of company employees depend on the role $r = 1, \ldots, n$ of the employee and we have trained n ML models $M_1, \ldots, M_n$ that predict access rights for employees with each role. It might be infeasible to retrain a large ML model that works well for all roles, due to not having access to historic access data, or the insufficient amount of data for each role. Using our proposed policy language, one can create a set of n allow policies "allow if the user has role r and M_r predicts allow with low uncertainty". Furthermore, one could even deploy a hierarchical prediction model that predicts the role r first, and grants access if all probabilistic ML predictions have low uncertainty.

The aforementioned paradigms require us to enrich the semantics of our policy languages with a notion of *uncertainty* that is absent from current languages such as Cedar [1] or XACML [3,52]. We thus introduce a new probabilistic access policy language **PAPL** with the intuition that some access rules are best implemented using the traditional, deterministic rule-based access control scheme, while some other rules may benefit from the capability of specifying the probability of random events, i.e., predicates over variables whose values are sampled from some probability distributions. The idea of combining symbolic and probabilistic/neural reasoning is not new, and has been studied extensively in many adjacent fields like neurosymbolic reasoning [24,25] or statistical relational learning [8,26]. Our work specifically focuses on adapting this philosophy to the application domain of access control, and also enabling automated reasoning to formally verify properties of the resulting probabilistic access policies. The theory of probability normally requires nonlinear reasoning[1], and it is not immediately clear whether we can implement *practical* SMT-based automated reasoning procedures for these policies. The key observation here is that we need an "interface" for the probabilistic part of the policy that restrains the form of probabilities that could be specified, so that we could reason about them using linear arithmetic. Specifically, we achieve this by allowing *deterministic* rules that involve *probabilities of random events defined by logical formulas* rather than *probabilistic* policies, which allows us to capture the semantics of probability theory axioms using linear arithmetic reasoning.

We implemented a prototype system that can parse and evaluate access requests against **PAPL** policies, and a sound and complete encoding of **PAPL** policies into linear integer and real arithmetic (LIRA) for automated reasoning with SMT solvers. We demonstrate the practicality of the language through two case studies, focusing on the potential improvement it could bring over MLBAC and existing deterministic policy languages.

The rest of the paper is organized as follows: an overview of access control process with **PAPL** policies (Sect. 2), the formalization of **PAPL** and the encoding of policies into SMT formulas decision (Sect. 3), implementation details and case studies (Sect. 4), and finally related work (Sect. 5).

[1] Consider two independent random events A and B where $\Pr[A] = p$ and $\Pr[B] = q$. Then $\Pr[A \cap B] = pq$.

2 Overview of the Probabilistic Access Control Paradigm

In this section, we provide intuitions on the semantics of **PAPL** access policies and the automated reasoning process through a hypothetical use case. Consider AR3D, an augmented reality (AR) service for AR glasses, that renders user-defined virtual objects and also provides an indoor navigation service akin to GPS-based applications like Apple Maps or Google Maps. These functionalities involve potentially private or proprietary virtual objects, visual features of the environment, and other sensitive data. This information could be safeguarded by *location-based* access control [6,7,17,30], similar to how similar information is secured through physical access control in reality. The access control system infers the user's location based on sensor data and checks access rights against relevant access policies accordingly.

Access Control Requirements. Imagine a company M wants to implement an access policy that only its employees can use AR3D within its buildings, also excluding private areas like restricted offices or conference rooms. Traditional methods like GPS are inadequate for indoor localization; instead, AR3D employs WiFi-based localization using the received signal strength intensity (RSSI) from wireless access points (WAPs). This localization procedure, effectively a regression problem in a high-dimensional space that involves complex nonlinearity, is usually implemented using ML. Here we suppose that a probabilistic ML model nondeterministically predicts user location and the variance in its predictions could be seen as a measure of uncertainty. We visualize the uncertain predictions of user trajectory using a model trained on a real-world indoor localization dataset [51] in Fig. 1. Given the variance in the predictions, considering uncertainty when making access decisions is essential for the robustness of the access control system. **PAPL** addresses this by enabling access policies to define access rights while referring to *probabilities that a user is in some space*. This approach is intuitively more robust than the deterministic predicate *mean predicted location of the user is in some space*, since mean can be affected by extreme values and also two predictions with the same mean but different variance can be treated differently by the access control system (in Sect. 4 we demonstrate that this can indeed be useful).

Evaluating Access Requests. When a user Alice requests access to some resource `M::internal::data::visual`, the access control system constructs an access request by collecting relevant attributes from the user. Here, suppose that the system collects a boolean indicating whether Alice is an M employee, and a vector for the RSSI of each WAP. The system then invokes the trained probabilistic ML models to infer the 3D location of the user. The output of these models are probability distributions (essentially conditioned on the training data and observations), e.g., `PredictX.posterior(rssi)`, and the predicted location of the user is modeled as random variables x, y, z sampled from these distributions.

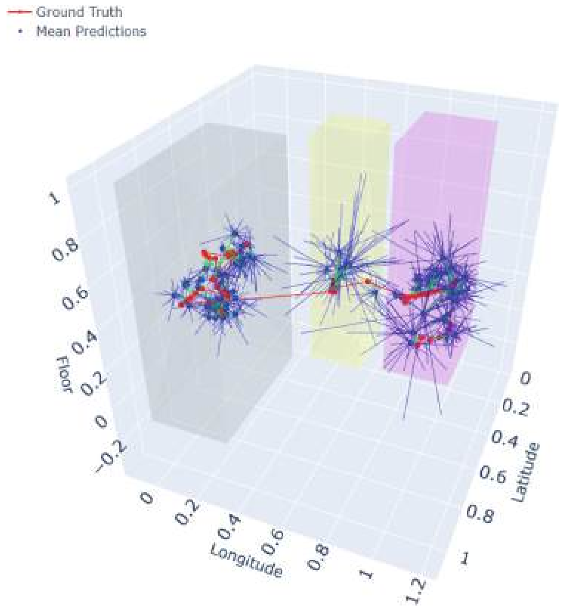


Fig. 1. Predicting user location based on RSSI of WAPs. Transparent boxes represent three buildings. The ground truth location and trajectory of the user at each time step is given by the red dots connected with red lines. At each time step, the RSSI from nearby WAPs is used to predict the user location in this 3D space. The blue dots are the mean prediction of the BNN and the thin blue lines emanating from the blue dots are randomly sampled predictions for each time step. The green lines connecting red and blue dots visualize errors of the mean predictions. The visualization clearly shows there is a varying degree of uncertainty in the predictions made by the BNN. (Color figure online)

This access request is given in Fig. 2c[2]. After the access request is constructed, it is evaluated against all relevant policies that may apply. Suppose that Figs. 2a and 2b are the only two relevant policies. To evaluate against these policies, the system needs to calculate probabilities of random events based on the *a posteriori* knowledge about the user location given in the access request. The system synthesizes a NumPyro program to estimate the probabilities (Fig. 2d) for all random events that appear in the policies. This program is constructed to collect the source code of the probabilistic ML model (in this case `PredictX`), run a large number of inferences given a new observation (the `rssi` vector), assign the values to variables based on information given in the access request context, and compute how many times the boolean predicates that correspond

[2] Notice that in the current design, arrays can only be used to supply a feature vector as input to the probabilistic ML models to compute the posterior probability distribution. Other parts of the policy cannot refer to the arrays or their elements.

to the random events in the policies evaluate to true in order to estimate the probability of these events.

Automated Reasoning. Over time, the company M and other users may have implemented complicated access policies that involve various buildings and spaces owned by M and other parties. Suppose that M wants formal guarantees that other existing allow policies in the system do not conflict with the company policy on its buildings, e.g., are not allowing principals without the `isMEmployee` attribute to access any portion of the buildings that M owns. This could be ensured by encoding the relevant policies into SMT formulas and invoke SMT solvers to do automated reasoning, which we will cover in the next section.

Threat Model. In this work, we focus on the semantics and automated verification of the probabilistic access policies. We thus assume that the device that collects attributes such as RSSI are uncompromised and trusted; the access policies are authentic; all communications are properly protected by cryptographic protocols; procedures include ML inferences, the construction of NumPyro programs, and the evaluation of policies are all executed on a trusted and secure server.

3 Formalization and SMT Encoding of PAPL Policies

In this section, we first define the formal syntax and semantics of requests and **PAPL** policies. Then we present a sound and complete encoding of **PAPL** policies into linear arithmetic formulas, and discuss how this could be useful to formally prove properties of policies.

3.1 Syntax and Semantics of Access Requests and Policies

Encoding of String Literals. Handling string literals and encoding conditions involving predicates on strings is not central for the purpose of this work. To simplify the presentation, we assume all string literals are encoded as integers and attributes include `Principal`, `Action`, and `Resource` are encoded as integer-typed variables. This limitation is not fundamental to the constructions in this paper and possible extensions to other SMT theories including strings have been explored by previous work [10].

Example 1. For the rest of this section, we assume the following integer encoding of string literals:

$$\texttt{"Alice"} \mapsto 0, \texttt{"AR3D::read"} \mapsto 1, \texttt{"M::internal::data::visual"} \mapsto 2\ .$$

Access Requests. The syntax of access requests is given in Fig. 3a, assuming the encoding for string literals has been applied. A request is parsed into a 4-tuple $\langle R_P, R_A, R_R, (M, D)\rangle$ containing information for `Principal`, `Action`, and `Resource`, and `Context`. The `Context` part contains a map M of deterministic variables to their values and the field D is a joint distribution from which the

```
{
    Effect == allow,
    Principal == *,
    Action == "AR3D::read",
    Resource == "M::internal::data::visual",
    Condition == {
        RequireVars == [x,y,z: Rand<real>,
        isMEmployee: bool],
        ConditionExp == isMEmployee and
            0.95 <= prob(
            M.building1.xmin <= x &
            x <= M.building1.xmax &
            M.building1.ymin <= y &
            y <= M.building1.ymax &
            M.building1.zmin <= z &
            z <= M.building1.zmax)
        }
}
```

(a) A policy that allows M employees to read proprietary data necessary for the AR3D app to function when it is *very* probable that they are inside a corporate building.

```
{
    Effect == deny,
    Principal == *,
    Action == "AR3D::read",
    Resource == *,
    Condition == {
        RequireVars == [x,y,z: Rand<real>],
        ConditionExp ==
            0.2 <= prob(
            M.building1.conf1.xmin <= x &
            x <= M.building1.conf1.xmax &
            M.building1.conf1.ymin <= y &
            y <= M.building1.conf1.ymax &
            M.building1.conf1.zmin <= z &
            z <= M.building1.conf1.zmax)
    }
}
```

(b) A policy that forbids anyone to use the AR3D app when the user is *possibly* inside a confidential conference room .

```
{
    Principal == "Alice",
    Action == "AR3D::read",
    Resource == "M::internal::data::visual",
    Context == {
        Det == {
            isMEmployee: bool = true,
        },
        Rand == {
            Features = {
                rssi: array<int>(250) =
                [-60, -68, -70, 100, ...],
            },
            Models = [
                PredictX,PredictY,PredictZ:
                    array<int>(250)
                        -> Rand<real>],
            RV = {
                x: Rand<real> =
                    PredictX(rssi),
                # similar for y and z
            }
        }
    }
}
```

(c) Example access request that contains a deterministic key-value map and the distributions from which random variables relevant for access control are sampled. The distributions are represented by NumPyro probabilistic programs.

```
def bnn(X, y_obs=None):
    net = random_flax_module(...)
    mean, rho = net(X)
    sigma = nn.softplus(rho)
    numpyro.sample("obs",
        dist.Normal(mean, sigma),
        obs=y_obs)

def PredictX(rssi, n_samples):
    # sample from trained bnn
    bnn_pred = Predictive(bnn,
        params=bnn_x_weights,
        num_samples=n_samples,
        return_sites=("obs"))
    return bnn_pred(PRNGKey(0),
        rssi)["obs"]

n_samples = 2000
x = PredictX(rssi, n_samples)
# predictions for y and z omitted
e1 = M.building1.xmin <= x &
    x <= M.building1.xmax & ...
    # similar predicates on y and z
prob1 = jnp.mean(e1)
# similarly get prob2 for policy in (b)
```

(d) We use NumPyro for probabilistic ML models and also to estimate the probabilities of random events. Probabilistic ML models such as `PredictX` sample `x` from a distribution as their predictions. Frequency is used to estimate probabilities of random events `e1` and `e2`.

Fig. 2. Performing access control for a particular access request based on access policies written in the **PAPL** language.

RequestConst	c	$::=$	$c_z \in \mathbb{Z} \mid c_q \in \mathbb{R} \mid c_b \in \mathbb{B}$	int/real/bool literal
RequestDVar	dv	$::=$	$z \mid q \mid b$	int/real/bool var
RequestRVar	rv	$::=$	$r_z \mid r_q$	random int/real var
Distribution	D	$::=$	$\mathcal{D}(rv_1, \dots, rv_d)$	probability distribution
Map	M	$::=$	$\emptyset$	empty map
		$\mid$	$M \cup \{dv \mapsto c\}$	deterministic valuation
Principal	R_P	$::=$	c_z	principal field
Action	R_A	$::=$	c_z	action field
Resource	R_R	$::=$	c_z	resource field
Request	R	$::=$	$\langle R_P, R_A, R_R, (M, D)\rangle$	access request

(a) Syntax for access requests.

Number	n	$::=$	$c_z \in \mathbb{Z} \mid c_q \in \mathbb{R}$	int/real literal
NumVar	v	$::=$	$z \mid q$	int/real var
Bool	bl	$::=$	$c_b \in \mathbb{B}$	bool literal
BoolVar	bv	$::=$	bv	bool var
LinearExp	l	$::=$	$n \mid nv \mid l + l$	linear expression
RandomEvent	e	$::=$	$l = l \mid l \leq l$	atoms
		$\mid$	$e \,\&\, e \mid (e \mid e) \mid \sim e$	boolean connective
ConditionNumVar	NV	$::=$	$\texttt{prob}(e)$	event probability
		$\mid$	v	int/real var
ConditionLinExp	L	$::=$	$n \mid n \cdot NV \mid L + L$	linear expression
ConditionFormula	C	$::=$	$bl \mid bv$	boolean atoms
		$\mid$	$L = L \mid L \leq L$	linear atoms
		$\mid$	$C \wedge C \mid C \vee C \mid \neg C$	boolean connective
Effect	E	$::=$	$\texttt{allow} \mid \texttt{deny}$	allow or deny policy
Principal	V_P	$::=$	$c_z \mid *$	principal or all
Action	V_A	$::=$	$c_z \mid *$	action or all
Resource	V_R	$::=$	$c_z \mid *$	resource or all
Policy	P	$::=$	$\langle E, V_P, V_A, V_R, C\rangle$	access policy

(b) Syntax for **PAPL** access policies.

Fig. 3. Syntax for **PAPL** policies and requests.

random variables are drawn[3]. Since attributes like `Principal` are also deterministic variables, we merge these into M and simply write (M, D) to represent an access request.

Example 2. Figure 2c gives a request (M, D) where M contains a boolean attribute `isMEmployee` $\mapsto \top$, and three integer attributes *principal* $\mapsto 0$, *action* $\mapsto 1$, and *resource* $\mapsto 2$. D gives the predicted user location as continuous, real-typed, independent random variables x, y, z, e.g., prediction for dimension

[3] The access control system defines the names and types of deterministic and random variables that are used consistently in both access requests and access policies.

x is given by a probabilistic ML model that takes as input `rssi` and generates predictions by sampling from a posterior distribution `PredictX(rssi)`.

Random Events. Syntax for random events e is given in Fig. 3b. Each event is a ground LIRA formula $e(V \cup X)$ over a set of free deterministic variables V and a set of free random variables X[4]. Given a map M containing valuations of all deterministic variables in e and a probability distribution D from which all random variables in e are drawn from, we use notation $e[v \mapsto M[v] : v \in dom(M)]$ or more succinctly $e[M]$ to denote the formula obtained by substituting all e variables that appear in M with their valuations. Assuming this substitution, we use $\Pr_{X \sim D}[e[M] = \top]$ to denote the probability for the random event to be evaluated to true when the random variables X is drawn from distribution D.

Example 3. The random event e_1 in Fig. 2a is given by the formula

$$\begin{aligned} &\texttt{M.building1.xmin} \leq x \wedge x \leq \texttt{M.building1.xmax} \\ \wedge\ &\texttt{M.building1.ymin} \leq y \wedge y \leq \texttt{M.building1.ymax} \\ \wedge\ &\texttt{M.building1.zmin} \leq z \wedge z \leq \texttt{M.building1.zmax} \end{aligned}$$

where x, y, z are real-typed free random variables (`M.building1.xmin`, etc. are real constants).

Probability Distributions. In this work, instead of pursuing a measure-theoretic formalization, we regard a probability distribution $\mathcal{D}$ as a function that maps random events described by LIRA formulas to rational numbers in $[0, 1]$. More concretely, given a set of arbitrary LIRA formulas $F = \{e_1, \ldots, e_n\}$, let $U = \{f_1 \wedge \cdots \wedge f_n : f_i \in \{e_i, \neg e_i\}\}$ be the set of all random events for which we care about their probabilities. We require that distribution $\mathcal{D}$ satisfies (c.f. Kolmogorov axioms of probability [34])

$$\begin{aligned} &\forall f \in U.\, 0 \leq \mathcal{D}(f) \leq 1 \\ &\forall f \in U.\, f \text{ is unsat} \implies \mathcal{D}(f) = 0 \\ &\forall f_1, f_2 \in U.\, f_1 \wedge f_2 = \bot \implies (\mathcal{D}(f_1 \vee f_2) = \mathcal{D}(f_1) + \mathcal{D}(f_2)) \end{aligned}$$

Access Policies. Full syntax for **PAPL** access policies is given in Fig. 3b. An access policy is parsed into a 5-tuple $P = \langle E, V_P, V_A, V_R, C \rangle$. The `Effect` E indicates whether a policy is an allow or deny policy. The fields V_P, V_A, V_R represent `Principal`, `Action`, and `Resource`. Notably, C describes the `Condition` under which this policy takes effect, represented by an LIRA formula with an uninterpreted function `prob`$(\cdot)$ that intends to represent the probability of random events. For convenience, we define a procedure $randEvents(P)$ that extracts the set of all random events mentioned in P, i.e., all e such that P contains an expression `prob`(e) in its `Condition` field. We define $\mathcal{I}[\![C]\!](M, D)$, or the *interpretation* of some condition formula C given access request $R = (M, D)$ as follows.

[4] Syntax of random events in the policies actually does not distinguish between deterministic versus random variables.

Let the interpretation of the function symbol prob(e) for some random event $e(V \cup X)$ be

$$\mathcal{I}[\![\texttt{prob}(e)]\!](M, D) \triangleq \Pr_{X \sim D}[e[M] = \top]$$

while the interpretation $\mathcal{I}[\![\cdot]\!]$ of other constructs in C are defined as if we are interpreting an LIRA formula according to an LIRA model M. We further define $\mathcal{I}[\![P]\!](M, D)$, or the interpretation of a (deny or allow) policy P given request $R = (M, D)$ as

$$\mathcal{I}[\![C]\!](M, D) \wedge \mathcal{I}[\![\mathit{principal} = V_P \wedge \mathit{action} = V_A \wedge \mathit{resource} = V_R]\!](M, D)$$

where the second clause of the conjunction simply checks if the principal, action, and resource fields in the policy matches the request. We say a policy P *applies to* a request $R = (M, D)$ if $\mathcal{I}[\![P]\!](M, D)$ is true. An access reqeust R is allowed by a set of policies S (denoted by $R \models S$) if and only if some allow policy in S applies to R and no deny policy in S applies to R [10].

Probabilistic Programs. Let E be a set of random events, $R = (M, D)$ be a request. A probabilistic program $\mathcal{PP}(E, R)$ estimates (or computes symbolically) the probability of random events in E given R. More precisely, it returns a mapping $PP(\cdot)$ from E to a rational number in $[0, 1]$ such that for each $e \in E$

$$PP(e) \approx \Pr_{X \sim D}[e[M] = \top] \ .$$

We mostly take a black-box view on probabilistic programs in this work. In practice, the probabilistic program may be implemented by either symbolic methods such as the sum-product probabilistic language (SPPL) [48] or sampling based methods such as Discrete Gibbs sampling or Mixed-HMC [38,53]. For the access control use cases (Sect. 4), it is difficult to train an SPPL model that performs as well as more expressive models like BNNs. Although we have an interesting observation that if the distribution from which the random variables are sampled can be written as an SPPL program, then we can use prob queries in SPPL to compute the probability of any random event exactly [48]. The sampling based methods only require that we could sample from distribution $\mathcal{D}$.

3.2 Automated Reasoning About Policies with Probabilities

Having defined the semantics of access requests and policies, we now encode policies as SMT formulas to enable automated reasoning about their behaviors. For a policy $P = \langle E, V_P, V_A, V_R, C \rangle$, we define its LIRA encoding $[\![P]\!]$ as

$$[\![P]\!] \triangleq [\![V_P]\!] \wedge [\![V_A]\!] \wedge [\![V_R]\!] \wedge [\![C]\!] \ .$$

For $[\![V_P]\!]$, we introduce a fresh integer symbol *principal* and define

$$[\![V_P]\!] = \begin{cases} \top, & \text{if } V_P = * \\ \mathit{principal} = V_P & \text{otherwise} \end{cases}$$

The encodings $[\![V_A]\!], [\![V_C]\!], [\![V_R]\!]$ are defined similarly. To define $[\![C]\!]$, we first introduce fresh real symbols prob_e for each $e \in \mathit{randEvents}(P)$ and then define $[\![C]\!]$ as $C[\texttt{prob}(e) \mapsto \mathit{prob}_e]$, i.e., with all uninterpreted function terms $\texttt{prob}(e)$ substituted with prob_e. The combined effect of a set of policies S is described by formula [10]

$$[\![S]\!] = \bigvee_{p \in S \text{ is allow}} [\![p]\!] \wedge \bigwedge_{q \in S \text{ is deny}} \neg [\![q]\!] \ .$$

Example 4. The LIRA encodings for the allow and deny policies in Fig. 2a and 2b are written as formulas P_1 and P_2, where

$$P_1 \triangleq \mathit{action} = 1 \wedge \mathit{resource} = 2 \wedge \mathit{isMEmployee} \wedge 0.95 \leq \mathit{prob}_1$$

$$P_2 \triangleq \mathit{action} = 1 \wedge 0.2 \leq \mathit{prob}_2$$

The effect of these two policies is encoded by $P_1 \wedge \neg P_2$.

The LIRA encoding $[\![P]\!]$ above simply introduces fresh real symbols prob_e for the event probabilities in P but ignores the fact that these are probabilities of events that might relate to each other in nontrivial ways. For example, given that $\Pr[x \geq 1 \wedge y = 0] \geq 0.5$, we should be able to deduce $\Pr[x \geq 0 \wedge y \geq 0] \geq 0.5$ according to the probability theory axioms. Algorithm 1 describes a procedure for discovering additional constraints these probabilities must satisfy. The algorithm works by computing a set U containing the probability of disjoint events, and then decomposing the probability prob_e for each random event $e \in \mathit{randEvents}(P)$ as a sum of elements in U.

Example 5. Consider a bijective map between events and symbols $K = \{(x < l \vee x \geq r) \leftrightarrow \mathit{prob}_1, (m \leq x < r) \leftrightarrow \mathit{prob}_2\}$ and an LIRA formula $C = l \leq r \wedge \mathit{prob}_1 \geq 0.6 \wedge (l \leq m \leq r) \wedge \mathit{prob}_2 > 0.5$, where l, m, r are real-typed deterministic variables and x is a real-typed random variable. Let $e_1 = (x < l \vee x \geq r), e_2 = (m \leq x < r)$. Given as input (K, C), Algorithm 1 enumerates the combinations $\{e_1 \wedge e_2, \neg e_1 \wedge e_2, e_1 \wedge \neg e_2, \neg e_1 \wedge \neg e_2\}$ and finds out that only $e_1 \wedge e_2$ is UNSAT given C. The algorithm returns a set of symbols that represents probabilities of disjoint events $U = \{u_1, u_2, u_3\}$ and a constraint on the probabilities $Q = 0 \leq u_1 \leq 1 \wedge 0 \leq u_2 \leq 1 \wedge 0 \leq u_3 \leq 1 \wedge u_1 + u_2 + u_3 = 1 \wedge \mathit{prob}_1 = u_1 \wedge \mathit{prob}_2 = u_2$.

Denotation of Policies. Following previous work [9], we define the *denotation* of a policy set S as the set of requests it allows: $\gamma(S) \triangleq \{R : R \models S\}$. Consider an LIRA formula $F(Y, X, U)$ over sets of variables $Y, X, U = \{u_1, \ldots, u_N\}$, and a bijective map $K : E \to Y$ that maps a set of random events E to a set of real symbols Y that represent their probabilities, where every $e \in E$ is a ground formula over a set of deterministic variables V and random variables X. For any $y \in Y$, let K_y^{-1} be the random event whose probability is represented by y. We define an *abstract denotation* $\gamma^\sharp(F, K)$ as a set of requests (M, D) such that the domain of M is V and

$$\exists u_1, \ldots, u_N . F[y \mapsto \Pr_{X \sim D}[K_y^{-1}[M] = \top] : y \in Y][M]$$

Algorithm 1: Computing the constraint formula for the reals representing probabilities for a set of policies.

```
Function constraints($K$, $C$)
    Input: A map $K$ from a set of random events $E$ to a set of symbols $prob_e$;
           an LIRA formula $C$ for the constraints that originates from the
           deterministic part of the policy.
    Output: A formula $Q$ in LIRA that encodes all constraints on the real
            constants created to represent probabilities must satisfy, and a set
            of symbols $U$ for the real-typed auxiliary symbols introduced.
    $U \leftarrow \emptyset$ ;                    // set of symbols to represent event probabilities
    $Q \leftarrow \top$ ;               // formula that contains constraints on symbols in $U$
    $E \leftarrow dom(K)$;
    $s \leftarrow$ Solver();
    $s$.push($C$);
    foreach $e \in E$ do
        /* We decompose $c_e$, the probability of event $e$, into a sum of
           probabilities of disjoint events.                                    */
        $c_e \leftarrow 0$;
    while $s$.check() = sat do
        /* Enumerating all satisfiable boolean combinations of $e \in E$ that are
           disjoint from each other and do not violate $C$. Random events that
           violate $C$ have probability 0 by axiom.                             */
        $m \leftarrow s$.getModel();
        /* Introducing fresh real symbol for the probability of each boolean
           combination.                                                         */
        $u \leftarrow$ mkReal();
        $U \leftarrow U \cup \{u\}$;
        /* Adding constraints on probability $u$ based on axioms of probability
           theory.                                                              */
        $Q \leftarrow Q \wedge 0 \leq u \wedge u \leq 1$;
        $b \leftarrow \top$ ;              // Blocking clause to avoid selecting the same model
        foreach $e \in E$ do
            if $m \models e$ then
                /* Event $e$ evaluates to $\top$ under model $m$, thus $u$ should appear
                   as a term in the decomposition for the probability of $e$.   */
                $c_e \leftarrow c_e + u$;
                $b \leftarrow b \wedge e$;
            else
                $b \leftarrow b \wedge \neg e$;
        $s$.push($\neg b$)
    foreach $e \in E$ do
        $prob_e \leftarrow K[e]$;
        $Q \leftarrow Q \wedge prob_e = c_e$;
    /* Probabilities of all satisfiable boolean combinations add up to 1.       */
    $Q \leftarrow Q \wedge \sum_{u \in U} u = 1$;
    return $U, Q$
```

holds. Intuitively, each symbol $y \in Y$ is intended to represent the probability of some random event, and F includes constraints on the auxiliary variables $u_1, \ldots, u_N$ and $y \in Y$ so that they behave like probabilities.

We now prove the soundness and completeness of our encoding algorithm (Algorithm 1).

Theorem 1 (Correctness of LIRA encoding). *Consider a policy P with LIRA encoding $[\![P]\!]$. Let E be the set of random events in P. Let K be a bijective map such that $e \mapsto prob_e$, where $prob_e$ is the symbol introduced in the encoding $[\![P]\!]$ for the probability of $e \in E$. Let $U, Q = \mathtt{constraints}(K, [\![P]\!])$ and $U = \{u_1, \ldots, u_N\}$. Then $\exists u_1, \ldots, u_N.([\![P]\!] \wedge Q)$ is satisfiable modulo LIRA if and only if there exists a request (M, D) such that $(M, D) \models P$.*

Corollary 1. *Given a set of policies $S = \{P_1, \ldots, P_n\}$. Let E be the set of all random events that appear in S. Let K be a bijective map from E to the set of probability symbols introduced in $[\![S]\!]$. Then*

$$\gamma^\sharp\left([\![S]\!] \wedge Q, K\right) = \gamma(S)$$

where $U, Q = \mathtt{constraints}\,(K, [\![S]\!])$.

Given Corollary 1, we can reason about the set of requests allowed by a policy P through an SMT formula to fulfill the requirements laid out in 2. Suppose we want to prove that a user policy P_1 is *not more permissive* than some company policy P_2, i.e., $\gamma(P_1) \subseteq \gamma(P_2)$. We first construct K, a bijective map between the random events in P_1 and P_2 and the introduced symbols in their encodings $[\![P_1]\!]$ and $[\![P_2]\!]$. Let $U, Q = \mathtt{constraints}(K, [\![P_1]\!] \wedge \neg [\![P_2]\!])$, we check if $[\![P_1]\!] \wedge \neg [\![P_2]\!] \wedge Q$ is unsatisfiable using an LIRA theory solver.

Example 6. Consider policies with non-trivial conditions $P_1 \triangleq l \leq r \wedge \mathtt{prob}(x < l \vee x \geq r) \geq 0.6$ and $P_2 \triangleq \neg(l \leq m \leq r) \vee \mathtt{prob}(m \leq x \wedge x < r) \leq 0.5$, for which a bijective map $K = \{(x < l \vee x \geq r) \leftrightarrow prob_1, (m \leq x < r) \leftrightarrow prob_2\}$ could be constructed along with the encodings $[\![P_1]\!] = l \leq r \wedge prob_1 \geq 0.6$ and $[\![P_2]\!] = \neg(l \leq m \leq r) \vee prob_2 \leq 0.5$. We construct $C = [\![P_1]\!] \wedge \neg [\![P_2]\!]$ and obtain $U, Q = \mathtt{constraints}(K, C)$ as in Example 5. Then

$$\begin{aligned} [\![P_1]\!] \wedge \neg [\![P_2]\!] \wedge Q = l \leq r \wedge prob_1 \geq 0.6 \\ &\wedge (l \leq m \leq r) \wedge prob_2 > 0.5 \\ &\wedge 0 \leq u_1 \leq 1 \wedge 0 \leq u_2 \leq 1 \wedge 0 \leq u_3 \leq 1 \\ &\wedge u_1 + u_2 + u_3 = 1 \wedge prob_1 = u_1 \wedge prob_2 = u_2 \end{aligned}$$

which is unsatisfiable modulo LIRA, since $prob_1 + prob_2 = u_1 + u_2 \leq 1$ by the last line of the formula but that contradicts the first two lines. We have thus proved that policy P_1 is not more permissive than P_2.

In the worst case, Algorithm 1 introduces 2^n additional real symbols when there are n random events in a policy P, resulting in large SMT formulas. Here we show that the number of additional auxiliary symbols will not be too large for particular kinds of condition formulas that are useful for reasoning about **PAPL** policies that implement location-based access control.

Theorem 2. *Given a policy P such that every probability term* `prob`(e) *refers to the probability for a point to be in a 3D space represented by a boolean combination of axis-aligned bounding boxes. If there are N bounding boxes in total in P, then the number of auxiliary symbols introduced in Algorithm 1 for P is bounded by $O(N^3)$.*

4 Implementation and Evaluation

We implemented a parser for access requests and policies in **PAPL** that could perform access control as outlined in Fig. 2. For evaluating requests against policies, we use the probabilistic programming language `NumPyro` [12,45] as the backend probabilistic programming language. We also implemented the procedure that encodes policies and the constraints on the fresh symbols that represent random event probabilities (Algorithm 1), and a procedure for checking relative permissivenss based on this encoding. We use `z3` [20] as the backend SMT solver for LIRA and the probabilistic programming language `NumPyro` [12,45] as the backend probabilistic programming language.

To demonstrate the usefulness and practicality of the **PAPL** language, we present two case studies to show that a combination of symbolic and deterministic rules and the ability to reason about uncertainty could lead to better outcomes.

Threats to Validity and Limitations. Due to the lack of access policy datasets available to the public [22] and also the novel nature of our proposed policy language, we have considered synthetic access control scenarios and policies in the case studies. Although such practices are standard in access control literature [43,44], this poses questions on whether the proposed solution is useful or performs as well in practice for real-world access control tasks. Also, we do not claim any contribution on the ML methods in this work. Thus, the ML model we have implemented in the case study might not be optimal for the datasets considered.

4.1 Case Study: Location-Based Access Control with Uncertainty

In this case study, we implement a system for location-based access control using WiFi-based indoor localization (see Sect. 2) using the **PAPL** policy language that considers uncertainty in the predications of user location. The key question we want to answer here is **RQ: Are PAPL policies more robust than deterministic policies for location-based access control?**

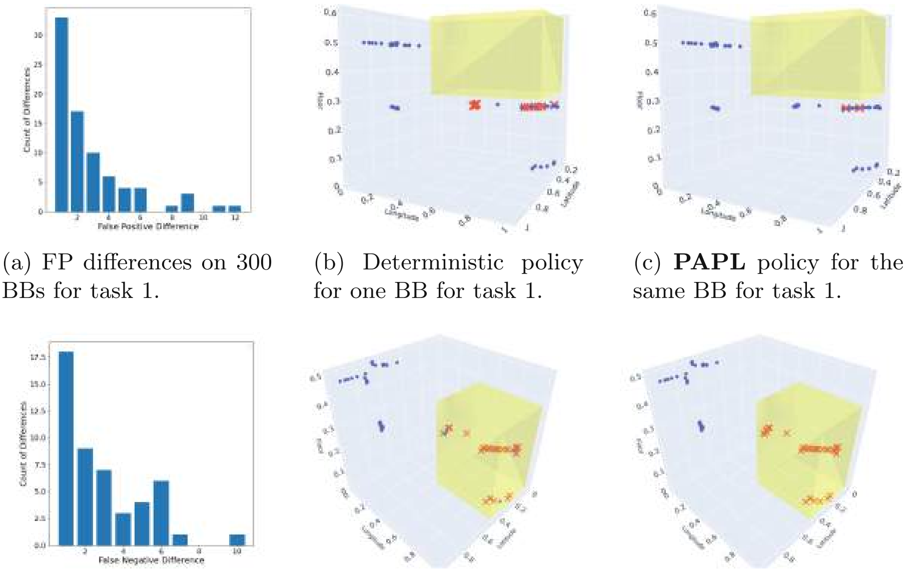
Dataset and Probabilistic ML Models. We use a WiFi localization dataset [51] comprised of trajectories of cellphone positions in terms of latitude, longitude, and floor levels, along with the RSSI collected from a fixed set of WAPs. We use Bayesian neural networks [27] to provide predictions of user location with uncertainty. We train three BNNs, each predicting the latitude, longitude, and floor level of the cellphone at each time step, given the observed RSSI as features. Each BNN model has two hidden layers of 128 and 64 units, and outputs both the mean and standard deviation of a prediction. The prior distribution of each network weight in the BNN models is initialized as a normal distribution $N(0, 1)$. All BNN models are trained to optimize the evidence lower bound (ELBO) through stochastic variance inference (SVI).

Access Control Tasks and Policies. We consider two access control tasks: (1) allow the user when the user is *probably* in a space, and (2) deny the user when the user *might be* in a space. For both tasks, we assume it is desirable to have high accuracy (the proportion of correct access decisions among all decisions), and err on the safe side. In other words, it is desirable to minimize the number of "false positives" in granting access, i.e., allowing what should have been denied, while not denying access too often for those that should have been allowed. For these tasks, we specify the following **PAPL** policies "allow the user when the probability for the user to be in a bounding box (BB) exceeds 0.7", and "deny the user when the probability for the user to be in a BB exceeds 0.2". Further investigation is needed for how to set the probability thresholds optimally for each access control context and the particular probabilistic ML models used. Basically, the thresholds could be used to balance security and usability of the access control system, i.e., the number of "false positives" (FP) and "false negatives" (FN).

Experimental Setup. We randomly generate 300 axis-aligned BBs and assess the access decisions made by the **PAPL** policy and the deterministic policy. We approximate the probability for the user to be in a BB by the empirical frequency of the event "predictions sampled from the posterior distribution of the BNN fall inside a BB" among 2000 samples. For a deterministic policy language, we could not make use of the uncertainty information and the best we could do is to compute the most probable location given the BNN's predictions by taking the mean of all predictions, and check if the mean location falls inside the BB.

Results. The overall accuracies for deciding access rights on the test set using both **PAPL** policies (0.9778 for task 1, 0.9681 for task 2) and deterministic policies (0.9787 for task 1, 0.9787 for task 2) are high. We then focus on comparing the number of "false positives". For task 1, an FP is a user location that is actually outside the allowing BB but is predicted to be in the BB and thus allowed by the access control system. For task 2, an FP is a user location that is actually inside the denying BB but is predicted to be outside and thus allowed by the system. The comparison of **PAPL** policies versus deterministic policies is shown in Fig. 4. Results show that compared to a deterministic policies language, using **PAPL** leads to more robust access control without sacrificing usability, i.e., it reduces FPs while maintaining the overall accuracy in its decisions.

Latency in Making Access Decisions. We have observed that the current implementation of the access control system on a Laptop needs up to a few seconds



(a) FP differences on 300 BBs for task 1.

(b) Deterministic policy for one BB for task 1.

(c) **PAPL** policy for the same BB for task 1.

(d) FP differences on 300 BBs for task 2.

(e) Deterministic policy for one BB for task 2.

(f) **PAPL** policy for the same BB for task 2.

Fig. 4. PAPL vs deterministic policy in the number of FP for 2 tasks on a user trajectory. The first row visualizes results for task 1. In Fig. 4a, the x-axis is the difference in #FPs and y-axis is the count of the BBs on which the deterministic versus **PAPL** policies exhibit that difference. Notably, for 2 BBs, using **PAPL** leads to more than 10 fewer FPs. Figure 4b and 4c visualizes whether the system predicts a user location is in the box (red cross) or not (blue dot), by deterministic and **PAPL** policies on the BB that yields the largest difference in FPs. Here all the points are actually outside the yellow box, thus the fewer the red crosses are the better. The second row similarly visualizes comparison on the FPs for task 2.

to evaluate an access request, which is inadequate for real-time access control. The main latency bottleneck is within the NumPyro programs that perform BNN inference and sampling to estimate random event probabilities. We have not tried to optimize the current proof-of-concept implementation for inference speed, and we conjecture that this overhead in performing access control could be reduced by running on customized hardware accelerators and batch-processing of access requests.

Scalability for Automated Reasoning. To demonstrate the scalability of the LIRA encoding procedure presented in Sect. 3 for reasoning about location policies, we create two **PAPL** policies that involve a variable number of BBs and record the time needed to prove relative permissivenss between these policies. Each policy is an allow policy containing a disjunction of bounding boxes. For "structured" comparisons, we generate the BBs and the policies such that the one policy is guaranteed to be less permissive than the other. And for "random" comparisons, we generate random BBs. The scaling of the average running time across 10 executions is shown in Fig. 5. It shows that the procedure usually finishes in a few minutes for policies involving up to a few thousand BBs on a Macbook Pro with M1 Pro processor.

4.2 Case Study: Administering Deny-Lists for Machine Learning Based Access Control

For the second case study, we consider a problem that involves access control policy administration, in particular implementing a deny-list in a machine learning based access control (MLBAC) system. Specifically, we have an MLBAC system running but we would like to implement *changes* in the access control rules to deny certain accesses when new requests are received. This problem is studied in literature [43], and ML based methods are proposed to fulfill this requirement. The key research question is **RQ: Can PAPL policies help reduce the number of mistakes made by MLBAC when implementing a deny-list?**

Dataset and Preparation. We use the Amazon employee access challenge dataset [29] from Kaggle for the case study. This dataset requires ML models to use eight features (encoded as integers) to predict whether an Amazon employee should be granted access to some resource, and evaluates models using AUC score, i.e., the area under the receiver operating characteristic (ROC) curve. We first divide the data available into training and testing sets. Since the integer features originally represent categorical variables, we apply target encoding on the training and test data. We notice that the dataset is highly imbalanced–the number of allowed accesses is much larger than the number of denied accesses, which might make the ML models bias towards predicting accesses as allowed. Thus, we also perform random oversampling on the training data.

Experiments. For the three experiments, we inject three synthetic deny-lists on top of the original dataset. Each deny-list is a predicate over the attributes, for

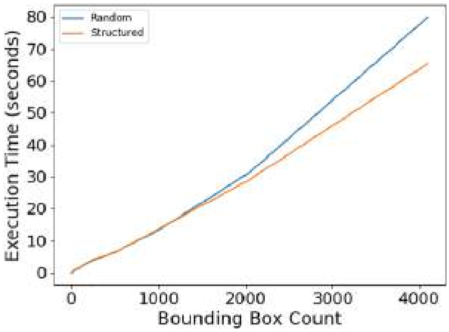


Fig. 5. Scalability for proving relative permissivenss of **PAPL** policies based on LIRA encodings.

example the first deny-list can be written as

$$\texttt{role\_family} = 19732 \vee \texttt{role\_rollup\_1} = 119062 \vee \texttt{role\_rollup\_2} = 118300$$

where `role_family`, `role_rollup_1`, and `role_rollup_2` are three features in access requests. A request is allowed only if it was allowed in the original dataset and is not included in the synthetic deny-list. Our main goal for this policy administration task is to enforce the deny-list, i.e., deny access requests if they conform to the injected deny rules, but we also observe how the system performs on the entire modified dataset. We implemented a random forest classifier and a BNN that predicts the allow probability for binary classification, based on which we considered four access control systems and we train them from scratch for each experiment. In the following, RF refers to the random forest classifier baseline considered by previous MLBAC work [43]. **PAPL** refers to a **PAPL** policy that first checks the access request using a symbolic deny rule that implements the deny-list. If the request is not governed by the rule, the system computes the empirical frequency of sampling a predicted probability of more 0.5 among all BNN predictions is greater than 0.8.

Results. Table 1 shows that in terms of AUC score for predicting access rights in this dataset, BNN offers comparable or even better predictions compared to RF. Table 2 shows that the **PAPL**-based access control system is effective in eliminating *all* false positives for the injected deny-list, and also for the whole synthetic dataset, at the cost of an increased number of FNs. A fundamental advantage of the **PAPL**-based access control system compared to RF is that the deny-list is guaranteed to be enforced correctly on all access requests. Also,

Table 1. RF vs BNN based on AUC score.

Exp	RF-AUC	BNN-AUC
1	0.8935	0.9422
2	0.7969	0.8841
3	0.6885	0.8122

Table 2. Comparing the number of FPs within the injected deny-list, the number of FPs and FNs on the entire test set with the injected deny-list, and the overall accuracy (percentage of correct predictions within the test set) for access control systems based on RF and **PAPL** using three experiments.

Exp	RF				**PAPL**			
	FP(list)	FP(all)	FN(all)	Acc	FP(list)	FP(all)	FN(all)	Acc
1	4/935	208/1254	250/5300	0.9301	0/935	121/1254	738/5300	0.8689
2	35/367	272/715	150/5839	0.9356	0/367	168/715	853/5839	0.8442
3	9/77	260/435	155/6119	0.9367	0/77	179/435	833/6119	0.8456

as mentioned in the previous case study, we could adjust the probability thresholds in the policies to balance FPs and FNs in an interpretable way within the framework provided by **PAPL**.

5 Related Work

Reasoning About Policies. Automated reasoning about policies encodes policies as SMT formulas and then invokes SMT solver like Z3 [20] to prove properties of policies [10,22,28,32,52]. Our work extends this scheme by presenting a way of encoding clauses involving probabilities into LIRA formulas.

Machine Learning for Access Control. In additional to traditional access control schemes, an alternative is to use machine learning to make access decisions [14,18,33,35,37,43,44]. The survey [42] offers a more comprehensive overview on this subject. The **PAPL** language offers an extension to existing MLBAC work by allowing policy writers to refer to the *uncertainty* in the ML predictions. Furthermore, **PAPL** allows combinations of ML models and traditional access policies, which is difficult to achieve in existing MLBAC methods.

Probabilistic Machine Learning. In certain application domains, it is important to capture *uncertainty* in the ML predictions [39]. Our work provides a method for using uncertainty in the particular domain of MLBAC, where knowledge about uncertainty in the ML models can be crucial in making more robust decisions. Within the realm of probabilistic/bayesian ML, it has been shown that Bayesian neural networks [27,40] could effectively accumulate domain knowledge (from similar tasks) in its prior to yield good uncertainty [36], and they can produce more robust predictions for out-of-distribution data [40], making it a good candidate model to be used in access control systems based on **PAPL**.

Probability and Programming. Research on probabilistic programming languages [8,12,13,21,45,48] (PPL) aims to bridge probabilistic reasoning together with general purpose programming diagrams. In particular, [13] presents a programming abstraction for representing and using uncertainty. In general, PPL provides ways to specify probabilistic models, do inferencing, and compute probabilities. Our work provides a scheme to *compute* and *reason* about probabilities in the context of access control, utilizing the capability of an existing PPL NumPyro [12].

Access Control. Location-based [6,7,17,30], risk-based [16,35], and context-aware [31,47,50] access control paradigms involves uncertain information, and are useful for different application domains, including augmented reality (AR) devices and applications. **PAPL** could be used to implement these paradigms as long as the uncertain information involved in access control could be expressed as probabilities of random events.

References

1. Cedar Language. https://www.cedarpolicy.com/en. Accessed 27 Jan 2024
2. Cloud leak: WSJ parent company dow jones exposed customer data. https://www.upguard.com/breaches/cloud-leak-dow-jones. Accessed 27 Jan 2023
3. eXtensible access control markup language (XACML) version 3.0. https://www.oasis-open.org/standard/xacmlv3-0. Accessed 27 Jan 2024
4. Another misconfigured Amazon S3 server leaks data of 50,000 Australian employees. SC Media (2017). https://www.scmagazine.com/news/breach/another-misconfigured-amazon-s3-server-leaks-data-of-50000-australian-employees
5. Akter, T., Dosono, B., Ahmed, T., Kapadia, A., Semaan, B.: "I am uncomfortable sharing what i can't see": privacy concerns of the visually impaired with camera based assistive applications. In: Proceedings of the 29th USENIX Conference on Security Symposium. SEC'20, USA. USENIX Association (2020). https://doi.org/10.5555/3489212.3489321
6. Ardagna, C., Cremonini, M., di Vimercati, S.D.C., Samarati, P.: Privacy-enhanced location-based access control. In: Gertz, M., Jajodia, S. (eds.) Handbook of Database Security, pp. 531–552. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-48533-1_22
7. Ardagna, C.A., Cremonini, M., Damiani, E., di Vimercati, S.D.C., Samarati, P.: Supporting location-based conditions in access control policies. In: Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security - ASIACCS '06, Taipei, Taiwan, p. 212. ACM Press (2006). https://doi.org/10.1145/1128817.1128850
8. Bach, S.H., Broecheler, M., Huang, B., Getoor, L.: Hinge-loss Markov random fields and probabilistic soft logic. J. Mach. Learn. Res. **18**(1), 3846–3912 (2017). https://doi.org/10.5555/3122009.3176853
9. Backes, J., et al.: Stratified abstraction of access control policies. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 165–176. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_9
10. 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
11. Bauer, L., Cranor, L.F., Reeder, R.W., Reiter, M.K., Vaniea, K.: Real life challenges in access-control management. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '09, New York, NY, USA, pp. 899–908. Association for Computing Machinery (2009). https://doi.org/10.1145/1518701.1518838
12. Bingham, E., et al.: Pyro: deep universal probabilistic programming. J. Mach. Learn. Res. **20**(1), 973–978 (2019). https://doi.org/10.5555/3322706.3322734
13. Bornholt, J., Mytkowicz, T., McKinley, K.S.: Uncertain<T>: A first-order type for uncertain data. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '14, pp. 51–66, New York, NY, USA. Association for Computing Machinery (2014). https://doi.org/10.1145/2541940.2541958
14. Cappelletti, L., Valtolina, S., Valentini, G., Mesiti, M., Bertino, E.: On the Quality of Classification Models for Inferring ABAC Policies from Access Logs. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 4000–4007 (2019). https://doi.org/10.1109/BigData47090.2019.9005959

15. Chen, S., Li, Z., Dangelo, F., Gao, C., Fu, X.: A case study of security and privacy threats from augmented reality (AR). In: 2018 International Conference on Computing, Networking and Communications (ICNC), pp. 442–446 (2018). https://doi.org/10.1109/ICCNC.2018.8390291
16. Cheng, P.C., Rohatgi, P., Keser, C., Karger, P.A., Wagner, G.M., Reninger, A.S.: Fuzzy multi-level security: an experiment on quantified risk-adaptive access control. In: 2007 IEEE Symposium on Security and Privacy (SP '07), pp. 222–230 (2007). https://doi.org/10.1109/SP.2007.21, iSSN: 2375-1207
17. van Cleeff, A., Pieters, W., Wieringa, R.: Benefits of location-based access control: a literature study. In: 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing, pp. 739–746 (2010). https://doi.org/10.1109/GreenCom-CPSCom.2010.148
18. Das, S., Mitra, B., Atluri, V., Vaidya, J., Sural, S.: Policy engineering in RBAC and ABAC. In: Samarati, P., Ray, I., Ray, I. (eds.) From Database to Cyber Security. LNCS, vol. 11170, pp. 24–54. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04834-1_2
19. De Guzman, J.A., Thilakarathna, K., Seneviratne, A.: Security and privacy approaches in mixed reality: A literature survey. ACM Comput. Surv. **52**(6), 110:1-110:37 (2019). https://doi.org/10.1145/3359626
20. 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
21. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: a probabilistic prolog and its application in link discovery. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence. IJCAI'07, San Francisco, CA, USA, pp. 2468–2473. Morgan Kaufmann Publishers Inc. (2007). https://doi.org/10.5555/1625275.1625673
22. Eiers, W., Sankaran, G., Li, A., O'Mahony, E., Prince, B., Bultan, T.: Quantifying permissiveness of access control policies. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22,, New York, NY, USA, pp. 1805–1817. Association for Computing Machinery (2022). https://doi.org/10.1145/3510003.3510233
23. Frank, M., Basin, D., Buhmann, J.M.: A class of probabilistic models for role engineering. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. CCS '08, New York, NY, USA, pp. 299–310. Association for Computing Machinery (2008). https://doi.org/10.1145/1455770.1455809
24. d'Avila Garcez, A., et al.: Neural-symbolic learning and reasoning: a survey and interpretation. Neuro-Symbolic Artif. Intell. State Art **342**(1), 327 (2022)
25. d'Avila Garcez, A., Lamb, L.C.: Neurosymbolic AI: the 3rd wave. Artif. Intell. Rev. **56**(11), 12387–12406 (2023). https://doi.org/10.1007/s10462-023-10448-w
26. Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning). The MIT Press (2007). https://doi.org/10.5555/1296231
27. Goan, E., Fookes, C.: Bayesian neural networks: an introduction and survey. In: Mengersen, K.L., Pudlo, P., Robert, C.P. (eds.) Case Studies in Applied Bayesian Data Science. LNM, vol. 2259, pp. 45–87. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-42553-1_3
28. Guelev, D.P., Ryan, M., Schobbens, P.Y.: Model-checking access control policies. In: Zhang, K., Zheng, Y. (eds.) ISC 2004. LNCS, vol. 3225, pp. 219–230. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30144-8_19

29. Hamner, B., kenmonta, Cukierski, W.: Amazon employee access challenge (2013). https://www.kaggle.com/c/amazon-employee-access-challenge
30. He, W., et al.: Rethinking access control and authentication for the home internet of things (IoT). In: Proceedings of the 27th USENIX Conference on Security Symposium, pp. 255–272. SEC'18, USA. USENIX Association (2018). https://doi.org/10.5555/3277203.3277223
31. Jana, S., et al.: Enabling fine-grained permissions for augmented reality applications with recognizers. In: Proceedings of the 22nd USENIX Conference on Security. SEC'13, USA, pp. 415-430. USENIX Association (2013). https://doi.org/10.5555/2534766.2534802
32. Jeffrey, A., Samak, T.: Model checking firewall policy configurations. In: 2009 IEEE International Symposium on Policies for Distributed Systems and Networks, pp. 60–67 (2009). https://doi.org/10.1109/POLICY.2009.32
33. Karimi, L., Abdelhakim, M., Joshi, J.: Adaptive ABAC Policy Learning: A Reinforcement Learning Approach (2021). https://doi.org/10.48550/arXiv.2105.08587
34. Kolmogoroff, A.: Grundbegriffe Der Wahrscheinlichkeitsrechnung. Springer, Heidelberg (1933). https://doi.org/10.1007/978-3-642-49888-6
35. Krautsevich, L., Lazouski, A., Martinelli, F., Yautsiukhin, A.: Towards attribute-based access control policy engineering using risk. In: Bauer, T., Großmann, J., Seehusen, F., Stølen, K., Wendland, M.F. (eds.) Risk Assessment and Risk-Driven Testing. LNCS, pp. 80–90. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07076-6_6
36. Lacoste, A., Oreshkin, B., Chung, W., Boquet, T., Rostamzadeh, N., Krueger, D.: Uncertainty in multitask transfer learning (2018). https://arxiv.org/abs/1806.07528
37. Liu, A., Du, X., Wang, N.: Efficient access control permission decision engine based on machine learning. Secur. Commun. Networks **2021**, e3970485 (2021). https://doi.org/10.1155/2021/3970485
38. Liu, J.S.: Peskun's theorem and a modified discrete-state Gibbs sampler. Biometrika **83**(3), 681–682 (1996). https://doi.org/10.1093/biomet/83.3.681
39. Murphy, K.P.: Probabilistic Machine Learning: An Introduction. The MIT press (2022)
40. Murphy, K.P.: Probabilistic Machine Learning: Advanced Topics. The MIT Press (2023)
41. Newcombe, R.A., Lovegrove, S.J., Davison, A.J.: DTAM: dense tracking and mapping in real-time. In: Proceedings of the 2011 International Conference on Computer Vision. ICCV '11, USA, pp. 2320–2327. IEEE Computer Society (2011). https://doi.org/10.1109/ICCV.2011.6126513
42. Nobi, M.N., Gupta, M., Praharaj, L., Abdelsalam, M., Krishnan, R., Sandhu, R.: Machine Learning in Access Control: A Taxonomy and Survey (2022). https://doi.org/10.48550/arXiv.2207.01739
43. Nobi, M.N., Krishnan, R., Huang, Y., Sandhu, R.: Administration of machine learning based access control. In: Atluri, V., Di Pietro, R., Jensen, C.D., Meng, W. (eds.) Computer Security – ESORICS 2022. LNCS, pp. 189–210. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17146-8_10
44. Nobi, M.N., Krishnan, R., Huang, Y., Shakarami, M., Sandhu, R.: Toward deep learning based access control. In: Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy. CODASPY '22, New York, NY, USA, pp. 143–154. Association for Computing Machinery (2022).https://doi.org/10.1145/3508398.3511497

45. Phan, D., Pradhan, N., Jankowiak, M.: Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro (2019). https://doi.org/10.48550/arXiv.1912.11554
46. Roesner, F., Kohno, T., Molnar, D.: Security and privacy for augmented reality systems. Commun. ACM **57**(4), 88–96 (2014). https://doi.org/10.1145/2580723.2580730
47. Roesner, F., Molnar, D., Moshchuk, A., Kohno, T., Wang, H.J.: World-driven access control for continuous sensing. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. CCS '14, pp. 1169–1181. Association for Computing Machinery (2014). https://doi.org/10.1145/2660267.2660319, event-place: New York, NY, USA
48. Saad, F.A., Rinard, M.C., Mansinghka, V.K.: SPPL: probabilistic programming with fast exact symbolic inference. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021, New York, NY, USA, pp. 804–819. Association for Computing Machinery (2021). https://doi.org/10.1145/3453483.3454078
49. Sinclair, S., Smith, S.W.: Preventative directions for insider threat mitigation via access control. In: Stolfo, S.J., Bellovin, S.M., Keromytis, A.D., Hershkop, S., Smith, S.W., Sinclair, S. (eds.) Insider Attack and Cyber Security: Beyond the Hacker. Advances in Information Security, pp. 165–194. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-77322-3_10
50. Templeman, R., Korayem, M., Crandall, D., Kapadia, A.: PlaceAvoider: Steering first-person cameras away from sensitive spaces. In: Proceedings 2014 Network and Distributed System Security Symposium. Internet Society (2014). https://doi.org/10.14722/ndss.2014.23014, event-place: San Diego, CA
51. Torres-Sospedra, J., et al.: UJIIndoorLoc: a new multi-building and multi-floor database for WLAN fingerprint-based indoor localization problems. In: 2014 International Conference on Indoor Positioning and Indoor Navigation (IPIN), pp. 261–270 (2014). https://doi.org/10.1109/IPIN.2014.7275492
52. Turkmen, F., den Hartog, J., Ranise, S., Zannone, N.: Analysis of XACML policies with SMT. In: Focardi, R., Myers, A. (eds.) POST 2015. LNCS, vol. 9036, pp. 115–134. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46666-7_7
53. Zhou, G.: Mixed hamiltonian monte carlo for mixed discrete and continuous variables. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. NIPS '20, Red Hook, NY, USA. Curran Associates Inc. (2020). https://doi.org/10.5555/3495724.3497158

Compositional Value Iteration with Pareto Caching

Kazuki Watanabe[1,2](✉), Marck van der Vegt[3], Sebastian Junges[3], and Ichiro Hasuo[1,2]

[1] National Institute of Informatics, Tokyo, Japan
kazukiwatanabe@nii.ac.jp
[2] The Graduate University for Advanced Studies (SOKENDAI), Kanagawa, Japan
[3] Radboud University, Nijmegen, The Netherlands

Abstract. The de-facto standard approach in MDP verification is based on value iteration (VI). We propose *compositional VI*, a framework for model checking compositional MDPs, that addresses efficiency while maintaining soundness. Concretely, compositional MDPs naturally arise from the combination of individual components, and their structure can be expressed using, e.g., string diagrams. Towards efficiency, we observe that compositional VI repeatedly verifies individual components. We propose a technique called *Pareto caching* that allows to reuse verification results, even for previously unseen queries. Towards soundness, we present two stopping criteria: one generalizes the optimistic value iteration paradigm and the other uses Pareto caches in conjunction with recent baseline algorithms. Our experimental evaluations shows the promise of the novel algorithm and its variations, and identifies challenges for future work.

1 Introduction

MDP Model Checking and Value Iteration. *Markov decision processes (MDPs)* are the standard model for sequential decision making in stochastic settings. A standard question in the verification of MDPs is: *what is the maximal probability that an error state is reached.* MDP model checking is an active topic in the formal verification community. *Value iteration (VI)* [44] is an iterative and approximate method whose performance in MDP model checking is well-established [11,29,30]. Several extensions with *soundness* have been proposed; they provide, in addition to under-approximations, also over-approximations with a desired precision [4,24,30,43,46], so that an approximate answer comes

K.W. and I.H. are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603) and the ASPIRE grant No. JPMJAP2301, JST. K.W. is supported by the JST grants No. JPMJFS2136 and JPMJAX23CU. S.J. is supported by the NWO Veni ProMiSe (222.147).

K. Watanabe and M. van der Vegt—Equal contribution.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 467–491, 2024.
https://doi.org/10.1007/978-3-031-65633-0_21

with an error bound. These sound algorithms are implemented in mature model checkers such as Prism [37], Modest [27], and Storm [32].

Compositional Model Checking. Even with these state-of-the-art algorithms, it is a challenge to model check large MDPs efficiently with high precision. Experiments observe that MDPs with more than 10^8 states are too large for those algorithms [35,53,54]—they simply do not fit in memory. However, such large MDPs often arise as models of complicated stochastic systems, e.g. in the domains of network and robotics. Furthermore, even small models may be numerically challenging to solve due to their structure [4,24,29].

Compositional model checking is a promising approach to tackle this scalability challenge. Given a compositional structure of a target system, compositional model checking executes a divide-and-conquer algorithm that avoids loading the entire state space at once, often solving the above memory problem. Moreover, reusing the model checking results for components can lead to speed-up by magnitudes. Although finding a suitable compositional structure for a given "monolithic" MDP is still open, many systems come with such an *a priori* compositional structure. For example, such compositional structures are often assumed in robotics and referred to as *hierarchical models* [5,23,31,35,40,48,51].

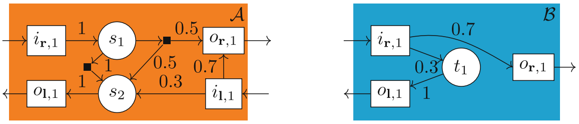


Fig. 1. open MDPs $\mathcal{A}$ and $\mathcal{B}$.

Recently, *string diagrams of MDPs* are introduced for compositional model checking [53,54]; the current paper adopts this formalism. There, MDPs are extended with (open) entrances and exits (Fig. 1), and they get composed by *sequential composition* $\fatsemi$ and *sum* $\oplus$. See Fig. 2, where the right-hand sides are simple juxtapositions of graphs (wires get connected in $\fatsemi$). This makes the formalism focused on sequential (as opposed to parallel) composition. This restriction eases the design of compositional algorithms; yet, the formalism is rich enough to capture the compositional structures of many system models.

Current Work: Compositional Value Iteration. In this paper, we present a *compositional value iteration (CVI)* algorithm that solves reachability probabilities of string diagrams of MDPs, operating in a divide-and-conquer manner along compositional structures. Our approximate VI algorithm comes with *soundness*—it produces error bounds—and exploits compositionality for *efficiency*.

Specifically, for soundness, we lift the recent paradigm of *optimistic value iteration (OVI)* [30] to the current compositional setting. We use it both for local (component-level) model checking and—in one of the two global VI stopping criteria that we present—for providing a global over-approximation.

For efficiency, firstly, we adopt a *top-down* compositional approach where each component is model-checked repeatedly, each time on a different weight $\mathbf{w}$, in a *by-need* manner. Secondly, in order to suppress repetitive computation on similar weights, we introduce a novel technique of *Pareto caching* that allows "approximate reuse" of model checking results. This closely relates to multi-objective probabilistic model checking [17,20,45], without the explicit goal of building Pareto curves. Our Pareto caching also leads to another (*sound*) global VI stopping criterion that is based on the approximate bottom-up approach [54].

Our algorithm is approximate (unlike the exact one in [53]), and top-down (unlike the *bottom-up* approximate one in [54]). Experimental evaluation demonstrates its performance thanks to the combination of these two features.

Contributions and Organization. We start with an overview (Sect. 2) that presents graphical intuitions. After formalizing the problem setting in Sect. 3, we move on to describe our technical contributions:

- *compositional value iteration* for string diagrams of MDPs where VI is run in a top-down and thus by-need manner (Sect. 4.2),
- the *Pareto caching* technique for reusing results for components (Sect. 5.2),
- two *global stopping criteria* that ensure soundness (Sect. 6).

We evaluate and discuss our approach through experiments (Sect. 7), show related work (Sect. 8), and conclude this paper (Sect. 9).

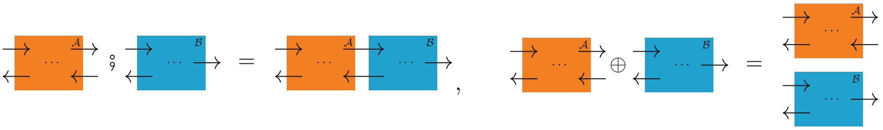


Fig. 2. sequential composition $\mathcal{A} \fatsemi \mathcal{B}$ and sum $\mathcal{A} \oplus \mathcal{B}$ of open MDPs. The framework is *bidirectional* (edges can be left- and right-ward); thus loops can arise in $\mathcal{A} \fatsemi \mathcal{B}$.

Notations. For a natural number m, we write $[m]$ for $\{1, \dots, m\}$. For a set X, we write $\mathcal{D}(X)$ for the set of distributions on X. For sets X, Y, we write $X \uplus Y$ for their disjoint union and $f\colon X \rightharpoonup Y$ for a partial function f from X to Y.

2 Overview

This section illustrates our take on CVI with so-called Pareto caches using graphical intuitions. We describe MDPs as string diagrams over so-called *open MDPs* [53]. Open MDPs, such as $\mathcal{A}, \mathcal{B}$ in Fig. 1, extend MDPs with *open ends* (entrances and exits). We use two operations $\fatsemi$ and $\oplus$; see Fig. 2. That figure also illustrates the *bidirectional* nature of the formalism: arrows can point left and right; thus acyclic MDPs can create cycles when combined. String diagrams come from category theory (see [53]) and they are used in many fields of computer science [8,9,22,52].

2.1 Approximate Bottom-Up Model Checking

The first compositional model checking algorithm for string diagrams of MDPs is in [53], which is exact. Subsequently, in [54], an *approximate* compositional model checking algorithm is proposed. This is the basis of our algorithm and we shall review it here. Consider, for illustration, the sequential composition $\mathcal{A} \fatsemi \mathcal{B}$ in Fig. 3, where the exit o_3 is the target. The algorithm from [54] proceeds in the following *bottom-up* manner.

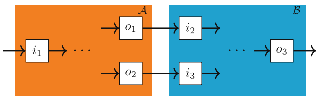


Fig. 3. $\mathcal{A} \fatsemi \mathcal{B}$

First Step: Model Checking Each Component. Firstly, model checking is conducted for component oMDPs $\mathcal{A}$ and $\mathcal{B}$ separately, which amounts to identifying an *optimal scheduler* for each. At this point, however, it is unclear what constitutes an optimal scheduler:

Example 1. In the MDP $\mathcal{A}$ in Fig. 3, let's say the reachability probabilities $\big(\mathrm{RPr}^{\sigma_1}(i_1 \to o_1), \mathrm{RPr}^{\sigma_1}(i_1 \to o_2)\big)$ are $(0.2, 0.7)$ under a scheduler σ_1, and $(0.6, 0.2)$ under another σ_2. One cannot tell which scheduler (σ_1 or σ_2) is better for the global objective (i.e. reaching o_3 in $\mathcal{A} \fatsemi \mathcal{B}$) since $\mathcal{B}$ is a black box.

Concretely, the context $\_ \fatsemi \mathcal{B}$ of $\mathcal{A}$ is unknown. Therefore we have to compute all candidates of optimal schedulers, instead of one. This set is given by, for each component $\mathcal{C}$ and its entrance i,

$$\left\{ \text{schedulers } \sigma \;\middle|\; \big(\mathrm{RPr}^{\sigma}(i \to o)\big)_{o:\mathcal{C}\text{ 's exit}} \text{ is Pareto optimal} \right\}. \tag{1}$$

Here the *Pareto optimality* is a usual notion from *multi-objective model checking* (e.g. [17,41]); here, it means that there is no scheduler σ' that *dominates* σ in the sense that $\mathrm{RPr}^{\sigma}(i \to o) \leq \mathrm{RPr}^{\sigma'}(i \to o)$ holds for each o and $<$ holds for some o. The two points from the example can be plotted, see Fig. 4.

The *Pareto curve*—the set of points $\big(\mathrm{RPr}^{\sigma}(i \to o)\big)_o$ for the Pareto optimal schedulers σ in (1)—will look like the dashed blue line in Fig. 4.The solid blue line is realizable by a convex combination of the schedulers σ_1 and σ_2. It is always below the Pareto curve.

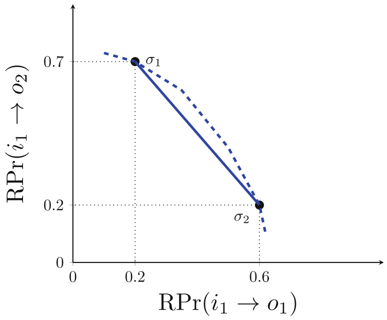


Fig. 4. Pareto-optimal points

The algorithm in [54] computes guaranteed under- and over-approximations (L, U) of Pareto-optimal points (1) for every open MDP. See Fig. 5; here the green area indicates the under-approximation, and the red area is the complement of the over-approximation, so that any Pareto-optimal points are guaranteed to be in their gap (white). These approximations are obtained by repeated application of (optimistic) value iteration on the open MDPs, i.e., a standard approach for verifying MDPs, based on [20, 47]. We formalize these notions in Sect. 5.1.

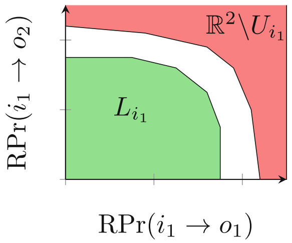


Fig. 5. approximations (L_{i_1}, U_{i_1}).

Second Step: Combination along. $\fatsemi, \oplus$ The second (inductive) step of the bottom-up algorithm in [54] is to combine the results of the first step—approximations as in Fig. 5, and the corresponding (near) optimal schedulers (1), for each component $\mathcal{C}$—along the operations $\fatsemi, \oplus$ in a string diagram.

Here we describe this second step through the example in Fig. 3. It computes reachability probabilities

$$\begin{aligned}\mathrm{RPr}^{\sigma,\tau}(i_1 \to o_3) = {} & \mathrm{RPr}^{\sigma}(i_1 \to o_1) \cdot \mathrm{RPr}^{\tau}(i_2 \to o_3) \\ & + \mathrm{RPr}^{\sigma}(i_1 \to o_2) \cdot \mathrm{RPr}^{\tau}(i_3 \to o_3)\end{aligned} \tag{2}$$

for each combination of Pareto-optimal schedulers σ (for $\mathcal{A}$) and τ (for $\mathcal{B}$) to find which combinations of σ, τ are Pareto optimal for $\mathcal{A} \fatsemi \mathcal{B}$.

The equality (2)—called the *decomposition equality* in [53]—enables compositional reasoning on Pareto-optimal points and on their approximations: Pareto-optimal schedulers for $\mathcal{A} \fatsemi \mathcal{B}$ can be computed from those for $\mathcal{A}$ and $\mathcal{B}$. This compositional reasoning can be exploited for performance. In particular, when the same component $\mathcal{A}$ occurs multiple times in a string diagram $\mathbb{D}$, the model checking result of $\mathcal{A}$ can be reused multiple times.

2.2 Key Idea I: From Bottom-Up to Top-Down

The bottom-up approaches compute the Pareto curves independent of the context of the open MDP. One key idea is to move from bottom-up to *top-down*, a direction followed by other compositional techniques too, see Sect. 8.

For illustration, consider the sequential composition $\mathcal{A} \fatsemi \mathcal{B}$ in Fig. 6; we have concretized $\mathcal{B}$ in Fig. 3. For this $\mathcal{B}$, it follows that $\mathrm{RPr}(i_2 \to o_3) = 0.8$ and $\mathrm{RPr}(i_3 \to o_3) = 0.3$. Therefore the equality (2) boils down to

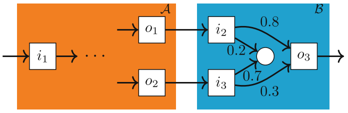


Fig. 6. $\mathcal{A} \fatsemi \mathcal{B}$

$$\mathrm{RPr}^{\sigma}(i_1 \to o_3) = 0.8 \cdot \mathrm{RPr}^{\sigma}(i_1 \to o_1) + 0.3 \cdot \mathrm{RPr}^{\sigma}(i_1 \to o_2). \tag{3}$$

The equation (3) is a significant simplification compared to (2):

- in (2), since the *weight* $\bigl(\mathrm{RPr}^{\tau}(i_2 \to o_3), \mathrm{RPr}^{\tau}(i_3 \to o_3)\bigr)$ is unknown, we must compute multidimensional Pareto curves as in Figs. 4 and 5;
- in (3), since the weight is known to be $(0.8, 0.3)$, we can solve the equation using standard single-objective model checking.

Exploiting this simplification is our first key idea. We introduce a systematic procedure for deriving weights (such as $(0.8, 0.3)$ above) that uses the context of an oMDP, i.e., it goes *top-down* along the string diagram. The procedure works for bi-directional sequential composition (thus for loops, cf. Fig. 2), not only for uni-directional as in Fig. 6. In the procedure, we first examine the context of a component $\mathcal{C}$, approximate a weight $\mathbf{w}$ for $\mathcal{C}$, and then compute maximum weighted reachability probabilities in $\mathcal{C}$. We formalize the approach in Sect. 4.2.

Potential performance advantages compared to the bottom-up algorithm in [54] should be obvious from Fig. 6. Specifically, the bottom-up algorithm draws a complete picture for Pareto-optimal points (such as Fig. 5) *once for all*, but a large part of this complete picture may not be used. In contrast, the top-down one draws the picture in a *by-need* manner, for a weight $\mathbf{w}$ only when the weight $\mathbf{w}$ is suggested by the context.

The top-down approximation of Pareto-optimal points is illustrated in Fig. 7. Here a weight $\mathbf{w}$ is the normal vector of the blue lines; the figure shows a situation after considering two weights.

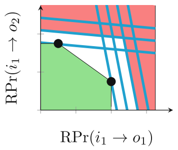


Fig. 7. top-down approximation.

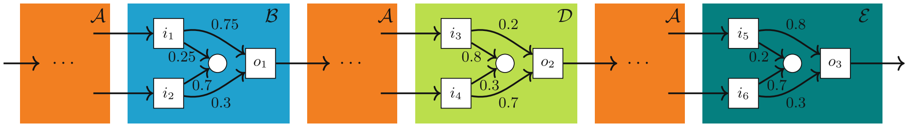


Fig. 8. $\mathcal{A} \fatsemi \mathcal{B} \fatsemi \mathcal{A} \fatsemi \mathcal{D} \fatsemi \mathcal{A} \fatsemi \mathcal{E}$, an example

2.3 Key Idea II: Pareto Caching

Our second key idea (*Pareto caching*) arises when we try to combine the last idea (top-down compositionality) with the key advantage of the bottom-up approach [54], namely *exploiting duplicates*. Consider the string diagram $\mathcal{A}\fatsemi\mathcal{B}\fatsemi\mathcal{A}\fatsemi\mathcal{D}\fatsemi\mathcal{A}\fatsemi\mathcal{E}$ in Fig. 8, for motivation, where we designate multiple occurrences of $\mathcal{A}$ by $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ for distinction, from left to right.

Let us run the top-down algorithm. The component $\mathcal{E}$ suggests the weight $(0.8, 0.3)$ for the two exits of $\mathcal{A}_3$, and $\mathcal{D}$ suggest the weight $(0.2, 0.7)$ for the exits of $\mathcal{A}_2$. Recalling that $\mathcal{A}_2$ and $\mathcal{A}_3$ are identical, the weighted optimization results for these two weights can be combined, leading to a picture like Fig. 7.

Now, in Fig. 8, we go on to the component $\mathcal{B}$. It suggests the weight $(0.75, 0.3)$.

- In the bottom-up approach [54], performance advantages are brought by *exploiting duplicates*, that is, by reusing the model checking result of a component $\mathcal{C}$ for its multiple occurrences.
- Therefore, also here, we wish to use the previous analysis results for $\mathcal{A}$—for the weights $(0.8, 0.3)$ and $(0.2, 0.7)$—for the weight $(0.75, 0.3)$.
- Intuitively, $(0.75, 0.3)$ seems close enough to $(0.8, 0.3)$, suggesting that we can use the previously obtained result for $(0.8, 0.3)$.

But this casts the following questions: what is it for two weights to be "close enough"? Is $(0.75, 0.3)$ really closer to $(0.8, 0.3)$ than to $(0.2, 0.7)$? Can we bound errors—much like in Sect. 2.1—that arise from this "approximate reuse"?

In Sect. 5.2, we use the existing theory on Pareto curves in multi-objective model checking from [17,20,45] to answer these questions. Intuitively, the previous analysis result (red and green regions) gets *queried* on a new weight $\mathbf{w}$ (the normal vector of the blue lines), as illustrated in Fig. 9. We call answering weighted reachability based on the Pareto curve *Pareto caching*. The technique can prevent many invocations of using VI to compute the weighted reachability for $\mathbf{w}$. The distance between the under- and over-approximations computed this way can be big; if so ("cache miss"), we run VI again for the weight $\mathbf{w}$.

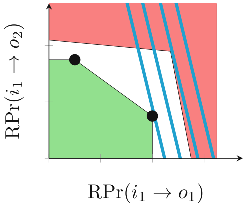


Fig. 9. Pareto caching

2.4 Global Stopping Criteria (GSCs)

On top of two key ideas, we provide two global stopping criteria (GSCs) in Sect. 6: one is based on the ideas from OVI [30] and the other is a symbiosis of the Pareto caches with the bottom-up approach. Although ensuring the termination of our algorithm in finite steps with our GSCs remains future work, we show that our GSCs are *sound*, that is, its output satisfies a given precision upon termination.

3 Formal Problem Statement

We recall (weighted) reachability in Markov decision processes (MDPs) and formalize string diagrams as their compositional representation. Together, this is the formal basis for our problem statement as already introduced above.

3.1 Markov Decision Process (MDP)

Definition 3.1 (MDP). An *MDP* $\mathcal{M} = (S, A, P)$ is a tuple with a finite set S of *states*, a finite set A of *actions*, and a *probabilistic transition function* $P \colon S \times A \rightharpoonup \mathcal{D}(S)$ (which is a partial function, cf. notations in Sect. 1).

A *(finite) path* (on $\mathcal{M}$) is a finite sequence of states $\pi := (\pi_i)_{i \in [m]}$. We write $\mathsf{FPath}_{\mathcal{M}}$ for the set of finite paths on $\mathcal{M}$. A *memoryless scheduler* σ is a function $\sigma \colon S \to \mathcal{D}(A)$; in this paper, memoryless schedulers suffice [20,44]. We say σ is *deterministic memoryless (DM)* if for each $s \in S$, $\sigma(s)$ is Dirac. We also write $\sigma \colon S \to A$ for a DM scheduler σ. The set of all memoryless schedulers on $\mathcal{M}$ is $\Sigma^{\mathcal{M}}$, and the set of all DM schedulers on $\mathcal{M}$ is $\Sigma^{\mathcal{M}}_{\mathrm{d}}$.

For a memoryless scheduler σ and a *target state* $t \in S$, the *reachability probability* $\mathrm{RPr}^{\mathcal{M},\sigma,t}(s)$ from a state s is given by $\mathrm{RPr}^{\mathcal{M},\sigma,t}(s) := \sum_{\pi \in \mathsf{FPath}_{\mathcal{M}}(t)} \mathrm{Pr}^{\mathcal{M}}_{\sigma,s}(\pi)$, where (i) the set $\mathsf{FPath}_{\mathcal{M}}(t) \subseteq \mathsf{FPath}_{\mathcal{M}}$ is defined by $\mathsf{FPath}_{\mathcal{M}}(t) := \{(\pi_i)_{i\in[m]} \in \mathsf{FPath}_{\mathcal{M}} \mid \mathrm{last}(\pi) = t, \text{ and } \pi_i \neq t \text{ for } i \in [m-1]\}$, and (ii) the probability $\mathrm{Pr}^{\mathcal{M}}_{\sigma,s}(\pi)$ is defined by $\mathrm{Pr}^{\mathcal{M}}_{\sigma,s}(\pi) := \prod_{i\in[m-1]} \sum_{a\in A} P(\pi_i, a, \pi_{i+1}) \cdot \sigma(\pi_{i-1})(a)$ if $\pi_1 = s$ and $\mathrm{Pr}^{\mathcal{M}}_{\sigma,s}(\pi) := 0$ otherwise.

Towards our compositional approach for a reachability objective, we must generalize the objective to a *weighted reachability probability* objective: we want to compute the *weighted sum*—with respect to a certain weight vector $\mathbf{w}$—over reachability probabilities to multiple target states. The standard reachability probability problem is a special case of this weighted reachability problem using a suitable unit vector $\mathbf{e}$ as the weight $\mathbf{w}$.

Definition 3.2 (weighted reachability probability). Let $\mathcal{M}$ be an MDP, and T be a set of target states. A *weight* $\mathbf{w}$ on T is a vector $\mathbf{w} := (w_t)_{t\in T} \in [0,1]^T$.

Let s be a state, and σ be a scheduler. The *weighted reachability probability* $\mathrm{WRPr}^{\mathcal{M},\sigma,T}(\mathbf{w}, s) \in [0,1]$ from s to T over σ with respect to a weight $\mathbf{w}$ is defined naturally by a weighted sum, that is, $\mathrm{WRPr}^{\mathcal{M},\sigma,T}(\mathbf{w}, s) := \sum_{t\in T} w_t \cdot \mathrm{RPr}^{\mathcal{M},\sigma,t}(s)$. We write $\mathrm{WRPr}^{\mathcal{M},T}_{\max}(\mathbf{w}, s)$ for the *maximum weighted reachability probability* $\sup_\sigma \mathrm{WRPr}^{\mathcal{M},\sigma,T}(\mathbf{w}, s)$. (The supremum is realizable; see e.g. [28].)

3.2 String Diagram of MDPs

Definition 3.3 (oMDP). An *open MDP* (oMDP) $\mathcal{A} = (M, \mathsf{IO})$ is a pair consisting of an MDP M with *open ends* $\mathsf{IO} = (I_{\mathbf{r}}, I_{\mathbf{l}}, O_{\mathbf{r}}, O_{\mathbf{l}})$, where $I_{\mathbf{r}}, I_{\mathbf{l}}, O_{\mathbf{r}}, O_{\mathbf{l}} \subseteq S$ are pairwise disjoint and each of them is totally ordered. The states in $I := I_{\mathbf{r}} \cup I_{\mathbf{l}}$ are the *entrances*, and the states in $O := O_{\mathbf{r}} \cup O_{\mathbf{l}}$ are the *exits*, respectively. We often use superscripts to designate the oMDP $\mathcal{A}$ in question, such as $I^{\mathcal{A}}$ and $O^{\mathcal{A}}$.

We write $\mathrm{arity}(\mathcal{A}) \colon (m_{\mathbf{r}}, m_{\mathbf{l}}) \to (n_{\mathbf{r}}, n_{\mathbf{l}})$ for the *arities* of $\mathcal{A}$, where $m_{\mathbf{r}} := |I_{\mathbf{r}}|$, $m_{\mathbf{l}} := |O_{\mathbf{l}}|$, $n_{\mathbf{r}} := |O_{\mathbf{r}}|$, and $n_{\mathbf{l}} := |I_{\mathbf{l}}|$. We assume that every exit s is a sink state, that is, $P(s,a)$ is undefined for any $a \in A$. We can naturally lift the definitions of schedulers and weighted reachability probabilities from MDPs to oMDPs: we will be particularly interested in the following instances; 1) the *weighted reachability probability* $\mathrm{WRPr}^{\mathcal{A},\sigma}(\mathbf{w},i) := \mathrm{WRPr}^{\mathcal{A},\sigma,O^{\mathcal{A}}}(\mathbf{w},i)$ from a chosen entrance i to the set $O^{\mathcal{A}}$ of all exits; and 2) the *maximum weighted reachability probability* $\mathrm{WRPr}^{\mathcal{A}}_{\max}(\mathbf{w},i) := \sup_{\sigma} \mathrm{WRPr}^{\mathcal{A},\sigma}(\mathbf{w},i)$ from i to $O^{\mathcal{A}}$ weighted by $\mathbf{w}$.

We define *string diagrams of MDPs* [53] syntactically, as syntactic trees whose leaves are oMDPs and non-leaf nodes are algebraic operations. The latter are syntactic operations and they are yet to be interpreted.

Definition 3.4 (string diagram of MDPs). A *string diagram* $\mathbb{D}$ *of MDPs* is a term adhering to the grammar $\mathbb{D} ::= \mathsf{c}_{\mathcal{A}} \mid \mathbb{D} \mathbin{;} \mathbb{D} \mid \mathbb{D} \oplus \mathbb{D}$, where $\mathsf{c}_{\mathcal{A}}$ is a constant designating an oMDP $\mathcal{A}$.

The above syntactic operations $;, \oplus$ are interpreted by the *semantic* operations below. The following definitions explicate the graphical intuition in Fig. 2.

Definition 3.5 (sequential composition $;$). Let $\mathcal{A}$, $\mathcal{B}$ be oMDPs, $\mathrm{arity}(\mathcal{A}) = (m_{\mathbf{r}}, m_{\mathbf{l}}) \to (l_{\mathbf{r}}, l_{\mathbf{l}})$, and $\mathrm{arity}(\mathcal{B}) = (l_{\mathbf{r}}, l_{\mathbf{l}}) \to (n_{\mathbf{r}}, n_{\mathbf{l}})$. Their *sequential composition* $\mathcal{A} \mathbin{;} \mathcal{B}$ is the oMDP (M, IO') where $\mathsf{IO}' = (I^{\mathcal{A}}_{\mathbf{r}}, I^{\mathcal{B}}_{\mathbf{l}}, O^{\mathcal{B}}_{\mathbf{r}}, O^{\mathcal{A}}_{\mathbf{l}})$, $M := \big((S^{\mathcal{A}} \uplus S^{\mathcal{B}}) \setminus (O^{\mathcal{A}}_{\mathbf{r}} \uplus O^{\mathcal{B}}_{\mathbf{l}}), A^{\mathcal{A}} \uplus A^{\mathcal{B}}, P\big)$ and P is

$$P(s,a,s') := \begin{cases} P^{\mathcal{D}}(s,a,s') & \text{if } \mathcal{D} \in \{\mathcal{A},\mathcal{B}\}\ ,\ s \in S^{\mathcal{D}}\ ,\ a \in A^{\mathcal{D}}\ ,\ \text{and } s' \in S^{\mathcal{D}}, \\ P^{\mathcal{A}}(s,a,o^{\mathcal{A}}_{\mathbf{r},i}) & \text{if } s \in S^{\mathcal{A}}\ ,\ a \in A^{\mathcal{A}}\ ,\ s' = i^{\mathcal{B}}_{\mathbf{r},i} \text{ for some } 1 \le i \le l_{\mathbf{r}}\ , \\ P^{\mathcal{B}}(s,a,o^{\mathcal{B}}_{\mathbf{l},i}) & \text{if } s \in S^{\mathcal{B}}\ ,\ a \in A^{\mathcal{B}}\ ,\ s' = i^{\mathcal{A}}_{\mathbf{l},i} \text{ for some } 1 \le i \le l_{\mathbf{l}}, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 3.6 (sum $\oplus$). Let $\mathcal{A}, \mathcal{B}$ be oMDPs. Their *sum* $\mathcal{A} \oplus \mathcal{B}$ is the oMDP (M, IO') where $\mathsf{IO}' = (I^{\mathcal{A}}_{\mathbf{r}} \uplus I^{\mathcal{B}}_{\mathbf{r}}, I^{\mathcal{A}}_{\mathbf{l}} \uplus I^{\mathcal{B}}_{\mathbf{l}}, O^{\mathcal{A}}_{\mathbf{r}} \uplus O^{\mathcal{B}}_{\mathbf{r}}, O^{\mathcal{A}}_{\mathbf{l}} \uplus O^{\mathcal{B}}_{\mathbf{l}})$, $M = (S^{\mathcal{A}} \uplus S^{\mathcal{B}}, A^{\mathcal{A}} \uplus A^{\mathcal{B}}, P)$, and P is given by $P(s,a,s') := P^{\mathcal{D}}(s,a,s')$ if $\mathcal{D} \in \{\mathcal{A},\mathcal{B}\}$, $s \in S^{\mathcal{D}}$, $a \in A^{\mathcal{D}}$, and $s' \in S^{\mathcal{D}}$, and otherwise $P(s,a,s') := 0$.

Definition 3.7 (operational semantics $[\![\mathbb{D}]\!]$). Let $\mathbb{D}$ be a string diagram of MDPs. The *operational semantics* $[\![\mathbb{D}]\!]$ is the oMDP which is inductively defined by Defs 3.5 and 3.6, with the base case $[\![\mathsf{c}_{\mathcal{A}}]\!] = \mathcal{A}$. Here we assume that every string diagram $\mathbb{D}$ has matching arities so that compositions are well-defined. We call $I^{[\![\mathbb{D}]\!]}$ and $O^{[\![\mathbb{D}]\!]}$ *global entrances* and *global exits* of $\mathbb{D}$, respectively.

For describing the occurrence of oMDPs and their duplicates in a string diagram $\mathbb{D}$, we formally define *nominal components* $\mathrm{nCP}(\mathbb{D})$ and *components* $\mathrm{CP}(\mathbb{D})$. The latter for graph-theoretic operations in our compositional VI (CVI) (Algorithm 1), while the former is for Pareto caching (Sect. 5.2). Examples are provided later in Ex. 3.10.

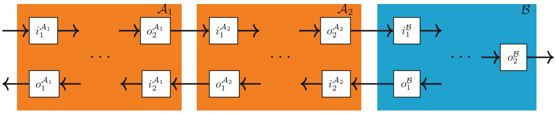


Fig. 10. $[\![\mathbb{D}]\!]$ in Ex. 3.10

Definition 3.8 ($\mathrm{nCP}(\mathbb{D})$, $\mathrm{CP}(\mathbb{D})$). The set $\mathrm{nCP}(\mathbb{D})$ of *nominal components* is the set of constants occurring in $\mathbb{D}$ (as a term). The set $\mathrm{CP}(\mathbb{D})$ of *components* is inductively defined by the following: $\mathrm{CP}(\mathsf{c}_{\mathcal{A}}) := \{\mathcal{A}\}$, and $\mathrm{CP}(\mathbb{E} * \mathbb{F}) := \mathrm{CP}(\mathbb{E}) \uplus \mathrm{CP}(\mathbb{F})$ for $* \in \{\fatsemi, \oplus\}$; here we count multiplicities, unlike $\mathrm{nCP}(\mathbb{D})$.

We introduce *local open ends* of string diagrams, in contrast to global open ends defined in Def. 3.7.

Definition 3.9 ($I_{\mathrm{lc}}(\mathbb{D})$, $O_{\mathrm{lc}}(\mathbb{D})$ (local)). The sets $I_{\mathrm{lc}}(\mathbb{D})$ and $O_{\mathrm{lc}}(\mathbb{D})$ of *local entrances* and *exits* of $\mathbb{D}$ are given by $I_{\mathrm{lc}}(\mathbb{D}) := \biguplus_{\mathcal{A} \in \mathrm{CP}(\mathbb{D})} I^{\mathcal{A}}$, and $O_{\mathrm{lc}}(\mathbb{D}) := \biguplus_{\mathcal{A} \in \mathrm{CP}(\mathbb{D})} O^{\mathcal{A}}$, respectively. Clearly we have $I^{[\![\mathbb{D}]\!]} \subseteq I_{\mathrm{lc}}(\mathbb{D})$, $O^{[\![\mathbb{D}]\!]} \subseteq O_{\mathrm{lc}}(\mathbb{D})$.

Example 3.10 Let $\mathbb{D} = \mathsf{c}_{\mathcal{A}} \fatsemi \mathsf{c}_{\mathcal{A}} \fatsemi \mathsf{c}_{\mathcal{B}}$, where $\mathcal{A}$ and $\mathcal{B}$ are from Fig. 1. The oMDP $[\![\mathbb{D}]\!]$ is shown in Fig. 10. Then $\mathrm{nCP}(\mathbb{D}) = \{\mathsf{c}_{\mathcal{A}}, \mathsf{c}_{\mathcal{B}}\}$, while $\mathrm{CP}(\mathbb{D}) = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{B}\}$ with subscripts added for distinction. We have $I^{[\![\mathbb{D}]\!]} = \{i_1^{\mathcal{A}_1}\}$ and $O^{[\![\mathbb{D}]\!]} = \{o_1^{\mathcal{A}_1}, o_2^{\mathcal{B}}\}$, and $I_{\mathrm{lc}}(\mathbb{D}) = \{i_1^{\mathcal{A}_1}, i_2^{\mathcal{A}_1}, i_1^{\mathcal{A}_2}, i_2^{\mathcal{A}_2}, i_1^{\mathcal{B}}\}$ and $O_{\mathrm{lc}}(\mathbb{D}) = \{o_1^{\mathcal{A}_1}, o_2^{\mathcal{A}_1}, o_1^{\mathcal{A}_2}, o_2^{\mathcal{A}_2}, o_1^{\mathcal{B}}, o_2^{\mathcal{B}}\}$. Note also that $O_{\mathrm{lc}}(\mathbb{D})$ does *not* suppress exits removed in sequential composition, such as $\{o_2^{\mathcal{A}_1}, o_1^{\mathcal{A}_2}, o_2^{\mathcal{A}_2}, o_1^{\mathcal{B}}\}$.

Problem: Near-Optimal Weighted Reachability Probability
Given a string diagram $\mathbb{D}$, an entrance $i \in I^{[\![\mathbb{D}]\!]}$, a weight $\mathbf{w} \in [0,1]^{O^{[\![\mathbb{D}]\!]}}$ over exits, and an error bound $\epsilon \in [0,1]$, compute an under-approximation $l \in [0,1]$ such that $l \leq \mathrm{WRPr}^{[\![\mathbb{D}]\!]}_{\max}(\mathbf{w}, i) \leq l + \epsilon$.

We remark that as a straightforward extension, we can also extract a scheduler that achieves the under-approximation.

4 VI in a Compositional Setting

We recap *value iteration (VI)* [3,44] and its extension to *optimistic value iteration (OVI)* [30] before presenting our *compositional VI (CVI)*.

4.1 Value Iteration (VI) and Optimistic Value Iteration (OVI)

VI relies on the characterization of maximum reachability probabilities as a *least fixed point (lfp)*, specifically the lfp $\mu\Phi_{\mathcal{M},T}$ of the *Bellman operator* $\Phi_{\mathcal{M},T}$: the Bellman operator $\Phi_{\mathcal{M},T}$ is an operator on the set $[0,1]^S$ that intuitively returns the $t+1$-step reachability probabilities given the t-step reachability probabilities. A formal treatment can be found in [55, Appendix B]. Then the *Kleene sequence* $\bot \leq \Phi_{\mathcal{M},T}(\bot) \leq \Phi^2_{\mathcal{M},T}(\bot) \leq \cdots$ gives a monotonically increasing sequence that converges to the lfp $\mu\Phi_{\mathcal{M},T}$, where $\bot$ is the least element. This also applies to weighted reachability probabilities.

While VI gives guaranteed under-approximations, it does not say *how close* the current approximation is to the solution $\mu\Phi_{\mathcal{M},T}$[1]. The capability of providing

[1] The challenge applies to VI in (our) undiscounted setting, where the Bellman operator is not a contraction operator. With discounting, one can easily approximate the gap.

guaranteed over-approximations as well is called *soundness* in VI, and many techniques come with soundness [24,30,43,46]. Soundness is useful for *stopping criteria*: one can fix an *error bound* $\eta \in [0,1]$; VI can terminate when the distance between under- and over-approximations is at most η.

Among sound VI techniques, in this paper we focus on optimistic VI (OVI) due to its proven performance record [11,29]. We use OVI in many places, specifically for 1) stopping criteria for local VIs in Sect. 4.2, 2) caching heuristics in Sect. 5.2, and 3) a stopping criterion for global (compositional) VI in Sect. 6.

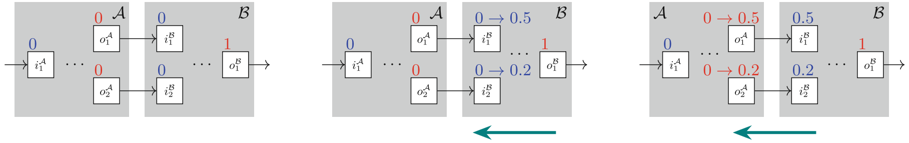
The main steps of OVI proceed as follows: 1) a VI iteration produces an under-approximation l for every state; 2) we heuristically pick an over-approximation candidate u, for example by $u := l + \epsilon$; and 3) we verify the candidate u by checking if $\Phi_{\mathcal{M},T}(u) \leq u$. If the last holds, then by the *Park induction principle* [42], u is guaranteed to over-approximate the lfp $\mu\Phi_{\mathcal{M},T}$. If it does not, then we refine l, u and try again. See [30] for details.

4.2 Going Top-Down in Compositional Value Iteration

We move on to formalize the story of Sect. 2.2. Algorithm 1 is a prototype of our proposed algorithm, where compositional VI is run in a top-down manner. It

Algorithm 1. A prototype of compositional value iteration (CVI)

```
    Input: a string diagram $\mathbb{D}$ of MDPs and a weight $\mathbf{w} \in [0,1]^{O^{[\![\mathbb{D}]\!]}}$, as in the target problem.
    Output: a function $f\colon I^{[\![\mathbb{D}]\!]} \to [0,1]$.
 1: initialize $g\colon I_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$ as the least element $\bot$ (i.e. everywhere 0)
 2: initialize $h\colon O_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$ as everywhere 0, except for global exits $o$ where $h(o) := w_o$ (depending on the weight $\mathbf{w} = (w_o)_o$)
 4: while not GlobalStoppingCriterion($g$) do
 5:     for each $\mathcal{A} \in \mathrm{CP}(\mathbb{D})$ do        ▷ for each component $\mathcal{A}$, counting multiplicities
 6:         $g_{\mathcal{A}} \leftarrow$ LocalVI($\mathcal{A}, h|_{O^{\mathcal{A}}}$)   ▷ run VI locally in $\mathcal{A}$ and obtain $g_{\mathcal{A}}\colon I^{\mathcal{A}} \to [0,1]$
11:     $g \leftarrow \coprod_{\mathcal{A}\in\mathrm{CP}(\mathbb{D})} g_{\mathcal{A}}$          ▷ $g\colon I_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$ is obtained by patching $(g_{\mathcal{A}})_{\mathcal{A}}$
12:     $h \leftarrow$ PropagateSeqComp($g, \mathbf{w}$)
                  ▷ $g$'s update is propagated to $h$ along sequential composition, see Fig. 11c
13: return $g|_{I^{[\![\mathbb{D}]\!]}}$                  ▷ restrict $g\colon I_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$ along $I^{[\![\mathbb{D}]\!]} \subseteq I_{\mathrm{lc}}(\mathbb{D})$
```



(a) Initialization, lines 1–2.

(b) Local VI, line 6.

(c) Propag. along ⨟, line 12.

Fig. 11. an overview of Algorithm 1. In the MDP $[\![\mathbb{D}]\!]$, the exit $o_1^{\mathcal{A}}$ and the entrance $i_1^{\mathcal{B}}$ get merged in $\mathcal{A} \fatsemi \mathcal{B}$ (Def. 3.5); here they are distinguished, much like in Def. 3.9. Numbers in red are the values of h; those in blue are the values of g.

will be combined with Pareto caching (Sect. 5.2) and the stopping criteria introduced in Sect. 6. A high-level view of Algorithm 1 is the iteration of the following operations: 1) running local VI in each component oMDP, and 2) propagating its result along sequential composition, from an entrance of a succeeding component, to the corresponding exit of a preceding component. See Fig. 11 for illustration. The algorithm maintains two main constructs: functions $g\colon I_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$ and $h\colon O_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$ that assign values to local entrances and exits, respectively. They are analogues of the value function $f\colon S \to [0,1]$ in (standard) VI (Sect. 4.1); g and h get iteratively increased as the algorithm proceeds.

Lines 4–12 are the main VI loop, where we combine local VI (over each component $\mathcal{A}$) and propagation along sequential composition. The algorithm `LocalVI` takes the target oMDP $\mathcal{A}$ and its "local weight" as arguments; the latter is the restriction $h|_{O^{\mathcal{A}}}\colon O^{\mathcal{A}} \to [0,1]$ of the function $h\colon O_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$. Any VI algorithm will do for `LocalVI`; we use OVI as announced in Sect. 4.1. The result of local VI is a function $g_{\mathcal{A}}\colon I^{\mathcal{A}} \to [0,1]$ for values over entrances of $\mathcal{A}$. These get patched up to form $g\colon I_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$ in line 11. The function $\coprod_{\mathcal{A}\in\mathrm{CP}(\mathbb{D})} g_{\mathcal{A}}$ is defined by obvious case-distinction: it returns $g_{\mathcal{A}}(i)$ for a local entrance $i \in I^{\mathcal{A}}$. Recall from Def. 3.9 that $I_{\mathrm{lc}}(\mathbb{D}) = \biguplus_{\mathcal{A}\in\mathrm{CP}(\mathbb{D})} I^{\mathcal{A}}$. In line 12, the values at entrances are propagated to the connected exits.

On `PropagateSeqComp` in line 12, its graphical intuition is in Fig. 11c; here are some details. We first note that the set $O_{\mathrm{lc}}(\mathbb{D})$ of local exits is partitioned into 1) global exits (i.e. those in $O^{[\![\mathbb{D}]\!]}$) and 2) those local exits that get removed by sequential composition. Indeed, by examining Defs 3.5 and 3.6, we see that sequential composition $\fatsemi$ is the only operation that removes local exits, and the local exits that are not removed eventually become global exits. It is also obvious (Def. 3.5) that each local exit o removed in sequential composition has a corresponding local entrance i_o. Using these, we define the function $h :=$ `PropagateSeqComp`$(g, \mathbf{w})$, of the type $O_{\mathrm{lc}}(\mathbb{D}) \to [0,1]$, as follows: $h(o) = w_o$ if o is a global exit (much like line 2); $h(o) = g(i_o)$ otherwise.

Theorem 4.1 *Algorithm 1 satisfies the following properties:*

1. *(Guaranteed under-approximation) For the output f of Algorithm 1, we have $f(i) \le \mathrm{WRPr}^{[\![\mathbb{D}]\!]}_{\max}(\mathbf{w}, i)$ for each $i \in I^{[\![\mathbb{D}]\!]}$.*
2. *(Convergence) Assume that `GlobalStoppingCriterion` is **false**. Algorithm 1 converges to the optimal value, that is, f converges to $\mathrm{WRPr}^{[\![\mathbb{D}]\!]}_{\max}(\mathbf{w}, I^{[\![\mathbb{D}]\!]})$.* □

The correctness of the under-approximation of Algorithm 1 follows easily from those of (non-compositional, asynchronous) VI. The convergence depends on the fact that line 6 of Algorithm 1 iterates over *all* components.

5 Pareto Caching in Compositional VI

In our formulation of Algorithm 1, there is no explicit notion of Pareto curves. However, in line 6, we do (implicitly) compute under-approximations on points on the Pareto curves. Here we recap approximate Pareto curves. We then show how we conduct *Pareto caching*, the key idea sketched in Sect. 2.3.

5.1 Approximating Pareto Curves

We formalize the Pareto curves illustrated in Sect. 2. For details, see [17,20,41,45]. Model checking oMDPs is a multi-objective problem, that determines different trade-offs between reachability probabilities for the individual exits.

Definition 5.1 (Pareto curve for an oMDP [54]). Let $\mathcal{A}$ be an oMDP, and i be a (chosen) entrance. Let $\mathbf{p}, \mathbf{p}' \in [0,1]^{O^{\mathcal{A}}}$. The relation $\preceq$ between them is defined by $\mathbf{p} \preceq \mathbf{p}'$ if $\mathbf{p}(o) \leq \mathbf{p}'(o)$ for each $o \in O^{\mathcal{A}}$. When $\mathbf{p} \prec \mathbf{p}'$ (i.e. $\mathbf{p} \preceq \mathbf{p}'$ and $\mathbf{p} \neq \mathbf{p}'$), we say $\mathbf{p}'$ *dominates* $\mathbf{p}$. Let σ be a scheduler for $\mathcal{A}$. We define the point *realized by* σ, denoted by $\mathbf{p}_i^{\sigma}$, by $\mathbf{p}_i^{\sigma}(o) := \mathrm{RPr}^{\mathcal{A},\sigma,o}(i)$, the reachability probability from i to o under σ.

The set Ach_i^{σ} of points *achievable* by σ is $\mathsf{Ach}_i^{\sigma} := \{\mathbf{p} \mid \mathbf{p} \preceq \mathbf{p}_i^{\sigma}\}$. The set Ach_i of *achievable points* is given by $\mathsf{Ach}_i := \bigcup_{\sigma \in \Sigma^{\mathcal{A}}} \mathsf{Ach}_i^{\sigma}$. The *Pareto curve* $\mathsf{Pareto}_i \subseteq [0,1]^{O^{\mathcal{A}}}$ is the set of maximal elements in Ach_i wrt. $\preceq$. We say a scheduler σ is *Pareto-optimal* if $\mathbf{p}_i^{\sigma} \in \mathsf{Pareto}_i$.

The set $\mathsf{Ach}_i \subseteq [0,1]^{O^{\mathcal{A}}}$ is convex, downward closed, and finitely generated by DM schedulers; it follows that, for our target problem, Pareto-optimal DM schedulers suffice. This is illustrated in Fig. 9, where a weight $\mathbf{w}$ is the normal vector of blue lines, and the maximum is achieved by a generating point for Ach_i.

The last observations are formally stated as follows.

Proposition 5.2 ([17,20,45]). *For any entrance $i \in I$, the set Ach_i of achievable points is finitely generated by DM schedulers, that is, $\mathsf{Ach}_i = \mathsf{DwConvCl}(\mathsf{Ach}_i^{\Sigma_d^{\mathcal{A}}})$. Here, $\mathsf{DwConvCl}(X)$ denotes the downward and convex closed set generated by $X \subseteq \mathbb{R}^n$, and $\mathsf{Ach}_i^{\Sigma_d^{\mathcal{A}}}$ is given by $\mathsf{Ach}_i^{\Sigma_d^{\mathcal{A}}} := \bigcup_{\sigma \in \Sigma_d^{\mathcal{A}}} \mathsf{Ach}_i^{\sigma}$, where $\Sigma_d^{\mathcal{A}}$ is the set of DM schedulers.*

Proposition 5.3 ([17,20,45]). *Given a weight $\mathbf{w} \in [0,1]^{O^{\mathcal{A}}}$ and an entrance i, there is a scheduler σ such that $\mathrm{WRPr}^{[\![\mathbb{D}]\!],\sigma}(\mathbf{w}, i) = \mathrm{WRPr}^{[\![\mathbb{D}]\!]}_{\max}(\mathbf{w}, i)$. Moreover, this σ can be chosen to be DM and Pareto-optimal.*

We now formulate *sound approximations* of Pareto curves, which is a foundation of our Pareto caching (and a global stopping criterion in Sect. 6).

Definition 5.4 (sound approximation [54]). Let i be an entrance. An *under-approximation* L_i of the Pareto curve Pareto_i is a downward closed subset $L_i \subseteq \mathsf{Ach}_i$; an *over-approximation* is a downward closed superset $U_i \supseteq \mathsf{Ach}_i$. A pair (L_i, U_i) is called a *sound approximation* of the Pareto curve Pareto_i. In this paper, we focus on L_i and U_i that are *finitely generated*, i.e. the convex and downward closures of some finite generators $L_i^{\mathrm{g}}, U_i^{\mathrm{g}} \subseteq [0,1]^{O^{\mathcal{A}}}$, respectively. A *sound approximation* of an oMDP $\mathcal{A}$ is a pair (L, U), where $L = (L_i)_{i \in I^{\mathcal{A}}}$, $U = (U_i)_{i \in I^{\mathcal{A}}}$, and (L_i, U_i) is a sound approximation for each entrance i.

5.2 Pareto Caching

We go on to formalize our second key idea, *Pareto caching*, outlined in Sect. 2.3.

In Def. 5.5, an index $\mathsf{c}_{\mathcal{A}} \in \mathrm{nCP}(\mathbb{D})$ is a *nominal* component that ignores multiplicities, since we want to reuse results for different occurrences of $\mathcal{A}$.

Definition 5.5 (Pareto cache). Let $\mathbb{D}$ be a string diagram of MDPs. A *Pareto cache* $\mathbf{C}$ is an indexed family $\mathbf{C} := \big((L^{\mathcal{A}}, U^{\mathcal{A}})\big)_{\mathsf{c}_{\mathcal{A}} \in \mathrm{nCP}(\mathbb{D})}$, where $(L^{\mathcal{A}}, U^{\mathcal{A}})$ is a sound approximation for each nominal component $\mathsf{c}_{\mathcal{A}}$, defined in Def. 5.4.

As announced in Sect. 2.3, a Pareto cache $\mathbf{C}$—its component $(L^{\mathcal{A}}, U^{\mathcal{A}})$, to be precise—gets *queried* on a weight $\mathbf{w} \in [0,1]^{O^{\mathcal{A}}}$. It is not trivial what to return, however, since the specific weight $\mathbf{w}$ may not have been used before to construct $\mathbf{C}$. The query is answered in the way depicted in Fig. 9, finding an extremal point where $L^{\mathcal{A}}$ intersects with a plane with its normal vector $\mathbf{w}$.

Definition 5.6 (cache read). Assume the above setting, and let i be an entrance of interest. The *cache read* $(L_i^{\mathcal{A}}(\mathbf{w}), U_i^{\mathcal{A}}(\mathbf{w})) \in [0,1]^2$ on $\mathbf{w}$ at i is defined by $L_i^{\mathcal{A}}(\mathbf{w}) := \sup_{\mathbf{p} \in L_i} \mathbf{w} \cdot \mathbf{p}$ and $U_i^{\mathcal{A}}(\mathbf{w}) := \sup_{\mathbf{p} \in U_i} \mathbf{w} \cdot \mathbf{p}$.

Recall from Sect. 5.1 that we can assume L_i and U_i are finitely generated as convex and downward closures. It follows [20,45] that each supremum above is realized by some generating point, much like in Prop. 5.3, easing computation.

We complement Algorithm 1 by Algorithm 2 that introduces our Pareto caching. Specifically, for the weight $h|_{O^{\mathcal{A}}}$ in question, we first compute the *error* $\max_{i \in I^{\mathcal{A}}} U_i^{\mathcal{A}}(h|_{O^{\mathcal{A}}}) - L_i^{\mathcal{A}}(h|_{O^{\mathcal{A}}})$ of the Pareto cache $\mathbf{C} = \big((L^{\mathcal{A}}, U^{\mathcal{A}})\big)_{\mathsf{c}_{\mathcal{A}}}$ with respect to this weight. The error can be greater than a prescribed bound η—we call this *cache miss*—in which case we run OVI locally for $\mathcal{A}$ (line 9). When the error is no greater than η—we call this *cache hit*—we use the cache read (Def. 5.6), sparing OVI on a component $\mathcal{A} \in \mathrm{CP}(\mathbb{D})$. In the case of a cache miss, the result (l, σ) of local OVI (line 9) is used also to update the Pareto cache $\mathbf{C}$ (line 10); see below.

Using a Pareto cache may prevent the execution of local VI on every component, which can be critical for the convergence of Algorithm 1; see Thm. 4.1. A simple solution is to disregard Pareto caches eventually.

Algorithm 2. Updating $g_{\mathcal{A}}$ with a Pareto cache $\mathbf{C}$ and a bound $\eta \in [0,1]$

```
3: initialize a Pareto cache $\mathbf{C}$ by $((\emptyset, [0,1]^{O^{\mathcal{A}}}))_{\mathsf{c}_{\mathcal{A}}}$

6:          if $\max_{i \in I^{\mathcal{A}}} \big(U_i^{\mathcal{A}}(h|_{O^{\mathcal{A}}}) - L_i^{\mathcal{A}}(h|_{O^{\mathcal{A}}})\big) \le \eta$ then
                      ▷ computing the error of cache $\mathbf{C}$ for weight $h|_{O^{\mathcal{A}}}$ (cf. Def. 5.6)
7:              $g_{\mathcal{A}} \leftarrow L^{\mathcal{A}}(h|_{O^{\mathcal{A}}})$          ▷ "cache hit"; use the cache read $L^{\mathcal{A}}(h|_{O^{\mathcal{A}}})$
8:          else
9:              $(l, \sigma) \leftarrow$ LocalOVI$(\mathcal{A}, h|_{O^{\mathcal{A}}}, \eta)$
                                  ▷ "cache miss"; run local VI as in Alg. 1
10:             $g_{\mathcal{A}} \leftarrow l$ and $\mathbf{C} \leftarrow$ Update$(\mathcal{A}, \mathbf{C}, l, \sigma, h|_{O^{\mathcal{A}}}, \eta)$     ▷ see the end of §5.2
```

Updating the Cache. Pareto caches get incrementally updated using the results for weighted reachabilities with different weights $\mathbf{w}$. We build upon data structures in [20,45]. Notable is the asymmetry between under- and over-approximations (L_i, U_i): we obtain 1) a *point in* L_i and 2) a *plane that bounds* U_i.

We update the cache after running OVI on a weight $\mathbf{w} \in [0,1]^{O^{\mathcal{A}}}$, which approximately computes the optimal weighted reachability to exits $o \in O^{\mathcal{A}}$. That is, it returns $l, u \in [0,1]$ such that

$$l \;\le\; \sup_{\sigma}\big(\mathbf{w} \cdot \big(\mathrm{RPr}^{\sigma}(i \to o)\big)_{o \in O^{\mathcal{A}}}\big) \;\le\; u. \tag{4}$$

Here i is any entrance and $\mathrm{RPr}^{\sigma}(i \to o)$ is the probability $\mathrm{RPr}^{\mathcal{A},\sigma,\{o\}}(i)$ in Sect. 3.1.

What are the "graphical" roles of l, u in the Pareto curve? The role of u is easier: it follows from (4) that any achievable reachability vector $\big(\mathrm{RPr}^{\sigma}(i \to o)\big)_o$ resides under the plane $\{\mathbf{p} \mid \mathbf{w} \cdot \mathbf{p} = u\}$. This plane thus bounds an over-approximation U_i. The use of l takes some computation. By (4), the existence of a good scheduler σ is guaranteed; but this alone does not carry any graphical information e.g. in Fig. 9. We have to go constructive, by extracting a near-optimal DM scheduler σ_0 (we can do so in VI) and using this fixed σ_0 to compute $\big(\mathrm{RPr}^{\sigma_0}(i \to o)\big)_o$. This way we can plot an achievable point—a corner point in Fig. 9—in L_i.

6 Global Stopping Criteria (GSC)

We present the last missing piece, namely global stopping criteria (*GSC* in short, in line 4 of Algorithm 1). It has to ensure that the computed underapproximation f is ϵ close to the exact reachability probability. We provide two criteria, called *optimistic* and *bottom-up*.

Optimistic GSC (Opt-GSC). The challenge in adapting the idea of OVI (see Sect. 4.1) to CVI is to define a suitable Bellman operator for CVI. Once we define such a Bellman operator for CVI, we can immediately apply the idea of OVI. For simplicity, we assume that CVI solves exactly in each local component (line 6 in Algorithm 1) without Pareto caching; this can be done, for example, by policy iteration [29]. Then, CVI (without Pareto caching and a global stopping criterion) on $\mathbb{D}$ is exactly the same as the (non-compositional) VI on a suitable *shortcut MDP* [54] of $\mathbb{D}$. Intuitively, a shortcut MDP summarizes a Pareto-optimal scheduler by a single action from a local entrance to exit, see [55, Appendix C] for the definition. Thus, we can regard the standard Bellman operator on the shortcut MDP as the Bellman operator for CVI, and define Opt-GSC as the standard OVI based on this characterisation. CVI with Opt-GSC (and Pareto caching) actually uses local under-approximations (not exact solutions) for obtaining a global under-approximation (line 7 in Algorithm 2 and line 9 in Algorithm 2), where the desired soundness property still holds. See [55, Appendix C] for more details.

Bottom-up GSC (BU-GSC). We obtain another global stopping criterion by composing Pareto caches—computed in Algorithm 2 for each component $\mathcal{A}$—in the bottom-up manner in [54] (outlined in Sect. 2.1). Specifically, 1) Algorithm 2 produces an over-approximation $U^{\mathcal{A}}$ for the Pareto curve of each component $\mathcal{A}$; 2) we combine $(U^{\mathcal{A}})_{\mathcal{A}}$ along $⨟$ and $\oplus$ to derive an over-approximation U of the global Pareto curve; and 3) this U is queried on the weight $\mathbf{w}$ in question (i.e. the input of CVI), in order to obtain an over-approximation u of the weighted reachability probabilities. BU-GSC checks if this over-approximation u is close enough to the under-approximation l derived from g in Algorithm 1.

Correctness. CVI (Algorithm 1 with Pareto caching under either GSC) is sound. The proof is in [55, Appendix C].

Theorem 6.1 (ϵ-soundness of CVI). *Given a string diagram $\mathbb{D}$, a weight $\mathbf{w}$, and $\epsilon \in [0, 1]$, if CVI terminates, then the output f satisfies*

$$f(i) \leq \mathrm{WRPr}^{[\![\mathbb{D}]\!]}_{\max}(\mathbf{w}, I^{[\![\mathbb{D}]\!]}) \leq f(i) + \epsilon,$$

for each $i \in I^{[\![\mathbb{D}]\!]}$.

Our algorithm currently comes with no termination guarantee; this is future work. Termination of VI (with soundness) is a tricky problem: most known termination proofs exploit the uniqueness of a fixed point of the Bellman operator, which must be algorithmically enforced e.g. by eliminating end components [10,24]. In the current compositional setting, end components can arise by composing components, so detecting them is much more challenging.

7 Empirical Evaluation

In this section, we compare the scalability of our approaches both among each other and in comparison with some existing baselines. We discuss the setup, give the results, and then give our interpretation of them.

Approaches. We examine our three main algorithms. Opt-GSC with either exact caching (OCVI^e) or Pareto-caching (OCVI^p), and BU-GSC with Pareto-caching (Symb). BU-GSC needs a Pareto cache, so we cannot run BU-GSC with an exact cache. We compare our approaches against two baselines: a *monolithic* (Mono) algorithm building the complete MDP $[\![\mathbb{D}]\!]$ and the *bottom-up* (BU) as explained in [54]. We use two *virtual* approaches that use a perfect oracle to select the fastest out of the specified algorithms: *baselines* is the best-of-the-baselines, while *novel* is the best of the three new algorithms. All algorithms are built on top of the probabilistic model checker Storm [32], which is primarily used for model building and (O)VI on component MDPs as well as operating on Pareto curves.

Setup. We run all benchmarks on a single core of an AMD Ryzen TRP 5965WX, with a 900 s time-out and a 16GB memory limit. We use *all* (scalable) benchmark instances from [54]. While these benchmarks are synthetic, they reflect typical structures found in network protocols and high-level planning domains. We require an overall precision of 10^{-4}, we run the Pareto cache with an acceptance precision of 10^{-5}, and solve the LPs in the upper-bound queries for the Pareto cache with an exact LP solver and a tolerance of 10^{-4}. The components are reverse topologically ordered, i.e., we always first analyse component MDPs towards the end of a given MDP $[\![\mathbb{D}]\!]$. To solve the component MDPs inside the VI, we use OVI for the lower bounds and precise policy iteration for the upper bounds. We use algorithms and data structures already present in Storm for maintaining Pareto curves [45], which use exact rational arithmetic for numerical stability. Although our implementation supports exact arithmetic throughout the code, in practice this leads to a significant performance penalty, performing up to 100 times slower. For algorithms not related to maintaining the Pareto cache, we opted for using 64-bit floating point arithmetic, which is standard in probabilistic model checking [11]. Using floating point arithmetic can produce unsound results [26]; we attempt to prevent unsound results in our benchmark. First, we check with our setup that our results are very close (error < 10^{-5}) to the exact solutions (when they could be computed). Second, we check that all results, obtained with different methods, are close. We evaluate the stopping criteria after ten iterations. These choices can be adapted using our prototypical implementation, we discuss some of these choices at the end of the discussion below.

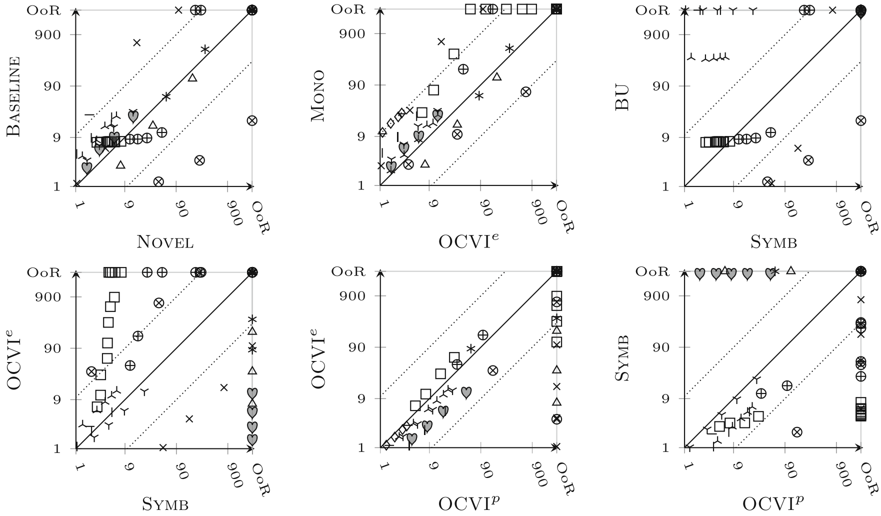


Fig. 12. Benchmark scatter plots, time in seconds, OoR=Out of Resources

Table 1. Performance for different algorithms. See **Results** for explanations.

				Baseline			Novel							
				Mono	BU		OCVI^e		OCVI^p			Symb		
$\mathbb{D}$	$\mathcal{M}$	$\lvert S\rvert$	$\lvert L\rvert$	t	t	$\lvert P\rvert$	t	t_s	t	t_s	$\lvert P\rvert$	t	t_s	$\lvert P\rvert$
`Birooms10`	`RmB`	1.1e+06	16	56	TO	TO	84	25	58	25	1155	TO	TO	TO
`Birooms100`	`RmS`	8.5e+05	16	15	TO	TO	32	8	TO	TO	TO	TO	TO	TO
`Birooms200`	`RmS`	3.4e+06	16	126	TO	TO	187	58	TO	TO	TO	TO	TO	TO
`Chains100`	`RmB`	1.1e+06	6	TO	7	46	57	26	27	24	40	4	0	44
`Chains3500`	`RmB`	3.7e+07	6	OOM	8	46	TO	TO	TO	TO	TO	8	1	40
`ChainsLoop500`	`Dice4`	4.9e+06	4	28	TO	TO	13	13	25	12	54	21	16	50
`ChainsLoop500`	`Dice5`	4.9e+06	4	25	TO	TO	13	13	47	20	66	TO	TO	TO
`Chains500`	`RmS`	4.2e+04	6	1	0	67	0	0	TO	TO	TO	0	0	38
`Rooms10`	`RmB`	1.1e+06	14	183	8	84	42	21	31	20	108	11	0	108
`Rooms250`	`RmB`	6.6e+08	14	OOM	OOM	OOM	TO	TO	TO	TO	TO	273	226	134
`Rooms500`	`RmS`	2.1e+07	13	TO	OOM	OOM	100	57	TO	TO	TO	TO	TO	TO

Results. We provide pairwise comparisons of the runtimes on all benchmarks using the scatter plots in Fig. 12[2]. Notice the log-log scale. For some of the benchmark instances, we provide detailed information in Tables 1 and 2, respectively. In Table 1, we give the identifier for the string diagram and the component MDPs, as well as the number of states in $[\![\mathbb{D}]\!]$. Then, for each of the five algorithms, we provide the timings in t, for each algorithm maintaining Pareto points, we give the number of Pareto points stored $|P|$, and for the three novel VI-based algorithms, we give the amount of time spent in an attempt to prove convergence (t_s). In Table 2, we focus on our three novel algorithms and the performance of the caches. We again provide identifiers for the models, and then for each algorithm, the total time spent by the algorithm, the time spent on inserting and retrieving items from the cache, as well as the fraction of cache hits H and the number of total queries Q. Thus, the number of cache hits is given by $H \cdot Q$. The full tables and more figures are given in [55, Appendix A].

Discussion. We make some observations. *We notice that the CVI algorithms collectively solve more benchmarks within the time out and speed up most benchmarks, see Fig. 12(top-l).*[3] We refer to benchmark results in Table 1.

OCVI^e Mostly Outperforms Mono, Fig. 12(top-c). The monolithic VI as typical in Storm requires a complete model, which can be prohibitively large. However, even for medium-sized models such as `Chains100-RmB`, the VI can run into time outs due to slow convergence. CVI with the exact cache (and even with no cache) quickly converges – highlighting that the grouping of states helps

[2] A point (x, y) means that the approach on the x-axis took x seconds and the tool on the y-axis took y seconds. Different shapes refer to different benchmark sets.

[3] We highlight that we use the benchmark suite that accompanied the bottom-up approach.

Table 2. Cache access times for CVI algorithms. See **Results** for explanations.

		OCVI^e					OCVI^p					Symb				
$\mathbb{D}$	$\mathcal{M}$	t	t_i	t_r	H	Q	t	t_i	t_r	H	Q	t	t_i	t_r	H	Q
Birooms10	RmB	84	0	0	0.03	206	58	14	4	0.66	1500	TO	TO	TO	TO	TO
Birooms100	RmS	32	0	0	0.24	65545	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Birooms200	RmS	187	1	3	0.25	477157	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Chains100	RmB	57	0	0	0.50	204	27	0	0	0.62	306	4	0	0	0.84	102
Chains3500	RmB	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	8	0	1	1.00	3502
ChainsLoop500	Dice4	13	0	0	0.84	15508	25	0	13	0.84	15592	21	0	5	1.00	5532
ChainsLoop500	Dice5	13	0	0	0.84	15511	47	0	26	0.84	15592	TO	TO	TO	TO	TO
Chains500	RmS	0	0	0	0.50	1004	TO	TO	TO	TO	TO	0	0	0	0.97	502
Rooms10	RmB	42	0	0	0.50	200	31	1	0	0.50	300	11	1	0	0.50	100
Rooms250	RmB	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	273	2	30	1.00	62499
Rooms500	RmS	100	0	0	0.50	500000	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

VI to converge. On the other hand, a model such as Birooms100-RmS highlights that the harder convergence check can yield a significant overhead.

Symb Mostly Outperforms BU, Fig. 12(top-r). For many models, the top-down approach as motivated in Sect. 4.2 indeed ensures that we avoid the undirected exploration of the Pareto curves. However, if the VI repeatedly asks for weights that are not relevant for the optimal scheduler, the termination checks fail and this yields a significant overhead.

OCVI^e and Symb Both Provide Clear Added Value, Fig. 12(bot-l). Both approaches can solve benchmarks within ten seconds that the other approach does not solve within the time-out. Both approaches are able to save significantly upon the number of iterations necessary. Symb suffers from the overhead of the Pareto cache, see below, whereas OCVI^e requires somewhat optimal values in all leaves, regardless of whether these leaves are important for reaching the global target. Therefore, Symb may profit from ideas from asynchronous VI and OCVI^e from adaptive schemes to decide when to run the termination check.

Pareto Cache Has a Significant Overhead, Fig. 12(bot-c/r) and Table 2. We observe that the Pareto cache consistently yields an overhead: In particular, OCVI^e often outperforms OCVI^p. The Pareto cache is essential for Symb. The overhead has three different sources. (1) *More iterations*: Birooms10-RmB illustrates how OCVI^e requires only 14% of the iterations of OCVI^p. Even with a 66% cache hit rate in OCVI^p, this means an overhead in the number of component MDPs analysed. The main reason is that reusing approximation can delay convergence[4]. (2) *Cache retrieval*: To obtain an upper bound, we must optimize over Pareto curves that contain tens of halfspaces, which are numerically not very stable. Therefore, Pareto curves in Storm are represented exactly. The linear program that must be solved is often equally slow[5] as actually solving the LP,

[4] Towards global convergence, we may eventually deactivate the cache.

[5] We use the Soplex LP solver [21] for exact LP solving, which is significantly faster than using, e.g., Z3. Soplex may return unknown, which we interpret as a cache miss.

especially for small MDPs. (3) *Cache insertion*: Cache insertion of lower bounds requires model checking Markov chains, as many as there are exits in the open MDPs. These times are pure overhead if this lower bound is never retrieved and can be substantial for large open MDPs.

Opportunities for Heuristics and Hyperparameters. We extensively studied variations of the presented algorithms. For example, a much higher tolerance in the Pareto cache can significantly speed up OCVI^p on the cost of not terminating on many benchmark cases and one can investigate a per-query strategy for retrieving and/or inserting cache results.

Interpretation of Results. Mono works well on models that fit into memory and exhibit little sharing of open MDPs. BU works well when the Pareto curves of the open MDPs can be accurately be approximated with few Pareto points, which, in practice, excludes open MDPs with more than 3 exits. CVI without caching and termination criteria resembles a basic kind of topological VI[6] on the monolithic MDP. CVI can thus improve upon topological VI either via the cache or via the alternative stopping criteria. Based on the experiments, we conjecture that

- the cache is efficient when the cost of performing a single reachability query is expensive — such as in the `Room10` model — while the cache hit rate is high.
- the symbiotic termination criterion (Symb) works well when some exits are not relevant for the global target, such as the `Chains3500` model, in which going backwards is not productive.
- the compositional OVI stopping criterion (OCVI^e/OCVI^p) works well when the likelihood of reaching all individual open MDPs is high, such as can be seen in the `ChainsLoop500-Dice4` model.

8 Related Work

We group our related work into variations of value iteration, compositional verification of MDPs, and multi-objective verification.

Value Iteration. Value iteration as standard analysis of MDPs [29] is widely studied. In the undiscounted, indefinite horizon case we study, value iteration requires an exponential number of iterations in theory, but in practice converges earlier. This motivates the search for sound termination criteria. Optimistic value iteration [30] is now widely adopted as the default approach [11,29]. To accelerate VI, various asynchronous variations have been suggested that prevent operating on the complete state space. In particular *topological VI* [2,15] and *(uni-directional) sequential VI* [25,28,36] aim to exploit an acyclic structure similar to what exists in uni-directional MDPs.

[6] Topological VI orders strongly connected components, whereas CVI uses the hierarchical structure. This can also lead to advantages.

Sequentially Composed MDPs. The exploitation of a compositional structure in MDPs is widely studied. In particular, the sequential composition in our paper is closely related to hierarchical compositions that capture how tasks are often composed of repetitive subtasks [5,6,23,31,35,48,50,51]. While we study a fully model-based approach, Jothimurugan et al. [33] provide a compositional reinforcement learning method whose sub-goals are induced by specifications. Neary et al. [40] update the learning goals based on the analysis of the component MDPs, but do not consider the possibility of reaching multiple exits. The widespread *option-framework* and variations such as *abstract VI* [34], aggregate policies [14,49] into additional actions to speed up convergence of value iterations and is often applied in model-free approaches. In the context of OVI, we must converge everywhere and the bottom-up stopping criterion is not easily lifted to a model-free setting.

Further Related Work. As a different type of compositional reasoning, *assume-guarantee reasoning* [7,12,16,18,19,38,39] is a central topic, and a compositional probabilistic framework [38] with the parallel composition $\|$ is also based on Pareto curves: extending string diagrams of MDPs for the parallel composition $\|$ is challenging, but an interesting future work. We mention that there are VIs on Pareto curves solving multi-objective simple stochastic games [1,13]. Due to the multi-objectivity, they maintain a *set of points* for each state during iterations; CVI solves single-objective oMDPs determined by weights, thus we maintain a single value for each state during iterations.

9 Conclusion

This paper investigates the verification of compositional MDPs, with a particular focus on approximating the behavior of the component MDPs via a Pareto cache and sound stopping criteria for value iteration. The empirical evaluation does not only demonstrate the efficacy of the novel algorithms, but also demonstrates the potential for further improvements, using asynchronous value iteration, efficient Pareto caches manipulations, and powerful compositional stopping criteria.

References

1. Ashok, P., Chatterjee, K., Kretínský, J., Weininger, M., Winkler, T.: Approximating values of generalized-reachability stochastic games. In: LICS, pp. 102–115. ACM (2020)
2. Azeem, M., Evangelidis, A., Kretínský, J., Slivinskiy, A., Weininger, M.: Optimistic and topological value iteration for simple stochastic games. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) Automated Technology for Verification and Analysis. ATVA 2022. LNCS, vol. 13505. Springer, Cham (2022).https://doi.org/10.1007/978-3-031-19992-9_18
3. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)

4. 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
5. Barry, J.L., Kaelbling, L.P., Lozano-Pérez, T.: Deth*: Approximate hierarchical solution of large Markov decision processes. In: IJCAI, pp. 1928–1935. IJCAI/AAAI (2011)
6. Barto, A.G., Mahadevan, S.: Recent advances in hierarchical reinforcement learning. Discret. Event Dyn. Syst. **13**(1–2), 41–77 (2003)
7. Bloem, R., Chatterjee, K., Jacobs, S., Könighofer, R.: Assume-guarantee synthesis for concurrent reactive programs with partial information. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 517–532. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_50
8. Bonchi, F., Gadducci, F., Kissinger, A., Sobocinski, P., Zanasi, F.: String diagram rewrite theory I: rewriting with Frobenius structure. J. ACM **69**(2), 14:1–14:58 (2022)
9. Bonchi, F., Holland, J., Piedeleu, R., Sobocinski, P., Zanasi, F.: Diagrammatic algebra: from linear to concurrent systems. Proc. ACM Program. Lang. **3**(POPL), 25:1–25:28 (2019)
10. Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
11. Budde, C.E., et al.: On correctness, precision, and performance in quantitative verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12479, pp. 216–241. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-83723-5_15
12. Chatterjee, K., Henzinger, T.A.: Assume-guarantee synthesis. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 261–275. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_21
13. Chen, T., Forejt, V., Kwiatkowska, M., Simaitis, A., Wiltsche, C.: On stochastic games with multiple objectives. In: Chatterjee, K., Sgall, J. (eds.) MFCS 2013. LNCS, vol. 8087, pp. 266–277. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40313-2_25
14. Ciosek, K., Silver, D.: Value iteration with options and state aggregation (2015). CoRR abs/1501.03959
15. Dai, P., Mausam, Weld, D.S., Goldsmith, J.J.: Topological value iteration algorithms. Artif. Intell. Res. **42**, 181–209 (2011)
16. Dewes, R., Dimitrova, R.: Compositional high-quality synthesis. In: André, É., Sun, J. (eds.) Automated Technology for Verification and Analysis. ATVA 2023. LNCS, vol. 14215. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-45329-8_16
17. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of Markov decision processes. Log. Methods Comput. Sci. **4**(4) (2008)
18. Finkbeiner, B., Passing, N.: Compositional synthesis of modular systems. Innov. Syst. Softw. Eng. **18**(3), 455–469 (2022)
19. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 112–127. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_11
20. Forejt, V., Kwiatkowska, M., Parker, D.: Pareto curves for probabilistic model checking. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 317–332. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_25

21. Gamrath, G., et al.: The SCIP optimization suite 7.0. Tech. Rep. 20-10, ZIB, Takustr. 7, 14195 Berlin (2020)
22. Ghani, N., Hedges, J., Winschel, V., Zahn, P.: Compositional game theory. In: LICS, pp. 472–481. ACM (2018)
23. Gopalan, N., et al.: Planning with abstract Markov decision processes. In: ICAPS, pp. 480–488. AAAI Press (2017)
24. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)
25. Hahn, E.M., Hartmanns, A.: A comparison of time- and reward-bounded probabilistic model checking techniques. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 85–100. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_6
26. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic. In: TACAS 2022. LNCS, vol. 13244, pp. 41–59. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_3
27. Hartmanns, A., Hermanns, H.: The modest toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
28. Hartmanns, A., Junges, S., Katoen, J., Quatmann, T.: Multi-cost bounded tradeoff analysis in MDP. J. Autom. Reason. **64**(7), 1483–1522 (2020)
29. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2023. LNCS, vol. 13993. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_24
30. 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
31. Hauskrecht, M., Meuleau, N., Kaelbling, L.P., Dean, T.L., Boutilier, C.: Hierarchical solution of Markov decision processes using macro-actions. In: UAI, pp. 220–229. Morgan Kaufmann (1998)
32. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022)
33. Jothimurugan, K., Bansal, S., Bastani, O., Alur, R.: Compositional reinforcement learning from logical specifications. In: NeurIPS, pp. 10026–10039 (2021)
34. Jothimurugan, K., Bastani, O., Alur, R.: Abstract value iteration for hierarchical reinforcement learning. In: AISTATS. Proceedings of Machine Learning Research, vol. 130, pp. 1162–1170. PMLR (2021)
35. Junges, S., Spaan, M.T.J.: Abstraction-refinement for hierarchical probabilistic models. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. CAV 2022. LNCS, vol. 13371. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_6
36. Klein, J., et al.: Advances in symbolic probabilistic model checking with PRISM. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 349–366. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_20
37. 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

38. Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Compositional probabilistic verification through multi-objective model checking. Inf. Comput. **232**, 38–65 (2013)
39. Majumdar, R., Mallik, K., Schmuck, A., Zufferey, D.: Assume-guarantee distributed synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(11), 3215–3226 (2020)
40. Neary, C., Verginis, C.K., Cubuktepe, M., Topcu, U.: Verifiable and compositional reinforcement learning systems. In: ICAPS, pp. 615–623. AAAI Press (2022)
41. Papadimitriou, C.H., Yannakakis, M.: On the approximability of trade-offs and optimal access of web sources. In: FOCS, pp. 86–92. IEEE Computer Society (2000)
42. Park, D.: Fixpoint induction and proofs of program properties. Machine intelligence **5**, 59–78 (1969)
43. Phalakarn, K., Takisaka, T., Haas, T., Hasuo, I.: Widest paths and global propagation in bounded value iteration for stochastic games. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 349–371. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_19
44. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley, Hoboken (1994)
45. Quatmann, T.: Verification of multi-objective Markov models. Phd thesis (2023). https://doi.org/10.18154/RWTH-2023-09669, https://publications.rwth-aachen.de/record/971553
46. 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
47. Quatmann, T., Katoen, J.-P.: Multi-objective optimization of long-run average and total rewards. In: TACAS 2021. LNCS, vol. 12651, pp. 230–249. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_13
48. Saxe, A.M., Earle, A.C., Rosman, B.: Hierarchy through composition with multitask LMDPs. In: ICML. Proceedings of Machine Learning Research, vol. 70, pp. 3017–3026. PMLR (2017)
49. Silver, D., Ciosek, K.: Compositional planning using optimal option models. In: ICML. icml.cc / Omnipress (2012)
50. Sutton, R.S., Precup, D., Singh, S.: Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. Artif. Intell. **112**(1–2), 181–211 (1999)
51. Vien, N.A., Toussaint, M.: Hierarchical monte-carlo planning. In: AAAI, pp. 3613–3619. AAAI Press (2015)
52. Watanabe, K., Eberhart, C., Asada, K., Hasuo, I.: A compositional approach to parity games. In: MFPS. EPTCS, vol. 351, pp. 278–295 (2021)
53. Watanabe, K., Eberhart, C., Asada, K., Hasuo, I.: Compositional probabilistic model checking with string diagrams of MDPs. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. CAV 2023. LNCS, vol. 13966. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37709-9_3
54. Watanabe, K., van der Vegt, M., Hasuo, I., Rot, J., Junges, S.: Pareto curves for compositionally model checking string diagrams of MDPs. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2024. LNCS, vol. 14571. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57249-4_14
55. Watanabe, K., van der Vegt, M., Junges, S., Hasuo, I.: Compositional value iteration with Pareto caching (2024). https://arxiv.org/abs/2405.10099, a longer version

Quantum Systems

Approximate Relational Reasoning for Quantum Programs

Peng Yan[1], Hanru Jiang[2], and Nengkun Yu[3](✉)

[1] University of Technology Sydney, Sydney, Australia
pengyan.edu@gmail.com

[2] Beijing Institute of Mathematical Sciences and Applications, Beijing, China
hanru@bimsa.cn

[3] Stony Brook University: Stony Brook, New York, USA
nengkunyu@gmail.com

Abstract. Quantum computation is inevitably subject to imperfections in its implementation. These imperfections arise from various sources, including environmental noise at the hardware level and the introduction of approximate implementations by quantum algorithm designers, such as lower-depth computations. Given the significant advantage of relational logic in program reasoning and the importance of assessing the robustness of quantum programs between their ideal specifications and imperfect implementations, we design a proof system to verify the approximate relational properties of quantum programs. We demonstrate the effectiveness of our approach by providing the first formal verification of the renowned low-depth approximation of the quantum Fourier transform. Furthermore, we validate the approximate correctness of the repeat-until-success algorithm. From the technical point of view, we develop approximate quantum coupling as a fundamental tool to study approximate relational reasoning for quantum programs, a novel generalization of the widely used approximate probabilistic coupling in probabilistic programs, answering a previously posed open question for projective predicates.

Keywords: Relational Hoare Logic · Approximating Reasoning · Quantum Programming Languages

1 Introduction

Program equivalence [11,18,41] is a central concept in many areas of computer science, including software engineering [31,36,54], translation validation of compilers [38], program optimization [30], and program analysis [6,15,35]. Relational verification aims to prove the relational properties between two programs. A typical Hoare-style relational judgment is of the form $\vdash c_1 \sim c_2 : \Psi \Rightarrow \Phi$ where c_1 and c_2 represent two compared programs, Ψ and Φ are relational assertions in the deterministic scenario [10], where relational Hoare logic (RHL) predicates are

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 495–519, 2024.
https://doi.org/10.1007/978-3-031-65633-0_22

binary relations over memories. The judgment states that for any initial memories m_1 and m_2 that satisfy the precondition Ψ, the resulting memories m_1' and m_2' should satisfy postcondition Φ. For probabilistic programs [7], probabilistic relational Hoare logic (pRHL) lifted the predicates into relations over probabilistic distributions on memories. Furthermore, [9] introduced extra parameters to allow approximate lifting of relations to distributions. To be specific, the judgments defined in approximate probabilistic relational Hoare logic (apRHL) are of the form $c_1 \sim_{\alpha,\delta} c_2 : \Psi \Rightarrow \Phi$ with parameters α, δ for reasoning about differential privacy.

Since the emergence of quantum programming languages, there have been various works [25,27,32,44,57,59–62] about the formal verification of quantum programs. Among techniques in program analysis, the *exact* relational logic for quantum programs attracts lots of attention [8,33,51]. Relational logic provides a more expressive approach to characterize the relation between two programs. For instance, direct verification of the equivalence between quantum programs S_1 and S_2 defined on register $\bar{q}$ requires checking the equivalence between $[\![S_1]\!](\rho)$ and $[\![S_2]\!](\rho)$ for all ρ in Hilbert space $\mathcal{H}_{\bar{q}}$ that involves enumerations of an infinite set. A quantum relational judgment concerning the quantum equivalence predicate can concisely explain the direct enumerations. However, none of the above works considers approximate reasoning that is universal in practice.

- It is implausible to physically implement quantum gates with perfect accuracy on the hardware level, and the need to consider approximations is likely inevitable. As noted by John Preskill, the noise in quantum gates will limit the size of reliable quantum circuits, and technologies for more accurate quantum gates are of great value in the Noisy Intermediate-Scale Quantum (NISQ) [42] era.
- On the software level, the NISQ nature of hardware signifies the importance of taking noise into account at the level of quantum algorithm design. More specifically, approximate computation can be more efficient and less erroneous than precise one since it can improve the depth of circuits and simplify the calculation. A good example is the approximate quantum Fourier transform [16], which achieves a lower circuit depth approximation of the exact quantum Fourier transform used in Shor's celebrated algorithm [45].

As for approximate reasoning in quantum settings, [66] discussed the robustness of quantum programs by introducing the concept of approximate satisfaction of predicates, [26] proposed a parameterized diamond norm to characterize the distance between an ideal program and a noisy one. Despite the significant advancements in quantum approximate reasoning and the recognition of the importance of relational reasoning, there remains a notable gap in the field — an absence of a robust logical framework for effectively reasoning about the relational properties between quantum programs approximately. In quantum approximate relational reasoning, the main obstacles are:

- There is no mathematical theory for a quantum version of approximate couplings, an open question in [8]. The lack of such a theory significantly affects the applications of exact quantum coupling and relations quantum Hoare logic. Usually, two quantum programs have different branching probabilities in the presence of noise or approximations. Under these circumstances, their corresponding quantum states have different traces, where *exact* quantum couplings on these states do not exist. The main difficulties in defining an approximate quantum coupling include defining a distance between quantum states, which can be highly nonlinear. Previous knowledge about probabilistic couplings may not directly apply: even for the exact quantum coupling, fundamental properties of probabilistic coupling [24] are no longer true [67].

- Designing an approximate relational quantum Hoare logic system is indeed highly challenging. The system needs to consider several factors, including infinite executions of quantum while loops, approximated quantum couplings, and the applicability of the logic rules. In quantum programming, a while loop can have infinite executions of the loop body because of the probabilistic feature of quantum measurements. Furthermore, when dealing with approximate quantum couplings, the system must handle the inherent uncertainty and approximation errors that arise when coupling with program branches.
- The applicability of logic rules adds another layer of complexity. To strike a balance between the accuracy of the logic rules and simplicity, efficiency, and usability is a crucial consideration when designing a logic system. Ensuring the logic rules are powerful yet easy to use for reasoning relational properties of complicated quantum programs requires careful consideration and analysis.

In this paper, we derive an approximate version of the existing quantum relational Hoare logic, thus making approximate relational reasoning feasible. Our judgment is of the form

$$S_1 \sim_\delta S_2 : A \Rightarrow B$$

where S_1 and S_2 represent compared quantum programs, A and B are projective quantum predicates over the whole system. The validity of our judgment is based on the idea of approximate (quantum) coupling and lifting. A state ρ is a δ-coupling for the state pair $\langle \rho_1, \rho_2 \rangle$ if trace distances $D(\rho_1, \mathrm{Tr}_2(\rho))$ and $D(\rho_2, \mathrm{Tr}_1(\rho))$ are both not bigger than δ. A state σ is a witness of the δ-lifting $\rho_1 \sim_P^\delta \rho_2$ if σ is a δ-coupling for the state pair $\langle \rho_1, \rho_2 \rangle$ and satisfies the predicate P $(P\sigma = \sigma)$. Informally, our judgment holds if for any quantum lifting $\rho_1 \sim_A^0 \rho_2$ of the inputs, there exists a witness of the δ-lifting $[\![S_1]\!](\rho_1) \sim_P^\delta [\![S_2]\!](\rho_2)$ of the outputs.[1]

Technical contributions include:

- *Approximate quantum liftings.* We propose a novel notion of approximate quantum liftings concerning projection-based quantum predicates to make approximate reasoning simple and powerful. We do not require two quantum states to have the same trace in approximate quantum lifting. In other words, the exact quantum coupling may not exist. We employ the existing distances, including trace distance and diamond norm, and define a "Hausdorff-like"

[1] See Sect. 2 and Sect. 5 for a detailed definition.

distance between projections incorporated with quantum coupling to be the metric of the approximation of the couplings.

- *Sound aqRHL.* We propose a formal relational judgment to incorporate the spirit of classical apRHL with a new quantum explanation based on the proposed approximate quantum liftings. A sound approximate quantum relational Hoare logic (aqRHL) is built based on our relational judgments. Our choice of quantum δ-lifting allows us to track the relational properties of two programs with different classical branching probabilities. In particular, our methodology allows us to compute proper upper bounds for approximate liftings for quantum equivalence relations, which plays a central role in characterizing the equivalence of quantum programs.
- *Application.* We demonstrate the first formal verification of the low-depth approximate quantum Fourier transform (QFT) with an error bound that is asymptotically equivalent to the one in [16]. Implementing QFT is a significant step in the development of quantum algorithms such as period finding [45], HHL algorithm [21] and quantum principal component analysis [34]. We also apply aqRHL, particularly the loop rule, to reason the repeat until success which is one of the essential loop programs in quantum computation. Other applications covered in the complete edition of this paper include the verification of appropriate decomposition of unitary gates, and the correctness of bit flip code against an arbitrary single-qubit error.

2 Preliminary and Notations

This section offers a brief introduction to quantum computation and necessary notations from [39].

The state space of a quantum system is a Hilbert space $\mathcal{H}$. The Dirac notation $|\psi\rangle$ denotes a unit complex vector (called *pure state* or *vector state*). The most important orthonormal basis of one-qubit system is the *computational basis*, i.e. $\{|0\rangle, |1\rangle\}$. Superposition is a key feature that makes quantum programs different from classical ones, such as a qubit being in the superposition $(|0\rangle \pm |1\rangle)/\sqrt{2}$. An operator acting on an d-dimensional Hilbert space $\mathcal{H}$ is represented as a $d \times d$ matrix. A positive semi-definite, Hermitian operator, ρ acting on $\mathcal{H}$, is called a *partial density operator* if its trace satisfies $\mathrm{Tr}(\rho) \leq 1$. Particularly, ρ is called a *density operator* if $\mathrm{Tr}(\rho) = 1$. The partial density operators can represent both pure and *mixed* quantum states. For a pure state $|\psi\rangle$, its partial density operator is $|\psi\rangle\langle\psi|$, where $\langle\psi|$ is the conjugate transpose of $|\psi\rangle$. For a mixed state which is a classical distribution $\{p_i\}$ over pure states $\{|\psi_i\rangle\}$, its partial density operator is $\sum_i p_i|\psi_i\rangle\langle\psi_i|$. We use $\mathcal{D}(\mathcal{H})$ to denote the set of all partial density operators acting on Hilbert space $\mathcal{H}$.

Let $\bar{q}_1$ and $\bar{q}_2$ be two independent registers in states $\rho_1 \in \mathcal{D}(\mathcal{H}_{\bar{q}_1})$ and $\rho_2 \in \mathcal{D}(\mathcal{H}_{\bar{q}_2})$ respectively, the composite register $\bar{q} = \{\bar{q}_1, \bar{q}_2\}$ is then in the state $\rho_1 \otimes \rho_2 \in \mathcal{H}_{\bar{q}} = \mathcal{H}_{\bar{q}_1} \otimes \mathcal{H}_{\bar{q}_2}$. *Partial trace* is a very useful tool for describing subsystems of a composite quantum system. Formally, the partial trace over $\mathcal{H}_{\bar{q}_1}$ is a mapping $\mathrm{Tr}_1(\cdot)$ from operators in $\mathcal{H}_{\bar{q}_1} \otimes \mathcal{H}_{\bar{q}_2}$ to operators in $\mathcal{H}_{\bar{q}_2}$ defined by $\mathrm{Tr}_1(|\varphi_1\rangle\langle\psi_1| \otimes |\varphi_2\rangle\langle\psi_2|) = \langle\psi_1|\varphi_1\rangle \cdot |\varphi_2\rangle\langle\psi_2|$ for any $|\psi_1\rangle, |\varphi_1\rangle \in \mathcal{H}_{\bar{q}_1}$ and $|\psi_2\rangle, |\varphi_2\rangle \in \mathcal{H}_{\bar{q}_2}$. The partial trace $\mathrm{Tr}_2(\cdot)$ can be defined symmetrically. If the composite system $\bar{q} = \{\bar{q}_1, \bar{q}_2\}$ is in the state ρ, then subsystems $\bar{q}_1$ and $\bar{q}_2$ are in states $\mathrm{Tr}_2(\rho)$ and $\mathrm{Tr}_1(\rho)$ respectively.

The evolution of an isolated quantum system can be characterized by a *unitary operator* U such that $U^\dagger U = UU^\dagger = I$, where † denotes the conjugate and transpose. Here we introduce some commonly used unitary operators, also known as "*gates*", that will be used in later examples:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$P(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix} \quad CNOT = \begin{pmatrix} I & 0 \\ 0 & X \end{pmatrix} \quad c\text{-}P(\theta) = \begin{pmatrix} I & 0 \\ 0 & P(\theta) \end{pmatrix}$$

The act of extracting information from a quantum system is known as *quantum measurement.* A measurement $\mathcal{M} = \{M_m\}$ is described by a set of linear operators over $\mathcal{H}$ such that $\sum_m M_m^\dagger M_m = I$, where the subscript m refers to the measurement outcome. Applying a quantum measurement $\mathcal{M}$ on $|\psi\rangle$, the probability of observing outcome m is $p_m = \langle\psi|M_m M_m^\dagger|\psi\rangle$, and the state after the measurement collapses into $M_m|\psi\rangle/\sqrt{p_m}$.

A *projection* is a linear operator P on $\mathcal{H}$ that satisfies $P^2 = P = P^\dagger$. This paper adopts the convention from [12] and recent work [51,66] that constrains quantum predicates to be Hilbert spaces or projections. The complete partial order over Hilbert subspaces is equivalent to the inclusion relation $\subseteq$. This choice of predicates enables us to define the assertion about quantum states.

Definition 1 (Support). *If $A = \sum_i \lambda_i|\psi_i\rangle\langle\psi_i|$, where $|\psi_i\rangle$ is an unit vector in $\mathcal{H}$ and $\lambda_i > 0$, then the* support *of A is the space spanned by $\{|\psi_i\rangle\}$. I.e.,* $\mathrm{supp}(A) = \mathrm{span}\{|\psi_i\rangle\}$.

Definition 2 (Satisfaction). *A partial density operator ρ satisfies a predicate P, denoted by $\rho \vDash P$, if* $\mathrm{supp}(\rho) \subseteq P$.

A general quantum operation, described by a superoperator $\mathcal{E}$, can be implemented by combining unitary transformations with quantum measurements by introducing ancilla systems and discarding post-measurement states. A superoperator always maps density matrices to partial density matrices and has the Kraus representation. Readers may refer to the system-environment model in Sect. 8.2 [39] for more details.

3 Quantum Programming Language

In this section, we review the syntax and semantics of the quantum while-language [59]. We use $var(S)$ to represent the set of all variables present in a quantum program S, and $\mathcal{H}_S = \otimes_{q \in var(S)} \mathcal{H}_q$ to denote the Hilbert space of all the quantum variables in program S. The syntax in Definition 3 is the same as [59] except that the conditional statement is replaced by the **if** statement.

Definition 3 (Syntax). *The following syntax defines the quantum programs:*

$$\begin{aligned}\text{(Stmts)}\quad S \ ::=\ & \textbf{skip} \mid q := |0\rangle \mid \bar{q} := U[\bar{q}] \mid S_1; S_2 \\ & \mid \textbf{if } (\square m \cdot \mathcal{M}[\bar{q}] = m \rightarrow S_m) \textbf{ fi} \mid \textbf{while } \mathcal{M}[\bar{q}] = 1 \textbf{ do } S \textbf{ od}\end{aligned}$$

The denotational semantics of the quantum while-language are presented in Fig. 1. By convention, we use $[\![S]\!]$ to denote the semantic function of a program S. Statement $q := |0\rangle$ initializes a variable q in state ρ to $|0\rangle\langle 0|$ while leaving other variables unchanged, where $|\psi\rangle_q\langle\varphi|$ denote the outer product of the vector states $|\psi\rangle$ and $|\varphi\rangle$ in $\mathcal{H}_q$. The statement $\bar{q} := U[\bar{q}]$ performs the unitary transition $\rho \mapsto U\rho U^\dagger$ over register $\bar{q}$. Quantum measurements work as guards to set a variable in a mixed state. For the if statement, if the measurement outcome is m, the input state ρ will collapse into $M_m\rho M_m^\dagger/p_m$ with probability $p_m = \mathrm{Tr}(M_m\rho M_m^\dagger)$ and then executes subprogram S_m. Here we absorb the probability p_m into the collapse state, and $M_m\rho M_m^\dagger$ represents the corresponding measurement output. The final output is the summation of the outputs of all branches. For the loop statement, $\mathcal{M}_0 \circ ([\![S]\!] \circ \mathcal{M}_1)^k$ denotes the k-th unrolling of loop statement.

$$[\![\textbf{skip}]\!](\rho) = \rho \qquad [\![q := |0\rangle]\!](\rho) = \textstyle\sum_n |0\rangle_q\langle n|\rho|n\rangle_q\langle 0|$$

$$[\![\bar{q} := U[\bar{q}]]\!](\rho) = U\rho U^\dagger \qquad [\![S_1; S_2]\!] = [\![S_2]\!] \circ [\![S_1]\!]$$

$$[\![\textbf{if } (\square m \cdot \mathcal{M}[\bar{q}] = m \rightarrow S_m) \textbf{ fi}]\!](\rho) = \textstyle\sum_m S_m \circ \mathcal{M}_m = \sum_m [\![S_m]\!](M_m\rho M_m^\dagger)$$

$$[\![\textbf{while } \mathcal{M}[\bar{q}] = 1 \textbf{ od } S \textbf{ od}]\!](\rho) = \textstyle\sum_{k=0}^{\infty} \mathcal{M}_0 \circ ([\![S]\!] \circ \mathcal{M}_1)^k(\rho)$$

$$\mathcal{M}_m(\rho) = M_m\rho M_m^\dagger \quad (\mathcal{R}_1 + \mathcal{R}_2)(\rho) = \mathcal{R}_1(\rho) + \mathcal{R}_2(\rho) \quad \mathcal{R}^0(\rho) = \rho \quad \mathcal{R}^n = \mathcal{R}^{n-1} \circ \mathcal{R}$$

$$\mathcal{R}_2 \circ \mathcal{R}_1(\rho) = \{\rho'' \mid \rho' = \mathcal{R}_1(\rho) \wedge \rho'' = \mathcal{R}_2(\rho') \wedge \rho, \rho', \rho'' \in \mathcal{D}(\mathcal{H})\}$$

Fig. 1. Denotational semantics of quantum while-language

Lemma 1 ([59]). *For any quantum while program S defined in Fig. 1, its denotational semantics function $[\![S]\!] : \mathcal{D}(\mathcal{H}) \mapsto \mathcal{D}(\mathcal{H})$ is a superoperator.*

4 Quantum Approximate Coupling and Liftings

4.1 Approximate Quantum Coupling and Lifting

We first review the quantum generalization of the classical trace distance, a commonly used metric for the difference between two (partial) quantum states.

Definition 4. *The trace distance of two partial density operators ρ and σ is $D(\rho, \sigma) \equiv \frac{1}{2}\mathrm{Tr}|\rho - \sigma|$, where $|A| = \sqrt{A^\dagger A}$ for any operator A, i.e., the positive square root of $A^\dagger A$.*

In the classical setting, two discrete distributions μ_1 and μ_2 over sets A_1 and A_2 are coupled by a distribution μ over $A_1 \times A_2$ if and only if the first and second marginals of μ are exactly μ_1 and μ_2, respectively. This notion of coupling for distributions naturally generalizes to an exact quantum coupling [8] for density matrices. Formally, we say ρ is an exact *coupling* for $\langle \rho_1, \rho_2 \rangle$ if $\mathrm{Tr}_1(\rho) = \rho_2$ and $\mathrm{Tr}_2(\rho) = \rho_1$. Commonly, the quantum measurements in two quantum programs produce different probability distributions. In such cases, *exact* quantum coupling does not exist between branches. We propose approximate quantum coupling parameterized by deviation δ to bound the trace distance between the "marginal" density matrices.

Definition 5 (Approximate Quantum Coupling). *Let $\rho_1 \in \mathcal{D}(\mathcal{H}_1)$ and $\rho_2 \in \mathcal{D}(\mathcal{H}_2)$, then $\rho \in \mathcal{D}(\mathcal{H}_1 \otimes \mathcal{H}_2)$ is a δ-*coupling *for $\langle \rho_1, \rho_2 \rangle$ if*

$$D(\rho_1, \mathrm{Tr}_2(\rho)) \leq \delta \qquad D(\rho_2, \mathrm{Tr}_1(\rho)) \leq \delta$$

The approximate quantum coupling degenerates to the exact version if $\delta = 0$. Like the classical case, approximate quantum coupling induces approximate semantics of projective predicates via approximate lifting.

Definition 6 (Approximate Quantum Lifting). *Let $\rho_1 \in \mathcal{D}(\mathcal{H}_1)$ and $\rho_2 \in \mathcal{D}(\mathcal{H}_2)$, let P be a projection onto a closed subspace of $\mathcal{H}_1 \otimes \mathcal{H}_2$, then $\rho \in D(\mathcal{H}_1 \otimes \mathcal{H}_2)$ is called a witness of the δ-lifting $\rho_1 \sim_P^\delta \rho_2$ if,*

1. *ρ is a δ-coupling for $\langle \rho_1, \rho_2 \rangle$;*
2. $\mathrm{supp}(\rho) \subseteq P$.

where δ is the deviation *from the exact quantum lifting.*

A valid approximate quantum lifting implies the existence of an approximate quantum coupling that satisfies a quantum predicate. The approximate lifting $\rho_1 \sim_P^\delta \rho_2$ degenerates into the exact lifting $\rho_1 \sim_P \rho_2$ when $\delta = 0$. One of the most important quantum predicates is the equivalence relation between two registers, as defined below [8].

Definition 7 (Equivalence). *Let register $\bar{p}$ and $\bar{q}$ are two disjoint registers of the same size. The quantum equivalence predicate over $(\bar{p}, \bar{q})$, denoted by $\equiv_{(\bar{p},\bar{q})}$, is the projection*

$$(I_{\bar{p}} \otimes I_{\bar{q}} + SWAP)/2$$

over subspace $\mathcal{H}_{\bar{p}} \otimes \mathcal{H}_{\bar{q}}$. $SWAP$ is the swap operator defined on $(\bar{p}, \bar{q})$ such that by $SWAP|\psi\rangle|\varphi\rangle = |\varphi\rangle|\psi\rangle$ for any $|\psi\rangle \in \mathcal{H}_{\bar{p}}$ and $|\varphi\rangle \in \mathcal{H}_{\bar{q}}$.

The quantum equivalence predicate in Definition 7 directly comes from a natural observation. In the probabilistic world, if two probability distributions μ_1 and μ_1 over X are the same, then there exists a coupling μ whose support lives in the identity relation $\{(a, a) \mid a \in X\}$. In quantum settings, this is not true due to superposition. For example, the exact coupling of the state $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ and itself is $|+\rangle \otimes |+\rangle$, which is not in the space spanned by $|0\rangle \otimes |0\rangle$ and $|1\rangle \otimes |1\rangle$.

Instead, we use the projection $(I + SWAP)/2$ to represent the corresponding symmetric space. By doing so, we have $(I + SWAP)(|+\rangle \otimes |+\rangle)/2 = |+\rangle \otimes |+\rangle$. The following lemma shows that approximate lifting concerning $\equiv$[2] effectively encodes the trace distance of two partial density matrices.

Lemma 2. *For any ρ_1, ρ_2, we have $\rho_1 \sim^{\delta}_{\equiv} \rho_2 \;\Leftrightarrow\; D(\rho_1, \rho_2) \leq 2\delta$. Particularly, if $\delta = 0$, we have $\rho_1 \sim_{\equiv} \rho_2 \;\Leftrightarrow\; \rho_1 = \rho_2$.*

The introduction of approximate couplings/liftings is necessary when the comparison can not match the desired predicate exactly. For example, the implementation of a unitary gate U can be approximated by a proper RUS circuit [13,40]. Given an input ρ, quantum measurement in each iteration of the RUS circuit will generate the desired output $U\rho U^\dagger$ with a probability p, where p is determined by the construction of the RUS circuit. If the iterations of the RUS circuit are unbounded, the desired state can eventually be achieved. The RUS algorithm's function $\mathcal{E}$ converges to U, as expressed as $\mathcal{E}(\rho) = \sum_{k=1}^{\infty} p(1-p)^{k-1} U\rho U^\dagger = U\rho U^\dagger$. In this case, the exact lifting can accurately describe the equivalence $\mathcal{E}(\rho) \sim_{\equiv} U\rho U^\dagger$ with no problem. However, in practical scenarios, there typically exists an upper bound N on the iteration count k, leading to an approximate equivalence denoted as $\mathcal{E}'(\rho) \sim^{\delta}_{\equiv} U\rho U^\dagger$, where $\mathcal{E}'$ represents the function with bounded looping and $\delta = (1-p)^N/2$.

4.2 Upper Bound of Approximation

The approximation usually arises when we use a desired postcondition to approximate an exact postcondition. Formally, let (A, B) be the pair of two projections A and B over Hilbert space $\mathcal{H}_1 \otimes \mathcal{H}_2$, the inference

$$\forall \rho_1, \rho_2, \rho_1 \sim_A \rho_2 \;\Rightarrow\; \rho_1 \sim^{\delta}_{B} \rho_2 \tag{1}$$

demonstrates a general way of introducing approximate reasoning. That is, given a witness of the exact lifting $\rho_1 \sim_A \rho_2$, does there exist a witness σ of the approximate lifting $\rho_1 \sim^{\delta \rho_2}_{B}$?. The optimal deviation δ in Eq. 1 is equivalent to the following quantity,

$$\delta = d(A, B) = \sup_{\rho \vDash A} \inf_{\sigma \vDash B} \max\{D(\mathrm{Tr}_1(\rho), \mathrm{Tr}_1(\sigma)), D(\mathrm{Tr}_2(\rho), \mathrm{Tr}_2(\sigma))\} \tag{2}$$

where $d(A, B)$ can be upper bounded by $\sup_{\rho \vDash A} \inf_{\sigma \vDash B} D(\rho, \sigma)$ introduced in [66].

In the following, we discuss a simple but important instance of Eq. 1 with $A = (U_1 \otimes U_2)B(U_1 \otimes U_2)^\dagger$ and B being the quantum equivalence predicate $\equiv$. Then Eq. 1 can be represented as follows,

$$\forall \rho_1, \rho_2, \rho_1 \sim_{\equiv} \rho_2 \;\Rightarrow\; U_1\rho_1U_1^\dagger \sim_A U_2\rho_2U_2^\dagger \;\Rightarrow\; U_1\rho_1U_1^\dagger \sim^{\delta}_{\equiv} U_2\rho_2U_2^\dagger$$

where δ can be upper bounded by $\|U_1 \cdot U_1^\dagger - U_2 \cdot U_2^\dagger\|_\diamond$. The diamond norm $\|\cdot\|_\diamond$ proposed by Kitaev [2] can better distinguish between two superoperators with the help of the power of quantum entanglement by introducing auxiliary qubits.

[2] The subscripts can be ignored without confusion.

Definition 8 (Diamond Norm). *Let $\mathcal{A} : \mathcal{L}(\mathcal{H}) \mapsto \mathcal{L}(\mathcal{H})$ with $\mathcal{L}(\mathcal{H})$ denoting the matrix space of $\mathcal{H}$,*

$$\|\mathcal{A}\|_\diamond \equiv \max_{\rho \in \mathcal{D}(\mathcal{H} \otimes \mathcal{H}')} \frac{1}{2} \mathrm{Tr}|(\mathcal{A} \otimes I_{\mathcal{H}'})(\rho)| \tag{3}$$

where $\mathcal{H}'$ denotes a copy of $\mathcal{H}$. The factor $1/2$ is added to keep consistent with trace distance.

It is straightforward to verify that $D(\mathcal{E}_1(\sigma), \mathcal{E}_2(\sigma)) \leq \|\mathcal{E}_1 - \mathcal{E}_2\|_\diamond$ for any $\sigma \in \mathcal{D}(\mathcal{H})$ for superoperators $\mathcal{E}_1, \mathcal{E}_2$ by choosing $\rho = \sigma \otimes I_{\mathcal{H}'} / 2^{Dim(\mathcal{H}')}$.[3] The distance between two superoperators can be computed efficiently [53]. Particularly, the distance between U_1 and U_2 is

$$\|U_1 \cdot U_1^\dagger - U_2 \cdot U_2^\dagger\|_\diamond = \begin{cases} \sin \alpha/2 & \alpha < \pi \\ 1 & \alpha \geq \pi \end{cases} \tag{4}$$

where α is the smallest arc containing the spectrum of $U_1^\dagger U_2$ [37].

5 Approximate Relational Logic

5.1 Judgment and Validity

Our logic, called aqRHL, "approximates" the quantum relational Hoare logic described in [8]. The judgments in aqRHL take the following form $S_1 \sim_\delta S_2 : A \Rightarrow B$, where S_1 and S_2 are quantum programs, A and B are projections over subspaces of $\mathcal{H}_{\bar{q}_1} \otimes \mathcal{H}_{\bar{q}_2}$ such that $\bar{q}_i$ contains all free variables of S_i, $\delta \in [0, 1/2]$ is referred to as the *deviation* from the exact quantum lifting, respectively. Registers $\bar{q}_1$ and $\bar{q}_2$ are often omitted since they rarely change along our reasoning and are often clear from the context.

Definition 9 (Validity). *The approximate relational judgement $S_1 \sim_\delta S_2 : A \Rightarrow B$ is valid, written as $\models S_1 \sim_\delta S_2 : A \Rightarrow B$, if and only if*

$$\forall \rho_1, \rho_2 . \, \rho_1 \sim_A \rho_2 \quad \Rightarrow \quad [\![S_1]\!](\rho_1) \sim_B^\delta [\![S_2]\!](\rho_2)$$

where A and B are projections. If the deviation δ equals zero, it will be omitted for simplicity.

In Definition 9, we choose projective predicates [12] over the joint system of two programs because such predicates are the quantum analog of binary relations, the predicates used in pRHL [9]. Moreover, this definition will become a judgment of [8] if $\delta = 0$. One of the most important applications of relational Hoare logic is to verify the equivalence between programs, as presented in the following lemma. Naturally, the approximate equivalence between programs can also be reasoned by the approximate relational judgment, where δ characterizes the deviation of approximation.

[3] $Dim(\mathcal{H})$ denotes the dimension of $\mathcal{H}$.

Lemma 3 (Program Equivalence). *Program S_1 is equivalent*[4] *to program S_2 if and only if $\vDash S_1 \sim S_2 : \equiv \Rightarrow \equiv$.*

The program equivalence can be expressed concisely with predicates being the equivalence relation, instead of checking whether two quantum programs perform uniformly by enumeration of an infinite number of states in Hilbert space. The following example shows that superposition makes quantum program equivalence more complex than its classical counterpart.

Example 1. Let S_1 and S_2 be two programs defined on a single bit or qubit. For classical programs, the state space for programs S_1 and S_2 is the set $\{|0\rangle, |1\rangle\}$. Let Ψ and Φ be the equivalence relation, the relational judgment $S_1 \sim S_2 : \Psi \ \Rightarrow \ \Phi$ holds for classical programs S_1 and S_2 if

$$[\![S_1]\!](|0\rangle\langle 0|) = [\![S_2]\!](|0\rangle\langle 0|) \qquad [\![S_1]\!](|1\rangle\langle 1|) = [\![S_2]\!](|1\rangle\langle 1|) \tag{5}$$

However, this conclusion no longer holds in quantum programs since the input state could be a superposition of $|0\rangle$ and $|1\rangle$. For example, let $S_1 ::=$ **skip** and $S_2 ::= q := Z[q]$, it is clear that S_1 and S_2 are not equivalent[5] although Eq. 5 still holds. To check quantum program equivalence, we need to verify the validity of $[\![S_1]\!](\rho) = [\![S_2]\!](\rho)$ for all ρ in the Hilbert space span$\{|0\rangle, |1\rangle\}$ rather than the set $\{|0\rangle, |1\rangle\}$, which involves enumerations of an infinite set.

5.2 Proof Rules

We are ready to provide some proof rules for our aqRHL judgments. These rules include construct-specific rules (two-sided and one-sided) and structural ones, as is typical in relational Hoare logic. Notice that rules for branching structures are discussed in Sect. 7 later.

Simple Rules. Figure 2 includes the two/one-sided proof rules for basic statements and sequence structure. The basic rules, namely [Skip], [Init], [Ut] are similar to their counterparts in [8] with $\delta = 0$, where they are presented in the forward variant. Here we use proj(A) to lift non-projection A to its support before assigning it as a predicate. Notice that the rule [Ut] gives the strongest postcondition, which means the reverse $\vdash \bar{q}_1 := U_1^{-1}[\bar{q}_1] \sim \bar{q}_2 := U_2^{-1}[\bar{q}_2] : (U_1 \otimes U_2)A(U_1^\dagger \otimes U_2^\dagger) \Rightarrow A$ still holds. The rule [Seq] demonstrates that the deviation grows linearly with respect to the sequences, which directly comes from the triangle inequality of trace distance. One-sided rules are necessary when two programs do not share the same structure. We have only listed the one-side rules (appended with "-L") for the left side, and similar rules apply to the right side symmetrically.

[4] That is, $[\![S_1]\!](\rho) = [\![S_2]\!](\rho)$ holds for any partial density operator ρ.

[5] $[\![S_1]\!](|\psi\rangle\langle\psi|) \neq [\![S_2]\!](|\psi\rangle\langle\psi|)$ for any superposition $|\psi\rangle = a|0\rangle + b|1\rangle$, $0 < |a|^2 + |b|^2 \leq 1$.

$$\begin{array}{rl}
(\textsc{Skip}) & \vdash \textbf{skip} \sim \textbf{skip} : A \Rightarrow A \\
(\textsc{Init}) & \vdash \bar{q}_1 := |0\rangle \sim \bar{q}_2 := |0\rangle : A \Rightarrow |0\rangle_{\bar{q}_1}\langle 0| \otimes |0\rangle_{\bar{q}_2}\langle 0| \otimes \mathrm{proj}(\mathrm{Tr}_{(\bar{q}_1,\bar{q}_2)}(A)) \\
(\textsc{Ut}) & \vdash \bar{q}_1 := U_1[\bar{q}_1] \sim \bar{q}_2 := U_2[\bar{q}_2] : A \Rightarrow (U_1 \otimes U_2)A(U_1^\dagger \otimes U_2^\dagger) \\
(\textsc{Seq}) & \dfrac{\vdash S_1 \sim_{\delta_1} S_2 : A \Rightarrow R \quad \vdash S_1' \sim_{\delta_2} S_2' : R \Rightarrow B}{\vdash S_1; S_1' \sim_{\delta_1+\delta_2} S_2; S_2' : A \Rightarrow B} \\
(\textsc{Init-L}) & \vdash \bar{q}_1 := |0\rangle \sim \textbf{skip} : A \Rightarrow |0\rangle_{\bar{q}_1}\langle 0| \otimes \mathrm{proj}(\mathrm{Tr}_{\mathcal{H}_{\bar{q}_1}}(A)) \\
(\textsc{Ut-L}) & \vdash \bar{q}_1 := U_1[\bar{q}_1] \sim \textbf{skip} : A \Rightarrow (U_1 \otimes I_2)A(U_1^\dagger \otimes I_2)
\end{array}$$

Fig. 2. Simple aqRHL rules.

Rules for Equivalence Relation. We address a scenario regarding the rules [Ut], where the precondition and postcondition are equivalence relations defined in Definition 7. We use diamond norm to bound the deviation in rule [Ut-id], where $U \cdot U^\dagger$ denotes the Kraus representation [39] of unitary U. The rule [Comp] permits reasoning equivalence between programs by introducing intermediate programs (Fig. 3).

$$\begin{array}{rl}
(\textsc{Ut-id}) & \vdash \bar{q}_1 := U_1[\bar{q}_1] \sim_{\|U_1 \cdot U_1^\dagger - U_2 \cdot U_2^\dagger\|_\diamond/2} \bar{q}_2 := U_2[\bar{q}_2] : \equiv \Rightarrow \equiv \\
(\textsc{Comp}) & \dfrac{\vdash S_1 \sim_{\delta_1} S_2 : \equiv \Rightarrow \equiv \qquad \vdash S_2 \sim_{\delta_2} S_3 : \equiv \Rightarrow \equiv}{\vdash S_1 \sim_{\delta_1+\delta_2} S_3 : \equiv \Rightarrow \equiv}
\end{array}$$

Fig. 3. Rules for Equivalence Relation

5.3 Soundness Theorem

Theorem 1. [Soundness] *For any program S_1, S_2, projections A and B, deviation δ, we have,*

$$\vdash S_1 \sim_\delta S_2 : A \Rightarrow B \quad \Rightarrow \quad \models S_1 \sim_\delta S_2 : A \Rightarrow B$$

The soundness of our proof system is proved with respect to the validity of judgments defined in 9, while the completeness remains an open question. For classical deterministic programs, relational Hoare logic has been demonstrated to be relatively complete for terminating programs with the help of providing additional supplementary one-sided rules. However, relative completeness does not extend to probabilistic programs. As highlighted in [5], the probabilistic coupling method lacks the robustness of the conductance method in demonstrating the rapid mixing of Markov chains. Building upon the work laid by [8], the quantum extension of probabilistic couplings and the introduction of approximation in our judgments further complicate this problem.

6 Approximate Quantum Fourier Transform

Objective. As a quantum analog of the classical discrete Fourier transform, *quantum Fourier transform* (QFT) [17] performs a linear transformation on quantum states and extracts the periodicity of the amplitudes of quantum states. Due to the imperfectness of quantum gates, the *approximate quantum Fourier transform* (AQFT) is proposed to improve the circuit depth of QFT for efficiency. Reference [17] proposes a direct AQFT based on ignoring gates related to high-order terms. Cleve and Watrous [16] parallelized the phase estimation procedure to perform AQFT with lower circuit depth. Let S_{QFT} and S_{AQFT} be the corresponding quantum programs for QFT and AQFT, respectively. This section uses our logic to derive the judgment of form $S_{\text{QFT}} \sim_\delta S_{\text{AQFT}} : \equiv \Rightarrow \equiv$ to reason about how well AQFT approximates QFT.

Specification. For an n qubit system, QFT on a *computational basis state* $|x\rangle = |x_1 x_2 \dots x_n\rangle$ is defined as the linear operation U such that

$$U|x\rangle = |\psi_x\rangle = \tfrac{1}{\sqrt{2^n}} \textstyle\sum_{y=0}^{N-1} (e^{2\pi i/N})^{x\cdot y}|y\rangle \tag{6}$$

where $N = 2^n$, $|\psi_x\rangle$ is called a *Fourier basis* state with respect to state $|x\rangle$, $x \cdot y$ denotes the multiplication between the binary representation of x and y. $|\psi_x\rangle$ can be described as $|\psi_x\rangle = |\mu_{0.x_n}\rangle|\mu_{0.x_{n-1}x_n}\rangle \cdots |\mu_{0.x_1 \dots x_n}\rangle$, where $|\mu_\theta\rangle = (|0\rangle + e^{2\pi i\theta}|1\rangle)/\sqrt{2}$, $0.x_i \dots x_j$ denotes the binary fraction $x_i/2 + x_{i+1}/4 + \cdots + x_j/2^{j-i+1}$. State $|\mu_\theta\rangle$ can be obtained by applying the phase shift gate $P(2\pi\theta)$ (mentioned in Sect. 2) on state $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$. The phase shift gate $P(2\pi\theta)$ can be decomposed as the sequence of gates $R_m = P(2\pi/2^m)$ since $P(\theta_1)P(\theta_2) = P(\theta_1 + \theta_2)$. The controlled R_m gate is denoted by $CR_m[(q_1, q_2)]$, which is the *c*-$P(\theta)$ gate (mentioned in Sect. 2) with $\theta = 2\pi/2^m$.

QFT can be parallelly implemented [16], as shown in Fig. 4. The unitary V generates the Fourier basis state $|\psi_x\rangle$ without erasing $|x\rangle$. The unitary *Add* introduces auxiliary $(k-1)n$ qubits to create $k-1$ replicas of Fourier basis state $|\psi_x\rangle$. The unitary oracle T introduces auxiliary n qubits to compute the corresponding phase parameter $|x\rangle$ of the Fourier basis state $|\psi_x\rangle$ without erasing $|\psi_x\rangle$. All these auxiliary qubits are not depicted in Fig. 4 since they are reset back to $|0\rangle$ after the computation.

We can perform approximate computations for oracles V and T to achieve a lower circuit depth. Oracle V can be approximated by ignoring CR_m gates of larger m. Oracle T can be approximated by performing quantum measurements followed by classical post-processing on measurement outcomes [28]. Let unitary V' and T' be the approximation of V and T respectively, the corresponding program S_{AQFT} is almost the same as S_{QFT} but with oracles V and T replaced by V' and T' respectively. Next, we use our logic to reason the approximate equivalence between programs S_{QFT} and S_{AQFT}. That is,

$$S_{\text{QFT}} \sim_{\delta_1+2\delta_2} S_{\text{AQFT}} : \equiv_{(\bar{q}_0,\bar{q}_0')} \otimes |0\rangle\langle 0|_{aux} \Rightarrow \equiv_{(\bar{q}_0,\bar{q}_0')} \otimes |0\rangle\langle 0|_{aux} \tag{7}$$

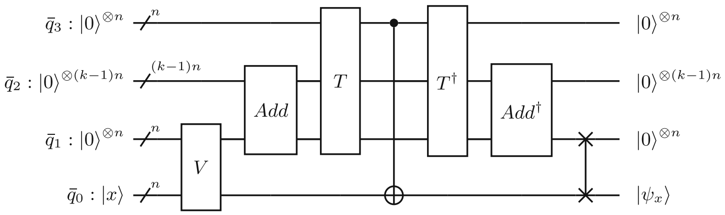

$\bar{q}_3 : |0\rangle^{\otimes n}$
n
$\bar{q}_2 : |0\rangle^{\otimes (k-1)n}$
$(k-1)n$
$\bar{q}_1 : |0\rangle^{\otimes n}$
n
$\bar{q}_0 : |x\rangle$
n
V
Add
T
$T^\dagger$
$Add^\dagger$
$|0\rangle^{\otimes n}$
$|0\rangle^{\otimes (k-1)n}$
$|0\rangle^{\otimes n}$
$|\psi_x\rangle$

Fig. 4. QFT circuit in [16]. Given a computational basis state $|x\rangle$ and corresponding Fourier basis state $|\psi_x\rangle$, unitary V performs mapping $|x\rangle|0\rangle^{\otimes n} \mapsto |x\rangle|\psi_x\rangle$, unitary Add performs mapping $|\psi_x\rangle|0\rangle^{\otimes n} \cdots |0\rangle^{\otimes n} \mapsto |\psi_x\rangle|\psi_x\rangle \cdots |\psi_x\rangle$, and unitary T performs mapping $|\psi_x\rangle \cdots |\psi_x\rangle|0\rangle^{\otimes n} \mapsto |\psi_x\rangle \cdots |\psi_x\rangle|x\rangle$.

$$\left\{ \equiv_{(\bar{q}_0,\bar{q}'_0)} \otimes |0\rangle\langle 0|_{(\bar{q}_1,\bar{q}_2,\bar{q}_3,\bar{q}'_1,\bar{q}'_2,\bar{q}'_3,\bar{r})} \right\} // P_0$$

$$(\bar{q}_0, \bar{q}_1) := V[(\bar{q}_0, \bar{q}_1)]; \sim_{\delta_1} (\bar{q}'_0, \bar{q}'_1) := V'[(\bar{q}'_0, \bar{q}'_1)];$$

$$\{ \ (V \otimes V) P_0 (V^\dagger \otimes V^\dagger) \ \} // P_1$$

$$(\bar{q}_1, \bar{q}_2) := Add[(\bar{q}_1, \bar{q}_2)]; \ \sim \ (\bar{q}'_1, \bar{q}'_2) := Add[(\bar{q}'_1, \bar{q}'_2)];$$

$$\{ \ (Add \otimes Add) P_1 (Add^\dagger \otimes Add^\dagger) \ \} // P_2$$

$$(\bar{q}_1, \bar{q}_2, \bar{q}_3) = T[(\bar{q}_1, \bar{q}_2, \bar{q}_3)]; \sim_{\delta_2} (\bar{q}'_1, \bar{q}'_2, \bar{q}'_3) = T'[(\bar{q}'_1, \bar{q}'_2, \bar{q}'_3)];$$

$$\{ \ (T \otimes T) P_2 (T^\dagger \otimes T^\dagger) \ \} // P_3$$

$$(\bar{q}_3, \bar{q}_0) := CNOT(\bar{q}_3, \bar{q}_0); \ \sim \ (\bar{q}_3, \bar{q}_0) := CNOT(\bar{q}'_3, \bar{q}'_0);$$

$$\{ \ (CNOT \otimes CNOT) P_3 (CNOT \otimes CNOT)^\dagger \ \} // P_4$$

$$(\bar{q}_1, \bar{q}_2, \bar{q}_3) := T^\dagger[(\bar{q}_1, \bar{q}_2, \bar{q}_3)]; \sim_{\delta_2} (\bar{q}'_1, \bar{q}'_2, \bar{q}'_3, \bar{r}) := T^\dagger[(\bar{q}'_1, \bar{q}'_2, \bar{q}'_3, \bar{r})];$$

$$\{ \ (T^\dagger \otimes T^\dagger) P_4 (T \otimes T) \ \} // P_5$$

$$(\bar{q}_1, \bar{q}_2) := Add^\dagger[(\bar{q}_1, \bar{q}_2)]; \ \sim \ (\bar{q}'_1, \bar{q}'_2) := Add^\dagger[(\bar{q}'_1, \bar{q}'_2)];$$

$$\{ \ (Add^\dagger \otimes Add^\dagger) P_5 (Add \otimes Add) \ \} // P_6$$

$$(\bar{q}_0, \bar{q}_1) := SWAP(\bar{q}_0, \bar{q}_1); \ \sim \ (\bar{q}'_0, \bar{q}'_1) := SWAP(\bar{q}'_0, \bar{q}'_1);$$

$$\{ \ (SWAP \otimes SWAP) P_6 (SWAP^\dagger \otimes SWAP^\dagger) \ \} // P_7 = P_0$$

Fig. 5. Proof sketch for programs S_{QFT} and S_{AQFT}. To easily refer to predicates, we label each assertion a name $//P_i$ on its right.

where $\delta = n\pi 2^{-k-1} + 2ne^{-k/8}$, $|0\rangle\langle 0|_{aux}$ denotes the tensor product of constant projections $|0\rangle\langle 0|$ over all qubits in other registers except $\bar{q}_0$ and $\bar{q}'_0$. The main proof sketch is shown in Fig. 5.

Create the Fourier Basis State. The computation of unitary U in Eq. 6 can be parallelized by individually preparing every $|\mu_\theta\rangle$ by the following unitary

$$Q_{t,i} : |0\rangle^{\otimes t}|x_1 \ldots x_n\rangle \mapsto |\mu_{0.x_i \ldots x_{i+t-1}}\rangle|0\rangle^{\otimes t-1}|x_1 \ldots x_n\rangle$$

in [16], where $i+t-1 \leq n$, qubits $x_1 \dots x_{i-1}$ and $x_{i+t} \dots x_n$ in $|x\rangle$ are not used. The unitary $Q_{t,i}$ acting on register $(\bar{q}, \bar{p})$ can be denoted as,

$$U_{GHZ}[\bar{q}]; CR_1[(\bar{p}[i], \bar{q}[1])]; \dots; CR_t[(\bar{p}[i+t-1], \bar{q}[t])]; U_{GHZ}^{\dagger}[\bar{q}]; H[\bar{q}[1]]$$

where U_{GHZ} denotes the unitary that generates a GHZ state, that is, $U_{GHZ}|0\rangle^{\otimes t} = (|0\rangle^{\otimes t} + |1\rangle^{\otimes t})/\sqrt{2}$. Registers $\bar{q}$ and $\bar{p}$ are of size t and n, respectively. $\bar{q}[i]$ denotes the i-th qubit in register $\bar{q}$. For example, Fig. 6 in [16] represents the circuit of unitary $Q_{4,i}$ on $|x\rangle$. Similar to the approximation in [17], unitary $Q_{t,i}$ can be approximated by ignoring CR_m gates of large m. That is, we could use $Q_{t,i}$ to approximate $Q_{t',i}$ if $1 \leq t < t' \leq n$. The approximation can be modeled by the judgment

$$\begin{gathered} \vdash (\bar{q}, \bar{p}) := Q_{t,i}[(\bar{q}, \bar{p})] \sim_{\delta(t,t')} (\bar{q}', \bar{p}') := Q_{t',i}[(\bar{q}', \bar{p}')] : \\ |0\rangle\langle 0|_{(\bar{q},\bar{q}')} \otimes \equiv_{(\bar{p},\bar{p}')} \Rightarrow \equiv_{(\bar{q}[1],\bar{q}'[1])} \otimes |0\rangle\langle 0|_{(\bar{q}[2,n],\bar{q}'[2,n])} \otimes \equiv_{(\bar{p},\bar{p}')} \end{gathered} \tag{8}$$

with $\delta(t,t') = \frac{1}{2} \sin \pi (2^{-t} - 2^{-t'})$. $P_{\bar{q}}$ denotes a projection P over the register $\bar{q}$. Particularly, $|\psi\rangle\langle\psi|_{\bar{q}}$ denotes the tensor product of $|\psi\rangle\langle\psi|$ over all qubits in register $\bar{q}$.

Figure 7 illustrates the circuit of the oracle V over register $(\bar{q}_0, \bar{r}, \bar{q}_1)$. To prepare each $|\mu_\theta\rangle$ in $|\psi_x\rangle$ individually, we need to prepare n copies of state $|x\rangle$ beforehand, which is achieved by the unitary C. The unitary C can be implemented by $CNOT$ gates in a binary tree architecture to achieve a circuit depth of $\log n$. To make it concise, the auxiliary qubits $q[2,n]$ in oracle $Q_{t,i}(\bar{q}, \bar{p})$ that reset back to $|0\rangle$ are ignored in Fig. 7 and the input of $Q_{t,i}$ is set as $|0\rangle|x\rangle$. The circuit for oracle V' is almost the same as Fig. 7 except that $Q_{t,i}$ is approximated by $Q_{k,i}$, where k $(0 < k < t \leq n)$ denotes the number of significant phase shift gates. Specifically, oracles V and V' can be represented as follows,

$$\begin{aligned} V &= C[(\bar{q}_0, \bar{r})]; Q_{1,n}[(\bar{q}_1[1], \bar{q}_0)]; Q_{2,n-1}[(\bar{q}_1[2], \bar{r}_1)]; \dots; Q_{n,1}[(\bar{q}_1[n], \bar{r}_{n-1})]; C^{\dagger}[(\bar{q}_0, \bar{r})] \\ V' &= C[(\bar{q}_0', \bar{r}')]; Q_{1,n}[(\bar{q}_1'[1], \bar{q}_0')]; Q_{2,n-1}[(\bar{q}_1'[2], \bar{r}_1')]; \dots; Q_{k,n-k+1}[(\bar{q}_1'[k], \bar{r}_{k-1}')]; \\ &\quad Q_{k,n-k+1}[(\bar{q}_1'[k+1], \bar{r}_k')]; \dots; Q_{k,n-k+1}[(\bar{q}_1'[n], \bar{r}_{n-1}')]; C^{\dagger}[(\bar{q}_0', \bar{r}')] \end{aligned}$$

where register $\bar{r} = \{\bar{r}_1, \dots, \bar{r}_{n-1}\}$ contains $n-1$ registers $\bar{r}_i$ initialized with $|0\rangle^{\otimes^n}$. Based on judgement 8, we have the following judgment

$$\vdash (\bar{q}_0, \bar{q}_1) := V[(\bar{q}_0, \bar{q}_1)] \sim_{\delta_1} (\bar{q}_0', \bar{q}_1') := V'[(\bar{q}_0', \bar{q}_1')] : P_0 \Rightarrow P_1 \tag{9}$$

where $\delta_1 = \sum_{i=k+1}^{n} \delta(k,i) = \frac{1}{2} \sum_{i=k+1}^{n} \sin \pi (2^{-k} - 2^{-i}) \leq n\pi 2^{-k-1}$. Notice that every register $\bar{r}_i$ in Fig. 7 is reset back to $|0\rangle$, thus the predicate $|0\rangle\langle 0|_{(\bar{r},\bar{r}')}$ on register $(\bar{r}, \bar{r}')$ can be ignored.

Replicate & Erase Fourier Basis State. We provide a brief overview of the functionality of the oracle *Add* as described in [16]. We begin with the state $|\psi_x\rangle|0\rangle^{\otimes^n} \cdots |0\rangle^{\otimes^n}$ and apply Hadamard gates $H^{\otimes^n}$ to each $|0\rangle^{\otimes^n}$, resulting in $|\psi_x\rangle|\psi_0\rangle \cdots |\psi_0\rangle$. Then, we apply telescoping subtraction $|x_1\rangle|x_2\rangle \cdots |x_k\rangle \rightarrow$

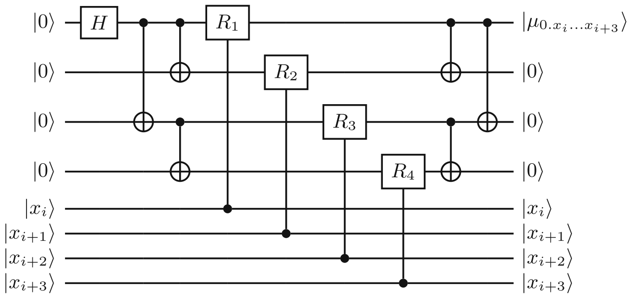
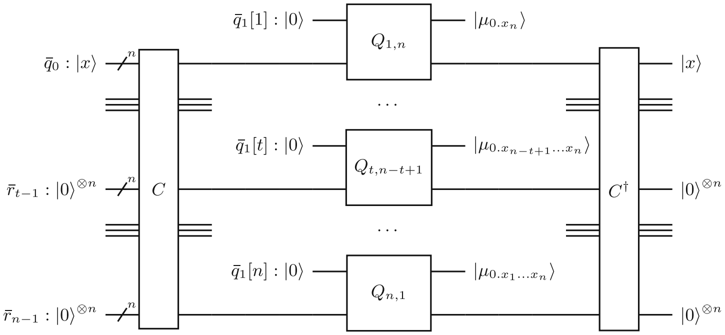


Fig. 6. Circuit for oracle $Q_{4,i}$ on state $|x_1 \ldots x_n\rangle$. Qubits $x_1 \ldots x_{i-1}$ and $x_{i+4} \ldots x_n$ are not used and ignored.



$\bar{q}_0 : |x\rangle$
$\bar{r}_{t-1} : |0\rangle^{\otimes n}$
$\bar{r}_{n-1} : |0\rangle^{\otimes n}$
n
n
n
C
$\bar{q}_1[1] : |0\rangle$
$Q_{1,n}$
$|\mu_{0.x_n}\rangle$
$\bar{q}_1[t] : |0\rangle$
$Q_{t,n-t+1}$
$|\mu_{0.x_{n-t+1}\ldots x_n}\rangle$
$\bar{q}_1[n] : |0\rangle$
$Q_{n,1}$
$|\mu_{0.x_1\ldots x_n}\rangle$
$C^\dagger$
$|x\rangle$
$|0\rangle^{\otimes n}$
$|0\rangle^{\otimes n}$

Fig. 7. Circuit for oracle V. Given a computational basis state $|x\rangle = |x_1 \ldots x_n\rangle$, unitary C performs the mapping $|x\rangle|0\rangle^{\otimes^n} \cdots |0\rangle^{\otimes^n} \mapsto |x\rangle^{\otimes^n}$, and unitary $Q_{t,i}$ performs the mapping $|0\rangle|x\rangle \mapsto |\mu_{0.x_i \ldots x_{i+t-1}}\rangle|x\rangle$.

$|x_1\rangle|x_2 - x_1\rangle \cdots |x_k - x_{k-1}\rangle$ to obtain $|\psi_x\rangle|\psi_x\rangle \ldots |\psi_x\rangle$. Reversely, we can use prefix addition $|x_1\rangle|x_2\rangle \cdots |x_k\rangle \to |x_1\rangle|x_1 + x_2\rangle \cdots |x_1 + x_2 + \cdots + x_k\rangle$ to eliminate the duplicates of the Fourier basis state. A $\log(k)$-depth tree of 3-2 adders can be used to generate two encoded numbers, followed by a quantum carry-lookahead adder of $\log(n)$-depth to add the encoded numbers. Since programs S_{AFT} and S_{QAFT} share the same procedure to replicate and erase Fourier basis states, we simplify replicating and erasing procedures by treating them as quantum oracles *Add* and *Add*† respectively. Then we use rule [UT] to get the following judgment.

$$\vdash (\bar{q}_1, \bar{q}_2) := Add[(\bar{q}_1, \bar{q}_2)] \sim (\bar{q}_1', \bar{q}_2') := Add[(\bar{q}_1', \bar{q}_2')] : P_1 \;\Rightarrow\; P_2 \tag{10}$$

$$\vdash (\bar{q}_1, \bar{q}_2) := Add^\dagger[(\bar{q}_1, \bar{q}_2)] \sim (\bar{q}_1', \bar{q}_2') := Add^\dagger[(\bar{q}_1', \bar{q}_2')] : P_5 \Rightarrow\; P_6 \tag{11}$$

Estimate the Phase of a Fourier State. The key to this step is based on the idea [3] that quantum measurement can be simulated by unitaries with the help of ancillary qubits. As shown in Fig 4, the oracle T generates the phase $|x\rangle$ in register $\bar{q}_3$ of the Fourier state $|\psi_x\rangle$, then the Fourier basis state $|x\rangle$ in register

$\bar{q}_0$ can be erased by the following $CNOT$ gate ($CNOT|x\rangle|x\rangle = |x\rangle|0\rangle$). The gate $T^\dagger$, the reverse of T, is applied subsequently to restore the state to the duplicates of $|\psi_x\rangle$. Given the input $|x\rangle$ in register $\bar{q}_0$, the whole process of erasing $|x\rangle$ works as $|x\rangle|\psi_x\rangle\cdots|\psi_x\rangle|0\rangle^{\otimes^n} \xrightarrow{T} |x\rangle|\psi_x\rangle\cdots|\psi_x\rangle|x\rangle \xrightarrow{CNOT} |0\rangle^{\otimes^n}|\psi_x\rangle\cdots|\psi_x\rangle|x\rangle \xrightarrow{T^\dagger} |0\rangle^{\otimes^n}|\psi_x\rangle\cdots|\psi_x\rangle|0\rangle^{\otimes^n}$ where the auxiliary register $\bar{q}_3$ is initialized with $|0\rangle^{\otimes^n}$ and reset back to $|0\rangle^{\otimes^n}$.

In order to reduce the circuit depth of oracle T, [16] parallelized the phase estimation procedure proposed by [29]. Given k copies of each $|\mu_{x2^{-i}}\rangle$, we perform two single-qubit measurements

$$\mathcal{M}_1 = \{M_1^0 = |\mu_0\rangle\langle\mu_0|, M_1^1 = |\mu_{\frac{1}{2}}\rangle\langle\mu_{\frac{1}{2}}|\} \quad \mathcal{M}_2 = \{M_2^0 = |\mu_{\frac{1}{4}}\rangle\langle\mu_{\frac{1}{4}}|, M_2^1 = |\mu_{\frac{3}{4}}\rangle\langle\mu_{\frac{3}{4}}|\}$$

on $k/2$ of the copies independently, where $\{|\mu_0\rangle, |\mu_{\frac{1}{2}}\rangle\}$ and $\{|\mu_{\frac{1}{4}}\rangle, |\mu_{\frac{3}{4}}\rangle\}$ are the eigenvectors of Pauli operators X and Y respectively. These measurements on copies of $|\psi_x\rangle$ would generate a distribution $\{p_{(x,i)}\}$ over a nk-bit string $|m_{(x,i)}\rangle$ of measurement outcomes. Then a reversible classical processing f is applied to infer x_i' based on measurement outcome $|m_{(x,i)}\rangle$, that is $|m_{(x,i)}\rangle|0\rangle \to |m_{(x,i)}\rangle|x_i'\rangle$, where the probability $p_{(x,i)}$ is close to 1 if $|x_i'\rangle = |x\rangle$, and a properly estimated $|x_i'\rangle$ can be used to erase the phase $|x\rangle$ on register $\bar{q}_0$. The following lemma is proved using Chernoff bound.

Lemma 4. [16] *Given any computational basis $|x\rangle$, measuring observables X and Y randomly generates a distribution $\{p_{(x,i)}\}$ over $\{|m_{(x,i)}\rangle\}$, followed by a classical processing that generates phase $|x_i'\rangle$ from $|m_{(x,i)}\rangle$. We have $Pr(|x_i'\rangle = |x\rangle) = p_{(x,i)} > 1 - 4ne^{-k/8}$.*

We can convert the above whole process into a unitary operation T' without actual measurements that can operate on data in superposition. First, the following unitary $U_M([\bar{q}_1, \bar{q}_2, \bar{r}])$,

$$U_M([\bar{q}_1, \bar{q}_2, \bar{r}]) := \otimes_{i=1}^n(\otimes_{j=1}^{k/2}U_X[(r[ik+j], p[ik+j])])\otimes(\otimes_{j=1+k/2}^{k}U_Y[(r[ik+j], p[ik+j])])$$

is applied to simulate measurements on copies of $|\psi_x\rangle$, where register $\bar{p} = \{\bar{q}_1, \bar{q}_2\}$ and auxiliary register $\bar{r}$ is initialized with $|0\rangle$. Unitary gates U_X and U_Y

$$U_X[(q_1, q_2)] := (H[q_1] \otimes I[q_2])CNOT[(q_1, q_2)](H[q_1] \otimes I[q_2]);$$
$$U_Y[(q_1, q_2)] := (H[q_1] \otimes I[q_2])CY[(q_1, q_2)](H[q_1] \otimes I[q_2])$$

introduce auxiliary qubit q_1 initialized with $|0\rangle$ to simulate single-qubit measurements $\mathcal{M}_1$ and $\mathcal{M}_2$ on $|\mu_{x2^{-i}}\rangle$ in qubit q_2.

$$|0\rangle|\mu_{x2^{-i}}\rangle \xrightarrow{U_X} \langle\mu_0|\mu_{x2^{-i}}\rangle \cdot |0\rangle|\mu_0\rangle + \langle\mu_{\frac{1}{2}}|\mu_{x2^{-i}}\rangle \cdot |1\rangle|\mu_{\frac{1}{2}}\rangle$$

$$|0\rangle|\mu_{x2^{-i}}\rangle \xrightarrow{U_Y} \langle\mu_{\frac{1}{4}}|\mu_{x2^{-i}}\rangle \cdot |0\rangle|\mu_{\frac{1}{4}}\rangle + \langle\mu_{\frac{3}{4}}|\mu_{x2^{-i}}\rangle \cdot |1\rangle|\mu_{\frac{3}{4}}\rangle$$

where $CY[(q_1, q_2)]$ denotes the controlled Pauli Y gate. Next, we set the outputs of auxiliary register $\bar{r}$ of U_M to be the input of oracle $O[(\bar{r}, \bar{q}_3)]$ such that

$$U_M|\mu_{x2^{-i}}\rangle \otimes |0\rangle \xrightarrow{O} \textstyle\sum_i \sqrt{p_{(x,i)}}|\varphi\rangle \otimes |x_i'\rangle$$

where oracle O denotes the corresponding quantum circuit of the classical processing f on measurement outcomes. Thus, the oracle T' can achieved by $U_M[(\bar{q_1}, \bar{q_2}, \bar{r})]$ and $O[(\bar{r}, \bar{q_3})]$ sequentially. By lemma 4, we would have

$$\begin{array}{l} \vdash (\bar{q_1}, \bar{q_2}, \bar{q_3}) := T[(\bar{q_1}, \bar{q_2}, \bar{q_3})] \sim_{\delta_2} (\bar{q_1'}, \bar{q_2'}, \bar{q_3'}, \bar{r}) := T'[(\bar{q_1'}, \bar{q_2'}, \bar{q_3'}, \bar{r})] : P_2 \;\Rightarrow\; P_3 \\ \vdash (\bar{q_1}, \bar{q_2}, \bar{q_3}) := T^{\dagger}[(\bar{q_1}, \bar{q_2}, \bar{q_3})] \sim_{\delta_2} (\bar{q_1'}, \bar{q_2'}, \bar{q_3'}, \bar{r}) := T'^{\dagger}[(\bar{q_1'}, \bar{q_2'}, \bar{q_3'}, \bar{r})] : P_4 \Rightarrow P_5 \end{array} \tag{12}$$

where $\delta_2 = 2ne^{-k/8}$.

Conclusion Finally, we use rule [SEQ] to sum up all judgments to get Eq. 7.

7 Measurements Conditions and Additional Proof Rules

7.1 Measurement Conditions

Additional constraints must be imposed on programs to establish feasible relational proof rules for those with complex structures, such as if and loop statements. In the classical pRHL approach in [7], the precondition $m_1 \Psi m_2$ satisfied by the initial memories m_1 and m_2 requires the guards e_1 and e_2 in the if or loop statements must be equal. Things get more complex in quantum programs since quantum mechanics are naturally probabilistic, and it is generally impossible to require two if statements to give the same measurement or with the same probability distributions. In [8], the term "synchronous execution" in quantum programs means that two quantum measurements $\mathcal{M}_1 = \{M_1^m\}$ and $\mathcal{M}_2 = \{M_2^m\}$ should produce the same distribution over branches for input ρ_1 and ρ_2, that is, $\mathrm{Tr}(M_1^m \rho_1 M_1^{m\dagger}) = \mathrm{Tr}(M_2^m \rho_2 M_2^{m\dagger})$. To study more general programs, we propose the approximate measurement conditions, which establish appropriate upper bounds for the deviations in our judgments.

Definition 10 (Approximate Measurement Condition). *Let $\mathcal{M}_1 = \{M_1^m\}$ and $\mathcal{M}_2 = \{M_2^m\}$ be two measurements in $\mathcal{H}_1$ and $\mathcal{H}_2$ that share the same set $\{m\}$ of measurement outcomes, respectively. The measurement condition*

$$\mathcal{M}_1 \approx_{\{\delta_m\}} \mathcal{M}_2 : A \Rightarrow \{(p_m, B_m)\} \tag{13}$$

means that for every measurement outcome m, we have

$$\forall \rho_1,\ \rho_2. \rho_1 \sim_A \rho_2 \ \Rightarrow \begin{cases} M_1^m \rho_1 M_1^{m\dagger} \sim_{B_m}^{\delta_m} M_2^m \rho_2 M_2^{m\dagger} \\ \max\{\mathrm{Tr}(M_1^m \rho_1 M_1^{m\dagger}), \mathrm{Tr}(M_2^m \rho_2 M_2^{m\dagger})\} \leq p_m \end{cases}$$

where $p_m \in [0,1]$. Deviations δ_m in the measurement condition can be ignored if they equal zero. We write predicate $\{(p_m, B_m)\}$ as $\{B_m\}$ for short if all $p_m = 1$.

7.2 Additional Proof Rules

Now, we introduce the rules for if and loop statements in Fig. 8. The rule [IF] requires the measurement condition in the premises to provide a bound on the whole approximation, where the deviation δ'_m of the branch body is scaled down by p_m. The rule [LP] does not require the synchronous execution of loop guards [8,9], or the speed bound at which loops converge [26]. Instead, the measurement condition only employs an upper bound p_1 on the probabilities of entering loop bodies for the first iteration. Rule [LP] requires $p_1 \in [0, 1)$ and provides better deviation if p_1 is smaller. If p_1 equals one initially, we can unroll loop statements several times to make p_1 less than one.

We derive rule [LP*] as an alternative to rule [LP] by incorporating more specific measurement conditions for the iterations of loops when we can not find a good A for rule [LP]. When doing approximate reasoning about loops, it is typical to set an upper bound N on the number of iterations. Notice that the factor λ_n is not an upper bound on the probability of entering $(n+1)$-th iteration except for $n = 0$. Overall, rule [LP*] is a direct application of rule [Seq] on a finite number of iterations, where measurement conditions are used to scale the deviations. Since we can always make the **skip** statement share the same probability distribution with any **if** and **while** statements, the measurement conditions for one-side rules [IF-L], [LP-L], and [LP*-L] are more straightforward.

$$\text{(IF)}\quad \frac{\mathcal{M}_1 \approx_{\{\delta_m\}} \mathcal{M}_2 : A \Rightarrow \{(p_m, B_m)\} \qquad \vdash S_{1m} \sim_{\delta'_m} S_{2m} : B_m \Rightarrow C}{\vdash \textbf{if } (\Box m \cdot \mathcal{M}_1[\bar{q}] = m \rightarrow S_{1m}) \textbf{ fi} \sim_{\sum_m \delta_m + p_m \cdot \delta'_m} \textbf{if } (\Box m \cdot \mathcal{M}_2[\bar{q}] = m \rightarrow S_{2m}) \textbf{ fi} : A \Rightarrow C}$$

$$\text{(LP)}\quad \frac{\mathcal{M}_1 \approx_{\{\delta_0, \delta_1\}} \mathcal{M}_2 : A \Rightarrow \{(p_0, B_0), (p_1, B_1)\} \qquad \vdash S_1 \sim_{\delta} S_2 : B_1 \Rightarrow A}{\vdash \textbf{while } \mathcal{M}_1[\bar{q}] = 1 \textbf{ do } S_1 \textbf{ od} \sim_{\frac{\delta_0 + \delta_1 + p_1\delta}{1 - p_1}} \textbf{while } \mathcal{M}_2[\bar{q}] = 1 \textbf{ do } S_2 \textbf{ od} : A \Rightarrow B_0}$$

$$\text{(LP*)}\quad \frac{\begin{array}{c}\forall 0 \le k < N.\, \mathcal{M}_1 \approx_{\{\alpha_k, \beta_k\}} \mathcal{M}_2 : A_k \Rightarrow \{(q_k, C), (p_k, B_k)\} \\ \mathcal{M}_1 \approx_{\{\alpha_N, 0\}} \mathcal{M}_2 : A_N \Rightarrow \{(1, C), (0, I)\} \qquad \vdash S_1 \sim_{\delta_k} S_2 : B_k \Rightarrow A_{k+1}\end{array}}{\vdash \textbf{while } \mathcal{M}_1[\bar{q}] = 1 \textbf{ do } S_1 \textbf{ od} \sim_{f(\alpha_k, \beta_k, p_k)} \textbf{while } \mathcal{M}_2[\bar{q}] = 1 \textbf{ do } S_2 \textbf{ od} : A_0 \Rightarrow C}$$

$$\text{(IF-L)}\quad \frac{\mathcal{M}_1 \approx I_2 : A \Rightarrow \{(p_m, B_m)\} \qquad \vdash S_{1m} \sim_{\delta_m} \textbf{skip} : B_m \Rightarrow C}{\vdash \textbf{if } (\Box m \cdot \mathcal{M}_1[\bar{q}] = m \rightarrow S_{1m}) \textbf{ fi} \sim_{\sum_m p_m \delta_m} \textbf{skip} : A \Rightarrow C}$$

$$\text{(LP-L)}\quad \frac{\mathcal{M}_1 \approx I_2 : A \Rightarrow \{(p_0, B_0), (p_1, B_1)\} \qquad \vdash S_1 \sim_{\delta} \textbf{skip} : B_1 \Rightarrow A}{\vdash \textbf{while } \mathcal{M}_1[\bar{q}] = 1 \textbf{ do } S_1 \textbf{ od} \sim_{p_1\delta/(1-p_1)} \textbf{skip} : A \Rightarrow B_0}$$

$$\text{(LP*-L)}\quad \frac{\begin{array}{c}\forall 0 \le k < N.\, \mathcal{M}_1 \approx I_2 : A_k \Rightarrow \{(q_k, C), (p_k, B_k)\} \qquad \lambda_n = \prod_{k=0}^{n} p_k \\ \mathcal{M}_1 \approx_{\{\delta_N, 0\}} I_2 : A_N \Rightarrow \{(1, C), (0, I)\} \qquad \vdash S_1 \sim_{\delta_k} \textbf{skip} : B_k \Rightarrow A_{k+1}\end{array}}{\vdash \textbf{while } \mathcal{M}_1[\bar{q}] = 1 \textbf{ do } S_1 \textbf{ od} \sim_{\lambda_{N-1}\delta_N + \sum_{n=0}^{N-1}(N-n)\lambda_n\delta_n)} \textbf{skip} : A_0 \Rightarrow C}$$

Fig. 8. Rules for branching structure in aqRHL. The deviation of rule (LP*) is given by $f(\alpha_k, \beta_k, p_k) = (\alpha_0 + \sum_{n=0}^{N-1} \lambda_n \alpha_{n+1}) + (N\beta_0 + \sum_{n=0}^{N-2}(N-n-1)\lambda_n\beta_{n+1}) + (\sum_{n=0}^{N-1}(N-n)\lambda_n\delta_n)$ with $\lambda_n = \prod_{k=0}^{n} p_k$.

Structural Rules. Unlike classical programs, the potential quantum entanglement between subsystems brings a unique challenge in constructing a general frame rule for quantum programs [8,51,64]. We derive a simple frame rule [Frame] to specify a specific instance that the predicate C on additional independent system $(\bar{r}_1, \bar{r}_2)$ is one-dimensional. Subscripts related to registers are displayed explicitly for clarity. Rule [Order] adds an order relation $\leq$ over deviations. In addition, an additional condition is introduced in rule [Approx] to allow switching postconditions at the cost of bringing approximation (Fig. 9).

$$\text{(Frame)}\quad \frac{\vdash_{(\bar{q}_1,\bar{q}_2)} S_1 \sim_\delta S_2 : A \Rightarrow B \quad (\bar{r}_1,\bar{r}_2) \cap var(S_1,S_2) = 0 \quad Dim(C_{(\bar{r}_1,\bar{r}_2)}) = 1}{\vdash_{(\bar{q}_1,\bar{r}_1),(\bar{q}_2,\bar{r}_2)} S_1 \sim_\delta S_2 : A \otimes C_{(\bar{r}_1,\bar{r}_2)} \Rightarrow B \otimes C_{(\bar{r}_1,\bar{r}_2)}}$$

$$\text{(Order)}\quad \frac{\vdash S_1 \sim_{\delta'} S_2 : A' \Rightarrow B' \quad A \subseteq A' \quad B' \subseteq B \quad \delta' \leq \delta}{\vdash S_1 \sim_\delta S_2 : A \Rightarrow B}$$

$$\text{(Approx)}\quad \frac{\vdash S_1 \sim S_2 : A \Rightarrow B \qquad \forall \rho_1, \rho_2.\ \rho_1 \sim_B \rho_2 \Rightarrow \rho_1 \sim_C^{\delta} \rho_2}{\vdash S_1 \sim_\delta S_2 :\ A \Rightarrow C}$$

Fig. 9. Structural aqRHL rules.

8 Related and Future Works

With the fast development of quantum hardware [50], various quantum programming languages [1,4,19,20,22,47,48] have been proposed for more straightforward implementation of quantum algorithms. Very recently, significant efforts have been devoted to the research of quantum logic and quantum program analysis [14,23,43,49,55,56,63,65] for these emerging quantum programs.

Comparison with Quantitative Robustness Reasoning. [26] develops semantics for erroneous quantum while programs and logic to prove robustness between an ideal program and a noisy one. [66] derives applied quantum Hoare logic by employing projection as predicates and reasons about the robustness of quantum programs, i.e., error bounds of outputs. These two works focus on single-program executions, while our work studies relational reasoning. In particular, the major differences are as follows. a). Different formula: In the logic formula of [26,66], the predicate lives in the space of the principle program. The predicate of our logic lives in the joint space of the two programs. b). Different scope of applications: The proof systems developed by [26,66] focus on studying the robustness of quantum programs, i.e., equivalence or closeness. Our choice of relational Hoare logic can reason about general relations beyond equivalence or closeness. We can reason relational properties between programs with different numbers of qubits. c). Different proof rules: The proof rules of [26,66] discuss programs with the same syntax statement, while our one-side rules can track relational properties for different statements.

Comparison with Relational Quantum Hoare Logics. Our work is primarily inspired by the quantum relational Hoare logics recently proposed by [8,33,51].

In particular, [8] suggests that casting approximate reasoning into the general framework of relational quantum Hoare logic remains open. Generally, two quantum while programs do not share the same probabilities for taking different paths or outcomes during their execution. Under such circumstances, exact quantum couplings cannot be found, as they only exist for partial density operators with identical trace. This mathematical condition significantly restricts the flexibility of the exact quantum relational Hoare logic. Our work provides a promising solution to this open question. In particular, by introducing approximate quantum coupling, our logic system offers a more general scope of applications. Our logic, aqRHL, is a quantum counterpart to apRHL [9], even from a technical point of view: aqRHL employs projective predicates [12] over the joint systems of the programs, a natural quantum counterpart of binary relations, the predicates used in apRHL.

Future Work. There are several promising directions for future work. Firstly, we would like to extend our theory to the hybrid system, i.e., programs with quantum and classical variables. Hybrid quantum-classical systems allow for the exploitation of quantum advantages while leveraging the existing classical computing infrastructure. A unified language incorporating both quantum and classical effects may offer advantages in analyzing hybrid programs [52]. Secondly, we will investigate the potential applications of the newly developed approximate relational quantum Hoare logic. Particularly, we are interested in applying it to the construction and verification of quantum cryptographic proofs and ensuring the correctness of optimized quantum compilers specifically designed for NISQ (Noisy Intermediate-Scale Quantum) devices. Lastly, it is interesting to incorporate recently developed tools such as quantum abstract interpretation [62] and quantum separation logic [64] to design over-approximation techniques [58]. Another interesting technique is Context-Free-Language Ordered Binary Decision Diagrams [46], which may serve as a backend representation and manipulation technique in studying quantum Hoare logics.

9 Conclusion

We resolve the open question of [8] by designing an approximate relational Hoare logic for robustly reasoning the relational properties of two programs. We showcase the success of our methodology by formally verifying the well-known low-depth approximation of the quantum Fourier transform, and the correctness of the repeat-until-success algorithm and bit flip code.

Acknowledgement. We thank our anonymous referees for their comments and suggestions on earlier versions of this paper. Hanru Jiang acknowledges the support from the National Natural Science Foundation of China under Grant No. 62202265, and the Beijing Natural Science Foundation under Grant No. Z220002.

References

1. Abhari, A.J., et al.: Scaffold: quantum programming language. Princeton University NJ Department of Computer Science, Tech. rep. (2012)
2. Aharonov, D., Kitaev, A., Nisan, N.: Quantum circuits with mixed states (1998). https://doi.org/10.48550/ARXIV.QUANT-PH/9806029, https://arxiv.org/abs/quant-ph/9806029
3. Aharonov, D., Kitaev, A., Nisan, N.: Quantum circuits with mixed states. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, pp. 20-30. STOC '98, Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/276698.276708
4. Aleksandrowicz, G., et al.: Qiskit: an open-source framework for quantum computing (2019). https://doi.org/10.5281/zenodo.2562111
5. Anil Kumar, V., Ramesh, H.: Coupling vs. conductance for the jerrum-sinclair chain*. Random Struct. Algorithms **18**(1), 1–17 (2001). https://doi.org/10.1002/1098-2418(200101)18:11::AID-RSA13.0.CO;2-7
6. Badihi, S., Akinotcho, F., Li, Y., Rubin, J.: ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 13–24. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409757
7. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 90–101. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480894
8. Barthe, G., Hsu, J., Ying, M., Yu, N., Zhou, L.: Relational proofs for quantum programs. Proc. ACM Program. Lang. **4**(POPL) (2019). https://doi.org/10.1145/3371089
9. Barthe, G., Köpf, B., Olmedo, F., Zanella-Béguelin, S.: Probabilistic relational reasoning for differential privacy. ACM Trans. Program. Lang. Syst. **35**(3) (2013). https://doi.org/10.1145/2492061
10. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 14–25. POPL '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/964001.964003
11. Bergstra, J., Tiuryn, J., Tucker, J.: Floyd's principle, correctness theories and program equivalence. Theor. Comput. Sci. **17**(2), 113–149 (1982). https://doi.org/10.1016/0304-3975(82)90001-9, https://www.sciencedirect.com/science/article/pii/0304397582900019
12. Birkhoff, G., Neumann, J.V.: The logic of quantum mechanics. Ann. Math. **37**(4), 823–843 (1936). http://www.jstor.org/stable/1968621
13. Bocharov, A., Roetteler, M., Svore, K.M.: Efficient synthesis of universal repeat-until-success quantum circuits. Phys. Rev. Lett. **114**, 080502 (2015). https://doi.org/10.1103/PhysRevLett.114.080502
14. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L., Yen, D.D.: An automata-based framework for verification and bug hunting in quantum circuits. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591270

15. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1027–1040. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314596
16. Cleve, R., Watrous, J.: Fast parallel circuits for the quantum fourier transform. In: Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 526–536. IEEE Computer Society, Redondo Beach, CA, USA (2000). https://doi.org/10.1109/SFCS.2000.892140
17. Coppersmith, D.: An approximate fourier transform useful in quantum factoring (2002). https://doi.org/10.48550/arxiv.quant-ph/0201067
18. Cousineau, G., Enjalbert, P.: Program equivalence and provability. In: Bečvář, J. (ed.) Mathematical Foundations of Computer Science, pp. 237–245. Springer, Berlin, Heidelberg (1979). https://doi.org/10.1007/3-540-09526-8_20
19. Developers, C.: Cirq (2021). https://doi.org/10.5281/zenodo.5182845, See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors
20. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. SIGPLAN Not. **48**(6), 333–342 (2013). https://doi.org/10.1145/2499370.2462177
21. Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. Phys. Rev. Lett. **103**, 150502 (2009). https://doi.org/10.1103/PhysRevLett.103.150502
22. Heim, B., et al.: Quantum programming languages. Nat. Rev. Phys. **2**, 709–722 (2020). https://doi.org/10.1038/s42254-020-00245-7
23. Hietala, K., Rand, R., Hung, S.H., Wu, X., Hicks, M.: A verified optimizer for quantum circuits. Proc. ACM Program. Lang. **5**(POPL) (2021). https://doi.org/10.1145/3434318
24. Hsu, J.: Probabilistic couplings for probabilistic reasoning (2017)
25. Huang, Y., Martonosi, M.: Statistical assertions for validating patterns and finding bugs in quantum programs. In: Proceedings of the 46th International Symposium on Computer Architecture, pp. 541–553. ISCA '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3307650.3322213
26. Hung, S., Hietala, K., Zhu, S., Ying, M., Hicks, M., Wu, X.: Quantitative robustness analysis of quantum programs. Proc. ACM Program. Lang. **3**(POPL), 31:1–31:29 (2019). https://doi.org/10.1145/3290344
27. Kakutani, Y.: A logic for formal verification of quantum programs. In: Proceedings of the 13th Asian Conference on Advances in Computer Science: Information Security and Privacy, pp. 79–93. ASIAN'09, Springer-Verlag, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10622-4_7
28. Kitaev, A.Y., Shen, A.H., Vyalyi, M.N.: Classical and Quantum Computation. American Mathematical Society, USA (2002). https://doi.org/10.1090/gsm/047
29. Kitaev, A.Y.: Quantum measurements and the abelian stabilizer problem. Electron. Colloquium Comput. Complex. **TR96-003** (1996). https://eccc.weizmann.ac.il/eccc-reports/1996/TR96-003/index.html
30. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. SIGPLAN Not. **44**(6), 327–337 (2009). https://doi.org/10.1145/1543135.1542513
31. Lahiri, S.K., Sinha, R., Hawblitzel, C.: Automatic root causing for program equivalence failures in binaries. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification, pp. 362–379. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_21

32. Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y.: Projection-based runtime assertions for testing and debugging quantum programs. Proc. ACM Program. Lang. **4**(OOPSLA) (2020). https://doi.org/10.1145/3428218
33. Li, Y., Unruh, D.: Quantum relational hoare logic with expectations. In: Bansal, N., Merelli, E., Worrell, J. (eds.) 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 198, pp. 136:1–136:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ICALP.2021.136, https://drops.dagstuhl.de/opus/volltexte/2021/14205
34. Lloyd, S., Mohseni, M., Rebentrost, P.: Quantum principal component analysis. Nat. Phys. **10**(9), 631–633 (2014). https://doi.org/10.1038/nphys3029
35. Lucanu, D., Rusu, V.: Program equivalence by circular reasoning. Form. Asp. Comput. **27**(4), 701–726 (2015). https://doi.org/10.1007/s00165-014-0319-6
36. Ming, J., Zhang, F., Wu, D., Liu, P., Zhu, S.: Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. IEEE Trans Reliab. **65**(4), 1647–1664 (2016). https://doi.org/10.1109/TR.2016.2570554
37. Nechita, I., Puchała, Z., Pawela, L., Życzkowski, K.: Almost all quantum channels are equidistant. J. Math. Phys. **59**(5), 052201 (2018). https://doi.org/10.1063/1.5019322
38. Necula, G.C.: Translation validation for an optimizing compiler. SIGPLAN Not. **35**(5), 83–94 (2000). https://doi.org/10.1145/358438.349314
39. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press, New York, 10th edn. (2011). https://doi.org/10.1017/CBO9780511976667
40. Paetznick, A., Svore, K.M.: Repeat-until-success: non-deterministic decomposition of single-qubit unitaries. Quantum Info. Comput. **14**(15-16), 1277–1301 (2014). https://doi.org/10.26421/QIC14.15-16-2
41. Pitts, A.: Operationally-Based Theories of Program Equivalence, pp. 241–298. Publications of the Newton Institute, Cambridge University Press, Cambridge (1997). https://doi.org/10.1017/CBO9780511526619.007
42. Preskill, J.: Quantum computing in the NISQ era and beyond. Quantum **2**, 79 (2018). https://doi.org/10.22331/q-2018-08-06-79
43. Rand, R.: Verification logics for quantum programs (2019)
44. Rand, R., Paykin, J., Zdancewic, S.: QWIRE Practice: formal verification of quantum circuits in Coq. Electron. Proc. Theor. Comput. Sci. **266**, 119–132 (2018). https://doi.org/10.4204/eptcs.266.8
45. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. SFCS '94, IEEE Computer Society, USA (1994). https://doi.org/10.1109/SFCS.1994.365700
46. Sistla, M., Chaudhuri, S., Reps, T.: CFLOBDDs: context-free-language ordered binary decision diagrams (2023)
47. Smith, R.S., Curtis, M.J., Zeng, W.J.: A practical quantum instruction set architecture (2016). https://arxiv.org/abs/1608.03355
48. Svore, K., et al.: Q#: enabling scalable quantum computing and development with a high-level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. RWDSL2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3183895.3183901

49. Tao, R., et al.: Giallar: push-button verification for the Qiskit quantum compiler. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 641–656. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523431
50. Trabesinger, A.: Quantum computing: towards reality. Nature **543**, S1 (2017). https://doi.org/10.1038/543S1a
51. Unruh, D.: Quantum relational Hoare logic. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290346
52. Voichick, F., Li, L., Rand, R., Hicks, M.: Qunity: a unified language for quantum and classical computing. Proc. ACM Program. Lang. **7**(POPL) (2023). https://doi.org/10.1145/3571225
53. Watrous, J.: Simpler semidefinite programs for completely bounded norms. Chicago J. Theor. Comput. Sci. **2013**, 8 (2013). http://cjtcs.cs.uchicago.edu/articles/2013/8/contents.html
54. Weimer, W., Fry, Z.P., Forrest, S.: Leveraging program equivalence for adaptive program repair: models and first results. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 356–366. ASE '13, IEEE Press, Silicon Valley, CA, USA (2013). https://doi.org/10.1109/ASE.2013.6693094
55. Xu, A., Molavi, A., Pick, L., Tannu, S., Albarghouthi, A.: Synthesizing quantum-circuit optimizers. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591254
56. Xu, M., et al.: Quartz: superoptimization of quantum circuits. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 625–640. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523433
57. Yan, P., Jiang, H., Yu, N.: On incorrectness logic for quantum programs. Proc. ACM Program. Lang. **6**(OOPSLA1) (2022). https://doi.org/10.1145/3527316
58. Yang, H.: Relational separation logic. Theor. Comput. Sci. **375**(1), 308–334 (2007). https://doi.org/10.1016/j.tcs.2006.12.036, https://www.sciencedirect.com/science/article/pii/S0304397506009261, festschrift for John C. Reynolds's 70th birthday
59. Ying, M.: Floyd–Hoare logic for quantum programs. ACM Trans. Program. Lang. Syst. **33**(6) (2012). https://doi.org/10.1145/2049706.2049708
60. Ying, M.: Foundations of Quantum Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2016). https://doi.org/10.1016/C2014-0-02660-3
61. Ying, M., Duan, R., Feng, Y., Ji, Z.: Predicate Transformer Semantics of Quantum Programs, pp. 311–360. Cambridge University Press, Cambridge (2009). https://doi.org/10.1017/CBO9781139193313.009
62. Yu, N., Palsberg, J.: Quantum abstract interpretation. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 542–558. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454061
63. Yuan, C., McNally, C., Carbin, M.: Twist: sound reasoning for purity and entanglement in quantum programs. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498691

64. Zhou, L., Barthe, G., Hsu, J., Ying, M., Yu, N.: A quantum interpretation of bunched logic & quantum separation logic. In: Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science. LICS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1109/LICS52264.2021.9470673
65. Zhou, L., Barthe, G., Strub, P.Y., Liu, J., Ying, M.: CoqQ: foundational verification of quantum programs. Proc. ACM Program. Lang. **7**(POPL) (2023). https://doi.org/10.1145/3571222
66. Zhou, L., Yu, N., Ying, M.: An applied quantum Hoare logic. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1149–1162. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314584
67. Zhou, L., Yu, N., Ying, S., Ying, M.: Quantum earth mover's distance, a no-go quantum Kantorovich-Rubinstein theorem, and quantum marginal problem. J. Math. Phys. **63**(10), 102201 (2022). https://doi.org/10.1063/5.0068344

QReach: A Reachability Analysis Tool for Quantum Markov Chains

Aochu Dai[1] and Mingsheng Ying[1,2](✉)

[1] Department of Computer Science and Technology, Tsinghua University,
Beijing 100084, China
dac22@mails.tsinghua.edu.cn
[2] Institute of Software, Chinese Academy of Sciences,
Beijing 100190, China
yingms@ios.ac.cn

Abstract. We present QReach, the first reachability analysis tool for quantum Markov chains based on decision diagrams CFLOBDD (presented at *CAV* 2023). QReach provides a novel framework for finding reachable subspaces, as well as a series of model-checking subprocedures like image computation. Experiments indicate its practicality in verification of quantum circuits and algorithms. QReach is expected to play a central role in future quantum model checkers.

Keywords: Reachability analysis · Quantum Markov chain · Quantum Model Checking

1 Introduction

A rapid growth of quantum computing hardware has been witnessed in the last few years. As a recent breakthrough, IBM has introduced its new quantum processor Condor, which breaks the 1000-qubit barrier. Many researchers share the belief that quantum computation will be scalable and stable enough for some meaningful quantum algorithms in the foreseeable future. In the era of Fault-Tolerant Quantum Computing (FTQC), quantum systems may be too complicated to be designed and verified manually. On the other hand, systematic ventures occurring in a quantum circuit or a communication protocol may differ significantly from those in classical systems and may be counterintuitive. The success of model checking techniques in classical computing and communication industry motivates us to extend it for analysis and verification of various temporal properties of a quantum system. Indeed, several model checking algorithms have been proposed for quantum automata and quantum Markov chains [10,11,18]. Additionally, some basic communication protocols like BB84 have passed the verification of the proposed quantum model checkers [2]. However, these quantum model checkers cannot be applied to larger quantum systems.

As is well known, the scalability of classical model checkers heavily relies on the data structures (in particular, various DDs (Decision Diagrams), e.g.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 520–532, 2024.
https://doi.org/10.1007/978-3-031-65633-0_23

ROBDD) employed in them for representing the system under checking. Several quantum generalisations of DDs have been introduced for modelling, simulation, and verification of (combinational) quantum circuits, like QMDD [12], TDD [9], and LimDD [17], providing different degrees of compression for quantum states and operators. Based on these diagram structures, some simulation or verification tools were developed for experimental tasks like equivalence checking and bug finding [5,6]. Decision diagrams have well-defined canonicity and regularization, which motivates us to implement quantum model-checking algorithms by means of DDs. Up to now, however, these quantum DDs have not been used in quantum model checking.

In this paper, we incorporate quantum DDs into quantum model checking for the first time. Quantum Markov chains (QMCs for short) have been adopted as a fundamental model of many quantum information processing systems (e.g. quantum communication protocols, semantics of quantum programs, etc). So, we choose to use QMCs as our system model. As is well-known, reachability analysis is a core task in classical model checking algorithms. In the quantum case, indeed, reachability analysis has been applied in quantum communication, quantum control and termination analysis of quantum programs among many others. Therefore, we focus on the issue of reachability analysis of quantum Markov chains. In addition, we decide to use Context-Free-Language Ordered Binary Decision Diagrams [15] (CFLOBDD for short), one of the most efficient quantum DDs as the backend of our tool to provide support for functionalities. We also refer to Quasimodo [14], a quantum circuit simulator based on CFLOBDD, for some of the code's implementation details. Supported by the efficiency of the DD representation, our tool is well scalable and has the potential to be expanded into large-scale quantum circuit model checkers in the future.

Contributions of the Paper: This paper introduces the first reachability analysis tool for QMCs, called QReach[1]. It can efficiently compute reachable subspaces of QMCs with the following techniques:

- Subspaces of QMCs, which are usually defined by atomic propositions in Birkhoff-von Neumman quantum logic, are represented as CFLOBDDs in QReach.
- Partitioning and frontier set simplification are introduced in our algorithm to reduce the size of data structures, in analogy to corresponding techniques in classical symbolic model checking.

2 Quantum Reachability Analysis

For convenience of the reader, we briefly review the model of QMCs and their reachable subspaces. Recall that a Markov chain (MC for short) is a pair $\langle S, P\rangle$, where S is a finite set of states and P is a transition probability matrix $P : S \times S \to [0,1]$ satisfying a normalization condition $\sum_{s' \in S} P(s,s') = 1$ for any $s \in S$. Similarly, a QMC is defined as a pair $\langle \mathcal{H}, \mathcal{E}\rangle$, where $\mathcal{H}$ is the state Hilbert

[1] Available at https://github.com/Acdimy/qreach-tools.

space of the quantum system under consideration and $\mathcal{E}$ is a quantum operation describing the evolution of the system, i.e. a mapping from a quantum state ρ to another $\mathcal{E}(\rho)$ (also called a quantum channel in quantum information literature). Table 1 gives a detailed comparison between classical MCs and QMCs:

Table 1. Classical Markov chains vs quantum Markov chains.

	(Discrete-time) Markov Chain	Quantum Markov Chain
State space	Finite or countable set	Finite-dimensional or separable Hilbert space
Initialization	Probability distribution: $\iota_{init} : S \to [0,1]$ $\sum_{s\in S} \iota_{init}(s) = 1$	Density matrix: $\rho = \sum_i p_i \vert\psi_i\rangle\langle\psi_i\vert$ $\sum p_i = 1$
Transition	Probability transition matrix: $P : S \times S \to [0,1]$ $\sum_{s'\in S} P(s,s') = 1$	Quantum operation: $\mathcal{E}(\rho) = \sum_i E_i \rho E_i^\dagger$ $\sum E_i^\dagger E_i = I$
Logic	Probabilistic temporal logic	Temporal extension of Birkhoff-von Neumann quantum Logic

- A pure state of an n-dimensional quantum system is represented by a unit complex vector $|\psi\rangle \in \mathbb{C}^n$. In Table 1, the density operator $\rho = \sum_i p_i|\psi_i\rangle\langle\psi_i|$ is a mathematical representation of ensemble $\{(p_i, |\psi_i\rangle\}$, meaning the system is in state $|\psi_i\rangle$ with probability p_i. Thus, ρ can be seen as a quantum analog of the initial probability distribution in a classical MC.
- The quantum operation $\mathcal{E}$ is a quantum generalization of the transition probability matrix in a classical MC. According to the principles of quantum mechanics, it can be mathematically modelled as $\mathcal{E}(\rho) = \sum_i E_i\rho E_i^\dagger$, where each E_i is an $n \times n$ complex matrices (called *Kraus matrices*) satisfying the condition $\sum E_i^\dagger E_i = I$ (the unit matrix), which is a counterpart of the normalization condition in a classical MC. In particular, the evolution of a closed quantum system is described by $\mathcal{E}(\rho) = U\rho U^\dagger$, where U is a unitary matrix, i.e. $UU^\dagger = U^\dagger U = I$.

Quantum Reachability Problem: Given a QMC $\mathcal{C} = \langle\mathcal{H}, \mathcal{E}\rangle$ and any initial state ρ in $\mathcal{H}$, compute the reachable subspace:

$$\mathcal{R}_{\mathcal{C}}(\rho) = \text{span}\{|\psi\rangle \in \mathcal{H} : |\psi\rangle \text{ is reachable from } \rho \text{ in } \mathcal{C}\} \tag{1}$$

Intuitively, $\mathcal{R}_{\mathcal{C}}(\rho)$ consists of not only the states reached in the execution path $\rho \to \mathcal{E}(\rho) \to \mathcal{E}^2(\rho) \to \cdots \to \mathcal{E}^n(\rho) \to \cdots$ but also their linear combinations.

Example 1 (Quantum random walk). Consider a quantum random walk on a 4-cycle [16] with the Hadamard operator as a coin c, shown in Fig. 1. The walking space is a 4-dimensional Hilbert space $\mathcal{H}_p$ on the bottom two qubits p_1, p_2, supported by four position states $\{|0\rangle_p, |1\rangle_p, |2\rangle_p, |3\rangle_p\}$ in the computational basis. After applying the coin H in each step, the evolution of the system is described by a quantum conditional (shift operator):

$$S = |0\rangle_c\langle 0| \otimes \sum_i |i+1\rangle_p\langle i| + |1\rangle_c\langle 1| \otimes \sum_i |i-1\rangle_p\langle i|$$

where the neighbor-state $|i+1\rangle$ and $|i-1\rangle$ are computed modulo 4. It means that the walker can simultaneously walk in different directions, which is the main difference between classic and quantum random walks.

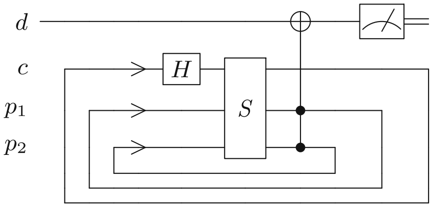


Fig. 1. Quantum random walk on a 4-cycle

This system can be modelled as a QMC with $\mathcal{E}$ being defined by the unitary operator $S(H_c \otimes I_p)$. Let it start in pure state $\rho = |000\rangle\langle 000|$. Using the tool QReach presented in this paper, one can compute that the reachable space of this QMC is the 6-dimensional space with linear independent basis $\{|000\rangle, |001\rangle + |111\rangle, |100\rangle - |110\rangle, |101\rangle + |001\rangle, |010\rangle, |011\rangle + |101\rangle\}$. It is surprising that not the whole 8-dimensional state space $\mathcal{H}_c \otimes \mathcal{H}_p$ is reachable although any position in $\mathcal{H}_p$ may be hit in some time.

3 Architecture and Data Structures

In this section, we elaborate on the architecture of QReach and some reasoning techniques for quantum circuits based on the CFLOBDD backend. Although our target is specified on quantum reachability analysis, we believe that some functionalities of QReach are also useful for other tasks.

3.1 Architecture of QReach

An overview of the architecture of QReach is presented in Fig. 3. For convenience of presentation, we describe QReach's procedure with Example 1. Let the system starting in state $\rho = |000\rangle\langle 000|$. To illustrate the capability of QReach for handling general quantum operations, we consider a faulty quantum random walk in which a bit-flip error happens in front of the Hadamard gate with probability p. The behavior of the system is then modelled by $\mathcal{E}(\rho) = E_0 \rho E_0^\dagger + E_1 \rho E_1^\dagger$,

```
    # Prepare the BitFlipError
    e = QError(”Bitflip”, loc=[0,0], params=[], p=0.5)

    # Prepare the quantum Markov chain
    qmc = QMarkov(cir_body=”qrw.qasm”, channels=[e])

    # Prepare model checker and its CFLOBDD backend
    qChecker = fromMarkovModel(qmc)
    qChecker = initWithStr(qChecker, str_list=[])

    # Execution and output
    reachable_dim = qChecker.reachability()
    qChecker.printProjector()
```

Fig. 2. A Demo for reachability analysis of QMCs. `channels` is a list of quantum errors and instructions like measurement and reset, containing their occurring positions.

where $E_0 = \sqrt{1-p}\; S(H_c \otimes I_p)$, and $E_1 = \sqrt{p}\; S(H_c \otimes I_p)(X \otimes I_p)$. Our purpose is to compute the reachable subspace of $\mathcal{E}$. An example Python program for this process is shown in Fig. 2.

We implement our symbolic quantum reachability analysis algorithm with CFLOBDD in a C++ core and provide Python interfaces for invoking. Some of the key components are explained below:

Input and Output. QReach accepts a composite input specification to represent a QMC. A quantum program written in the QASM format [7] is parsed as the main circuit body of the QMC. Some specified quantum errors, measurements, and other non-unitary channels can be coded as a supplement in the `Qchannel` type. Once results are obtained, some subspace characters (e.g. dimensions and support vectors) can be output.

Simulation. A well-formed toolkit Quasimodo [14] based on CFLOBDD was implemented for quantum algorithm simulation. Simulation for quantum circuits, or in other words, applying a sequence of quantum gates on a pure state, is an essential process during the reachability analysis. Therefore, we referred to some of the codes' implementation details from Quasimodo. Specifically, we adopted Quasimodo's Pybind architecture, which links C++ APIs and Python. However, a simulation framework like Quasimodo cannot handle situations in reachability analysis like mixed states and super-operators. We fixed these issues, added some new features, and stored gate sequences and intermediate projectors to make the simulation execution process not just sequential.

These methods are covered in `fromMarkovModel()` and `reachability()`, while still available to invoke independently for other purposes. For example, `qchecker.u3()` conducts normal U3 gate simulation on the state vector in the current workspace; `setProjector()` and `applyProjector()` methods provide data manipulation in *Projector* and *State vector* as shown in Fig. 3. Detailed techniques used in our CFLOBDD simulator different from that in previous works [14,15] will be discussed in Sect. 3.2.

Reachability Analysis. The efficient algorithm for quantum reachability analysis to be elaborated in Sect. 4 has been implemented in QReach. We also implemented the interfaces for some of mathematical tricks in [19] (e.g. Choi matrix transformation and maximally entangled state preparation) on CFLOBDDs, which will be critical in a future extension of QReach for computing reachability probabilities (rather than subspaces) of QMCs.

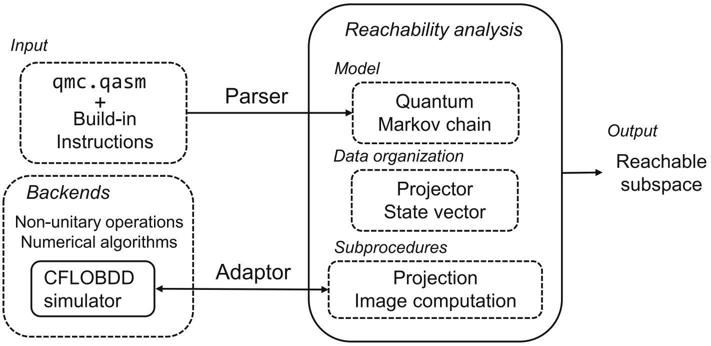


Fig. 3. Architecture of QReach.

3.2 CFLOBDD for Quantum Reachability Analysis

Now we introduce our CFLOBDD backend, which implements some support for numerical algorithms and quantum instructions. In particular, we illustrate how to expand simulation of a quantum system to its reachability analysis.

As a newly proposed DD-based structure, CFLOBDD attracts our attention due to its distinctive features compared to other DDs for quantum systems. CFLOBDD adopts a single-entry, multi-exit, non-recursive, hierarchical finite-state machine architecture [15]. From a programming perspective, a certain form of procedure call is invoked, leading to some exponential compression over BDDs. Following the name “context-free language”, the incoming and outgoing edges of groupings are matched according to certain principles. Figure 4 provides a general insight into how the edges of CFLOBDDs are matched. The representing capability of the CFLOBDD is exploited in our tool QReach for symbolic reachability analysis of QMCs.

Like any other type of DDs, the compression capability of CFLOBDDs only stands out in specific instances. In these cases, canonical reduced forms reuse parts in a DD and save storage from the raw data. However, in general circumstances, the number of nodes required to represent a large-scale matrix or vector is still exponential. Reordering strategies for reduced ordered BDDs to optimize storage usage are hard (NP-complete) [3]. This problem becomes even more severe in algebraic DDs [1], where non-Boolean values make it harder to

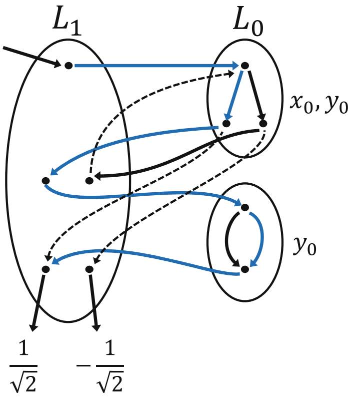


Fig. 4. CFLOBDD for Hadamard gate. Indices are represented in the L_0 groupings consisting of fork nodes and don't care nodes. A path from the entry of the topmost grouping to the terminal values denotes an assignment to all indices. For example, the bold blue path corresponds with the $\{x_0 = 0, y_0 = 1\}$ entry of the Hadamard matrix, resulting a value $\frac{1}{\sqrt{2}}$. (Color figure online)

find similar structures in a diagram. Therefore, we cannot simply represent a quantum operation or a projector as a single CFLOBDD without a partition strategy. Unlike classical symbolic model checking, a quantum circuit is usually difficult to partition due to entanglements. We chose an alternative in the QReach backend: using an augmented simulation method to calculate quantum operations. Thus, only state vectors and single quantum gates need to be stored, rather than the whole matrix.

In particular, circuits in QMCs are usually complicated, involving noises and dynamic operations. To handle them, we strengthen CFLOBDD with the following techniques:

Non-unitary Operators. Apart from normal quantum gates like Hadamard, Pauli, and generic U3 rotation gates, we specifically support two-dimensional matrices of any form, covering those non-unitary operators in quantum noises and measurements; for instance, amplitude damping channels with operators:

$$E_0 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{bmatrix}, \qquad E_1 = \begin{bmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{bmatrix}$$

Another example of non-unitary operators is Z-basis measurements. The post measurement states are obtained by applying $P_0 = |0\rangle\langle 0|$ and $P_1 = |1\rangle\langle 1|$ respectively, followed by normalizations.

Basic Methods Extensions. Some operations are added to CFLOBDD as a basis for top-level algorithms, incorporating the normalization and conjugate transpose. In addition to these methods, an optimizing trick for complex number representation is applied. We used a simplified version of the method proposed in [20], constructing a Hash function and unique table for complex numbers.

Numerical Algorithms. Numerical algorithms are critical in QReach. Based on them, some operations that are particularly important in modelling quantum systems (e.g. partial trace and Choi matrix) are now available in QReach (see Example 2). A key methodology is to decompose operands of calculations into base vectors, replacing complicated operations with matrix-vector multiplication or inner products of vectors. In Sect. 4, the high-level description of reachability analysis algorithm also embodies this idea.

Example 2 (Partial trace). Consider a quantum system composed of qubits A and B in state ρ_{AB}. Then the state of A can be described by the partial trace operator:

$$\rho_A \equiv tr_B(\rho_{AB}) := \sum_i (I_A \otimes \langle i|_B)\rho_{AB}(I_A \otimes |i\rangle_B)$$

For simplicity, suppose the system is in a pure state $|\psi\rangle = |0\rangle|\lambda\rangle + |1\rangle|\mu\rangle$. Tracing out qubit A, qubit B should be in the mixed state $\rho = |\lambda\rangle\langle\lambda| + |\mu\rangle\langle\mu|$. In QReach, this procedure is conducted in the following steps to avoid redundant matrix manipulations and adjustments to CFLOBDD's internal structures: (1) Perform a Z-basis measurement on A and get unnormalized post measurement states $|0\rangle|\lambda\rangle$ and $|1\rangle|\mu\rangle$; (2) Apply X gate to A conditionally on the measurement result one; (3) Let the collection $S = \{|0\rangle|\lambda\rangle, |0\rangle|\mu\rangle\}$. Then S can be viewed as ρ's representation and participate in later calculations. In some cases, like reachability analysis, the norm of a state vector is not essential and could be omitted thereby. Note that after these operations qubit A remains in a tensored zero state, because the number of qubits in a CFLOBDD is required to be an exponential power of 2. We apply an X gate on the measure-one result to make the effect looks like resetting a qubit.

4 Algorithm for Reachability Analysis

The existing algorithm for reachability analysis of QMCs is based on Choi matrix representation of quantum operations introduced in [19]. In this section, we propose a more efficient algorithm for the same purpose (see Algorithm 1).

Our algorithm is a natural extension of reachability analysis in classical model checking using a BFS-based technique. The main difference is that we are dealing with reachable *subspaces* of the Hilbert space $\mathcal{H}$ rather than *subsets* of a finite set of states in the classical case. Therefore, Algorithm 1 traverses each possible *dimension* of a finite-dimensional Hilbert space non-repetitively rather than each reachable state as in the classical case. Note that the code segment from Line 7 to Line 12 is the process of extracting vectors orthogonal to those that have been searched, which is similar to frontier set simplification in classical symbolic model checking.

The nontrivial subprocedures in Algorithm 1 differing from that in classical reachability analysis are the projection and image computation (Line 5 and Line 7). To reduce the representing and temporal cost in the algorithm, our basic idea

Algorithm 1. Computing reachable space

```
Input: Super operator $\mathcal{E}$ in Hilbert space $\mathcal{H}$ with dimension $d$; set of initial states $P$
Output: A set of orthogonal basis $P'$ of reachable subspace of $\mathcal{H}$
$P' \leftarrow Gram\_Schmidt(P)$,    $cnt \leftarrow size(P')$
Initialize queue $Q$ with $P'$
while $Q$ not empty and $cnt < d$ do
    $curr\_state \leftarrow Q$.pop()
    $expanded\_states \leftarrow \texttt{Image}(\mathcal{E}, curr\_state)$
    for $s$ in $expanded\_states$ do
        $s \leftarrow s - \texttt{Project}(P', s)$
        $s \leftarrow \texttt{normalize}(s)$
        if $s$ is not zero vector then
            $Q$.push($s$)
            $P'$.append($s$)
            $cnt \leftarrow cnt + 1$
        end if
    end for
end while
return $P'$
```

is to take eigenvectors into calculation instead of the whole matrix. In practice, most of the projections are low-rank, which ensures the efficiency of this idea.

Implementation in QReach. For image computation, we exploit the simulation functionality of our backend data structure CFLOBDD. The runtime of the backend's simulation highly determines our algorithm's efficiency. In this step, non-unitary operations like noises, measurements, qubit resets, and deallocations will be simulated by the augmented simulator introduced in the past section. All the simulations of channels together make up the image computation of Kraus representations $\mathcal{E}(\rho) = \sum_i E_i \rho E_i^\dagger$.

A projector onto a subspace of the Hilbert space can be represented by a set of orthogonal support vectors of the subspace. This technique can be viewed as a quantum version of partitioning, which usually appears as forms of disjunctions and conjunctions in classical symbolic model checking [4]. Formally, let $\mathcal{P}$ be the projector onto a subspace with an orthonormal basis $\{|i\rangle\}$, that is, $\mathcal{P} = \sum |i\rangle\langle i|$, then we set P to be the set of $|i\rangle$'s, and

$$\texttt{Project}(P, |s\rangle) = \sum \langle s|i\rangle^* |i\rangle$$

We conduct conjugate-transposing on $|s\rangle$ instead of $|i\rangle$ to invoke vector operations as few as possible. The computational complexity of subprocedure `Image` and `Project` are both $O(d^2)$ given a constant number of Kraus matrices.

The following theorem shows the correctness and complexity of our algorithm.

Theorem 1. *The output P' and input P of Algorithm 1 satisfies: $span(P') = \mathcal{R}_{\mathcal{C}}(\rho) = \bigvee_{i=0}^{d-1} supp(\mathcal{E}^i(\rho))$, where ρ is the initial state, and $supp(\rho) = span(P)$. The time complexity of Algorithm 1 is $O(d^3)$*

Proof. Following the theorem 1 in [19], for $d = dim(\mathcal{H})$, and any density operator ρ in $\mathcal{H}$,

$$\mathcal{R}_{\mathcal{C}}(\rho) = \text{supp}\left(\sum_{i=0}^{d-1} \mathcal{E}^i(\rho)\right) \tag{2}$$

And all reachable states can be reached in at most d iterations. The correctness is proved in three steps:

i) The main *while* loop terminates in at most d steps;
ii) When terminates, $\text{span}(\texttt{Image}(\mathcal{E}, P')) = \text{span}(P')$;
iii) $\mathcal{R}_{\mathcal{C}}(P') = \mathcal{R}_{\mathcal{C}}(\rho)$.

At last, combining the complexity of broad-first-search and subprocedures, the Algorithm 1 has complexity $O(d^3)$, which is an improvement over $O(d^{4.7454})$ in [19]. □

The dimension of a Hilbert space often grows exponentially larger in quantum systems. Therefore, our algorithm will inevitably become inefficient on quantum circuits with more than twelve qubits. Although limited on the scale of quantum systems, Algorithm 1 is stable for the number of quantum operations and the dimension of initial spaces. The BFS strategy and the frontier set simplification ensure that only dimensions that are reached for the first time can be counted.

5 Use Cases and Experiments

Some cases are studied in this section, providing insight into practical applications of our tool QReach for quantum reachability analysis in the future. All experiments were conducted on a personal computer with hardware configurations: Intel i5-13600kf CPU with 14 cores; 32GB RAM. The experimental results are presented in Table 2.

Grover Search. The Grover search algorithm provides a quadratic speedup over a series of classical search algorithms [8]. The main idea of the Grover search is to apply a quantum subroutine iteratively which leads to a rotation from the initial state to the target state. QReach's reachability analysis under some float precision shows that during iterations, the state is always located in the 2-dimensional subspace spanned by the initial state and the target state.

Quantum Random Walk. We tested quantum random walk circuits (QRW) with different numbers of qubits which have a similar structure with Example 1. To demonstrate more functionalities of QReach, mixed initial states and amplitude dumping noises are introduced in front of the Hadamard gate. The (dimension of) reachable space computed by QReach for these quantum circuits are given in Table 2.

Table 2. Experimental results. The noise in QRW is amplitude dumping. For RUS, the circuits implement $(I+2iZ)/\sqrt{5}$, $(2X+\sqrt{2}Y+Z)/\sqrt{7}$, and $(3I+2iZ)/\sqrt{13}$ respectively.

	#Qubits	Ope. type	Initial dim.	Time(s)	#Edges	Reachable dim.
Grover	7	Unitary	1	0.0252	268	2
	15	Unitary	1	0.0303	350	2
	31	Unitary	1	0.0516	432	2
	63	Unitary	1	0.0885	514	2
QRW	3	Unitary	1	0.0284	435	6
	5	Unitary	1	0.0561	1337	10
	7	Unitary	1	0.196	7512	34
	9	Unitary	2	379.64	608097	512
	7	Noise	2	0.446	10211	64
	9	Noise	2	110.70	447811	512
	10	Noise	2	117.36	421038	1024
RUS	3	Measure	1	0.0262	130	2
	2	Measure	1	0.0216	74	2
	2	Measure	1	0.0203	60	1

Repeat-Until-Success Circuits. Repeat-until-success (RUS) circuits [13] are a type of circuit that decides whether to repeat or terminate based on the measurement results. It is usually used to design circuits with fewer non-Clifford gates or ancilla qubits. In QReach, it can be modelled as a quantum Markov chain with measurements and qubit resetting as parts of the channel. We tested some of the examples in [13]. It is clear that the reachable dimensions should be 2 or 1, depending on whether the resulting quantum states differ by only one global phase if the measurement succeeds or fails (Fig. 5).

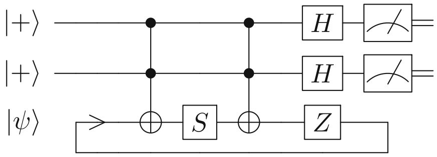


Fig. 5. A repeat-until-success circuit for gate $V_3 = (I + 2iZ)/\sqrt{5}$.

There is a consensus that every future tool released in quantum model checking must face the problem of finding broader applications. Besides these cases, we are improving the scope of QReach and exploring more possible applications on sequential quantum circuits and protocols.

References

1. Bahar, R.I., et al.: Algebric decision diagrams and their applications. Formal Methods Syst. Des. **10**, 171–206 (1997)
2. Baltazar, P., Chadha, R., Mateus, P.: Quantum computation tree logic-model checking and complete calculus. Int. J. Quantum Inform. **6**(02), 219–236 (2008)
3. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is np-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996)
4. Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **13**(4), 401–424 (1994)
5. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L.: AUTOQ: an automata-based quantum circuit verifier. In: Enea, C., Lal, A. (eds.) International Conference on Computer Aided Verification, pp. 139–153. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37709-9_7
6. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L., Yen, D.D.: An automata-based framework for verification and bug hunting in quantum circuits. Proc. ACM Program. Lang. **7**(PLDI), 1218–1243 (2023)
7. Cross, A., et al.: OpenQASM 3: a broader and deeper quantum assembly language. ACM Trans. Quantum Comput. **3**(3), 1–50 (2022)
8. Grover, L.K.: Quantum computers can search rapidly by using almost any transformation. Phys. Rev. Lett. **80**(19), 4329 (1998)
9. Hong, X., Zhou, X., Li, S., Feng, Y., Ying, M.: A tensor network based decision diagram for representation of quantum circuits. ACM Trans. Des. Autom. Electron. Syst. (TODAES) **27**(6), 1–30 (2022)
10. Mateus, P., Ramos, J., Sernadas, A., Sernadas, C.: Temporal logics for reasoning about quantum systems. Semantic Tech. Quantum Comput., 389–413 (2009)
11. Mateus, P., Sernadas, A.: Weakly complete axiomatization of exogenous quantum propositional logic. Inf. Comput. **204**(5), 771–794 (2006)
12. Niemann, P., Wille, R., Miller, D.M., Thornton, M.A., Drechsler, R.: QMDDs: efficient quantum function representation and manipulation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(1), 86–99 (2015)
13. Paetznick, A., Svore, K.M.: Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. Quantum Info. Comput. **14**(15–16), 1277–1301 (2014)
14. Sistla, M., Chaudhuri, S., Reps, T.: Symbolic quantum simulation with Quasimodo. In: International Conference on Computer Aided Verification. pp. 213–225. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_11
15. Sistla, M.A., Chaudhuri, S., Reps, T.: CFLOBDDs: context-free-language ordered binary decision diagrams. ACM Trans. Program. Lang. Syst. (2023)
16. Venegas-Andraca, S.E.: Quantum walks: a comprehensive review. Quantum Inf. Process. **11**(5), 1015–1106 (2012)
17. Vinkhuijzen, L., Coopmans, T., Elkouss, D., Dunjko, V., Laarman, A.: LIMDD: a decision diagram for simulation of quantum computing including stabilizer states. Quantum **7**, 1108 (2023)
18. Ying, M., Feng, Y.: Model Checking Quantum Systems: Principles and Algorithms. Cambridge University Press (2021)

19. Yu, N., Ying, M.: Reachability and termination analysis of concurrent quantum programs. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 69–83. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_7
20. Zulehner, A., Hillmich, S., Wille, R.: How to efficiently handle complex values? Implementing decision diagrams for quantum computing. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–7. IEEE (2019)

Measurement-Based Verification of Quantum Markov Chains

Ji Guan[1](✉), Yuan Feng[2], Andrea Turrini[1,3](✉), and Mingsheng Ying[4]

[1] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{guanj,turrini}@ios.ac.cn

[2] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

[3] Institute of Intelligent Software, Guangzhou 511458, China

[4] Centre for Quantum Software and Information, University of Technology Sydney, Sydney, NSW 2007, Australia

Abstract. Model-checking techniques have been extended to analyze quantum programs and communication protocols represented as quantum Markov chains, an extension of classical Markov chains. To specify qualitative temporal properties, a subspace-based quantum temporal logic is used, which is built on Birkhoff-von Neumann atomic propositions. These propositions determine whether a quantum state is within a subspace of the entire state space. In this paper, we propose the measurement-based linear-time temporal logic MLTL to check quantitative properties. MLTL builds upon classical linear-time temporal logic (LTL) but introduces quantum atomic propositions that reason about the probability distribution after measuring a quantum state. To facilitate verification, we extend the symbolic dynamics-based techniques for stochastic matrices described by Agrawal et al. (JACM 2015) to handle more general quantum linear operators (super-operators) through eigenvalue analysis. This extension enables the development of an efficient algorithm for approximately model checking a quantum Markov chain against an MLTL formula. To demonstrate the utility of our model-checking algorithm, we use it to simultaneously verify linear-time properties of both quantum and classical random walks. Through this verification, we confirm the previously established advantages discovered by Ambainis et al. (STOC 2001) of quantum walks over classical random walks and discover new phenomena unique to quantum walks.

1 Introduction

Model checking is a formal verification technique that is used to ensure the correctness of a system based on a given specification [1]. In recent years, model checking has been applied to quantum systems and has become a powerful tool for verifying the behaviors and properties of quantum programs or communication protocols [2,3]. Similar to the classical case, the main components of model

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 533–554, 2024.
https://doi.org/10.1007/978-3-031-65633-0_24

checking quantum systems are the system model and the temporal logic, which are used to mathematically describe the evolution of the system and specify its temporal properties, respectively.

System Model. Quantum Markov Chains (QMCs), which are the quantum extension of classical *Markov chains* (MCs), provide an exceptional paradigm for modeling the evolution of quantum systems in various scenarios, including quantum control [4], quantum information theory [5], quantum programming [6], and quantum communication systems [7]. Notably, quantum walks, which are a special class of QMCs, have been successfully employed in the design of quantum algorithms (for a survey of this research line, see [8,9]). A QMC $\mathcal{Q}$ is defined as a triple $\mathcal{Q} = (\mathcal{H}, \mathcal{E}, \rho_0)$ that corresponds to a classical Markov chain (S, P, s_0), where $\mathcal{H}$ is a finite-dimensional Hilbert (linear) state space instead of the finite state set S, $\mathcal{E}$ is a *super-operator* on $\mathcal{H}$ instead of the transition stochastic matrix P on S, and ρ_0, which is a *density matrix*, represents the initial state instead of s_0. Intuitively, the super-operator $\mathcal{E}(\,\cdot\,)$, which is a linear mapping, models the dynamics of the system and transforms a state (density matrix) ρ into another state $\mathcal{E}(\rho)$. Some special cases of QMCs have emerged, such as open quantum walks [10] and classical-quantum (super-operator valued) Markov chains [11], where the latter resemble classical Markov chains but with the transition stochastic matrix $P = \{p_{i,j}\}_{i,j\in S}$ being replaced by a transition set $\{\mathcal{E}_{i,j}\}_{i,j\in S}$ of super-operators, while still maintaining the finite state set S.

Temporal Logics. The dynamic extension of Birkhoff-von Neumann quantum logic [12] was proposed to specify a wide range of temporal properties of quantum systems. In this approach, atomic propositions are used to describe qualitative properties of a quantum system, represented as closed subspaces of the system's state Hilbert space (whether or not a quantum state ρ is in a subspace $\mathcal{X}$ of $\mathcal{H}$). Furthermore, QMCs are abstracted as subspace transition systems, and the temporal properties of interest are represented by infinite sequences of sets of atomic propositions (subspaces) [3]. This subspace-based temporal logic allows for the specification of linear-time properties, such as invariants and safety properties, for quantum automata (a simplified form of QMCs) [13]. Model-checking algorithms have been developed for the subspace-based temporal logic in [14], and the (un)decidability of model checking quantum automata has been studied in [15]. However, a limitation of model checking QMCs against subspace-based temporal logics is that it can only handle qualitative properties, which means that only simple examples can be checked. Given that the power of quantum systems lies in their probabilistic nature, it is crucial to be able to check probabilistic (quantitative) properties.

To address this issue, in this paper, we propose a measurement-based linear-time temporal logic to capture these properties and develop a model-checking algorithm to check the quantitative properties of QMCs. More specifically, we observe that the properties of the quantum systems in question can be described using quantum measurements, which extract classical (probabilistic) information from quantum states. Building on this observation, we introduce measurement-based atomic propositions to describe static quantitative properties, namely, the measurement outcome probability of a quantum state under a measurement. This can be seen as a generalization of the subspace-based atomic propositions of the Birkhoff-von Neumann quantum logic [12], where a quantum state ρ in a subspace $\mathcal{X} \subseteq \mathcal{H}$ can be regarded as having a measurement outcome probability of 1 under the measurement onto $\mathcal{X}$. By combining with standard linear-time temporal logic (LTL) [1,16], we obtain MLTL, the measurement-based LTL, to specify the temporal quantitative properties of quantum systems.

In order to develop an algorithm for model checking QMCs against MLTL formulas, we extend the Thiagarajan's approximate verification [17] of a stochastic transition matrix P to encompass more general linear operators in quantum systems, e.g., the super-operators. The key technique for this generalization is based on the eigenvalue analysis of QMCs, which simplifies the previous work based on the *Bottom Strongly Connected Component (BSCC) decomposition* [1] of the state space of classical Markov chains [17]. Subsequently, we provide an effective procedure for the approximate model checking of QMCs against MLTL formulas. In Sect. 6, we provide several case studies to illustrate how our model and algorithm can be applied in a quantum walk. These case studies help to verify the previously established advantages of quantum walks over classical random walks, as discovered by Ambainis et al. [18]. Additionally, we explore new phenomena unique to quantum walks when we verify the same MLTL formulas on both types of walks.

In summary, this paper makes the following main contributions:

1. *Introducing* a quantum temporal logic, called measurement-based linear-time logic (MLTL), which allows for specifying quantitative properties of QMCs.
2. *Generalizing* symbolic dynamics-based verification techniques of the transition stochastic matrix given in [17] to more general quantum linear operators (super-operators) by *eigenvalue analysis*; based on this, a model-checking algorithm for QMCs against MLTL is developed.
3. *Verifying* numerous quantitative properties of quantum walks through our model-checking algorithm as case studies. This serves to validate the established advantages of quantum walks over their classical counterparts and discover new phenomena unique to quantum walks.

1.1 Related Works and Challenges

To provide a suitable context for our work, let us delve deeper into the discussion of related works and the challenges we encounter in this paper.

Hybrid vs. Quantum Temporal Logic. In a previous study [11,19], a specialized type of quantum Markov chain known as super-operator-valued quantum Markov chain was proposed. This model was designed for the purpose of modeling quantum programs and quantum cryptographic protocols. Additionally, a quantum extension of the probabilistic computation tree logic (PCTL) called quantum computation tree logic (QCTL) was introduced, along with a model-checking algorithm specifically tailored for this Markov model. In a subsequent development, algorithms for verifying ω-regular properties were also introduced [20].

However, these hybrid temporal logic approaches heavily rely on the classical state graph and are not applicable to quantum systems. This is because quantum systems have a continuous state space and an infinite number of states, making it impossible to obtain a connection graph. In order to address this limitation and specify the properties of quantum Markov chains, we propose a measurement-based linear-time temporal logic (MLTL) approach that does not require a connection graph. This allows us to directly reason about the transitions of measurement outcome probability distributions. Our MLTL approach adopts classical LTL with quantum physical interpretation, providing the advantage of utilizing existing classical techniques and facilitating contributions from the classical model-checking community to the field of quantum computing.

Classical vs. Quantum Markov Chains. The main technique used for model checking classical Markov chains in [17] involves studying the periodicity of states in sub-chains obtained through BSCC decomposition [1], a widely-used method in the field of model checking Markov chains. However, when it comes to the quantum extension of this decomposition, as described in [21], the BSCC decomposition of QMCs is not unique but there are infinitely many decompositions due to the continuous state space, unlike the classical case. Consequently, we cannot directly generalize Thiagarajan's *approximate verification* [17]. To overcome this unique challenge posed by quantum mechanics, we propose a method based on eigenvalue analysis to directly explore the periodic properties of QMCs. As a result, QMCs are not always periodically stable like classical Markov chains. We provide a way to determine the periodic stability depending on the initial states of QMCs. With these efforts, we successfully extend the idea of approximate model checking to work for QMCs, pushing the application boundary of such model-checking techniques to more general linear systems.

Model Checking Quantum Walks. As a result of the phenomenon known as *quantum interference* [22], quantum walks can propagate at significantly faster or slower rates compared to their classical counterparts [18]. Quantum walks have gained attention due to their potential application in the development of randomized algorithms, with various quantum algorithms incorporating this concept [8,9]. Notably, in certain search problems, quantum walks can offer quadratic or exponential speedups compared to classical algorithms.

Previously, researchers have conducted case-by-case studies on the dynamic properties of quantum walks, requiring the introduction of various techniques depending on the specific property and underlying topological structure of the walks. In this paper, we introduce a model-checking method that overcomes this limitation by allowing for the automatic verification of a wide range of properties of the walks. To model quantum walks, we utilize QMCs, and to specify the relevant dynamical properties, we employ MLTL formulas. By simultaneously verifying classical and quantum walks using our model-checking algorithm, we can confirm the advantages of quantum walks that have already been established in [18], as well as discover new phenomena that are distinct from classical random walks. We anticipate that these new phenomena, discovered by our model-checking algorithm, will contribute to the development of more efficient quantum walk-based algorithms with enhanced speedup capabilities.

2 Preliminaries

In this section, we aim to explain the three components that appear in a QMC $\mathcal{Q} = (\mathcal{H}, \mathcal{E}, \rho_0)$, as well as quantum measurements, an essential part of our MLTL.

Quantum State Space. $\mathcal{H}$. The *state space* of a quantum system is a finite-dimensional linear space $\mathcal{H}$, which is commonly known as the *Hilbert space* in the field of quantum computing. A *quantum pure state* is represented by a unit complex column vector ψ in $\mathcal{H}$. In the field of quantum computing, the *bra-ket notation* is widely used to represent quantum states, making it easier to perform calculations that frequently arise in quantum mechanics. This notation uses angle brackets, $\langle$ and $\rangle$, along with a vertical bar $|$, to construct "bras" and "kets" that represent row and column vectors, respectively. The following list provides the notation used in this paper to represent linear algebra concepts:

1. $|\psi\rangle$ represents a unit complex column vector (quantum pure state) in $\mathcal{H}$, labeled with ψ;
2. $\langle\psi| := |\psi\rangle^\dagger$ denotes the complex conjugate and transpose of $|\psi\rangle$;
3. $\langle\psi_1|\psi_2\rangle := \langle\psi_1||\psi_2\rangle$ represents the inner product of $|\psi_1\rangle$ and $|\psi_2\rangle$;
4. $|\psi_1\rangle\langle\psi_2| := |\psi_1\rangle \cdot \langle\psi_2|$ denotes the outer product of $|\psi_1\rangle$ and $|\psi_2\rangle$.

It is important to note that any vector in $\mathcal{H}$ can be linearly represented by a *computational basis*, which is a set of mutually orthogonal unit vectors. In order to compare with classical Markov chains denoted as (S, P, μ_0), we use the finite state set $S = \{s_0, \ldots, s_{d-1}\}$ to label the computational basis of $\mathcal{H}$, with dimension d, as $\{|s_0\rangle, \ldots, |s_{d-1}\rangle\}$. Here, $|s_k\rangle$ is a unit column vector with the k-th element being 1 and the remaining elements being 0 (the index starts from 0). Then $\mathcal{H}$ is denoted as $\mathcal{H} = \mathrm{span}\{|s_0\rangle, \ldots, |s_{d-1}\rangle\}$.

Using this basis, any quantum state $|\psi\rangle$ in $\mathcal{H}$ can be expressed as a linear combination of $\{|s_0\rangle, \ldots, |s_{d-1}\rangle\}$ with complex coefficients a_k: $|\psi\rangle = \sum_{k=0}^{d-1} a_k|s_k\rangle$ with the normalization condition $\langle\psi|\psi\rangle = \sum_{k=0}^{d-1} a_k a_k^* = 1$, where a_k^* is the complex conjugate of a_k. In the case of a 2-dimensional space, we have:

$$|s_0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \qquad |s_1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \qquad |\psi\rangle = \begin{pmatrix}a_0\\a_1\end{pmatrix} \qquad \langle\psi| = \begin{pmatrix}a_0^*, a_1^*\end{pmatrix}.$$

It is evident that a quantum pure state has the capability to depict a *superposition* of state set $S = \{s_0, \ldots, s_{d-1}\}$ as $|\psi\rangle = \sum_{k=0}^{d-1} a_k|s_k\rangle$. The superposition is a unique feature of quantum systems and the main reason for the advantages of quantum algorithms over their classical counterparts [23].

Quantum Mixed State ρ. In quantum mechanics, uncertainty is a common characteristic of quantum systems, arising from quantum noise and measurements. To describe the uncertainty of possible quantum pure states, the concept of *quantum mixed state* ρ on $\mathcal{H}$ is introduced. It can be represented as

$$\rho = \sum_k p_k |\psi_k\rangle\langle\psi_k|. \tag{1}$$

Here, $\{(p_k, |\psi_k\rangle)\}_k$ represents an ensemble, indicating that the quantum state is at $|\psi_k\rangle$ with probability p_k. This concept can also be used to describe the uncertainty of a classical probability distribution, where each $|\psi_k\rangle$ represents s_k by $|\psi_k\rangle = |s_k\rangle$. Specifically, a (row) probability distribution $\mu = (p_0, \ldots, p_{d-1})$ over the state set $S = \{s_0, \ldots, s_{d-1}\}$ can be represented by a quantum mixed state. This representation involves a diagonal matrix on $\mathcal{H}$, where the diagonal elements correspond to the probabilities p_k as follows.

$$\rho_\mu = \sum_k p_k |s_k\rangle\langle s_k| = \mathrm{diag}(p_0, \ldots, p_{d-1}). \tag{2}$$

Hence, a quantum mixed state ρ is an extension of a probability distribution μ. Generally, the quantum uncertainty is more complex because the ensemble decomposition in Eq. (1) of a quantum state ρ can have infinitely many variants. This means that ρ can represent multiple ensembles $\{(p_k, |\psi_k\rangle)\}_k$ simultaneously.

From a mathematical perspective, a quantum mixed state $\rho \in \mathcal{L}(\mathcal{H})$ is a linear operator (d-by-d matrix) on $\mathcal{H}$ that satisfies three conditions: 1) *Hermitian* $\rho^\dagger = \rho$; 2) *positive semi-definite* $\langle\psi|\rho|\psi\rangle \geq 0$ for all $|\psi\rangle \in \mathcal{H}$; and 3) *unit trace* $\mathrm{tr}(\rho) = \sum_k \langle s_k|\rho|s_k\rangle = 1$, where $\mathrm{tr}(\rho)$ is the trace of ρ and represents the sum of the diagonal elements of ρ. Here, $\mathcal{L}(\mathcal{H})$ denotes the set of linear operators on $\mathcal{H}$. Let $\mathcal{D}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{H})$ be the set of all quantum mixed states on $\mathcal{H}$. To avoid any ambiguity, in the subsequent discussion, the term "quantum states" will specifically refer to quantum mixed states, given that we are considering the broader scenario.

Quantum Evolution $\mathcal{E}$. In the realm of quantum computing, the evolution of a quantum system is commonly represented by the equation

$$\rho' = \mathcal{E}(\rho). \tag{3}$$

Here, $\mathcal{E}$ is referred to as a *super-operator*. Mathematically, $\mathcal{E}(\,\cdot\,)$ is a linear mapping from $\mathcal{L}(\mathcal{H})$ to $\mathcal{L}(\mathcal{H})$, allowing for the transformation of one quantum state ρ into another ρ'. As stated by the *Kraus representation theorem* [24], $\mathcal{E}$ can be characterized by a finite set of d-by-d matrices $\{\, E_k : 0 \leq k \leq m-1 \,\} \subseteq \mathcal{L}(\mathcal{H})$, where $m \leq d^2$. The expression is given by $\mathcal{E}(\rho) = \sum_{k=0}^{m-1} E_k \rho E_k^\dagger$ for all $\rho \in \mathcal{D}(\mathcal{H})$.

This representation also satisfies the trace-preserving condition $\sum_k E_k^\dagger E_k = I$, where I is the identity matrix on $\mathcal{H}$ and † denotes the complex conjugate and transpose of matrices. In other words, for all $\rho \in \mathcal{D}(\mathcal{H})$, we have $\mathrm{tr}(\mathcal{E}(\rho)) = \mathrm{tr}(\rho)$ by $\mathrm{tr}(\mathcal{E}(\rho)) = \mathrm{tr}(\sum_k E_k \rho E_k^\dagger) = \mathrm{tr}(\sum_k E_k^\dagger E_k \rho) = \mathrm{tr}(\rho)$.

An example of the use of super-operators can be found in Sect. 3.1, where a super-operator is employed to represent the evolution of quantum walks. In the degenerate scenario, the Kraus operator is simplified to only include a *unitary matrix* U on $\mathcal{H}$ (where $U^\dagger U = U^\dagger U = I$), and $\mathcal{E}(\rho) = U\rho U^\dagger$.

Quantum Measurement. To extract information from a quantum state, a *quantum measurement* is performed. This measurement yields a classical outcome which is represented as a probability distribution over the possible results. Mathematically, a quantum measurement is described by a set $\{M_k\}_{k\in\mathcal{O}}$ of positive semi-definite matrices on the state (Hilbert) space $\mathcal{H}$, where $\mathcal{O}$ is a finite set of possible outcomes. The measurement process is probabilistic: if the quantum system is in a state ρ before the measurement, the probability of obtaining the outcome k is given as follows.

$$p_k = \operatorname{tr}(M_k\rho).$$

Note that the measurement $\{M_k\}_{k\in\mathcal{O}}$ satisfies the *unity condition* $\sum_k M_k = I$, which guarantees that the total probability of all outcomes is equal to 1. In other words, $\sum_k \operatorname{tr}(M_k\rho) = \operatorname{tr}(\sum_k M_k\rho) = \operatorname{tr}(\rho) = 1$.

In the case where we want to extract the classical probability distribution μ encoded in the quantum state ρ_μ as shown in Eq. (2), we can choose the measurement $\{M_k = |s_k\rangle\langle s_k|\}_{s_k\in S}$. In this scenario, the measurement probability of obtaining outcome s_k is given by

$$\operatorname{tr}(|s_k\rangle\langle s_k|\rho_\mu) = \sum_l \langle s_l||s_k\rangle\langle s_k|\rho_\mu|s_l\rangle = \langle s_k|\rho_\mu|s_k\rangle = \langle s_k|(\sum_j p_j|s_j\rangle\langle s_j|)|s_k\rangle = p_k.$$

The above equations rely on the mutual orthogonality of the computational basis $\{|s_0\rangle, \ldots, |s_{d-1}\rangle\}$, meaning that $\langle s_j|s_l\rangle = 0$ for $j \neq l$, and also the fact that each $|s_k\rangle$ is normalized, represented by $\langle s_k|s_k\rangle = 1$.

It should be noted that after the measurement, the state will collapse or be altered, depending on the measurement outcome k, which distinguishes quantum computation from classical computation. For instance, in the case of a *projection measurement* denoted as $\{P_k\}_{k\in\mathcal{O}}$, the state after obtaining outcome k is given by $P_k\rho P_k/p_k$ with $p_k = \operatorname{tr}(P_k\rho)$. Here, each positive semi-definite matrix P_k represents a projection operator ($P_k^2 = P_k$), and a specific example of a projection measurement is the above measurement $\{|s_k\rangle\langle s_k|\}_{s_k\in S}$. Another concrete example is provided in Example 2 for the case of quantum walks. For other scenarios involving post-measurement states that are not encountered in this paper, please refer to [23, Section 2.2.3].

3 Quantum Markov Chains

In this section, we present the formal definition of QMCs. For a more detailed discussion, we refer the interested readers to [3].

Definition 1. *A QMC is a tuple $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$, where $\mathcal{H}$ is a finite-dimensional Hilbert space, $\mathcal{E}$ is a super-operator on $\mathcal{H}$, and $\rho_0 \in \mathcal{D}(\mathcal{H})$ is an initial state.*

The execution of $\mathcal{G}$ is naturally described by the trajectory of quantum states:

$$\sigma(\mathcal{G}) := \rho_0, \mathcal{E}(\rho_0), \mathcal{E}^2(\rho_0), \ldots. \tag{4}$$

QMCs are a direct extension of classical Markov chains and can simulate their execution; see the following example.

Example 1 (Classical Markov chains as QMCs). Not surprisingly, any classical Markov chain (S, P, μ_0) can be effectively encoded as a QMC. We can use $\mathcal{H} = \mathrm{span}\{|s_0\rangle, \ldots, |s_{d-1}\rangle\}$ to encode S, and $\mathcal{E}$ can be a super-operator with Kraus operators $\{E_{k,l} = \sqrt{p_{k,l}}|s_l\rangle\langle s_k|\}_{s_k, s_l \in S}$ that encode the probabilities $p_{k,l}$ of P.

It can be easily verified that $\mathcal{E}$ is a valid super-operator, i.e., $\sum_{k,l} E_{k,l}^\dagger E_{k,l} = I$. Furthermore, let $\rho_0 = \sum_{s \in S} \mu_0(s)|s\rangle\langle s|$ encode the initial probability distribution μ_0. Then, the QMC $(\mathcal{H}, \mathcal{E}, \rho_0)$ can fully simulate the behavior of (S, P, μ_0) in the sense that for all $n \geq 0$:

$$\mathcal{E}^n(\rho_0) = \sum_{k=0}^{d-1} \mu_n(s_k)|s_k\rangle\langle s_k| = (\mu_n(s_0), \ldots, \mu_n(s_{d-1})).$$

Here, $\mu_n = \mu_0 P^n$, $\mu_n(s_k)$ represents the k-th entry of μ_n, and $\mathcal{E}^0 = \mathrm{id}_{\mathcal{H}}$, which is the *identity super-operator* with only one Kraus operator $\{I\}$. The proof follows a straightforward induction on n.

3.1 Quantum Walks

The case study of this paper is to explore the new and advanced properties of quantum walks compared to classical random walks by model checking QMCs. This exploration begins with modeling quantum walks using QMCs. Before presenting this, we introduce one-dimensional quantum walks with absorbing boundaries in the following example. Furthermore, we also need the bra-ket notation $|\psi_1\rangle|\psi_2\rangle := |\psi_1\rangle \otimes |\psi_2\rangle \in \mathcal{H}_1 \otimes \mathcal{H}_2$, which represents the composition (tensor product) of $|\psi_1\rangle$ and $|\psi_2\rangle$ in the Hilbert spaces $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively.

Example 2 (Quantum walk with absorbing boundaries). We consider an unbiased quantum walk on a one-dimensional lattice indexed from s_0 to s_d, with the boundaries s_0 and s_d being absorbing.

State Space. Let $\mathcal{H}_p = \mathrm{span}\{|s_0\rangle, \ldots, |s_d\rangle\}$ be a $(d+1)$-dimensional Hilbert space, where the pure state (unit vector) $|s_k\rangle$ represents the position s_k for each $0 \leq k \leq d$. In order to facilitate the evolution of the quantum walk, an additional coin space is required. Let $\mathcal{H}_c$ be the coin (direction) space, which is a 2-dimensional Hilbert space with orthonormal basis states $|L\rangle$ and $|R\rangle$, indicating the directions left and right, respectively. Therefore, the state space of the quantum walk is $\mathcal{H} = \mathcal{H}_p \otimes \mathcal{H}_c$.

Initial State. The initial state is $\rho_0 = |\psi_0\rangle\langle\psi_0|$ with $|\psi_0\rangle = |s_k\rangle|X\rangle$, indicating the initial position s_k and direction X, where $0 \le k \le d$ and $X \in \{L, R\}$.

Evolution. Each step of the walk consists of three operations:

First, measure the current position of the system to determine whether it is absorbing positions s_0 or s_d. If the position is s_0 or s_d, then the walk terminates. The measurement is described by

$$\{M_{yes} = (|s_0\rangle\langle s_0| + |s_n\rangle\langle s_n|) \otimes I_c, M_{no} = I - M_{yes}\}.$$

Here, I_c and I are the identity operators on $\mathcal{H}_c$ and $\mathcal{H}$, respectively. According to the principles of measurements, from the current state ρ, the walk terminates at state ρ_{yes} with probability p_{yes}, and it continues with state ρ_{no}, with probability p_{no}, where

$$p_x = \mathrm{tr}(M_x\rho) \quad \text{and} \quad \rho_x = M_x\rho M_x^\dagger / p_x \quad \text{for} \quad x = yes, no.$$

Second, apply an unbiased "coin-tossing" (unitary) operator on the coin space $\mathcal{H}_c$. This operator, denoted as U_H, is given by:

$$U_H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = |+\rangle\langle L| + |-\rangle\langle R|.$$

Here, $|\pm\rangle = \frac{1}{\sqrt{2}}(|L\rangle \pm |R\rangle)$. The operator U_H represents the *Hadamard operator* on $\mathcal{H}_c$, where the probabilities of going left and right are both equal to 0.5.

Third, perform a shift (unitary) operator on the space $\mathcal{H}$. The shift operator, denoted as U_S, is given by:

$$U_S = \sum_{k=0}^{d} |s_{k\ominus 1}\rangle\langle s_k| \otimes |L\rangle\langle L| + |s_{k\oplus 1}\rangle\langle s_k| \otimes |R\rangle\langle R|.$$

The intuitive meaning of the operator U_S is that the system walks one step left or right based on the coin state on $\mathcal{H}_c$. Here, $\oplus$ and $\ominus$ represent addition and subtraction modulo $d + 1$, respectively.

In summary, in each step, given the current state ρ as the input, the quantum walk transforms ρ into $\rho' = U\rho_{no}U^\dagger$ with a probability p_{no}, where $U = U_S(I_p \otimes U_H)$ and I_p is the identity operator on $\mathcal{H}_p$. Additionally, the walk terminates at state ρ_{yes} with a probability p_{yes}. The resulting state ρ' then serves as the input state for the subsequent step of the quantum walk.

For a better understanding, Appendix A in the full version of our paper [25] provides a visual representation (Fig. 1) that showcases the evolution of the quantum walk in Example 2 from a physical perspective.

Now, we demonstrate how to represent the quantum walk given in Example 2 using a QMC. To begin, we introduce the super-operator $\mathcal{E}$, which incorporates the unitary operator U representing the combined evolution of the second and third operations of the quantum walk. For any given $\rho \in \mathcal{D}(\mathcal{H})$, we let $\mathcal{E}(\rho)$ be

$$\mathcal{E}(\rho) = UM_{no}\rho M_{no}^\dagger U^\dagger + M_{yes}\rho M_{yes}^\dagger.$$

It is worth noting that $M_{no}M_{yes} = 0$ and $M_{yes}^2 = M_{yes}$, indicating that once the QMC terminates, its state remains unchanged. By using induction on the number of steps, we can easily verify that the evolution of the quantum walk can be modeled by the QMC $(\mathcal{H}_p \otimes \mathcal{H}_c, \mathcal{E}, \rho_0 = |\psi_0\rangle\langle\psi_0|)$.

4 Measurement-Based Linear-Time Temporal Logic

In this section, we introduce a specification language called *Measurement-based Linear-time Temporal Logic* (MLTL) for describing the properties of quantum systems. MLTL is similar to ordinary LTL, but its atomic propositions are interpreted in the context of quantum computing. It also expands on the subspace-based atomic propositions introduced by Birkhoff and von Neumann [12].

4.1 Measurement-Based Atomic Propositions

As mentioned in Sect. 2, a quantum measurement is the process of extracting classical information from quantum states. A quantum measurement can be represented by a finite set $\{M_k\}_{k\in\mathcal{O}}$ of positive semi-definite matrices with the unity condition $\sum_k M_k = I$. Each matrix M_k is called a *measurement operator* and it is associated with the probability $\mathrm{tr}(M_k\rho)$ of obtaining the outcome k when measuring the quantum state ρ. Mathematically, a measurement operator M is positive semi-definite and satisfies $M \leq I$, which means that $I - M$ is also positive semi-definite.

In the following discussion, our main focus is on the measurement operator M_k and its associated probability $\mathrm{tr}(M_k\rho)$. Therefore, we will often disregard the specific outcome value k and simply refer to the measurement operator as M. Additionally, we will not explicitly mention the measurement consisting of M, as a binary quantum measurement $\{M, I - M\}$ can be determined based on M itself (as seen in Example 2 with the measurement operators M_{yes} and M_{no}).

Our atomic propositions are designed to estimate the probability of the outcome $\mathrm{tr}(M\rho)$ after measuring the quantum state ρ, given the measurement operator M.

Definition 2. *Given a Hilbert space $\mathcal{H}$,*

1. *an atomic proposition in $\mathcal{H}$ is a pair $\langle M, \mathcal{I}\rangle$, where M represents a measurement operator on $\mathcal{H}$ and $\mathcal{I} \subseteq [0, 1]$ is an interval;*
2. *a state $\rho \in \mathcal{D}(\mathcal{H})$ satisfies $\langle M, \mathcal{I}\rangle$, written $\rho \models \langle M, \mathcal{I}\rangle$, if the outcome probability of the measurement operator M applied to ρ falls within the interval $\mathcal{I}$, that is, $\mathrm{tr}(M\rho) \in \mathcal{I}$.*

In terms of the expressive power of our measurement-based atomic propositions, it is worth noting that they extend the existing atomic propositions in both the classical and quantum domains.

Classical: the atomic proposition $\langle M, \mathcal{I}\rangle$ of ρ expands upon the proposition $\langle s_k, \mathcal{I}\rangle$ of μ in interval linear-time temporal logic. This classical logic has been used to specify (static) properties of classical Markov chains [17] and continuous-time Markov chains [26]. The proposition asserts that the probability of state $s_k \in S$ in distribution μ over S falls within the interval $\mathcal{I}$. The extension is achieved by utilizing $\rho_\mu = \sum_k \mu(s_k)|s_k\rangle\langle s_k|$ in Eq. (2) and $M = |s_k\rangle\langle s_k|$, resulting in $\mathrm{tr}(M\rho_\mu) = \mu(s_k)$. Consequently, $\mathrm{tr}(M\rho_\mu) \in \mathcal{I}$ if and only if $\mu(s_k) \in \mathcal{I}$.

Quantum: furthermore, an atomic proposition $\langle M, \mathcal{I}\rangle$ of a mixed state ρ can encode the subspace-based proposition $\mathcal{X} \subseteq \mathcal{H}$ of a pure state $|\psi\rangle$ in the Birkhoff-von Neumann quantum logic. This proposition asserts that $|\psi\rangle \in \mathcal{X}$. This observation is made by setting $M = P_{\mathcal{X}}$, which represents the projection onto $\mathcal{X}$, $\mathcal{I} = [1, 1]$, and $\rho = |\psi\rangle\langle\psi|$. As a result, we can conclude that $\rho = |\psi\rangle\langle\psi| \models \langle P_{\mathcal{X}}, [1, 1]\rangle$ (i.e., $\mathrm{tr}(P_{\mathcal{X}}\rho) = 1$) is equivalent to $|\psi\rangle \in \mathcal{X}$.

To demonstrate the practicality of our atomic propositions, we will present a series of specific instances in Example 3 later. These examples will effectively illustrate the characteristics and properties of quantum walks.

From an algorithmic perspective, we gather a finite number of pairs $\langle M, \mathcal{I}\rangle$ as the set of atomic propositions that are based on quantum measurements. This set is denoted as AP.

4.2 Quantum Linear-Time Temporal Logic

In this section, we enhance the linear-time temporal logic [1,16] by incorporating the newly introduced measurement-based atomic propositions. This will result in the formation of our measurement-based linear-time temporal logic MLTL.

The MLTL formulas, which involve the use of measurement-based AP, are defined according to the following syntax:

$$\varphi ::= \mathbf{true} \mid a \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 U \varphi_2,$$

where $a = \langle M, \mathcal{I}\rangle \in AP$. We can also derive additional standard Boolean operators and temporal modalities such as $\Diamond$ (*eventually*) and $\Box$ (*always*) using conventional methods.

The semantics of MLTL is also defined in a familiar manner. For any infinite word ξ over 2^{AP} and any MLTL formula φ over AP, the satisfaction relation $\xi \models \varphi$ is defined by induction on the structure of φ:

- $\xi \models \mathbf{true}$ always holds;
- $\xi \models a$ iff $a \in \xi[0]$;
- $\xi \models \neg\varphi$ iff it is not the case that $\xi \models \varphi$ (written $\xi \not\models \varphi$);
- $\xi \models \varphi_1 \vee \varphi_2$ iff $\xi \models \varphi_1$ or $\xi \models \varphi_2$;
- $\xi \models \bigcirc\varphi$ iff $\xi[1+] \models \varphi$; and
- $\xi \models \varphi_1 U \varphi_2$ iff there exists $k \geq 0$ such that $\xi[k+] \models \varphi_2$ and for each $0 \leq j < k$, $\xi[j+] \models \varphi_1$,

where $\xi[k]$ and $\xi[k+]$ denote the $(k+1)$-th element and the $(k+1)$-th suffix of ξ, respectively. The indexes start from zero so that, say, $\xi = \xi[0+]$. In addition, the *semantics* $\mathcal{L}_\omega(\varphi)$ of φ is defined as the language containing all infinite words over 2^{AP} that satisfy φ: $\mathcal{L}_\omega(\varphi) = \{\, \xi \in (2^{AP})^\omega : \xi \models \varphi \,\}$.

Now, let us extend the satisfaction relation $\rho \models a$ to $\mathcal{G} \models \varphi$ between a QMC $\mathcal{G}$ and an MLTL formula φ. To this end, we introduce the labeling function:

$$L\colon \mathcal{D}(\mathcal{H}) \to 2^{AP}, \qquad L(\rho) = \{\, a \in AP : \rho \models a \,\} \tag{5}$$

which assigns to each quantum state ρ the set of atomic propositions in AP satisfied by ρ. We further extend the labeling function to sequences of quantum states by setting $L(\rho_0, \rho_1, \ldots) = L(\rho_0), L(\rho_1), \ldots$ as usual. Then we define:

$$\mathcal{G} \models \varphi \text{ if and only if } L(\sigma(\mathcal{G})) \in \mathcal{L}_\omega(\varphi)$$

where $\sigma(\mathcal{G})$ is the state trajectory of $\mathcal{G}$ as defined in Eq. (4).

We now exhibit realistic settings where our approach leads to valuable insights for the quantum walk presented in Example 2.

Example 3 (Quantum walk with absorbing boundaries, continued). Given a finite set of intervals $\{\mathcal{I}_l \subseteq [0,1]\}_{l=0}^{L}$, let

$$AP = \{\, \langle M_{s_k}, \mathcal{I}_l \rangle : M_{s_k} = |s_k\rangle\langle s_k| \otimes I_c, 0 \le k \le d, 0 \le l \le L \,\}$$

with the atomic proposition $\langle M_{s_k}, \mathcal{I}_l \rangle$ asserting that $\mathrm{tr}(M_{s_k}\rho) \in \mathcal{I}_l$ for $0 \le k \le d$ and $0 \le l \le L$. This allows us to trace the probability distribution on all positions (including boundaries) of the quantum walk.

First, we can discuss the advantages of quantum walks over their classical counterparts, as discovered by Ambainis et al. in [18]. When given the initial state $|s_1\rangle|R\rangle$, the absorbing probability at position 0 tends to $1/\sqrt{2}$ in the limit as $d \to \infty$, whereas in the classical case, the value is 1. This property can be expressed as the MLTL formula $\varphi_0 = \Diamond\Box\langle M_{s_0}, \mathcal{I}_0 \rangle$, where $\mathcal{I}_0 = [1/\sqrt{2} - \gamma, 1/\sqrt{2} + \gamma]$ and $\gamma > 0$ is a given precision parameter.

Next, we examine two properties of interest that demonstrate significant differences between quantum walks and their classical counterparts. To the best of our knowledge, these findings are new.

1. In the classical case, the absorbing probability at position d is always smaller than 0.5 if the walk starts from the middle position and d is even. However, this does not necessarily hold for quantum walks. Let $\varphi_1 = \Box\langle M_{s_d}, \mathcal{I}_1 \rangle$, where $\mathcal{I}_1 = [0, 0.5)$, be the MLTL formula that expresses this property. Assuming the initial state is $|s_{d/2}\rangle|R\rangle$, we will see in Sect. 6 that φ_1 is false when $d = 20$.
2. Let $\mathcal{I}_2 = (0.4, 1]$ and $\varphi_2 = \Box(\langle M_{s_{d-1}}, \mathcal{I}_2 \rangle \implies \langle M_{s_1}, \mathcal{I}_2 \rangle)$, which states that at any given time point, if the probability at position $d-1$ is larger than 0.4, then the probability at position 1 is also larger than 0.4. In the classical case, φ_2 is true due to the symmetry of the distribution over positions. However, as shown in Sect. 6, φ_2 does not hold in the quantum case. Therefore, the distribution of the unbiased quantum walk over positions is asymmetric even when the walk starts from the middle position.

5 Model Checking Algorithm

In this section, we present an algorithm that can be used to approximately verify the satisfaction of $\mathcal{G} \models \varphi$. For the convenience of the reader, we put all proofs of theoretical results in the appendix.

Using the notations in Eq. (5), we can formally define the model checking problem for $\sigma(\mathcal{G})$ against MLTL formulas as follows.

Problem 1. Given a QMC $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$, a labeling function L, and an MLTL formula φ, the task is to decide whether $\mathcal{G} \models \varphi$, which means determining whether $L(\sigma(\mathcal{G})) \in \mathcal{L}_\omega(\varphi)$.

Although MLTL extends LTL with quantum atomic propositions, the traditional model-checking techniques for LTL cannot be directly applied to solve Problem 1. This is because the state space of a QMC is continuous and uncountably infinite, even in the case where the state space is finite-dimensional. In contrast, classical LTL model checking deals with a discrete and finite state set. However, QMCs can simulate Markov chains as seen in Example 1, and interval LTL formulas in [17] can be represented by MLTL formulas as discussed in Sect. 4. Despite this, the counter-example presented in [17] shows that the language $\{L(\sigma(\mathcal{G}))\}$ is generally not ω-regular. Therefore, the standard approach [27] of model checking ω-regular languages cannot directly solve Problem 1 either.

To address this, we turn to the problem of approximate verification for QMCs, following the techniques introduced in [17]. The main idea in [17] involves studying the periodicity of states in *finitely many* sub-chains obtained through BSCC decomposition [1], and a key property of Markov chains known as *periodic stability*, which ensures their stability. However, extending this idea to the quantum setting is not straightforward. It has been proven that the BSCC decomposition of QMCs is not unique, but rather has *infinitely many* possibilities [21,28,29] due to the continuous state space. Additionally, we will demonstrate below that QMCs do not exhibit periodic stability.

Definition 3. *A QMC* $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$ *is called* periodically stable *if there exists an integer* $\theta > 0$ *such that* $\lim_{n\to\infty} \mathcal{E}^{n\theta}(\rho_0) = \rho^*$ *for some limiting quantum state* ρ^**. The smallest value of* θ*, if it exists, is referred to as the* period *of* $\mathcal{G}$ *and is denoted as* $p(\mathcal{G})$*. Moreover,* $\{\, \mathcal{E}^k(\rho^*) : 0 \leq k < p(\mathcal{G}) \,\}$ *are called the* periodically stable states *of* $\mathcal{G}$ *as* $\lim_{n\to\infty} \mathcal{E}^{n\theta}(\mathcal{E}^k(\rho_0)) = \mathcal{E}^k(\rho^*)$*.*

In the classical case, any Markov chain (S, P, μ_0) is periodically stable [30]. This means that there exists an integer $\theta > 0$ such that $\lim_{n\to\infty} \mu_0 P^{n\theta} = \mu^*$ for some limiting distribution μ^*. Furthermore, θ is independent of μ_0. However, this property does not hold for QMCs, as demonstrated by the following example.

Example 4. Let $\mathcal{H} = \text{span}\{|s_0\rangle, |s_1\rangle\}$ and $U = |s_0\rangle\langle s_0| + e^{i2\pi\psi}|s_1\rangle\langle s_1|$ be a unitary operator on $\mathcal{H}$, where ψ is an irrational number. It can be proven that the

QMC $(\mathcal{H}, \mathcal{E}_U, \rho_0)$, where $\mathcal{E}_U(\rho) = U\rho U^\dagger$, is not periodically stable for any generic initial state ρ_0. In fact, a simple calculation reveals that

$$\mathcal{E}_U^{n\theta}(\rho_0) = \rho_{00} \cdot |s_0\rangle\langle s_0| + \rho_{11} \cdot |s_1\rangle\langle s_1| + e^{-i2\pi\psi n\theta}\rho_{01} \cdot |s_0\rangle\langle s_1| + e^{i2\pi\psi n\theta}\rho_{10} \cdot |s_1\rangle\langle s_0|$$

where $\rho_{ij} = \langle s_i|\rho|s_j\rangle$. It is worth noting that since ψ is irrational, the set $\{ e^{i2\pi\psi m} : m \in \mathbb{N} \}$ is dense in the unit circle [31]. Therefore, for any integer $\theta > 0$, the limit $\lim_{n\to\infty} \mathcal{E}_U^{n\theta}(\rho_0)$ cannot exist, except when $\rho_{01} = \rho_{10} = 0$.

Note that the operator $\mathcal{E}_U$ in Example 4 has four eigenvalues (with multiplicity taken into account): 1, 1, $e^{-i2\pi\psi}$, and $e^{i2\pi\psi}$. The corresponding eigenvectors are $|s_0\rangle\langle s_0|$, $|s_1\rangle\langle s_1|$, $|s_0\rangle\langle s_1|$, and $|s_1\rangle\langle s_0|$, respectively. We have proven that the system $(\mathcal{H}, \mathcal{E}_U, \rho_0)$ is periodically stable if and only if the initial state ρ_0 does not have any components in the directions of $|s_0\rangle\langle s_1|$ and $|s_1\rangle\langle s_0|$. Interestingly, this is precisely why a QMC cannot be periodically stable (see Appendix B of the full version of our paper [25]). To put it in another way, a QMC is periodically stable only if the initial quantum state does not contain any components in the directions defined by the eigenvectors of the relevant super-operator (linear operator) corresponding to eigenvalues of the form $e^{i2\pi\psi}$ where ψ is an irrational number. This result also offers an efficient method to verify the periodic stability of a given QMC $\mathcal{G}$ (and determine the period $p(\mathcal{G})$). Specifically, symbolic computation can be used to check whether the eigenvalues of QMCs are irrational by analyzing their algebraic representations and mathematical properties. Such a method can be utilized to confirm the periodic stability of the quantum walk in Example 2. Therefore, the approximate verification technique described in this paper can be applied to the quantum walk.

Proposition 1. *Given a QMC $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$ with $\dim(\mathcal{H}) = d$, there is a way to check whether $\mathcal{G}$ is periodically stable with computational complexity $\mathcal{O}(d^8)$. If it is indeed stable, we can compute the period $p(\mathcal{G})$ with a complexity of $\mathcal{O}(d^8)$.*

The evaluation process for periodic stability, as described in Proposition 1, involves examining the eigenvalues and eigenvectors of the super-operator $\mathcal{E}$. To calculate these eigenvalues and eigenvectors of $\mathcal{E}$ with Kraus operators $\{E_k\}_k$, we can use the matrix representation $\mathcal{M}_\mathcal{E}$ [32] of $\mathcal{E}$, given by $\mathcal{M}_\mathcal{E} = \sum_k E_k \otimes E_k^*$. Here, E_k^* represents the entry-wise complex conjugate of E_k. The linear operator (matrix) $\mathcal{M}_\mathcal{E}$ acts on $\mathcal{H} \otimes \mathcal{H}$. Based on this, we can derive the following lemma:

Lemma 1. *For a non-zero $A \in \mathcal{L}(\mathcal{H})$, A is an eigenvector of $\mathcal{E}$ corresponding to the eigenvalue λ if and only if $|A\rangle$ is an eigenvector of $\mathcal{M}_\mathcal{E}$ corresponding to the eigenvalue λ. In other words, $\mathcal{E}(A) = \lambda A$ if and only if $\mathcal{M}_\mathcal{E}|A\rangle = \lambda|A\rangle$.*

In this context, $|A\rangle \in \mathcal{H} \otimes \mathcal{H}$ represents the vectorization of A, denoted by $|A\rangle := (A \otimes I)|\Omega\rangle$. Here, $|\Omega\rangle$ denotes the (unnormalized) maximally entangled state on $\mathcal{H} \otimes \mathcal{H}$ [23]. Assuming $\mathcal{H} = \operatorname{span}\{|s_0\rangle, \ldots, |s_{d-1}\rangle\}$, the maximally entangled state can be expressed as $|\Omega\rangle = \sum_{k=0}^{d-1} |s_k\rangle|s_k\rangle$. In particular, for a quantum state $\rho \in \mathcal{D}(\mathcal{H})$, we write $|\rho\rangle = (\rho \otimes I)|\Omega\rangle$.

Now, our attention can be directed towards approximately model checking the QMC $\mathcal{G}$ that is periodically stable. To achieve this with a desired level of accuracy ε, we can adopt the following approach. First and foremost, we need to identify the states of the chain that are periodically stable as defined in Definition 3. After a sufficient number of steps, any state on the trajectory $\sigma(\mathcal{G})$ will be close to one of the periodically stable states. This approach can also be applied to the labeled trajectory $L(\sigma(\mathcal{G}))$. By doing so, we can obtain an ω-regular language. Following that, we can utilize the standard Büchi automata approach of model checking ω-regular languages to analyze the obtained language. Therefore, in the following subsections, we will outline the step-by-step process for handling this.

5.1 Periodically Stable States

Given a periodically stable QMC $(\mathcal{H}, \mathcal{E}, \rho_0)$, we can obtain the periodically stable states by employing a specific super-operator called the *stabilizer* of $\mathcal{G}$, denoted by $\mathcal{E}_\phi$. This stabilizer is generated by the super-operator $\mathcal{E}$.

Lemma 2. *If a QMC $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$ is periodically stable with period $p(\mathcal{G})$, then the set $\{\mathcal{E}_\phi(\mathcal{E}^k(\rho_0))\}_{0\leq k<p(\mathcal{G})}$ consists of the periodically stable states of $\mathcal{G}$. The computational complexity of obtaining such a set is $\mathcal{O}(d^8)$, where $d = \dim(\mathcal{H})$.*

The super-operator $\mathcal{E}_\phi$ is constructed from $\mathcal{E}$ by retaining the eigenvectors corresponding to eigenvalues with a magnitude of one, which do not vanish in the evolution $\mathcal{E}^n(\rho)$ as n tends to infinity. Specifically, let $\mathcal{M}_\mathcal{E}$ be the matrix representation of $\mathcal{E}$ with Jordan decomposition $\mathcal{M}_\mathcal{E} = SJS^{-1}$, where $J = \bigoplus_{k=0}^{K} J_k(\lambda_k) = \bigoplus_{k=0}^{K} \lambda_k P_k + N_k$. Here, λ_k represents the eigenvalues of $\mathcal{M}_\mathcal{E}$, P_k is a projector onto the corresponding (generalized) eigenvector space, and N_k is the corresponding nilpotent part. Furthermore, according to [7, Proposition 6.2], the geometric multiplicity of any λ_k with a magnitude of one (i.e., $|\lambda_k| = 1$) is equal to its algebraic multiplicity, i.e., $N_k = 0$. Then we define $J_\phi := \bigoplus_{k:|\lambda_k|=1} P_k$ as the projector onto the eigenspace corresponding to eigenvalues with a magnitude of one. By [7, Proposition 6.3], it is confirmed that $\mathcal{M}_{\mathcal{E}_\phi} = SJ_\phi S^{-1}$ is indeed the matrix representation of some super-operator $\mathcal{E}_\phi$.

5.2 Neighborhood of Quantum States

Now we proceed to introduce the concepts of (symbolic) neighborhoods for (sequences of) quantum states using the labeling function L.

Definition 4. *Let $\rho \in \mathcal{D}(\mathcal{H})$ be a quantum state and $\varepsilon > 0$. The (symbolic) ε-neighborhood $\mathcal{N}_\varepsilon(\rho)$ of ρ is the subset of 2^{AP} defined as*

$$\mathcal{N}_\varepsilon(\rho) := \{\, L(\rho') : \rho' \in \mathcal{D}(\mathcal{H}), \|\rho - \rho'\| < \varepsilon \,\},$$

where $\|A\| := \sqrt{\mathrm{tr}(A^\dagger A)}$ represents the Schatten 2-norm (also known as the Hilbert-Schmidt norm) for the linear operator $A \in \mathcal{L}(\mathcal{H})$.

Now we show that, after a certain number of steps, the symbols $L(\mathcal{E}^n(\rho_0))$ will be enclosed within the ε-neighborhood of one of the periodically stable states.

Lemma 3. *Consider a periodically stable QMC $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$ with period $p(\mathcal{G})$. Let $\eta_k = \mathcal{E}_\phi(\mathcal{E}^k(\rho_0))$, for each $0 \leq k < p(\mathcal{G})$, as the periodically stable states of $\mathcal{G}$. Then for any $\varepsilon > 0$, there exists an integer $K^\varepsilon > 0$ such that for any $n \geq K^\varepsilon$,*

$$L(\mathcal{E}^n(\rho_0)) \in \mathcal{N}_\varepsilon(\eta_{n \bmod p(\mathcal{G})}).$$

Moreover, the time complexity of computing K^ε is in $\mathcal{O}(d^8)$, where $d = \dim(\mathcal{H})$.

With Lemma 3, we can define the concept of the (symbolic) neighborhood of trajectories for periodically stable QMCs.

Definition 5. *Given a periodically stable QMC $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$ and $\varepsilon > 0$, the (symbolic) ε-neighborhood of the trajectory $\sigma(\mathcal{G})$ of $\mathcal{G}$ is defined to be the language $\mathcal{N}_\varepsilon(\sigma(\mathcal{G}))$ over $(2^{AP})^\omega$ such that $\xi \in \mathcal{N}_\varepsilon(\sigma(\mathcal{G}))$ if and only if*

- *$\xi[n] = L(\mathcal{E}^n(\rho_0))$ for all $0 \leq n \leq K^\varepsilon - 1$ and*
- *$\xi[n] \in \mathcal{N}_\varepsilon(\eta_{n \bmod p(\mathcal{G})})$ for all $n \geq K^\varepsilon$,*

where the states $\{\, \eta_k : 0 \leq k < p(\mathcal{G}) \,\}$ and K^ε are as given in Lemma 3.

5.3 Approximate Verification of Quantum Markov Chains

Using Definition 5, we can formulate and address the problem of approximate model checking for QMCs against MLTL formulas in the following manner.

Problem 2. Given a periodically stable QMC $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$, a labeling function L, an MLTL formula φ, and $\varepsilon > 0$, decide whether

1. $\mathcal{G}$ *ε-approximately satisfies φ from below*, denoted $\mathcal{G} \models_\varepsilon \varphi$; that is, whether $\mathcal{N}_\varepsilon(\sigma(\mathcal{G})) \cap \mathcal{L}_\omega(\varphi) \neq \emptyset$;
2. $\mathcal{G}$ *ε-approximately satisfies φ from above*, denoted $\mathcal{G} \models^\varepsilon \varphi$; that is, whether $\mathcal{N}_\varepsilon(\sigma(\mathcal{G})) \subseteq \mathcal{L}_\omega(\varphi)$.

To justify that Problem 2 is indeed an approximate version of Problem 1, we first note that $L(\sigma(\mathcal{G})) \in \mathcal{N}_\varepsilon(\sigma(\mathcal{G}))$. Then we have three cases:

1. if $\mathcal{G} \not\models_\varepsilon \varphi$, then $\mathcal{N}_\varepsilon(\sigma(\mathcal{G})) \cap \mathcal{L}_\omega(\varphi) = \emptyset$, and hence $L(\sigma(\mathcal{G})) \notin \mathcal{L}_\omega(\varphi)$ $(\mathcal{G} \not\models \varphi)$;
2. if $\mathcal{G} \models^\varepsilon \varphi$, then $\mathcal{N}_\varepsilon(\sigma(\mathcal{G})) \subseteq \mathcal{L}_\omega(\varphi)$, and hence $L(\sigma(\mathcal{G})) \in \mathcal{L}_\omega(\varphi)$ $(\mathcal{G} \models \varphi)$;
3. if neither $\mathcal{G} \not\models_\varepsilon \varphi$ nor $\mathcal{G} \models^\varepsilon \varphi$, then whether or not $\mathcal{G} \models \varphi$ is unknown.

If we find ourselves in the third scenario (the unknown case), we have the option to decrease the value of ε by half and then repeat the approximate model-checking process described in the previous scenarios. We can continue this process until we reach either of the first two cases, which will provide us with either a negative or affirmative answer to Problem 1. In exceptional cases, diminishing ε may not lead to termination. To prevent this, a predetermined number

Algorithm 1. ModelCheck($\mathcal{G}, AP, L, \varphi, \varepsilon$)

```
Require: A periodically stable QMC $\mathcal{G} = (\mathcal{H}, \mathcal{E}, \rho_0)$ with Kraus operators $\{E_k\}_k$, a
    finite set of (measurement-based) atomic propositions $AP$, a labeling function $L$,
    an MLTL formula $\varphi$, and $\varepsilon > 0$.
Ensure: true, false, or unknown, where true indicates $\mathcal{G} \models \varphi$, false indicates $\mathcal{G} \not\models$
    $\varphi$, and unknown stands for an inconclusive answer.
 1: Put $\mathcal{M}_\mathcal{E} = \sum_k E_k \otimes E_k^*$
 2: Get $p(\mathcal{G})$ and $M_{\mathcal{E}_\phi} = SJ_\phi S^{-1}$ by the Jordan decomposition form $\mathcal{M}_\mathcal{E} = SJS^{-1}$
 3: Put $|\rho_0\rangle = (\rho_0 \otimes \mathcal{I})|\Omega\rangle$
 4: for each $k \in \{0, 1, \dots, p(\mathcal{G}) - 1\}$ do
 5:     Set $|\eta_k\rangle$ to be $M_{\mathcal{E}_\phi} M_\mathcal{E}^k |\rho_0\rangle$
 6:     Compute $\mathcal{N}_\varepsilon(\eta_k)$ by semi-definite programming
 7: end for
 8: Get $K^\varepsilon$ by solving some inequalities
 9: for each $k \in \{0, 1, \dots, K^\varepsilon - 1\}$ do
10:     Put $\rho_k = \mathcal{E}^k(\rho_0)$ and compute $L(\rho_k)$
11: end for
12: Put $\zeta_k = \eta_{(K^\varepsilon + k) \bmod p(\mathcal{G})}$ for $0 \le k < p(\mathcal{G})$
13: Let $\mathcal{N}_\varepsilon(\sigma(\mathcal{G}))$ be the $\omega$-regular language
            $\{L(\rho_0)\}\{L(\rho_1)\} \cdots \{L(\rho_{K^\varepsilon - 1})\} \cdot (\mathcal{N}_\varepsilon(\zeta_0)\mathcal{N}_\varepsilon(\zeta_1) \cdots \mathcal{N}_\varepsilon(\zeta_{p(\mathcal{G})-1}))^\omega$
14: Construct the NBA $A_\varphi$ for $\varphi$                         // standard construction
15: Construct the NBA $A_\mathcal{G}$ accepting $\mathcal{N}_\varepsilon(\sigma(\mathcal{G}))$// standard lasso-shaped construction
16: if $\mathcal{L}(A_\mathcal{G}) \cap \mathcal{L}(A_\varphi) = \emptyset$ then              // standard Büchi automata operation
17:     return false
18: else if $\mathcal{L}(A_\mathcal{G}) \subseteq \mathcal{L}(A_\varphi)$ then                 // standard Büchi automata operation
19:     return true
20: else
21:     return unknown
22: end if
```

of iterations can be set for reducing epsilon. Determining when the procedure terminates seems difficult, and we would like to leave it as future work.

Finally, to solve Problem 2, we represent $\mathcal{N}_\varepsilon(\sigma(\mathcal{G}))$ in Definition 5 as the ω-regular expression

$$\mathcal{N}_\varepsilon(\sigma(\mathcal{G})) = \{L(\rho_0)\} \cdot \{L(\mathcal{E}(\rho_0))\} \cdots \{L(\mathcal{E}^{K^\varepsilon - 1}(\rho_0))\} \cdot \left(\mathcal{N}_\varepsilon(\zeta_0) \cdots \mathcal{N}_\varepsilon(\zeta_{p(\mathcal{G})-1})\right)^\omega$$

where $\zeta_k = \eta_{(K^\varepsilon + k) \bmod p(\mathcal{G})}$, $0 \le k < p(\mathcal{G})$, and for any two sets X and Y, $X \cdot Y = \{ xy : x \in X, y \in Y \}$. Thus $\mathcal{N}_\varepsilon(\sigma(\mathcal{G}))$ is ω-regular and standard techniques [1,33] can be employed to check $\mathcal{N}_\varepsilon(\sigma(\mathcal{G})) \cap \mathcal{L}_\omega(\varphi) = \emptyset$ and $\mathcal{N}_\varepsilon(\sigma(\mathcal{G})) \subseteq \mathcal{L}_\omega(\varphi)$.

Theorem 1. *The verification problem outlined in Problem 2 can be addressed using Algorithm 1 within a time complexity of* $\mathcal{O}(2^{\mathcal{O}(|\varphi|)} \cdot (K^\varepsilon + p(\mathcal{G})) + d^8)$. *Here,* $d = \dim(\mathcal{H})$, $|\varphi|$ *represents the size of MLTL formula* φ, *and* $p(\mathcal{G})$ *and* K^ε *are as given in Proposition 1 and Lemma 3, respectively.*

Algorithm 1 summarizes our techniques proposed above to answer Problem 2. Starting from lines 1 to 7, we make use of the Jordan decomposition of the matrix representation $\mathcal{M}_{\mathcal{E}}$ of $\mathcal{E}$ to determine the period $p(\mathcal{G})$ and the stabilizer $\mathcal{E}_{\phi}$. These computations allow us to obtain the periodically stable states $\{\eta_k\}_{k=0}^{p(\mathcal{G})-1}$ of $\mathcal{G}$ and their corresponding symbolic ε-neighborhood $\{\mathcal{N}_{\varepsilon}(\eta_k)\}_{k=0}^{p(\mathcal{G})-1}$ based on the given approximation level ε. The steps involved in these computations utilize Proposition 1 and Lemma 2. Afterwards, in line 8, we determine the truncation number K^{ε} using Lemma 3. Subsequently, we compute $\{L(\rho_k)\}_{k=0}^{K^{\varepsilon}-1}$, which represents the symbols in AP of the first K^{ε} quantum states in the trajectory $\sigma(\mathcal{G})$, from lines 9 to 11. Finally, in lines 12 and 13, we obtain an ω-regular language $\mathcal{N}_{\varepsilon}(\sigma(\mathcal{G}))$ that represents the symbolic neighborhoods of the evolution $\sigma(\mathcal{G})$. The Büchi automaton A_{φ} for the MLTL formula φ is constructed at line 14 by means of a standard construction for an LTL formula φ (see, e.g., [33]) while the Büchi automaton $A_{\mathcal{G}}$ at line 15 is obtained by an ordinary lasso-shaped construction: it is enough to insert a new state between each letter, make the state joining the stem and the lasso part accepting, and use the accepting state as the target of the last action in the lasso. The two operations on Büchi automata at lines 16 and 18 are standard: intersection and emptiness reduce to automata product and strongly connected components decomposition, respectively, which require quadratic time (cf. [33]). Language inclusion, however, in general, requires exponential time and is PSPACE-complete (cf. [33]); in our case, however, we can remain in quadratic time by replacing the check $\mathcal{L}(A_{\mathcal{G}}) \subseteq \mathcal{L}(A_{\varphi})$ with $\mathcal{L}(A_{\mathcal{G}}) \cap \mathcal{L}(A_{\neg\varphi}) = \emptyset$, since constructing the Büchi automata A_{φ} and $A_{\neg\varphi}$ requires the same asymptotic effort (cf. [33, Section 4.6.1]).

Remark 1. The complexity of approximately verifying MCs against interval LTL formulas in the work of Agrawal et al. [17] is challenging to analyze and remains unknown. This is because the BSCC decomposition analysis, which is relied upon, is not an analytical method. Our model-checking algorithm, however, overcomes this by utilizing the Jordan normal form to create a linear-algebraic representation of the graph-theoretic (BSCC-based) model-checking algorithm mentioned in [17]. Consequently, our model-checking algorithm (Algorithm 1) can effectively verify MC models against interval LTL formulas. This capability stems from the ability of QMCs to emulate MCs, as illustrated in Example 1, and the greater expressiveness of our MLTL compared to interval LTL in [17], as discussed in Sect. 4. Notably, MCs are inherently periodically stable, which extends to the modeled QMCs, making our model-checking algorithm suitable for this context. Additionally, while Agrawal et al. [17] did not offer a complexity analysis for their algorithm, our method establishes the first upper bound on the computational complexity of approximate model-checking for MCs against interval LTL specifications from their work. Specifically, when using our algorithm for model checking MCs, the complexity is reduced to $\mathcal{O}(2^{\mathcal{O}(|\varphi|)} \cdot (K^{\varepsilon} + p(\mathcal{G})) + d^4)$, where d represents the number of states in the MC. This reduction is due to the fact that the complexity of computing the Jordan decomposition of the d-by-d stochastic matrix P in MCs is $\mathcal{O}(d^4)$. As a result, the time complexity of calculating K^{ε} and $p(\mathcal{G})$ is also reduced to $\mathcal{O}(d^4)$.

6 Case Studies

To demonstrate the utility of the model-checking techniques proposed in this paper, we conducted case studies on quantum walks to investigate their temporal linear-time properties. We completed a prototype for implementing our model-checking algorithm and used it to automatically model check all MLTL formulas provided in Example 3. The prototype was built using Python for the quantum part to generate an ω-regular language and Java for the classical part to model check the language against LTL formulas by calling Spot, a platform for LTL and ω-automata manipulation [34].

Before conducting the verification process, it is crucial to ensure the periodic stability of the QMC model for the quantum walk in Example 2. By Proposition 1, this can be achieved by computing the eigenvalues of the model and confirming that they only have 1 as the eigenvalue with a magnitude of one. Armed with this information, we can then employ Algorithm 1 to verify the properties outlined in Example 3. The experimental results for these verifications can be found in Table 1.

The first experiment in Table 1 confirms the advantages of quantum walks over classical random walks, which was previously established by Ambainis et al. in [18]. The remaining two experiments aim to uncover new properties of quantum walks. Moreover, we also utilized Algorithm 1 to verify these same properties for one-dimensional unbiased classical random walks with absorbing boundaries, which yielded contrasting results (**false** for the first experiment and **true** for the last two experiments in Table 1). This confirms that these properties are unique phenomena specific to quantum walks.

Table 1. The experimental result for position number $d = 20$ and different initial states $\rho_0 = |\psi_0\rangle\langle\psi_0|$.

$\vert\phi_0\rangle$	Formula φ	Parameter ε		
		0.5	0.25	0.125
$\vert s_1\rangle\vert R\rangle$	$\Diamond\Box\langle M_{s_0}, [1/\sqrt{2}-0.1, 1/\sqrt{2}+0.1]\rangle$	**unknown**	**unknown**	**true**
$\vert s_{10}\rangle\vert R\rangle$	$\Box\langle M_{s_{20}}, [0, 0.5)\rangle$	**unknown**	**unknown**	**false**
	$\Box(\langle M_{s_{19}}, (0.4, 1]\rangle \implies \langle M_{s_1}, (0.4, 1]\rangle)$	**unknown**	**unknown**	**false**

7 Conclusion

In this paper, we proposed a new quantum logic called measurement-based linear-time temporal logic (MLTL) to specify the quantitative properties of quantum Markov chains (QMCs). For model checking MLTL formulas, an algorithm was developed. The measurement-based atomic propositions of MLTL build upon the subspace-based quantum atomic propositions introduced by Birkhoff and von Neumann [12]. Furthermore, we demonstrated the practical applicability of our model-checking algorithm in quantum walks. This not only confirms the previously established advantages discovered by Ambainis et al. [18] of quantum walks over classical random walks, but also uncovers new phenomena that are unique to quantum walks.

As future work, we note that the quantum walk in Example 2 can also be written as a while-loop quantum program [32], and the absorbing probabilities are exactly the termination probabilities of the program. Indeed, any quantum program written in the WHILE language can be modeled by a QMC [6]. So our model-checking approach can be naturally applied in the verification of quantum program properties specified as MLTL formulas. Therefore, we plan to apply our model-checking algorithms to verify quantum programs. Moreover, it would be intriguing to broaden the application of the methods presented in this paper to verify the behavior of non-periodically stable QMCs. One possible approach is that any non-periodically stable QMC will be close to a periodically stable one with arbitrary precision, as any irrational number (eigenvalue) will be close to a rational number (eigenvalue) with the same precision.

Acknowledgments. We would like to thank the anonymous reviewers for their valuable suggestions. This work was partly supported by the Youth Innovation Promotion Association, Chinese Academy of Sciences (Grant No. 2023116), the Key Research Program of the Chinese Academy of Sciences (Grant No. ZDRW-XX-2022-1), and the Australian Research Council (Grant No: DP220102059).

This work is part of the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant no. 101008233.

References

1. Baier, C., Katoen, J.-P.: Principles of model checking. MIT press (2008)
2. Ying, M., Feng, Y.: Model-checking quantum systems. Nat. Sci. Rev. **6**(1), 28–31 (2019)
3. Ying, M., Feng, Y.: Model Checking Quantum Systems: Principles and Algorithms. Cambridge University Press (2021)
4. Ticozzi, F., Viola, L.: Quantum Markovian subsystems: Invariance, attractivity, and control. IEEE Trans. Autom. Control **53**(9), 2048–2063 (2008)
5. Guan, J., Feng, Y., Ying, M.: The structure of decoherence-free subsystems. arXiv preprint arXiv:1802.04904 (2018)
6. Ying, M.: Foundations of Quantum Programming. Morgan Kaufmann (2016)
7. Wolf, M.M.: Quantum channels & operations: Guided tour. Lecture notes available at. https://mediatum.ub.tum.de/doc/1701036/document.pdf (2012)
8. Ambainis, A.: Quantum walks and their algorithmic applications. Inter. J. Quantum Inform. **1**, 507–518 (2003)
9. Kempe, J.: Quantum random walks: an introductory overview. Contemp. Phys. **44**(4), 307–327 (2003)
10. Attal, S., Petruccione, F., Sabot, C., Sinayskiy, I.: Open quantum random walks. J. Stat. Phys. **147**(4), 832–852 (2012)
11. Feng, Y., Nengkun, Yu., Ying, M.: Model checking quantum Markov chains. J. Comput. Syst. Sci. **79**(7), 1181–1198 (2013)
12. Birkhoff, G., von Neumann, J.: The logic of quantum mechanics. Annals Math., 823–843 (1936)

13. Moore, C., Crutchfield, J.P.: Quantum automata and quantum grammars. Theoretical Comput. Sci. **237**(1–2), 275–306 (2000)
14. Ying, M., Li, Y., Nengkun, Yu., Feng, Y.: Model-checking linear-time properties of quantum systems. ACM Trans. Comput. Logic (TOCL) **15**(3), 22 (2014)
15. Li, Y., Ying, M.: (Un)decidable problems about reachability of quantum systems. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 482–496. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_33
16. Pnueli, A.: The temporal logic of programs. In : 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pp. 46–57. IEEE Computer Society (1977)
17. Agrawal, M., Akshay, S., Genest, B., Thiagarajan, P.S.: Approximate verification of the symbolic dynamics of Markov chains. J. ACM (JACM) **62**(1), 2 (2015)
18. Ambainis, A., Bach, E., Nayak, A., Vishwanath, A., Watrous, J.: One-dimensional quantum walks. In: Proceedings of the thirty-third annual ACM symposium on Theory of computing (STOC), pp 37–49. ACM (2001)
19. Feng, Y., Hahn, E.M., Turrini, A., Zhang, L.: QPMC: a model checker for quantum programs and protocols. In: Bj$\mathcal{O}$rner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 265–272. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_17
20. Feng, Y., Hahn, E.M., Turrini, A., Ying, S.: Model checking omega-regular properties for quantum Markov chains. In: LIPIcs-Leibniz International Proceedings in Informatics, vol. 85. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
21. Ying, S., Feng, Y., Yu, N., Ying, M.: Reachability probabilities of quantum markov chains. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 334–348. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_24
22. Venegas-Andraca, S.E.: Quantum walks: a comprehensive review. Quantum Inf. Process. **11**(5), 1015–1106 (2012)
23. Nielsen, M.A., Chuang, I.L.: Quantum computation and quantum information. Cambridge University Press (2010)
24. Choi, M.-D.: Completely positive linear maps on complex matrices. Linear Algebra Appl. **10**(3), 285–290 (1975)
25. Guan, J., Feng, Y., Turrini, A., Ying, M.: Measurement-based verification of quantum markov chains. arXiv preprint arXiv:2405.05825 (2024)
26. Guan, J., Yu, N.: A probabilistic logic for verifying continuous-time markov chains. In: TACAS 2022. LNCS, vol. 13244, pp. 3–21. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_1
27. Vardi, M.Y.: Probabilistic linear-time model checking: an overview of the automata-theoretic approach. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 265–276. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48778-6_16
28. Baumgartner, B., Narnhofer, H.: The structures of state space concerning quantum dynamical semigroups. Rev. Math. Phys. **24**(02), 1250001 (2012)
29. Guan, J., Feng, Y., Ying, M.: Decomposition of quantum Markov chains and its applications. J. Comput. Syst. Sci. (2018)
30. Gallager, R.G.: Discrete stochastic processes, vol. 321. Springer Science & Business Media (2012)
31. Hardy, G.H., Wright, E.M.: An introduction to the theory of numbers. Oxford university press (1979)
32. Ying, M., Nengkun, Yu., Feng, Y., Duan, R.: Verification of quantum programs. Sci. Comput. Program. **78**, 1679–1700 (2013)

33. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
34. Duret-Lutz, A., et al.: From Spot 2.0 to Spot 2.10: What's new? In: Proceedings of the 34th International Conference on Computer Aided Verification (CAV 2022), vol. 13372 LNCS. pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9

Simulating Quantum Circuits by Model Counting

Jingyi Mei(✉), Marcello Bonsangue, and Alfons Laarman

Leiden University, Leiden, The Netherlands
{j.mei,m.m.bonsangue,a.w.laarman}@liacs.leidenuniv.nl

Abstract. Quantum circuit compilation comprises many computationally hard reasoning tasks that lie inside #P and its decision counterpart in PP. The classical simulation of universal quantum circuits is a core example. We show for the first time that a strong simulation of universal quantum circuits can be efficiently tackled through weighted model counting by providing a linear-length encoding of Clifford+T circuits. To achieve this, we exploit the stabilizer formalism by Knill, Gottesmann, and Aaronson by reinterpreting quantum states as a linear combination of stabilizer states. With an open-source simulator implementation, we demonstrate empirically that model counting often outperforms state-of-the-art simulation techniques based on the ZX calculus and decision diagrams. Our work paves the way to apply the existing array of powerful classical reasoning tools to realize efficient quantum circuit compilation; one of the obstacles on the road towards quantum supremacy.

Keywords: Quantum Computing · Quantum Circuit Simulation · Satisfiability · #SAT · Weighted Model Counting · Stabilizer Formalism

1 Introduction

Classical simulation of quantum computing [3,17,70,77] serves as a crucial task in the verification [6,37,56,68], synthesis [2,13,62] and optimization [15,63] of quantum circuits. In addition, improved classical simulation methods aid the search for a quantum supremacy [8,39]. Due to the inherent exponential size of the underlying representations of quantum states and operations, classical simulation of quantum circuits is a highly non-trivial task that comes in two flavors [55]: Weak simulators only sample the probability distribution over measurement outcomes, i.e, they implement the “bounded-error” BQP-complete problem that a quantum computer solves, whereas strong circuit simulators can compute the amplitude of any basis state, solving a #P-complete problem [28,40].

Like SAT solvers [12,30], (weighted) model counting (WMC) tools have shown great potential for solving problem instances arising from industrial applications [54,60], despite the #P-completeness of the problem of counting the number (or weights) of satisfying assignments. Because strong quantum circuit

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 555–578, 2024.
https://doi.org/10.1007/978-3-031-65633-0_25

simulation is #P-complete, WMC is a natural fit that nonetheless has not yet been exploited. To do so, we provide the first encoding of the strong simulation problem of universal quantum circuits as a WMC problem. Perhaps surprisingly, this encoding is linear in the number of gates m or qubits n, i.e., $\mathcal{O}(n+m)$.

One of the key properties of our encoding is that it does not use complex numbers to encode the probability amplitudes of quantum states, which would prohibit the use of all modern model counters. We achieve this by exploiting a generalization [76] of the Gottesman-Knill theorem [35], which in effect rewrites the density matrix of any quantum state as a linear combination of stabilizer states [33] with real coefficients. It turns out that many exact WMC tools do support negative weights out of the box. Our encoding thus empowers them to reason directly about the constructive and destructive interference ubiquitous in quantum algorithms.

Like many, our method builds on the Solovay-Kitaev theorem [27] that in particular shows that the Clifford+T gate set is universal for quantum computing, meaning that this gate set can efficiently approximate any unitary operator. Since our encoding also supports arbitrary rotation gates (phase shift P, R_X, R_Y and R_Z), and since these rotations are non-Clifford gates, it can also support other universal gate sets like Clifford+R_X or Clifford+P, which allows for easier approximations. One important example is Quantum Fourier Transformation (QFT), which can be simulated with $O(n)$ rotation gates while needs at least $O(n\log(n))$ T gates to approximate [51]. Moreover, since the hardness of exact reasoning about quantum circuits depends on the gate set (the classes EQP and NQP are parameterized by the gate set as it defines the realizable unitaries), this flexibility significantly enhances the power of our strong simulation approach.

We implement our encoding in an open-source tool QCMC. We demonstrate the scalability and feasibility of the proposed encoding through experimental evaluations based on three classes of benchmarks: random Clifford+T circuits mimicking quantum chemistry applications [75] and oracles, and quantum algorithms from MQT bench [58]. We compare the results of our method against state-of-the-art circuit simulation tools QuiZX [41] and Quasimodo [64], respectively based on the ZX-calculus [25] and CFLOBDD [65]. QCMC simulates important quantum algorithms like QAOA, W-state, VQE, and others which can not be directly supported by QuiZX. Additionally, QCMC outperforms Quasimodo on almost all random circuits and uses orders of magnitudes less memory than Quasimodo on all benchmarks.

In sum, this paper makes the following contributions.

- a generalized stabilizer formalism defined in terms of stabilizer groups, which form a basis for our encoding;
- the first encoding for circuits in various universal quantum gate sets as a weighted model counting problem, which is also linear in size;
- an implementation based on the weighted model counting tool GPMC;
- new benchmarks for the WMC competition [4], and insights on improving model counters and samplers for applications in quantum computing.

2 Preliminaries

2.1 Quantum Computing

Similar to bits in classical computing, a quantum computer operates on quantum bits, or short as *qubits*. A bit is either 0 or 1, while a qubit can be in states $|0\rangle$, $|1\rangle$ or a *superposition* of both. Here '$|\rangle$' is the *Dirac notation*, representing a unit column vector, i.e., $|0\rangle = [1,0]^{\mathrm{T}}$ and $|1\rangle = [0,1]^{\mathrm{T}}$, while $\langle\psi|$ denotes the complex conjugate and transpose of $|\psi\rangle$, that is a row vector: $\langle\psi| = |\psi\rangle^{\dagger}$.

In this paper, we fix n to be the number of the qubits. Let $\mathcal{H}$ be a Hilbert space. An n-qubit quantum state is a 2^n-dimension unit column vector in $\mathcal{H}$. In the case of $n = 1$, a pure state $|\psi\rangle$ is written as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers in $\mathbb{C}$ satisfying $|\alpha|^2 + |\beta|^2 = 1$. To extend from single-qubit states to multiple-qubit states we use the tensor or Kronecker product, which is defined as follows: given $r_A \times c_A$ matrix A and $r_B \times c_B$ matrix B, the $r_A r_B \times c_A c_B$ matrix $A \otimes B$ is

$$A \otimes B = \begin{bmatrix} A_{00}B & A_{01}B & \dots & A_{0c_A}B \\ A_{10}B & A_{11}B & \dots & A_{1c_A}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{r_A0}B & A_{r_A1}B & \dots & A_{r_Ac_A}B \end{bmatrix}.$$

A *computational basis state* $|\vec{x}\rangle$ for $\vec{x} \in \{0,1\}^n$ is a vector with all entries set to 0 except at index $\vec{x}$, which is 1. For two computational basis states $|\vec{x}\rangle, |\vec{y}\rangle$, we have that $|\vec{x}\rangle \otimes |\vec{y}\rangle = |\vec{x}\vec{y}\rangle$. For example, a two-qubit state $|0\rangle \otimes |0\rangle$ equals $|00\rangle = [1,0,0,0]^{\mathrm{T}}$. Sometimes we represent a pure quantum state $|\psi\rangle$ by the density operator obtained as the product $|\psi\rangle\langle\psi|$ of the state with itself. Density operators U are Hermitian matrices, i.e., they satisfy $U^{\dagger} = U$.

Operations on quantum states are given by *quantum gates*. For an n-qubit quantum system, a (global) quantum gate is a function $\mathbf{G}: \mathcal{H} \to \mathcal{H}$, which can be described by $2^n \times 2^n$ *unitary matrix* U, i.e., with the property that $UU^{\dagger} = U^{\dagger}U = I$. A quantum gate is *local* when it works on a subspace of a quantum system, which can be extended to a *global quantum gate* by applying identity operators on unchanged qubits, i.e. a local quantum gate U on qubit j can be represented as a global quantum gate $U_j = I^{\otimes j} \otimes U \otimes I^{\otimes n-j-1}$. Examples of quantum gates are the 2×2 *Pauli matrices* (or *Pauli gates*) X, Y, Z, and the identity matrix I:

$$\sigma[10] \triangleq X \triangleq \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \sigma[11] \triangleq Y \triangleq \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \sigma[01] \triangleq Z \triangleq \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \sigma[00] \triangleq I \triangleq \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Together with the identity matrix I, the Pauli matrices form a basis for 2×2 Hermitian matrix space. Let PAULI_n be the set of the tensor product of n Pauli operators (a "Pauli string"). The so-called Pauli group is generated by multiplication of the local Pauli operators X, Z as follows: $\mathcal{P}_n = \langle X_0, Z_0 \dots, X_{n-1}, Z_{n-1}\rangle$. Structurally, the Pauli group can be written as

$\mathcal{P}_n = \{\lambda P \mid P \in \text{PAULI}_n, \lambda \in \{\pm 1, \pm i\}\}$. For instance, we have $-X \otimes Y \otimes Z \in \mathcal{P}_3$. The Pauli strings form a basis for the full Hermitian matrix space [48,52].

Let $[m]$ be $\{0, \ldots, m-1\}$. The evolution of a quantum system can be modeled by a *quantum circuit*, which is a sequence of quantum gates $C \equiv (\mathbf{G}^0, \ldots, \mathbf{G}^{m-1})$, Here $\mathbf{G}^t$ represents a global quantum gate at time step $t \in [m]$. Let U^t be the unitary matrix for $\mathbf{G}^t$. Then C can be expressed by the unitary matrix $U = U^{m-1} \cdots U^0$.

An important class of quantum circuits is the so-called *Clifford group*, as it can describe interesting quantum mechanical phenomena such as entanglement, teleportation, and superdense encoding. More importantly, they are widely used in quantum error-correcting codes [18,66] and measurement-based quantum computation [59]. The Clifford group is a set of unitary operators that map the Pauli group to itself through conjugation, i.e. all the $2^n \times 2^n$ unitary matrices U such that $UPU^\dagger \in \mathcal{P}_n$ for all $P \in \mathcal{P}_n$. It is generated by the local Hadamard (H) and phase (S) gate, and the two-qubit control-not gate (CX, $CNOT$):

$$H \triangleq \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad S \triangleq \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \quad \text{and } CX \triangleq \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Recall that U_j performs U on j-th qubit. Similarly, we denote by CX_{ij} the unitary operator taking the i-th qubit as the control qubit and j-qubit as the target to execute a controlled-not gate. Clifford circuits are circuits only containing gates from the Clifford group.

A (projective) measurement is given by a set of *projectors* $\{\mathbb{P}_0, \ldots, \mathbb{P}_{k-1}\}$ — one for each measurement outcome $[k]$— satisfying $\sum_{j \in [k]} \mathbb{P}_j = I$. A linear operator $\mathbb{P}$ is a projector if and only if $\mathbb{P}\mathbb{P} = \mathbb{P}$. For example, given a three-qubit system, measuring the first two qubits under computational basis is given by the measurement $\{\ |0\rangle\langle 0| \otimes |0\rangle\langle 0| \otimes I,\quad |1\rangle\langle 1| \otimes |0\rangle\langle 0| \otimes I,\quad |0\rangle\langle 0| \otimes |1\rangle\langle 1| \otimes I,\quad |1\rangle\langle 1| \otimes |1\rangle\langle 1| \otimes I\ \}$.

Weak simulation is the problem of sampling the measurement outcomes according to the probability distribution induced by the semantics of the circuit. In this work, we focus on strong simulation as defined in Definition 1. Here we assume, without loss of generality, that a circuit is initialized to the all-zero state: $|0\rangle^{\otimes n}$.

Definition 1. (Strong simulation [21]). *Given an n-qubit quantum circuit C and a measurement $M = \{\mathbb{P}_0, \ldots, \mathbb{P}_{k-1}\}$, a strong simulation of circuit C computes the probability of getting any outcome $l \in [k]$, that is, the value $\langle 0|^{\otimes n} C^\dagger \mathbb{P}_l C |0\rangle^{\otimes n}$, up to a number of desired bits of precision.*

The Gottesman-Knill theorem [35] shows that Clifford circuits can be strongly stimulated by classical algorithms in polynomial time and space.

2.2 Stabilizer Groups

The stabilizer formalism [35] is a subset of quantum computing that can be effectively simulated on a classical computer. A state $|\varphi\rangle$ is said to be *stabilized* by a quantum unitary operator U if and only if it is a $+1$ eigenvector of U, i.e., $U|\varphi\rangle = |\varphi\rangle$. For example, we say $|0\rangle$ is *stabilized* by Z as $Z|0\rangle = |0\rangle$. Similarly, $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ is stabilized by X, and all states are stabilized by I. The *stabilizer states* form a strict subset of all quantum states which can be uniquely described by maximal commutative subgroups of the Pauli group $\mathcal{P}_n$, which is called *stabilizer group*. The elements of the stabilizer group are called *stabilizers*. Recall that the Clifford group is formed by unitary operators mapping the Pauli group to itself. This leads to the fact that stabilizer states are closed under operators from the Clifford group.

Given an n-qubit stabilizer state $|\psi\rangle$, let $\mathcal{S}_{|\psi\rangle}$ be the stabilizer group of $|\psi\rangle$. While the elements of a Pauli group $\mathcal{P}_n$ either commute or anticommute, a stabilizer group $\mathcal{S}$ must be abelian, because if $P_1, P_2 \in \mathcal{S}_{|\varphi\rangle}$ anticommute, i.e, $P_1P_2 = -P_2P_1$, there would be a contradiction: $|\varphi\rangle = P_1P_2|\varphi\rangle = -P_2P_1|\varphi\rangle = -|\varphi\rangle$. In particular, $-I^{\otimes n}$ can never be a stabilizer. In fact, a subgroup $\mathcal{S}$ of $\mathcal{P}_n$ is a stabilizer group for an n-qubit quantum state if and only if it is an abelian group without $-I^{\otimes n}$. Therefore, for any Pauli string $P \in \text{PAULI}_n$, if $\lambda P \in \mathcal{S}$, then it holds that $\lambda = \pm 1$, since for all $\lambda P \in \mathcal{S}$, we have $(\lambda P)|\varphi\rangle = (\lambda P)^2|\varphi\rangle = \lambda^2 I|\varphi\rangle = \lambda^2|\varphi\rangle \Rightarrow \lambda = \pm 1$.

Any stabilizer group $\mathcal{S}$ can be specified by a set of generators $\mathcal{G}$ so that every element in $\mathcal{S}$ can be obtained through matrix multiplication on $\mathcal{G}$, denoted as $\langle \mathcal{G} \rangle = \mathcal{S}$. The set of generators $\mathcal{G}$ needs not to be unique and has order $|\mathcal{G}| = n$, where n represents the number of qubits, and the corresponding stabilizer group $\mathcal{S}$ has order $2^{|\mathcal{G}|}$.

Example 1. The Bell state $|\Phi_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ can be represented by the following stabilizer generators written in square form:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \equiv \left\langle \begin{matrix} X \otimes X \\ Z \otimes Z \end{matrix} \right\rangle \equiv \left\langle \begin{matrix} X \otimes X \\ -Y \otimes Y \end{matrix} \right\rangle. \qquad \triangle$$

We can relate the (generators of the) stabilizer group directly to the stabilizer state $|\psi\rangle$, as the density operator of the stabilizer state can be written in a linear combination of Pauli matrices as follows [69].

$$|\psi\rangle\langle\psi| = \prod_{G \in \mathcal{G}_{|\psi\rangle}} \frac{I + G}{2} = \frac{1}{2^n} \sum_{S \in \mathcal{S}_{|\psi\rangle}} S. \tag{1}$$

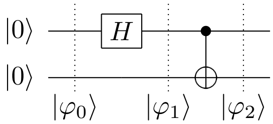
where $S = \pm P$ for $P \in \text{PAULI}_n$ in Clifford circuits. If a Clifford gate U is applied to the above state, i.e. $U|\psi\rangle$, and let $S \in \mathcal{S}_{|\psi\rangle}$, the corresponding stabilizers of $U|\psi\rangle$ can be obtained by $USU^\dagger$ since $USU^\dagger U|\psi\rangle = U|\psi\rangle$. Thus we have $\mathcal{S}_{U|\psi\rangle} = \{USU^\dagger \mid S \in \mathcal{S}_{|\psi\rangle}\}$ and the density operator of the resulting state will be obtained by conjugating U on $|\psi\rangle\langle\psi|$, i.e. $U|\psi\rangle\langle\psi|U^\dagger = \frac{1}{2^n}\sum_{S \in \mathcal{S}_{|\psi\rangle}} USU^\dagger$. To be specific, consider performing a Clifford gate U_j, denoting a single qubit gate U applied to j-th qubit as given in previous section, to a stabilizer $S = \pm P_0 \otimes \ldots \otimes P_{n-1}$. We have $U_jSU_j^\dagger = \pm P_0 \otimes \ldots \otimes UP_jU^\dagger \otimes \ldots \otimes P_{n-1}$. Since U_j is a Clifford gate, $U_jSU_j^\dagger \in \mathcal{P}_n$. Thus the j-th entry needs to be updated, which can be done in constant time

Table 1. Lookup table for the action of conjugating Pauli gates by Clifford gates. The subscripts "c" and "t" stand for "control" and "target". Adapted from [32].

Gate	In	Out	Gate	In	Out
H	X	Z	CX	$I_c \otimes X_t$	$I_c \otimes X_t$
	Y	$-Y$		$X_c \otimes I_t$	$X_c \otimes X_t$
	Z	X		$I_c \otimes Y_t$	$Z_c \otimes Y_t$
S	X	Y		$Y_c \otimes I_t$	$Y_c \otimes X_t$
	Y	$-X$		$I_c \otimes Z_t$	$Z_c \otimes Z_t$
	Z	Z		$Z_c \otimes I_t$	$Z_c \otimes I_t$

following the rules in Table 1 for H and S. Applying a two-qubit gate CX_{ij} is similar to updating the sign and Pauli matrices in i-th and j-th position (see Table 1), which also takes constant time. Overall, updating all generators after performing one Clifford gate can be done in $O(n)$ time.

Example 2. The Bell state $|\Phi_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is a stabilizer state, as it can be obtained by the following circuit, which evaluates to $CX_{01} \cdot H_0 \cdot |00\rangle = |\Phi_{00}\rangle$:



We can simulate the above circuit with the stabilizer formalism. The stabilizer generator set for each time step can be obtained using the rules shown in Table 1:

$$\left\langle \begin{matrix} Z \otimes I \\ I \otimes Z \end{matrix} \right\rangle \xrightarrow{H_0} \left\langle \begin{matrix} HZH^\dagger \otimes I \\ HIH^\dagger \otimes Z \end{matrix} \right\rangle = \left\langle \begin{matrix} X \otimes I \\ I \otimes Z \end{matrix} \right\rangle \xrightarrow{CX_{0,1}} \left\langle \begin{matrix} CX(X \otimes I)CX^\dagger \\ CX(I \otimes Z)CX^\dagger \end{matrix} \right\rangle = \left\langle \begin{matrix} X \otimes X \\ Z \otimes Z \end{matrix} \right\rangle . \qquad \triangle$$

The above explains the essence of the Gottesmann-Knill theorem [35], which states that Clifford circuits can be classically simulated by describing n-qubit stabilizer states by their stabilizer generator set instead of a 2^n complex vector. It is important to note that Clifford gates do not constitute a universal set of quantum gates. However, by adding any non-Clifford gate, such as the $T = |0\rangle\langle 0| + e^{i\pi/4}|1\rangle\langle 1|$ gate, it becomes possible to approximate any unitary operator with arbitrary accuracy, as shown in [14,42]. Moreover, the space of n-qubit density matrices has a basis in the n-qubit stabilizer states [33], which allows us in Sect. 3 to extend the stabilizer formalism to a general quantum state.

2.3 Weighted Model Counting

Weighted model counters can solve probabilistic reasoning problems with real-valued probabilities like Bayesian inference [20]. By reinterpreting quantum states in the stabilizer-state basis, we obviate the need for complex amplitudes typical in quantum computing, as shown in the next section. It turns out that existing weighted model counting tools, like GPMC [36], already support negative weights (see Sect. 4).

Let $\mathbb{B}$ be the Boolean set $\{0, 1\}$. A propositional formula $F\colon \mathbb{B}^V \Rightarrow \mathbb{B}$ over a finite set of Boolean variables V is *satisfiable* if there is an assignment $\alpha \in \mathbb{B}^V$ such that $F(\alpha) = 1$. We define the set of all satisfiable assignments of a propositional formula F as $SAT(F) := \{\alpha \mid F(\alpha) = 1\}$. We write an assignment α as a *cube* (a conjunction of literals, i.e. positive or negative variables), e.g., $a \wedge b$, or shorter ab.

A weight function $W\colon \{\overline{v}, v \mid v \in V\} \Rightarrow \mathbb{R}$ assigns a real-valued weight to positive literals v (i.e., $v = 1$) and the negative literals $\overline{v}$ (i.e., $v = 0$). We say a variable v is *unbiased* iff $W(v) = W(\overline{v}) = 1$. Given an assignment $\alpha \in \mathbb{B}^V$, let $W(\alpha(v)) = W(v = \alpha(v))$ for $v \in V$. For a propositional formula F over variables in V and weight function W, we define *weighted model counting* as follows.

$$MC_W(F) \triangleq \sum_{\alpha \in \mathbb{B}^V} F(\alpha) \cdot W(\alpha), \text{ where } W(\alpha) = \prod_{v \in V} W(\alpha(v)).$$

Example 3. Given the formula $F = v_1 v_2 \vee \overline{v_1} v_2 \vee v_3$ over $V = \{v_1, v_2, v_3\}$, there are two satisfying assignments: $\alpha_1 = v_1 v_2 v_3$ and $\alpha_2 = \overline{v_1} v_2 v_3$. We define the weight function W as $W(v_1) = -\frac{1}{2}$, $W(\overline{v_1}) = \frac{1}{3}$ and $W(v_2) = \frac{1}{4}$, $W(\overline{v_2}) = \frac{3}{4}$, while v_3 remains unbiased. The weight of F can be computed as $MC_W(F) = -\frac{1}{2} \times \frac{1}{4} \times 1 + \frac{1}{3} \times \frac{1}{4} \times 1 = -\frac{1}{24}$. △

3 Encoding Quantum Circuits as Weighted CNF

3.1 Generalized Stabilizer Formalism

Clifford circuits together with T gates generate states beyond stabilizer states, enabling universal quantum computation. As is shown in Table 1, Clifford gates map the set of Pauli matrices to itself, keeping stabilizers within the Pauli group. In contrast, T gates can transform a Pauli matrix into a linear combination of Pauli matrices. To be specific, Table 2 gives the action of T gates on different Pauli gates. Given a Pauli string, after performing Clifford+T gates, we will get a summation of weighted Pauli strings, e.g., $T_0 \cdot (X \otimes X) \cdot T_0^\dagger = \frac{1}{\sqrt{2}}(X \otimes X + Y \otimes X)$ where $T_0 = T \otimes I$ as defined in Sect. 2. This leads to the definition of *generalized stabilizer state* extended from standard stabilizer formalism.

Definition 2. *In an n-qubit quantum system, a generalized stabilizer state $|\psi\rangle$ is the simultaneous eigenvector, with eigenvalue 1, of a group containing 2^n commuting unitary operators S. The set of S is a generalized stabilizer group.*

The above definition is adapted from [76] by defining a generalized stabilizer state using a generalized stabilizer group instead of generalized stabilizer generators. In this way, we can easily get the corresponding stabilizer state by a weighted summation of its stabilizers to avoid multiplications between Pauli strings (the middle part of Eq. 1). The following proposition is also adapted from [76], where they demonstrate that any pure state can be uniquely described by a set of stabilizers. We additionally show that there exists a set of stabilizers, forming a group, which uniquely describes any pure state.

Proposition 1. *For any pure state $|\varphi\rangle$ in an n-qubit quantum system, there exists a generalized stabilizer group $\mathcal{S}_{|\varphi\rangle}$ such that $|\varphi\rangle\langle\varphi| = \frac{1}{2^n} \cdot \sum_{S \in \mathcal{S}_{|\varphi\rangle}} S$.*

Proof. Given a pure state $|\varphi\rangle$, we have $|\varphi\rangle = U|0\rangle^{\otimes n}$, where U is a unitary operator. Let $\mathcal{S}_{|\varphi\rangle} = \{USU^\dagger \mid S \in \mathcal{S}_{|0\rangle^{\otimes n}}\}$, which is an isomorphic group to $\mathcal{S}_{|0\rangle^{\otimes n}}$ since U is unitary. For any $S \in \mathcal{S}_{|\varphi\rangle}$, we have $S' \in \mathcal{S}_{|0\rangle^{\otimes n}}$ satisfying $S = US'U^\dagger$ and $S|\varphi\rangle = US'U^\dagger U|0\rangle^{\otimes n} = US'|0\rangle^{\otimes n} = U|0\rangle^{\otimes n} = |\varphi\rangle$. Hence $S_{|\varphi\rangle}$ is the generalized stabilizer group of state $|\varphi\rangle$. Furthermore, we have $|\varphi\rangle\langle\varphi| = U|0\rangle\langle 0|U^\dagger = U(\frac{1}{2^n}\sum_{S' \in \mathcal{S}_{|0\rangle^{\otimes n}}} S')U^\dagger = \frac{1}{2^n}\sum_{S' \in \mathcal{S}_{|0\rangle^{\otimes n}}} US'U^\dagger = \frac{1}{2^n} \cdot \sum_{S \in \mathcal{S}_{|\varphi\rangle}} S$. In fact, given that any generalized stabilizer $S \in \mathcal{S}_{|\varphi\rangle}$ is a Hermitian matrix, since $S = US'U^\dagger = US'^\dagger U^\dagger = S^\dagger$ where $S' \in \mathcal{S}_{|0\rangle^{\otimes n}}$, according to [48, Lemma 1], it can be expressed as $S = \sum_{P \in \mathrm{PAULI}_n} \lambda_P P$, where $\lambda_P \in \mathbb{R}$. Thus a generalized stabilizer is always a linear combination of stabilizers. □

For an n-qubit quantum space, let *GStab* be the set of generalized stabilizer states, which is also the set of all pure states. Let $\mathcal{Q}$ be the set of quantum states generated from a Clifford+T circuit starting from the all-zero state, i.e. $\mathcal{Q} = \{U_{m-1} \dots U_0|0\rangle^{\otimes n} \mid U_i \in \{H, S, CX, T\}\}$. We have $\mathit{Stab} \subset \mathcal{Q} \subset \mathit{GStab}$ and any pure state in *GStab* can be approximated by some state in $\mathcal{Q}$ with arbitrary accuracy [27]. For any $|\varphi\rangle \in \mathit{Stab}$, we have $\mathcal{S}_{|\varphi\rangle} = \{\pm P \mid P \in \mathrm{PAULI}_n\}$. For any $|\psi\rangle \in \mathit{GStab}$, we have $\mathcal{S}_{|\psi\rangle} = \{\sum \lambda_P P \mid P \in \mathrm{PAULI}_n, \lambda_P \in \mathbb{R}\}$. E.g., when applying Clifford+T gates on $|\varphi\rangle$, the stabilizer group $\mathcal{S}_{|\psi\rangle}$ has λ_P equal to 0 or $\frac{1}{\sqrt{2}^k}$ for $k \in \mathbb{N}^+$ (as derived by updating $\mathcal{S}_{|\varphi\rangle}$ based on Table 1 and Table 2). Combining this with Proposition 1, we may flatten summations of generalized stabilizers:

$$|\psi\rangle\langle\psi| = \frac{1}{2^n} \sum_{S \in \mathcal{S}_{|\psi\rangle}} \sum_{\lambda_P P \in S} \lambda_P P = \frac{1}{2^n} \sum_{P \in \mathrm{PAULI}_n} \lambda_P P, \qquad (2)$$

Hence the density operator is determined by a summation of weighted Pauli strings and there is no need to distinguish which of the 2^n generalized stabilizers a Pauli string belongs to. (Stated differently, the density matrix can be decomposed in the Pauli basis [48].) We exploit this fact in our encoding and Example 4.

Example 4. Reconsider the circuit in Example 2 and add T, CX and H gates:

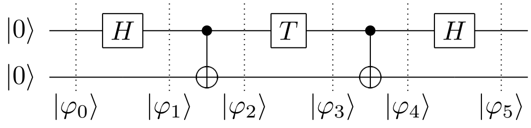


Table 2. Lookup table for the action of conjugating Pauli gates by T gates.

Gate	In	Out	Short
T	X	$\frac{1}{\sqrt{2}}(X+Y)$	$\triangleq X'$
	Y	$\frac{1}{\sqrt{2}}(Y-X)$	$\triangleq Y'$
	Z	Z	

Continuing from Example 2, we obtain the following generalized generators, where $X' = \frac{1}{\sqrt{2}}(X+Y)$ and $Z' = \frac{1}{\sqrt{2}}(Z-Y)$ as in Table 2.

$$\underbrace{\left\langle \begin{matrix} X \otimes X \\ Z \otimes Z \end{matrix} \right\rangle}_{|\varphi_2\rangle} \xrightarrow{T_0} \underbrace{\left\langle \begin{matrix} X' \otimes X \\ Z \otimes Z \end{matrix} \right\rangle}_{|\varphi_3\rangle} \xrightarrow{CX_{0,1}} \underbrace{\left\langle \begin{matrix} X' \otimes I \\ I \otimes Z \end{matrix} \right\rangle}_{|\varphi_4\rangle} \xrightarrow{H_0} \underbrace{\left\langle \begin{matrix} Z' \otimes I \\ I \otimes Z \end{matrix} \right\rangle}_{|\varphi_5\rangle} .$$

In our encoding, as in the above definition of generalized stabilizer states, we let satisfying assignments represent not just the generator set, but the entire stabilizer groups. These groups are: $\mathcal{S}_{|\varphi_2\rangle} = \{X \otimes X, Z \otimes Z, -Y \otimes Y, I \otimes I\}$, $\mathcal{S}_{|\varphi_3\rangle} = \{X' \otimes X, Z \otimes Z, -Y' \otimes Y, I \otimes I\}$, $\mathcal{S}_{|\varphi_4\rangle} = \{X' \otimes I, I \otimes Z, X' \otimes Z, I \otimes I\}$ and $\mathcal{S}_{|\varphi_5\rangle} = \{Z' \otimes I, I \otimes Z, Z' \otimes Z, I \otimes I\}$, where $Y' = \frac{1}{\sqrt{2}}(Y - X)$.

Finally, according to Eq. 2, we may equally expand e.g. $\mathcal{S}_{|\varphi_5\rangle}$ to: $\mathcal{S}'_{|\varphi_5\rangle} = \{\frac{1}{\sqrt{2}}Z \otimes I, \ -\frac{1}{\sqrt{2}}Y \otimes I, \ I \otimes Z, \ \frac{1}{\sqrt{2}}Z \otimes Z, \ -\frac{1}{\sqrt{2}}Y \otimes Z, \ I \otimes I\}$. △

3.2 Encoding Clifford+T Circuits

Since generalized stabilizer states can be determined by a sum of weighted Pauli strings as shown in Eq. 2, we encode a state by Boolean constraints whose satisfying assignments represent those weighted Pauli strings. The idea is to encode the sign, the Pauli string, and the weights separately. We will start with encoding the Pauli string and the sign.

A Pauli string $P \in \text{PAULI}_n$ can be encoded by $2n$ Boolean variables as $\sigma[x_0, z_0] \otimes \ldots \otimes \sigma[x_{n-1}, z_{n-1}]$, where the j-th Pauli matrix is indicated by variables x_j and z_j. To encode the sign, only one Boolean variable r is needed. We introduce weighted model counting to interpret the sign by defining $W(r) = -1$ and $W(\overline{r}) = 1$. Thus $\pm P \in \pm\text{PAULI}_n$ can be interpreted as $(-1)^r \otimes_{i \in [n]} \sigma[x_i, z_i]$. For example, consider the Boolean formula $r \neg x_0 z_0 x_1 z_1$. Its only satisfying assignment is $\{r \to 1, x_0 \to 0, z_0 \to 1, x_1 \to 1, z_1 \to 1\} \equiv -Z \otimes Y$. Without loss of generality, we set the initial state to be all zero state $|0 \ldots 0\rangle$, whose stabilizer group is $\mathcal{S}_{|0\ldots0\rangle} = \{\otimes_{i \in [n]} P_i \mid P_i \in \{Z, I\}\} \equiv \{(-1)^r \otimes_{i \in [n]} \sigma[x_i, z_i] \mid x_i = 0, z_i \in \{0,1\} \text{ and } r = 0\}$. Hence the Boolean formula for the initial state is defined as $F_{init}(\boldsymbol{x}^0, \boldsymbol{z}^0, r^0) \triangleq \neg r^0 \wedge \bigwedge_{j \in [n]} \neg x_j^0$, where we use superscripts, e.g., r^t, x_0^t, z_0^t, to denote variables at time step t (after t gates from the circuit have been applied). Note that we assign a weight to r only on the final time step, as explained later.

Table 3. Boolean variables under the action of conjugating one T gate. Here we omit the sign $(-1)^{r^t}$ for all operators G and sign $(-1)^{r^{t+1}}$ for all $TGT^\dagger$.

G	$x^t z^t r^t$	$TGT^\dagger$	x^{t+1}	z^{t+1}	r^{t+1}	u
I	$00\ r^t$	I	0	z^t	r^t	0
Z	$01\ r^t$	Z				
X	$10\ r^t$	$\frac{1}{\sqrt{2}}(X+Y)$	1	{0,1}	r^t	1
Y	$11\ r^t$	$\frac{1}{\sqrt{2}}(Y-X)$			$r^t \oplus \neg z^{t+1}$	

Example 5. Consider the initial state $|00\rangle$ in Example 4. The corresponding constraint at time step 0 is $\neg r^0 \neg\, x_0^0 \neg\, x_1^0$, which has satisfying assignments:

$$\left\{\begin{array}{l} \{r^0 \to 0, x_0^0 \to 0, x_1^0 \to 0, z_0^0 \to 1, z_1^0 \to 1\} \\ \{r^0 \to 0, x_0^0 \to 0, x_1^0 \to 0, z_0^0 \to 1, z_1^0 \to 0\} \\ \{r^0 \to 0, x_0^0 \to 0, x_1^0 \to 0, z_0^0 \to 0, z_1^0 \to 1\} \\ \{r^0 \to 0, x_0^0 \to 0, x_1^0 \to 0, z_0^0 \to 0, z_1^0 \to 0\} \end{array}\right\} \equiv \left\{\begin{array}{c} Z \otimes Z \\ Z \otimes I \\ I \otimes Z \\ I \otimes I \end{array}\right\} \qquad \triangle$$

While we encode those signed Pauli strings using variables from $\{x_j, z_j, r \mid j \in [n]\}$, to encode the weights, we introduce new variables u. When a T_j is performed and $x_j = 1$, which means executing a T gate on j-th qubit with certain stabilizer being either $\pm X$ or $\pm Y$, we set $u = 1$ to indicate a branch of the operator, i.e. $TXT^\dagger = X' = \frac{1}{\sqrt{2}}(X + Y)$ and $TYT^\dagger = Y' = \frac{1}{\sqrt{2}}(Y - X)$. Therefore, for each generalized stabilizer state in a circuit with m gates and n qubits, the encoding uses the set of variables $V^t = \{x_j^t, z_j^t, r^t, u_t \mid j \in [n]\}$, where $t \in \{0, \ldots, m\}$ denotes a time step. Table 3 illustrates the details of how the Boolean variables in V^t change over a T gate. Here each satisfying assignment indicates a weighted Pauli string, for example, there are two assignments for a stabilizer $\frac{1}{\sqrt{2}}(X + Y)$.

Using Table 1 and Table 3, given a single-qubit Clifford+T gate $\mathbf{G}_j$ on qubit j at time step t (or $CX_{j,k}$ on qubits j, k), we can derive a Boolean formula $F_{\mathbf{G}_j}(V^t, V^{t+1})$, abbreviated as $\mathbf{G}_j^t$, as in the following.

$$\begin{aligned} H_j^t &\triangleq r^{t+1} \Longleftrightarrow r^t \oplus x_j^t z_j^t \quad \wedge \quad z_j^{t+1} \Longleftrightarrow x_j^t \quad \wedge \quad x_j^{t+1} \Longleftrightarrow z_j^t \\ S_j^t &\triangleq r^{t+1} \Longleftrightarrow r^t \oplus x_j^t z_j^t \quad \wedge \quad z_j^{t+1} \Longleftrightarrow x_j^t \oplus z_j^t \\ CX_{j,k}^t &\triangleq r^{t+1} \Longleftrightarrow r^t \oplus x_j^t z_k^t (x_k^t \oplus \neg z_j^t) \quad \wedge \quad x_k^{t+1} \Longleftrightarrow x_k^t \oplus x_j^t \quad \wedge \\ & z_j^{t+1} \Longleftrightarrow z_j^t \oplus z_k^t \\ T_j^t &\triangleq x_j^{t+1} \Longleftrightarrow x_j^t \quad \wedge \quad x_j^t \vee (z_j^{t+1} \Leftrightarrow z_j^t) \quad \wedge \\ & r_i^{t+1} \Longleftrightarrow r^t \oplus x_j^t z_j^t \neg\, z_j^{t+1} \quad \wedge \quad u_t \Longleftrightarrow x_j^t. \end{aligned} \tag{3}$$

The above omits additional constraints $v^{t+1} \Leftrightarrow v^t$ for all unconstrained time-step-$t+1$ variables, i.e., for all $v^t \in V_l^t$ with $l \not= j, k$. In fact, it is not necessary to allocate new variables for those unconstrained time-step-$t+1$ variables. The constraint $v^{t+1} \Leftrightarrow v^t$ can be effectively implemented by reusing the Boolean variables v^t for v^{t+1}. Thus for each time step, only a constant number of new variables need to be allocated. For instance, when applying H_j gate, we only need one new variable r^{t+1}, since we can reuse the variable of x_j^t for z_j^{t+1} and z_j^t for x_j^{t+1}. And when applying CX_{ij} gate, we need three new variables for r^{t+1}, x_j^{t+1} and z_j^{t+1}. Additionally, since variables for all x_j^0 and z_j^0 with $j \in [n]$ are allocated initially, and as shown below, performing a measurement introduces no new variable, the size of our encoding is $O(n + m)$.

To this end, given a Clifford+T circuit $C = (\mathbf{G}^0, \ldots, \mathbf{G}^{m-1})$ without measurements, we can build the following Boolean constraint.

$$F_C(V^0, \ldots, V^m) \triangleq F_init(V^0) \wedge \bigwedge_{t \in [m]} F_{\mathbf{G}^t}(V^t, V^{t+1}). \tag{4}$$

The satisfying assignments of our encoding will represent weighted Pauli strings, so that we can get the density operator at time m by ranging over satisfying assignments $\alpha \in SAT(F_C)$:

$$\rho^m = \sum_{\alpha \in SAT(F_C)} F_C(\alpha) \cdot W(\alpha) \cdot \bigotimes_{j \in [n]} \sigma[\alpha(x_j^m), \alpha(z_j^m)], \tag{5}$$

where $W(r^m) = -1$, $W(\overline{r}^m) = 1$, $W(u_t) = \frac{1}{\sqrt{2}}$, $W(\overline{u}_t) = 1$ for all $t \in \{0, \ldots, m\}$ (and all other variables are unbiased). So we will get the weight as $W(\alpha) = W(\alpha(r^m)) \prod_{t \in [m]} W(\alpha(u_t))$. As mentioned before, we only assign weights to r^m where m is the final time step. We allocate a new r^{t+1} for each time step t as we always get r^{t+1} from a constraint related to r^t, but the sign of the final state is given by r^m. So we leave r^t unbiased except when t is the final time step. Additionally, all the weights are real numbers —there are no complex numbers— enabling the application of a classical weighted model counter that allows negative weights. It is worth noting that in Eq. 2, a generalized density matrix is a linear combination of stabilizers, which can be further decomposed to a sum of weighted Pauli strings. In the encoding, each weighted Pauli string corresponds to a satisfying assignment. In the case of Clifford circuits, since each stabilizer is a single Pauli string, the satisfying assignments can be interpreted as stabilizers. Therefore, for Clifford circuits, there are 2^n satisfying assignments for a n-qubit circuit at each time step. While for Clifford+T circuits, the number of satisfying assignments exceeds 2^n.

Example 6. Reconsider Example 4, after solving the constraint $F_{init}(V^0) \wedge H_0^0 \wedge CX_{0,1}^1 \wedge T_0^2 \wedge CX_{0,1}^3 \wedge H_0^4$, the satisfying assignments encoding $|\varphi_5\rangle$ will be

$$\left\{\begin{array}{l} \{r^5 \to 0, x_0^5 \to 0, x_1^5 \to 0, z_0^5 \to 1, z_1^5 \to 0, u_2 \to 1\}, \\ \{r^5 \to 1, x_0^5 \to 1, x_1^5 \to 0, z_0^5 \to 1, z_1^5 \to 0, u_2 \to 1\}, \\ \{r^5 \to 0, x_0^5 \to 0, x_1^5 \to 0, z_0^5 \to 0, z_1^5 \to 1, u_2 \to 0\}, \\ \{r^5 \to 0, x_0^5 \to 0, x_1^5 \to 0, z_0^5 \to 1, z_1^5 \to 1, u_2 \to 1\}, \\ \{r^5 \to 1, x_0^5 \to 1, x_1^5 \to 0, z_0^5 \to 1, z_1^5 \to 1, u_2 \to 1\}, \\ \{r^5 \to 0, x_0^5 \to 0, x_1^5 \to 0, z_0^5 \to 0, z_1^5 \to 0, u_2 \to 0\} \end{array}\right\} \equiv \left\{\begin{array}{r} \frac{1}{\sqrt{2}} Z \otimes I \\ -\frac{1}{\sqrt{2}} Y \otimes I \\ I \otimes Z \\ \frac{1}{\sqrt{2}} Z \otimes Z \\ -\frac{1}{\sqrt{2}} Y \otimes Z \\ I \otimes I \end{array}\right\}$$

where $W(\overline{r}^5) = 1$, $W(r^5) = -1$, $W(u_2) = \frac{\sqrt{2}}{2}$ and $W(\overline{u}_2) = 1$. Here we omit the satisfying assignments for $\{r^t, x_0^t, x_1^t, z_0^t, z_1^t \mid 0 \leq t \leq 4\}$. △

3.3 Encoding Arbitrary Rotation Gates

Our encoding can be extended to other non-Clifford gates, which we demonstrate by adding rotation gates $R_X(\theta)$, $R_Y(\theta)$, and $R_Z(\theta)$, where θ is an angle in radians. The matrices for these gates are given in Table 4.

Table 4. Lookup table for the action of conjugating Pauli gates by rotation gates.

Gate	Matrix Form	In	Out
$R_X(\theta)$	$\begin{bmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix}$	X	X
		Y	$\cos(\theta)Y + \sin(\theta)Z$
		Z	$\cos(\theta)Z - \sin(\theta)Y$
$R_Y(\theta)$	$\begin{bmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix}$	X	$\cos(\theta)Z + \sin(\theta)X$
		Y	Y
		Z	$\cos(\theta)X - \sin(\theta)Z$
$R_Z(\theta)$	$\begin{bmatrix} \exp(-i\frac{\theta}{2}) & 0 \\ 0 & \exp(i\frac{\theta}{2}) \end{bmatrix}$	X	$\cos(\theta)X + \sin(\theta)Y$
		Y	$\cos(\theta)Y - \sin(\theta)X$
		Z	Z

In particular, we have $T = \exp(-i\frac{\pi}{8})R_Z(\frac{\pi}{4})$, $S = \exp(-i\frac{\pi}{4})R_Z(\frac{\pi}{2})$ and $X = -iR_X(\pi)$, $Y = -iR_Y(\pi)$, $Z = -iR_Z(\pi)$. Note however that the stabilizer formalism discards the global phase of a state as it updates stabilizers by conjugation, e.g., $TPT^\dagger = \left(\exp(-i\frac{\pi}{8})R_Z(\frac{\pi}{4})\right) P \left(\exp(-i\frac{\pi}{8})R_Z(\frac{\pi}{4})\right)^\dagger = R_Z(\frac{\pi}{4})P\, R_Z(\frac{\pi}{4})^\dagger$. Based on Table 4, the constraints for rotation gates are as shown below, where we write RX for R_X, RY for R_Y and RZ for R_Z. We keep the coefficients $\cos(\theta)$ and $\sin(\theta)$ by defining the weights of the new variables as $W(u_{1t}) = \cos(\theta)$, $W(u_{2t}) = \sin(\theta)$, and $W(\overline{u}_{1t}) = W(\overline{u}_{2t}) = 1$.

$$\begin{aligned}
RX_j^t \triangleq\ & z_j^{t+1} \iff z_j^t \quad\wedge\quad z_j^t \vee (x_j^{t+1} \Leftrightarrow x_j^t) \quad\wedge\quad r^{t+1} \iff r^t \oplus z_j^t \neg\, x_j^t x_j^{t+1} \quad\wedge \\
& u_{1t} \iff z_j^t(x_j^t x_j^{t+1} \vee \neg x_j^t \neg\, x_j^{t+1}) \quad\wedge\quad u_{2t} \iff z_j^t(\neg x_j^t x_j^{t+1} \vee x_j^t \neg\, x_j^{t+1}). \\
RY_j^t \triangleq\ & (x_j^t \oplus z_j^t) \iff (x_j^{t+1} \oplus z_j^{t+1}) \quad\wedge\quad (x_j^t \oplus \neg z_j^t) \implies (x_j^t z_j^t \Leftrightarrow x_j^{t+1} \Leftrightarrow z_j^{t+1}) \wedge \\
& r^{t+1} \iff r^t \oplus z_j^t z_j^{t+1} \quad\wedge\quad u_{1t} \iff (x_j^{t+1} z_j^t \oplus x_j^t z_j^{t+1}) \quad\wedge \\
& u_{2t} \iff (x_j^{t+1} x_j^t \oplus z_j^{t+1} z_j^t). \\
RZ_j^t \triangleq\ & x_j^{t+1} \iff x_j^t \quad\wedge\quad x_j^t \vee (z_j^{t+1} \Leftrightarrow z_j^t) \quad\wedge\quad r^{t+1} \iff r^t \oplus x_j^t z_j^t \neg\, z_j^{t+1} \quad\wedge \\
& u_{1t} \iff x_j^t(z_j^t z_j^{t+1} \vee \neg z_j^t \neg\, z_j^{t+1}) \quad\wedge\quad u_{2t} \iff x_j^t(\neg z_j^t z_j^{t+1} \vee z_j^t \neg\, z_j^{t+1}).
\end{aligned}$$

3.4 Measurement

We now consider projective measurement both on a single qubit and on multiple qubits of a quantum system. Single-qubit measurement [43] can be used to extract only one bit of information from a n-qubits quantum state, effectively protecting quantum information [57]. It is also used in random quantum circuits, contributing to the study of quantum many-body physics [31]. Measurement on multiple qubits is generally used in quantum algorithms, such as in Grover and Shor algorithms, to get the final result. We implement both measurements using *Pauli measurement*, where projectors are Pauli strings.

Single-Qubit Pauli Measurement. Let $\mathbb{P}_{k,0} = I \otimes \cdots \otimes |0\rangle\langle 0|_k \otimes \cdots \otimes I = \frac{1}{2}(I^{\otimes n} + Z_k)$ and $\mathbb{P}_{k,1} = I \otimes \cdots \otimes |1\rangle\langle 1|_k \otimes \cdots \otimes I = \frac{1}{2}(I^{\otimes n} - Z_k)$ for $k \in [n]$. When measuring k-th qubit of a n-qubit state $|\psi\rangle$ using projectors $\{\mathbb{P}_{k,0}, \mathbb{P}_{k,1}\}$, we get two possible outcomes: 0 with probability $p_{k,0}$ and 1 with probability $p_{k,1}$. It follows that $p_{k,0} = \mathrm{tr}(\mathbb{P}_{k,0}|\psi\rangle\langle\psi|)$, where tr is the trace mapping [52]. As shown in Eq. 2, the density operator $|\psi\rangle\langle\psi|$ can be written as a weighted sum of Pauli strings, i.e., $|\psi\rangle\langle\psi| = \frac{1}{2^n}\sum \lambda_P P$ for $P \in \mathrm{PAULI}_n$, $\lambda_P \in \mathbb{R}$. The probabilities $p_{k,0}$ and $p_{k,1}$ can be obtained as

$$\begin{aligned} p_{k,0} &= \mathrm{tr}(\mathbb{P}_{k,0}\tfrac{1}{2^n}\sum \lambda_P P) = \tfrac{1}{2^n}\sum \tfrac{1}{2}(\mathrm{tr}(I^{\otimes n}\lambda_P P) + \mathrm{tr}(Z_k \lambda_P P)), \\ p_{k,1} &= \mathrm{tr}(\mathbb{P}_{k,1}\tfrac{1}{2^n}\sum \lambda_P P) = \tfrac{1}{2^n}\sum \tfrac{1}{2}(\mathrm{tr}(I^{\otimes n}\lambda_P P) - \mathrm{tr}(Z_k \lambda_P P)). \end{aligned} \tag{6}$$

In Eq. 6, the trace $\mathrm{tr}(I^{\otimes n}P)$ (resp. $\mathrm{tr}(Z_k P)$) is non-zero if and only if $P = \lambda_P I^{\otimes n}$ (resp. $P = \lambda_P Z_k$) where $\lambda_P \in \mathbb{R}$. This follows from two facts: First, given an n-qubit Pauli string $P = P_0 \otimes \ldots \otimes P_{n-1}$, where P_i are Pauli matrices, the trace $\mathrm{tr}(P) = \mathrm{tr}(P_0)\cdots\mathrm{tr}(P_{n-1})$ is non-zero if and only if $P = I^{\otimes n}$, since Pauli matrices are *traceless*, i.e., $\mathrm{tr}(X) = \mathrm{tr}(Y) = \mathrm{tr}(Z) = 0$. Second, given $P_1, P_2 \in \mathrm{PAULI}_n$ we have $P_1 P_2 = I^{\otimes n}$ if and only if $P_1 = P_2$. Together with the fact that $\mathrm{tr}(I^{\otimes n}) = 2^n$, we can now simplify Eq. 6 to:

$$p_{k,0} = \frac{1}{2^n}\sum \tfrac{1}{2}(\mathrm{tr}(I^{\otimes n}\lambda_P P) + \mathrm{tr}(Z_k \lambda_P P)) = \frac{1}{2}(\lambda_{I^{\otimes n}} + \lambda_{Z_k}) \tag{7}$$

Similarly, we have $p_{k,1} = \frac{1}{2}(\lambda_{I^{\otimes n}} - \lambda_{Z_k})$. In other words, to get the probability of obtaining outcome 0 when measuring the k-th qubit, we need to sum up the weights of all elements:

$$Z_k \equiv z_k \wedge \bigwedge_{j\in[n]} \overline{x}_j \wedge \bigwedge_{j\in[n], j\neq k} \overline{z}_j \quad \text{and} \quad I^{\otimes n} \equiv \bigwedge_{j\in[n]} \overline{x}_j \wedge \bigwedge_{j\in[n]} \overline{z}_j$$

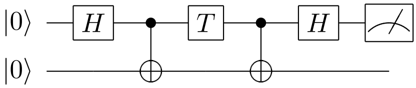
in the flattened Pauli strings of Eq. 2. Thus the measurement can be encoded as follows, for the final time step m.

$$F_{\mathbb{P}_{k,b}}(V^m) \triangleq w \wedge \bigwedge_{j\in[n]} \overline{x}_j^m \wedge \bigwedge_{j\in[n], j\neq k} \overline{z}_j^m, \tag{8}$$

where $W(w) = \frac{1}{2}$ to represent the constant factor in $p_{k,0}$ and the weight of z_k is unbiased when $b = 0$ or $W(z_k^m) = -1$, $W(\overline{z}_k^m) = 1$ when $b = 1$.

A Clifford+T circuit with a single-qubit Pauli measurement at the end can be encoded by conjoining the constraint of initial state and gates in Eq. 4 with the one for the measurement at the end, as Example 7 illustrates.

Example 7. Consider the circuit in Example 6 and assume we want to perform single-qubit Pauli measurement on the first qubit using $\{\frac{I\otimes I+Z\otimes I}{2}, \frac{I\otimes I-Z\otimes I}{2}\}$.



By adding the measurement constraint $F_{\mathbb{P}_{0,0}} \triangleq w \wedge \overline{x}_0^5 \wedge \overline{x}_1^5 \wedge \overline{z}_1^5$ to the circuits constraints in Example 6, we get $F_{init}(V^0) \wedge H_0^0 \wedge CX_{0,1}^1 \wedge T_0^2 \wedge CX_{0,1}^3 \wedge H_0^4 \wedge F_{\mathbb{P}_{0,0}}(V^5)$. The satisfying assignments will be the subset of the solutions in Example 6 with the constant variable w:

$$\left\{\begin{array}{l} \{r^5 \to 0, x_0^5 \to 0, x_1^5 \to 0, z_0^5 \to 1, z_1^5 \to 0, u_2 \to 1, w \to 1\}, \\ \{r^5 \to 0, x_0^5 \to 0, x_1^5 \to 0, z_0^5 \to 0, z_1^5 \to 0, u_2 \to 0, w \to 1\} \end{array}\right\} \equiv \left\{\begin{array}{r} \frac{1}{2\sqrt{2}} Z \otimes I \\ \frac{1}{2} I \otimes I \end{array}\right\}$$

where we only show the variables in V^5 representing the final state. The resulting probability is $W(\overline{r}^5)W(u_2)W(w) + W(\overline{r}^5)W(\overline{u}_2)W(w) = \frac{1}{2\sqrt{2}} + \frac{1}{2}$. △

Multi-qubit Pauli Measurement. Similar to the single-qubit Pauli measurement, we can resolve the constraint of n-qubit Pauli measurement based on the measurement projector $\mathbb{P}$. Given a quantum state $|\psi\rangle\langle\psi| = \frac{1}{2^n}\sum_P \lambda_P P$, let $Q \subseteq [n]$ be the set of qubits being measured. Without loss of generality, we explain how to obtain the probability of obtaining the outcome zero (0) for all qubits $q \in Q$. The projector measuring qubits in Q is defined as $\mathbb{P}_Q \triangleq \bigotimes_{q\in Q} P_q$ where $P_q = (I + Z)/2$ for $q \in Q$ and $P_q = I$ for $q \in [n] \setminus Q$. We can derive the constraint $F_{\mathbb{P}_q}(x_q^m, z_q^m) = \overline{x}_q^m$ for $q \in Q$ and no constraint for $q \in [n] \setminus Q$. By conjoining them, we obtain the measurement constraint $F_{\mathbb{P}_Q} = w \wedge \bigwedge_{q\in[n]} \overline{x}_q^m \wedge \bigwedge_{q\in[n]\setminus Q} \overline{z}_q^m$ where w is a constant variable for the factor and $W(w) = \frac{1}{2^{|Q|}}$.

We conclude Sect. 3 with Proposition 2, which also shows that our encoding implements a strong simulation of a universal quantum circuit.

Proposition 2. *Given an n-qubit quantum circuit C, a subset of qubits $Q \subseteq [n]$ and the WMC encoding of the corresponding simulation problem $F(V^0, \ldots, V^m) = F_C(V^0, \ldots, V^m) \wedge F_{\mathbb{P}_Q}(V^m)$ with according weight function W, the weighted model count of F equals the probability of the measurement outcome corresponding to $\mathbb{P}_Q$ (outcome 0 for all qubits $q \in Q$) on circuit C, i.e., $MC_W(F) = \langle 0|^{\otimes n} C^\dagger \mathbb{P}_Q C |0\rangle^{\otimes n}$.*

4 Experiments

To show the effectiveness of our approach, we implemented a WMC-based simulator in a tool called QCMC. It reads quantum circuits in QASM format [26], encodes them to Boolean formulas in conjunctive normal form (CNF) as explained in Sect. 3, and then uses the weighted model counter GPMC [36] to solve these constraints. We choose GPMC as it is the best solver supporting negative weights in model counting competition 2023 [4]. The resulting implementation and evaluation are publicly available at [46].

We compare our method against two state-of-the-art tools: QuiZX [41] based on ZX calculus [25] and Quasimodo [64] based on CFLOBDD [65]. In particular, this empirical analysis is performed on two families of circuits: (i) random Clifford+T circuits, which mimic hard problems arising in quantum chemistry [75] and quantum many-body physics [31]; (ii) random circuits mimicking oracle implementations; (iii) all benchmarks from the public benchmark suite MQT Bench [58], which includes many important quantum algorithms like QAOA, VQE, QNN and Grover. All experiments have been conducted on a 3.5 GHz M2 Machine with MacOS 13 and 16 GB RAM. We set the time limit to be 5 min (300 s) and include the time to read a QASM file, construct the weighted CNF, and perform the model counting in all reported runtimes.

4.1 Results

First, we show the limits of three methods using the set of benchmarks generated by [41]. They construct random circuits with a given number of T gates by exponentiating Pauli unitaries in the form of $\exp(-i(2k+1)\frac{\pi}{4}P)$ where P is a Pauli string and $k \in \{1, 2\}$. We reuse their experimental settings, which gradually increase the T count (through Pauli exponentiation) for $n = 50$ qubits, and we add an experiment with $n = 100$ qubits. Accordingly, we generate 50 circuits with different random seeds for each $n = 50$ and $T \in [0\text{–}100]$ and each $n = 100$ and $T \in [0\text{–}180]$. We then perform a single-qubit Pauli measurement on the first qubit. We plot the minimal time needed and the rate of successfully getting the answer in 5 min among all 50 simulation runs in Fig. 1.

Second, we also consider random circuits that more resemble typical oracle implementations — random quantum circuits with varying qubits and depths, which comprise the CX, H, S, and T gates with appearing ratio 10%, 35%, 35%, 20% [56]. The resulting runtimes can be seen in Fig. 2.

In addition to the random circuits, we empirically evaluated our method on the MQTBench benchmark set [58], measuring all qubits, as is typical in most quantum algorithms. We present a representative subset of results in Table 5. The complete results can be found in [47]. All benchmarks are expanded to the Clifford+T+R gate set, where R denotes $\{R_X, R_Y, R_Z\}$. The first two columns list the number of qubits n and the number of gates $|G|$. Columns T and R give the number of T gates and rotation gates. Then, the performance of the WMC-based tool QCMC, the performance of the ZX calculus-based tool QuiZX (ZX), and the performance of CFLOBDD-based tool Quasimodo (CFLOBDD). The performances are given by the runtime and the corresponding memory usage. Given that the variation in each run is minimal, we present the results of a single run for all experiments.

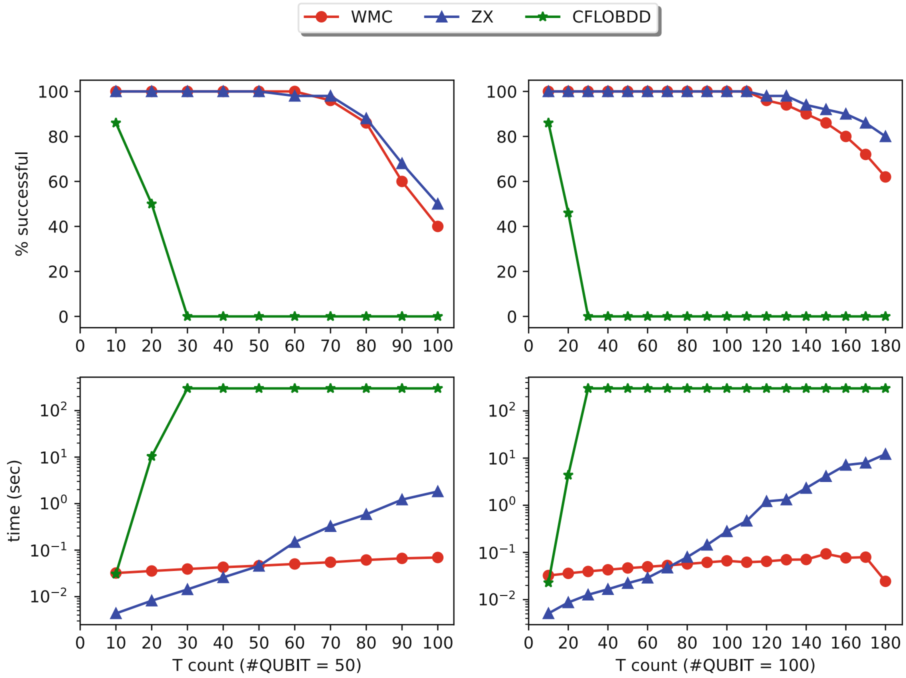


Fig. 1. The upper two figures, are percentages of random 50- and 100-qubit circuits with increasing depth which can be successfully measured in 5 min. The below two figures, both of which have y-axes on a logarithmic scale, are the minimum running time among the 50 samples for each configuration.

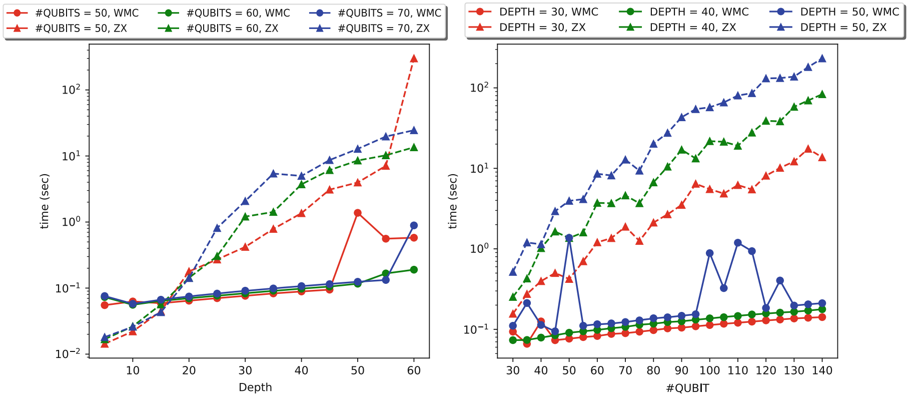


Fig. 2. Computational basis measurement of typical random Clifford+T circuits. (Both vertical axes are on a logarithmic scale.) CFLOBDD runs out of time for all benchmarks so we do not add it here.

4.2 Discussion

For all cases, QuiZX gives algebraic answers while the QCMC and CFLOBDD methods give numerical answers. Because of the imprecision of floating-point arithmetic, we consider a equal to b if $|a - b| < 10^{-8}$. With this equality tolerance, all three methods produce the same answers.

For random circuits, Fig. 1 illustrates that the minimum runtime barely increases for QCMC, while it seems exponential for QuiZX (note the log scale). However, when the number of qubits is $n = 50$ and the T count is larger than 70, or when, $n = 100$ and the T count is larger than 110, QuiZX has a better success rate, i.e., it completes more simulations than QCMC in 5 min. In contrast, CFLOBDD exhibits the lowest success rate among the three methods. When it comes to a typical random Clifford+T circuit Fig. 2 shows that the runtime of both QCMC and ZX exhibits a clear correlation with the size of the circuits, while CFLOBDD can not solve all benchmarks in 5 min. The proposed implementation consistently outperforms QuiZX by one to three orders of magnitude especially when the size is getting larger (note again the log scale). However, the story changes when considering structural quantum circuits.

For MQT benchmarks, Table 5 shows that QCMC performs better than QuiZX except for GHZ state and Graph State where QCMC is slightly slower in milliseconds. CFLOBDD significantly surpasses QCMC on Grover and quantum walk algorithms, primarily due to the decision diagram-based method's proficiency in handling circuits featuring large reversible parts and oracles. While for those circuits featuring a large number of rotation gates with various rotation angles, like Graph state, QFT, and VQE, QCMC demonstrates clear advantages. This distinction arises from the fact that when dealing with rotation gates, it might happen that two decision diagram nodes that should be identical in theory, differ by a small margin in practice, obstructing node merging [53]. In contrast, the WMC approach —also numerical in nature— avoids explicit representation of all satisfying assignments, by iteratively computing a sum of products. This not only avoids blowups in space use but, we hypothesize, also avoids numerical instability, a problem that has plagued numerical decision-diagram based approaches [53,56]. In terms of memory usage, CFLOBDD always uses more than 340 MB, in some cases uses more than 6 GB (graph state, $n = 64$), while QCMC and QuiZX use less than 13 MB (OS reported peak resident set size).

Overall, both QCMC and QuiZX outperform CFLOBDD in handling random circuits. Moreover, QCMC has better runtime performance than QuiZX. For structural circuits, QuiZX faces a limitation as it does not efficiently support rotation gates with arbitrary angles, so it is incapable of simulating many quantum algorithms, like VQE, directly. In terms of runtime, CFLOBDD is better at circuits featuring structure, while QCMC performs better at circuits with arbitrary rotation gates. However, CFLOBDD has a significantly higher memory cost compared to both QCMC and QuiZX.

Table 5. Results of verifying circuits from MQT bench. For cases within time limit, we give their running time (sec) and corresponding memory usage (MB). We use × when QuiZX does not support certain benchmarks, while >300 represents a timeout (5 min). For those benchmarks having a timeout or are not supported, we omit their memory usage by ⋆.

Algorithm	n	$\|G\|$	T	R	WMC		ZX		CFLOBDD	
					t(sec)	RSS(MB)	t(sec)	RSS(MB)	t(sec)	RSS(MB)
GHZ State	32	32	0	0	0.044	10.5	0.007	12.28	0.03	354.81
	64	64	0	0	0.049	10.5	0.008	12.30	0.03	356.02
	128	128	0	0	0.048	10.75	0.013	12.44	0.04	355.77
Graph State	16	64	0	0	0.046	10.125	0.005	12.44	0.06	355.68
	32	128	0	0	0.045	10.5	0.008	12.40	0.11	355.39
	64	256	0	0	0.045	10.63	0.015	12.47	243.22	6116.93
Grover's (no ancilla)	4	162	8	58	0.18	10.5	89.18	12.06	0.04	345.56
	5	470	0	195	3.86	11.5	>300	⋆	0.07	345.73
	6	1314	0	552	>300	⋆	>300	⋆	0.21	346.52
QAOA	7	63	0	28	0.03	10.5	>300	⋆	0.05	355.98
	9	81	0	36	0.036	10.13			0.05	355.89
	11	99	0	44	0.035	10.75			0.06	355.42
QFT	16	520	0	225	0.02	10.75	0.09	12.30	6.58	516.19
	32	2064	0	961	0.03	11.375	0.16	12.53	>300	⋆
	64	8224	0	3969	0.11	35.83	0.57	12.32	>300	⋆
QNN	16	1119	0	400	>300	⋆	×	⋆	57.36	2232.14
	32	3775	0	1312					>300	⋆
	64	13695	0	4672					>300	⋆
Quantum Walk (no ancilla)	5	1071	24	448	70.79	12.70	×	⋆	0.13	345.19
	6	2043	24	844	>300	⋆			0.28	345.58
	7	3975	24	1624	>300	⋆			0.80	347.58
VQE	14	236	0	82	0.23	10.875	×	⋆	2.84	610.54
	15	253	0	88	0.49	10.875			6.29	680.64
	16	270	0	94	0.51	10.875			17.97	943.67
W-state	32	435	0	124	0.11	11.125	×	⋆	>300	⋆
	64	883	0	252	0.28	11.87				
	128	1779	0	508	0.66	13.25				

5 Related Work

In this section, we give an overview of the related work on classical simulation of quantum computing with a focus on those methods applying SAT-based solvers.

SAT-based solvers have proven successful in navigating the huge search spaces encountered in various problems in quantum computing [49,73], initial attempts have been made to harness the strengths of satisfiability solvers for the simulation of quantum circuits. For instance, [11] implements a simulator for Clifford circuits based on a SAT encoding (our encoding of H, S, CX in Eq. 3 is similar to theirs). The authors also discuss a SAT encoding for universal quantum circuits, which however requires exponentially large representations, making it impractical. Besides quantum circuits, [10] presents symQV, a framework for verifying quantum programs in a quantum circuit model, which allows the encoding of the verification problem in an SMT formula, which can then be checked with a δ-complete decision process. There is an SMT theory for quantum computing [23].

Another method is based on decision diagrams (DDs) [1,16], which represent many Boolean functions succinctly, while allowing manipulation operations without decompression. DD methods for pseudo-Boolean functions include Algebraic DDs (ADD) [9,24,70] and various "edge-valued" ADDs [44,61,67,74]. The application of DDs to quantum circuit simulation, by viewing a quantum state as a pseudo-Boolean function, was pioneered with QuiDDs [71] and further developed with Quantum Multi-valued DDs [50], Tensor DDs [38] and CFLOBDDs [65]. All but CFLOBDD are essentially ADDs with complex numbers.

Another way is to translate quantum circuits into ZX-diagrams [25], which is a graphical calculus for quantum circuits equipped with powerful rewrite rules.

Classical simulation is commonly used for the verification of quantum circuits, with extensive research focused on their equivalence checking [5,7,72]. It can also be applied to bug hunting in quantum circuits. In [22], the authors proposed a tree automata to compactly represent quantum states and gates algebraically, framing the verification problem as a Hoare triple.

6 Conclusions

In this work, we proposed a generalized stabilizer formalism formulated in terms of a stabilizer group. Based on this, we provided an encoding for various universal gate sets as a weighted model counting problem with only real weights, obviating the need for complex numbers that are not supported by existing WMC tools. Besides T gates, we also extended our encoding to general rotation gates. Furthermore, we demonstrated how to perform computational basis measurements using this encoding, enabling strong quantum circuit simulation.

We have implemented our method in an open-source tool QCMC. To give empirical results on the practicality of our method, we have applied it to a variety of benchmarks comparing one based on ZX-calculus and one based on decision diagrams. Experimental results show that our approach outperforms both in several cases, particularly with circuits of large sizes. The performance of our approach is quite different from the other approaches, demonstrating the unique potential of WMC in various use cases.

This work provides a new benchmark paradigm for weighted model counting problems. It would be interesting to see whether WMC tools can be improved for this novel application domain. In the future, it could also be worthwhile to apply our encoding to approximate weighted model counting [19,29]. Moreover, using weighted samplers [34,45], we could achieve weak circuit simulation with the same encoding. The main obstacle now is incorporating negative weights into approximate weighted model counters and samplers. Additionally, it would be valuable to explore more applications of this encoding, such as checking the equivalence of two quantum circuits and entanglement purification.

Acknowledgements. We thank all anonymous reviewers for their helpful comments. This work is supported by the Dutch National Growth Fund, as part of the Quantum Delta NL program.

References

1. Akers. Binary decision diagrams. IEEE Trans. Comput. C-27(6), 509–516 (1978)
2. Amy, M., Bennett-Gibbs, O., Ross, N.J.: Symbolic synthesis of Clifford circuits and beyond. Electron. Proc. Theor. Comput. Sci. **394**, 343–362 (2023)
3. Anders, S., Briegel, H.J.: Fast simulation of stabilizer circuits using a graph-state representation. Phys. Rev. A **73**(2) (2006)
4. Hecher, M., Fichte, J.: Model counting competition (2023). https://mccompetition.org/. Accessed 01 Jul 2024
5. Ardeshir-Larijani, E., Gay, S.J., Nagarajan, R.: Equivalence checking of quantum protocols. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 478–492. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_33
6. Ardeshir-Larijani, E., Gay, S.J., Nagarajan, R.: Verification of concurrent quantum protocols by equivalence checking. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 500–514. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_42
7. Ardeshir-Larijani, E., Gay, S.J., Nagarajan, R.: Verification of concurrent quantum protocols by equivalence checking. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 500–514. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_42
8. Arute, F., et al.: Quantum supremacy using a programmable superconducting processor. Nature **574**, 505–510 (2019)
9. Iris Bahar, R., et al.: Algebraic decision diagrams and their applications. In: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD), pp. 188–191 (1993)
10. Bauer-Marquart, F., Leue, S., Schilling, C.: symQV: automated symbolic verification of quantum programs. In: Chechik, M., Katoen, J.-P., Leucker, M., ed, Formal Methods, pp. 181–198. Springer International Publishing, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_12
11. Berent, L., Burgholzer, L., Wille, R.: Towards a SAT encoding for quantum circuits: a journey from classical circuits to clifford circuits and beyond. In: Meel, K.S., Strichman, O., ed., 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022), volume 236 of Leibniz International Proceedings in Informatics (LIPIcs), pp. 18:1–18:17, Dagstuhl, Germany (2022). Schloss Dagstuhl – Leibniz-Zentrum für Informatik
12. Biere, A., Heule, M., van Maaren, H., Walsh, T., ed. Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
13. Brand, S., Coopmans, T., Laarman, A.: Quantum graph-state synthesis with SAT. In: Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023, volume 3545 of CEUR Workshop Proceedings, pp. 1–13. CEUR-WS.org (2023)
14. Bravyi, S., Gosset, D.: Improved classical simulation of quantum circuits dominated by Clifford gates. Phys. Rev. Lett. **116**, 250501 (2016)

15. Bravyi, S., Shaydulin, R., Shaohan, H., Maslov, D.: Clifford circuit optimization with templates and symbolic Pauli gates. Quantum **5**, 580 (2021)
16. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
17. Burgholzer, L., Wille, R.: Improved DD-based equivalence checking of quantum circuits. In: 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 127–132. IEEE (2020)
18. Calderbank, A.R., Shor, P.W.: Good quantum error-correcting codes exist. Phys. Rev. A **54**, 1098–1105 (1996)
19. Chakraborty, S., Fremont, D., Meel, K., Seshia, S., Vardi, M.: Distribution-aware sampling and weighted model counting for sat. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 28 (2014)
20. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. **172**(6), 772–799 (2008)
21. Chen, Y., Chen, Y., Kumar, R., Patro, S., Speelman, F.: QSETH strikes again: finer quantum lower bounds for lattice problem, strong simulation, hitting set problem, and more. arXiv preprint arXiv:2309.16431 (2023)
22. Chen, Y.-F., Chung, K.-M., Lengál, O., Lin, J.-A., Tsai, W.-L., Yen, D.-D.: An automata-based framework for verification and bug hunting in quantum circuits. Proc. ACM Program. Lang., 7(PLDI) (2023)
23. Chen, Y.-F., Rümmer, P., Tsai, W.-L.: A theory of cartesian arrays (with applications in quantum circuit verification). In: International Conference on Automated Deduction, pp. 170–189. Springer (2023). https://doi.org/10.1007/978-3-031-38499-8_10
24. Clarke, E.M., McMillan, K.L., Zhao, X., Fujita, M., Yang, J.: Spectral transforms for large Boolean functions with applications to technology mapping. In: Proceedings of the 30th international Design Automation Conference, pp. 54–60 (1993)
25. Coecke, B., Duncan, R.: Interacting quantum observables: categorical algebra and diagrammatics. New J. Phys. **13**(4), 043016 (2011)
26. Cross, A., et al.: OpenQASM3: a broader and deeper quantum assembly language. ACM Trans. Quantum Comput. **3**(3), 1–50 (2022)
27. Dawson, C.M., Nielsen, M.A.: The Solovay-Kitaev algorithm. Quantum Inf. Comput. **6**(1), 81–95 (2006)
28. Van den Nest, M.: Classical simulation of quantum computation, the gottesman-knill theorem, and slightly beyond. Quantum Inf. Comput. **10**(3), 258–271 (2010)
29. Ermon, S., Gomes, C.P., Sabharwal, A., Selman, B.: Embed and project: discrete sampling with universal hashing. In: Advances in Neural Information Processing Systems, vol. 26 (2013)
30. Feng, N., Marsso, L., Sabetzadeh, M., Chechik, M.: Early verification of legal compliance via bounded satisfiability checking. In: Enea, C., Lal, A., ed., Computer Aided Verification, pp. 374–396. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37709-9_18
31. Fisher, M.P.A., Khemani, V., Nahum, A., Vijay, S.: Random quantum circuits. Ann. Rev. Condensed Matter Phys. **14**(1), 335–379 (2023)
32. García, H.J., Markov, I.L., Cross, A.W.: On the geometry of stabilizer states. Quantum Inf. Comput. **14**(7&8), 683–720 (2014)
33. Gay, S.J.: Stabilizer states as a basis for density matrices. CoRR, abs/1112.2156 (2011)
34. Golia, P., Soos, M., Chakraborty, S., Meel, K.S.: Designing samplers is easy: The boon of testers. In: 2021 Formal Methods in Computer Aided Design (FMCAD), pp. 222–230. IEEE (2021)

35. Gottesman, D.: Stabilizer codes and quantum error correction. PhD thesis, California Institute of Technology (1997)
36. Hashimoto, K.: GPMC (2020). https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/GPMC
37. Hong, X., Feng, Y., Li, S., Ying, M.: Equivalence checking of dynamic quantum circuits. In: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ICCAD '22, New York, NY, USA (2022). Association for Computing Machinery
38. Hong, X., Zhou, X., Li, S., Feng, Y., Ying, M.: A tensor network based decision diagram for representation of quantum circuits. ACM Trans. Design Autom. Electr. Syst. **27**(6), 60:1–60:30 (2022)
39. Huang, C., et al.: Classical simulation of quantum supremacy circuits, Mario Szegedy (2020)
40. Jozsa, R., Van den Nest, M.: Classical simulation complexity of extended Clifford circuits. Quantum Inf. Comput. **14**(7–8), 633–648 (2014)
41. Kissinger, A., van de Wetering, J.: Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions. Quantum Sci. Technol. **7**(4), 044001 (2022)
42. Kliuchnikov, V.: Synthesis of unitaries with Clifford+T circuits. arXiv e-printsarXiv:1306.3200 (2013)
43. Kocia, L., Sarovar, M.: Classical simulation of quantum circuits using fewer gaussian eliminations. Phys. Rev. A **103**, 022603 (2021)
44. Lai, Y.-T., Pedram, M., Vrudhula, S.B.K.: EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. IEEE Trans. Comput.-Aid. Design Integr. Circ. Syst. **13**(8), 959–975 (1994)
45. Meel, K.S., Yang, S., Liang, V.: INC: a scalable incremental weighted sampler. In: FMCAD 2022, vol. 3, p. 205. TU Wien Academic Press (2022)
46. Mei, J.: The Quokka# repository. https://github.com/System-Verification-Lab/Quokka-Sharp. Accessed 29 Mar 2024
47. Mei, J., Bonsangue, M., Laarman, A.: Simulating quantum circuits by model counting. arXiv:2403.07197 (2024)
48. Mei, J., Coopmans, T., Bonsangue, M., Laarman, A.: Equivalence checking of quantum circuits by model counting. In: IJCAR (accepted for publication), Preprint available at arXiv:2403.18813 (2024)
49. Meuli, G., Soeken, M., De Micheli, G.: SAT-based CNOT, T quantum circuit synthesis. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 175–188. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_12
50. Miller, D.M., Thornton, M.A.: QMDD: a decision diagram structure for reversible and quantum circuits. In: 36th International Symposium on Multiple-Valued Logic (ISMVL'06), pp. 30–30. IEEE (2006)
51. Nam, Y., Su, Y., Maslov, D.: Approximate quantum fourier transform with $o(n\log(n))$ t gates. NPJ Quantum Inf. **6**(1), 26 (2020)
52. Nielsen, M.A., Chuang, I.L.: Quantum Information and Quantum Computation. Cambridge University Press, Cambridge, vol. 2, issue 8, p. 23 (2000)
53. Niemann, P., Zulehner, A., Drechsler, R., Wille, R.: Overcoming the tradeoff between accuracy and compactness in decision diagrams for quantum computation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(12), 4657–4668 (2020)
54. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: IJCAI'15, pp. 3141–3148. AAAI Press (2015)
55. Pashayan, H., Bartlett, S.D., Gross, D.: From estimation of quantum probabilities to simulation of quantum circuits. Quantum **4**, 223 (2020)

56. Peham, T., Burgholzer, L., Wille, R.: Equivalence checking of quantum circuits with the ZX-calculus. IEEE J. Emerging Sel. Top. Circ. Syst. **12**(3), 662–675 (2022)
57. Polla, S., Anselmetti, G.-L.R., O'Brien, T.E.: Optimizing the information extracted by a single qubit measurement. Phys. Rev. A **108**, 012403 (2023)
58. Quetschlich, N., Burgholzer, L., Wille, R.: MQT bench: benchmarking software and design automation tools for quantum computing. Quantum **7**, 1062 (2023)
59. Raussendorf, R., Briegel, H.J.: A one-way quantum computer. Phys. Rev. Lett. **86**, 5188–5191 (2001)
60. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: International Conference on Theory and Applications of Satisfiability Testing (2004)
61. Sanner, S., McAllester, D.: Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI'05, pp. 1384–1390, San Francisco, CA, USA (2005). Morgan Kaufmann Publishers Inc
62. Schneider, S., Burgholzer, L., Wille, R.: A SAT encoding for optimal Clifford circuit synthesis. In: Proceedings of the 28th Asia and South Pacific Design Automation Conference, ASPDAC '23. ACM (2023)
63. Shaik, I., van de Pol, J.: Optimal layout synthesis for quantum circuits as classical planning. arXiv preprint arXiv:2304.12014 (2023)
64. Sistla, M., Chaudhuri, S., Reps, T.: Symbolic quantum simulation with quasimodo. In: Enea, C., Lal, A., editors, Computer Aided Verification, pp. 213–225. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_11
65. Sistla, M., Chaudhuri, S., Reps, T.: Weighted context-free-language ordered binary decision diagrams. arXiv preprint arXiv:2305.13610 (2023)
66. Steane, A.M.: Error correcting codes in quantum theory. Phys. Rev. Lett. **77**, 793–797 (1996)
67. Tafertshofer, P., Pedram, M.: Factored edge-valued binary decision diagrams. Formal Methods Syst. Design **10**(2), 243–270 (1997)
68. Thanos, D., Coopmans, T., Laarman, A.: Fast equivalence checking of quantum circuits of Clifford gates. In: André, É., Sun, J., eds, Automated Technology for Verification and Analysis, pp. 199–216. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-45332-8_10
69. Tóth, G., Gühne, O.: Entanglement detection in the stabilizer formalism. Phys. Rev. A **72**, 022340 (2005)
70. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Improving gate-level simulation of quantum circuits. Quantum Inf. Process. **2**(5), 347–380 (2003)
71. Viamontes, G.F., Markov, I.L., Hayes, J.P.: High-performance QuIDD-based simulation of quantum circuits. In: Proceedings Design, Automation and Test in Europe Conference and Exhibition, vol. 2, pp. 1354–1355 (2004)
72. Wang, Q., Li, R., Ying, M.: Equivalence checking of sequential quantum circuits. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **41**(9), 3143–3156 (2022)
73. Wille, R., Zhang, H., Drechsler, R.: ATPG for reversible circuits using simulation, Boolean satisfiability, and pseudo Boolean optimization. In: 2011 IEEE Computer Society Annual Symposium on VLSI, pp. 120–125 (2011)
74. Wilson, N.: Decision diagrams for the computation of semiring valuations. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, pp. 331–336 (2005)
75. Wright, J., et al.: Numerical simulations of noisy quantum circuits for computational chemistry. Materials Theory **6**(1), 18 (2022)

76. Zhang, Y., Tang, Y., Zhou, Y., Ma, X.: Efficient entanglement generation and detection of generalized stabilizer states. Phys. Rev. A **103**, 052426 (2021)
77. Zulehner, A., Wille, R.: One-pass design of reversible circuits: combining embedding and synthesis for reversible logic. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(5), 996–1008 (2018)

Author Index

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 579–582, 2024.
https://doi.org/10.1007/978-3-031-65633-0