

Alexander Nadel / Kristin Yvonne Rozier (Eds.)

PROCEEDINGS OF THE 23RD CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2023



Academic Press



fmcad.²³

Alexander Nadel / Kristin Yvonne Rozier (Eds.)

PROCEEDINGS OF THE 23RD CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED
DESIGN – FMCAD 2023

Conference Series: Formal Methods in Computer-Aided Design

Volume 4

Conference Series: Formal Methods in Computer-Aided Design

Series edited by:

Warren A. Hunt, Jr., The University of Texas at Austin
Austin, TX 78705 | hunt@cs.utexas.edu

Georg Weissenbacher, TU Wien
Karlsplatz 13, 1040 Vienna, Austria | georg.weissenbacher@tuwien.ac.at

The Conference on Formal Methods in Computer-Aided Design (FMCAD) is an annual conference on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to researchers in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers formal aspects of computer-aided system design including verification, specification, synthesis, and testing.

Information on this publication series and the volumes published therein is available at www.tuwien.ac.at/academicpress.

Volume 4 edited by:

Alexander Nadel, Intel and Technion - Israel Institute of Technology
Technion City, Haifa, Israel | alexander.nadel@cs.tau.ac.il

Kristin Yvonne Rozier, Iowa State University of Science and Technology
Ames, Iowa, USA | kyrozier@iastate.edu

Alexander Nadel / Kristin Yvonne Rozier (Eds.)

PROCEEDINGS OF THE 23RD CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2023

Cite as:

Nadel, A. & Rozier, K. Y. (Eds.). (2023). *Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design – FMCAD 2023*. TU Wien Academic Press. <https://doi.org/10.34727/2023/isbn.978-3-85448-060-0>

TU Wien Academic Press, 2023

c/o TU Wien Bibliothek
TU Wien
Resselgasse 4, 1040 Wien
academicpress@tuwien.ac.at
www.tuwien.at/academicpress



This work is licensed under a Creative Commons attribution 4.0 international license (CC BY 4.0).
<https://creativecommons.org/licenses/by/4.0/>

ISBN (online): 978-3-85448-060-0
ISSN (online): 2708-7824

Available online: <https://doi.org/10.34727/2023/isbn.978-3-85448-060-0>

Media proprietor: TU Wien, Karlsplatz 13, 1040 Wien
Publisher: TU Wien Academic Press
Publication series editor: Warren A. Hunt, Jr. and Georg Weissenbacher
Editors (responsible for the content): Alexander Nadel and Kristin Yvonne Rozier

Preface

These are the proceedings of the twenty-third International Conference on Formal Methods in Computer-Aided Design (FMCAD), which was held in Ames, Iowa, USA from October 24 – October 27, 2023. FMCAD was first held in 1996, and was a bi-annual conference until 2006, when the FMCAD and CHARME conferences merged into a single FMCAD conference, and since then has been held annually. FMCAD 2023 is the twenty-third edition in the series, covering formal aspects of computer-aided system design including verification, specification, synthesis, and testing. It provides a leading forum to researchers in academia and industry to present and discuss groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems.

The program of FMCAD 2023 consists of four tutorials, three invited talks, a student forum, and the main program consisting of presentations of 31 accepted peer-reviewed papers.

The tutorial day featured four presentations:

- *Developing an Open-Source, State-of-the-Art Symbolic Model-Checking Framework for the Model-Checking Research Community* by The NSF:CCRI Project Investigators (Kristin Y. Rozier, Natarajan Shankar, Cesare Tinelli, Moshe Vardi)
- *MiniZinc for Formal Methods* by Peter J. Stuckey
- *Local Search and Its Application in CDCL/CDCL(T) solvers for SAT/SMT* by Shaowei Cai
- *NASA's core Flight System Framework Overview/Tutorial* by David Swartwout

and the main conference featured three invited talks:

- *Reasoning about quantifiers in SMT: the QSMA algorithm* by Maria Paola Bonacina
- *Distribution Testing: The New Frontier for Formal Methods* by Kuldeep Meel
- *Formal Methods for Trusted AI* by Bettina Könighofer

FMCAD 2023 received 73 submissions out of which the committee decided to accept 31 for publication. Each submission received at least four reviews, except for two submissions which received three reviews. The topics of the accepted papers include machine learning, model checking, hardware validation, SAT & SMT solving, avionics, security, synthesis and others. Among the accepted papers, there are 23 regular papers (20 long and 3 short) and 8 tool/case study papers (all long).

FMCAD 2023 hosted the eleventh edition of the Student Forum, which has been held annually since 2013 and provides a platform for graduate students at any career stage to introduce their research to the FMCAD community. The FMCAD Student Forum 2023 was organized by Mikoláš Janota and Nina Narodytska and featured short presentations of 16 accepted contributions. The proceedings provide a detailed description of the Student Forum and lists all accepted contributions.

Organizing this event was made possible by the support of a large number of people and our sponsors. The program committee members and additional reviewers, listed on the following pages, did an excellent job providing detailed and insightful reviews. The reviews helped us build a strong program and helped the authors improve their submissions. We thank each and everyone of them for dedicating their time and providing their expertise. We thank our web master Yogev Shalmon, our sponsorship chair Yoni Zohar and the Student Forum organizers Mikoláš Janota and Nina Narodytska. We thank Georg Weissenbacher both for his exceptional assistance in organizing the event, communicating to us the decisions of the steering committee, as well as being the publication chair.

Holding a conference like FMCAD would not be feasible without the financial support of our sponsors. We would like to express our gratitude to our sponsors (in alphabetical order): AWS, Cadence, Futurewei, GE Aerospace, Siemens, Synopsys and Toyota.

The conference proceedings are available as Open Access Proceedings published by TU Wien Academic Press, and through the IEEE Xplore Digital Library. Last but not least, we thank all authors who submitted their papers to FMCAD 2023 (accepted or not), and whose contributions and presentations form the core of the conference.

We are grateful to everyone who presented their paper, gave a keynote or gave a tutorial. We thank all attendees of FMCAD for supporting the conference and making FMCAD an engaging and enjoyable event.

October 2023

Kristin Y. Rozier Iowa State University, IA, USA
Alexander Nadel Intel Corporation and Technion, Israel

Organizing Committee

Program Co-Chairs

Kristin Y. Rozier
Alexander Nadel

Iowa State University, IA, USA
Intel Corporation and Technion, Israel

Student Forum Chairs

Mikoláš Janota
Nina Narodytska

Czech Technical University, Czechia
VMware Research, CA, USA

Sponsorship Chair

Yoni Zohar

Bar Ilan University, Israel

Web Chair

Yogev Shalmon

Intel Corporation and Open University, Israel

Publication Chair

Georg Weissenbacher

TU Wien, Austria

FMCAD Steering Committee

Clark Barrett
Armin Biere
Ruzica Piskac
Anna Slobodova
Georg Weissenbacher

Stanford University, CA, USA
University of Freiburg, Germany
Yale University, CT, USA
Intel Corporation, TX, USA
TU Wien, Austria

Program Committees

FMCAD 2023 Program Committee

Alessandro Abate	Oxford
Guy Amir	Hebrew University
Clark Barrett	Stanford University
Per Bjesse	Synopsys Inc.
Roderick Bloem	Graz University of Technology
Ivana Cerna	Masaryk University
Supratik Chakraborty	IIT Bombay
Sylvain Conchon	Universite Paris-Sud
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Rohit Dureja	IBM
Grigory Fedjukovich	Florida State University
Mathias Fleury	University of Freiburg
Amit Goel	Amazon
Alberto Griggio	Fondazione Bruno Kessler
Arie Gurfinkel	University of Waterloo
Liana Hadarean	Amazon Web Services
Ziyad Hanna	Cadence Design Systems
William Harrison	Two Six Technologies
Bo-Yuan Huang	Intel
Alan Jović	University of Zagreb
Daniela Kaufmann	TU Wien
Tim King	Google
Stepan Kochemazov	ISDCT SB RAS, ITMO University
Rebekah Leslie-Hurd	Rain
Andreas Löw	Imperial College London
Kuldeep Meel	University of Toronto
Baoluo Meng	GE Research
Naoko Okubo	Japan Aerospace Exploration Agency (JAXA)
Andrew Reynolds	University of Iowa
Philipp Riemmer	University of Regensburg
Cristoph Scholl	University of Freiburg
Roberto Sebastiani	University of Trento
Shaowei Cai	Chinese Academy of Sciences
Natasha Sharygina	Università della Svizzera Italiana (USI Lugano)
Christoph Stickel	The Mathworks
Christoph Torens	DLR (German Aerospace Center)
Nestan Tsikaridze	Stanford University
Yakir Vitzel	Technion
Georg Weissenbacher	TU Wien
Michael Whalen	Amazon Web Services, Inc.
Shufang Zhu	Oxford

FMCAD 2023 Student Forum Committee

Haniel Barbosa	Universidade Federal de Minas Gerais
Jaroslav Bendik	Certora
Armin Biere	University of Freiburg
Martin Blicha	University of Lugano
Nikolaj Bjørner	Microsoft Research
Martin Nyx Brain	University of London
Isabel Garcia Contreras	University of Waterloo
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Katalin Fazekas	TU Wien
Mathias Fleury	University of Freiburg
Arie Gurfinkel	University of Waterloo
Antti Hyvärinen	Università della Svizzera Italiana (USI Lugano)
Martin Jonáš	Fondazione Bruno Kessler
Daniela Kaufmann	TU Wien
Konstantin Korovin	The University of Manchester
Giles Reger	AWS and The University of Manchester
Andrew Reynolds	University of Iowa
Corina Pasareanu	NASA and Carnegie Mellon University
Mathias Preiner	Stanford
Karem Sakallah	University of Michigan
Mark Santolucito	Barnard College
Carsten Sinz	Karlsruhe Institute of Technology
Nestan Tsiskaridze	Stanford University
Tom van Dijk	University of Twente
Florian Zuleger	TU Wien

Additional Reviewers

Ahlbrecht, Alexander
Athavale, Anagha

Bayless, Samuel
Blich, Martin
Blumensath, Achim
Bombardelli, Alberto
Britikov, Konstantin

Ernst, Gidon
Esen, Zafer

Fraer, Ranan

Gacek, Andrew
Gajavelly, Raj Kumar

Hader, Thomas
Hamza, Ameer
Hjort, Håkan

Isac, Omri

Justino, Daniel
Jünger, Franz

Kiesl-Reiter, Benjamin
Kobayashi, Tsutomu

Le, Nham
Leslie-Hurd, Joe
Liang, Chencheng
Lukina, Anna
Lundgren, Lars

Mann, Makai
Masina, Gabriele
Meggendorfer, Tobias
Micheli, Andrea
Möhle, Sibylle
Morettin, Paolo

Neider, Daniel
Noetzli, Andres

Oertel, Andy
Otoni, Rodrigo

Paul, Saswata
Prabhu, Sumanth
Preiner, Mathias
Priya, Siddharth

Rappoport, Omer
Rath, Jakob
Redondi, Gianluca
Riley, Daniel
Roveri, Marco

S, Akshay
Schirmer, Sebastian
Sharma, Vaibhav
Singhania, Nimit
Somech, Nir
Spallitta, Giuseppe

Temel, Mertcan
Tiemeyer, Andreas

Ueda, Yasushi

Varanasi, Sarat Chandra

Westphal, Bernd
Wilson, Amalee
Wolfovitz, Guy
Wu, Haoze

Zelazny, Tom
Zavalía, Lucas
Zohar, Yoni

Table of Contents

Invited Talks

Reasoning about Quantifiers in SMT: The QSMA Algorithm.....	1
<i>Maria Paola Bonacina</i>	
Distribution Testing: The New Frontier for Formal Methods.....	2
<i>Kuldeep Meel</i>	
Formal Methods for Trusted AI.....	3
<i>Bettina Könighofer</i>	

Tutorials

Developing an Open-Source, State-of-the-Art Symbolic Model-Checking Framework for the Model-Checking Research Community.....	4
<i>Kristin Y. Rozier, Natarajan Shankar, Cesare Tinelli, Moshe Vardi</i>	
MiniZinc for Formal Methods.....	5
<i>Peter J. Stuckey</i>	
Local Search and Its Application in CDCL/CDCL(T) solvers for SAT/SMT.....	6
<i>Shaowei Cai</i>	
NASA's core Flight System Framework Overview.....	7
<i>David Swartwout</i>	

Student Forum

The FMCAD 2022 Student Forum.....	8
<i>Mikoláš Janota, Nina Narodytska</i>	

Neural Networks and Machine Learning

Formally Explaining Neural Networks within Reactive Systems.....	10
<i>Shahaf Bassan, Guy Amir, Davide Corsi, Idan Refaeli, and Guy Katz</i>	
Lightweight Online Learning for Sets of Related Problems in Automated Reasoning.....	23
<i>Haoze Wu, Christopher Hahn, Florian Lonsing, Makai Mann, Raghuram Ramanujan, and Clark Barrett</i>	
DelBugV: Delta-Debugging Neural Network Verifiers.....	34
<i>Raya Elsaleh and Guy Katz</i>	

Model Checking

Towards Compositional Hardware Model Checking Certification.....	44
<i>Emily Yu, Nils Froleys, Armin Biere, and Keijo Heljanko</i>	

Btor2MLIR: A Format and Toolchain for Hardware Verification	55
<i>Joseph Tafese, Isabel Garcia-Contreras, and Arie Gurfinkel</i>	
Data-Driven Learning of Strong Conjunctive Invariants	64
<i>Arkesh Thakkar and Deepak D'Souza</i>	
Automating Cutoff-based Verification of Distributed Protocols	75
<i>Shreesha G. Bhat and Kartik Nagar</i>	
Optimal Bounded Partial Order Reduction	86
<i>Iason Marmanis and Viktor Vafeiadis</i>	
Hardware	
Datapath Verification via Word-Level E-Graph Rewriting	92
<i>Samuel Coward, Emiliano Morini, Bryan Tan, Theo Drane, and George A. Constantinides</i>	
μ ArchiFI: Formal Modeling and Verification Strategies for Microarchitectural Fault Injections	101
<i>Simon Tollec, Mihail Asavoaie, Damien Couroussé, Karine Heydemann, and Mathieu Jan</i>	
Sylvia: Countering the Path Explosion Problem in the Symbolic Execution of Hardware Designs ...	110
<i>Kaki Ryan and Cynthia Sturton</i>	
Binary decision diagrams on modern hardware	122
<i>Samuel Pastva and Thomas Henzinger</i>	
SAT	
Proofs for Incremental SAT with Inprocessing	132
<i>Benjamin Kiesl-Reiter and Michael W. Whalen</i>	
Verified Encodings for SAT Solvers	141
<i>Cayden R. Codel, Jeremy Avigad, and Marijn J. H. Heule</i>	
SAT-Based Quantified Symmetric Minimization of the Reachable States of Distributed Protocols ...	152
<i>Katalin Fazekas, Aman Goel, and Karem A. Sakallah</i>	
BIG Backbones	162
<i>Nils Froleys, Emily Yu, and Armin Biere</i>	
SMT	
Local Search For SMT On Linear and Multilinear Real Arithmetic	168
<i>Bohan Li and Shaowei Cai</i>	
Mariposa: Measuring SMT Instability in Automated Program Verification	178
<i>Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn J. H. Heule, and Bryan Parno</i>	
A Procedure for SyGuS Solution Fitting via Matching and Rewrite Rule Discovery	189
<i>Abdallah Mohamed, Andrew Reynolds, Clark Barrett, and Cesare Tinelli</i>	
Partitioning Strategies for Distributed SMT Solving	199
<i>Amalee Wilson, Andres Noetzli, Andrew Reynolds, Byron Cook, Cesare Tinelli, and Clark Barrett</i>	

Avionics

CRV: An Automated Resiliency Reasoner for System Design Models	209
<i>Daniel Larraz, Robert Lorch, Moosa Yahyazadeh, M. Fareed Arif, Omar Chowdhury, and Cesare Tinelli</i>	
Towards a Correct-by-Construction Design of Integrated Modular Avionics	221
<i>Baolu Meng, Joyanta Debnath, Sarat Chandra Varanasi, Emmanuel Manolios, Michael Durling, Saswata Paul, Daniel Prince, Saif Alsabbagh, Richard Haadsma, Craig McMillan, Chi Zhang, and Tim Oates</i>	
Fortis: A Tool for Analysis and Repair of Robust Software Systems	228
<i>Changjian Zhang, Ian Dardik, Rômulo Meira-Góes, David Garlan, and Eunsuk Kang</i>	
A provably correct floating-point implementation of Well Clear Avionics Concepts	237
<i>Nikson Bernardes Fernandes Ferreira, Mariano M. Moscato, Laura Titolo, and Mauricio Ayala-Rincón</i>	

Security and Synthesis

Formal Verification of Correctness and Information Flow Security for an In-Order Pipelined Processor	247
<i>Ning Dong, Roberto Guanciale, Mads Dam, and Andreas Lööw</i>	
Modular System Synthesis	257
<i>Kanghee Park, Keith J. C. Johnson, Loris D’Antoni, and Thomas Reps</i>	
Modelling and Verification of Security-Oriented Resource Partitioning Schemes	268
<i>Adwait Godbole, Leiqi Ye, Yatin A. Manerkar, and Sanjit A. Seshia</i>	

Cyber-Physical Systems

Lift-off: Trustworthy ARMv8 semantics from formal specifications	274
<i>Kait Lam and Nicholas Coughlin</i>	
Cycle and Commute: Rare-Event Probability Verification for Chemical Reaction Networks	284
<i>Landon Taylor, Bryant Israelsen, and Zhen Zhang</i>	
Conformance Testing for Stochastic Cyber-Physical Systems	294
<i>Xin Qin, Navid Hashemi, Lars Lindemann, Jyotirmoy V. Deshmukh</i>	
MediK: Towards Safe Guideline-based Clinical Decision Support	306
<i>Manasvi Saxena, Shuang Song, and Lui Sha</i>	

Reasoning about Quantifiers in SMT: The QSMA algorithm

Maria Paola Bonacina
Università degli Studi di Verona
Verona, Italy
mariapaola.bonacina@univr.it

Abstract—Automated reasoning is a key enabling technology for formal methods. Automated theorem provers (ATP) for first-order and lately higher-order logic and solvers for satisfiability modulo theories (SMT) showcase impressive power and amazing sophistication. However, ATP systems reason well about formulas with arbitrary quantification and free symbols, while SMT solvers reason well about ground formulas with defined symbols. Since formulas from applications involve both quantifiers and defined symbols, a hiatus remains open. QSMA is a new algorithm for quantifiers in SMT [4]. QSMA stands for Quantified Satisfiability Modulo theory and Assignment. Currently, QSMA works for one theory with a unique intended model, so that models differ only in the assignment of values to variables. QSMA accepts arbitrary formulas, viewing all quantifiers as existential by double negation. Since QSMA operates a recursive descent over the tree structure of the formula, peeling off quantifiers and instantiating variables, each call works modulo an assignment. By building under- and over- approximations of the formula, QSMA zooms in on a model or finds that none exists. The implementation of QSMA in the YicesQS solver [6] built on top of the Yices 2 solver [5] exhibited excellent performances in linear rational arithmetic and nonlinear arithmetic [4]. Integrating QSMA in the CDSAT framework for conflict-driven satisfiability in a union of theories [1]–[3] is the next challenge.

REFERENCES

- [1] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Conflict-driven satisfiability for theory combination: transition system and completeness. *J. Autom. Reason.*, 64(3):579–609, March 2020.
- [2] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. CDSAT for nondisjoint theories with shared predicates: arrays with abstract length. In Antti Hyvärinen and David Déharbe, editors, *Proc. SMT-20*, volume 3185 of *CEUR Proceedings*, pages 18–37. CEUR WS-org, August 2022.
- [3] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Conflict-driven satisfiability for theory combination: lemmas, modules, and proofs. *J. Autom. Reason.*, 66(1):43–91, February 2022.
- [4] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Christophe Vauthier. QSMA: a new algorithm for quantified satisfiability modulo theory and assignment. In Brigitte Pientka and Cesare Tinelli, editors, *Proceedings of the 29th International Conference on Automated Deduction (CADE)*, volume 14132 of *Lecture Notes in Artificial Intelligence*, pages 78–95. Springer, 2023.
- [5] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.
- [6] Stéphane Graham-Lengrand. The yicesQS solver web page. <https://github.com/disteph/yicesQS>, 2023. Last seen on September 14, 2023.

Distribution Testing: The New Frontier for Formal Methods

Kuldeep Meel
University of Toronto
Toronto, Canada
meel@cs.toronto.edu

Abstract—The dominant guiding philosophy in the first sixty years of Computer Science was for designers to design systems that were always correct, and to accept nothing less as users. But times have changed: Users and designers are accustomed to systems with statistical components and behaviors. What does it mean for the formal methods community?

In this talk, we argue that such a dramatic change in the acceptance and design of systems presents exciting opportunities to make fundamental contributions: we need to rethink the notions and techniques for the design of specifications and verification methodologies. In particular, we will focus on the systems whose behaviors are not naturally captured by symbolic relations but rather require reliance on probability distributions. We will discuss our recent efforts in designing formal methodologies for testing whether a sampling subroutine generates a desired distribution. We will showcase the challenges, opportunities, and rewards in our journey.

Formal Methods for Trused AI

Bettina Könighofer

Graz University of Technology

Graz, Austria

bettina.koenighofer@tugraz.at

Abstract—The enormous influence of systems deploying AI is contrasted by the growing concerns about their safety and the relative lack of trust by the society. This talk will focus on a few aspects of trustworthy AI: safety, accountability, and explainability. First, we will discuss recent work on evaluating safety for systems deploying deep learning, and correct-by-construction runtime assurance methods to enforce safety during runtime (aka shielding). For accountability, we outline the potential of formal computing tools to analyse the decisions of autonomous agents and to assign responsibility. Finally, we approach explainability from the automata learning perspective. We will discuss recent automata learning approaches which are able to learn compact probabilistic models for high-dimensional environments and outline how learned environmental models can effectively be used to understand and to evaluate the decisions of the agent.

Developing an Open-Source, State-of-the-Art Symbolic Model-Checking Framework for the Model-Checking Research Community

Kristin Y. Rozier
Iowa State University
Ames, IA, USA
kyrozier@iastate.edu

Natarajan Shankar
SRI
Menlo Park, CA, USA
shankar@cs.sri.com

Cesare Tinelli
University of Iowa
Iowa City, IA, USA
cesare-tinelli@uiowa.edu

Moshe Vardi
Rice University
Houston, TX, USA
vardi@cs.rice.edu

Abstract—As model checking becomes more integrated into the standard design and verification process for safety-critical systems, the platforms for model checking research have become more limited. Previous options have become closed-source or industry tools; current research platforms don't have support for expressive specification languages needed for verifying real systems. Our goal is to fill the current gap in model checking research platforms: building a freely-available, open-source, scalable model checking infrastructure that accepts expressive models and efficiently interfaces with the currently-maintained state-of-the-art back-end algorithms to provide an extensible research and verification tool. With extensive involvement from the research community, we have been creating a community resource with a well-documented intermediate representation to enable extensibility, and a web portal, facilitating new modeling languages and back-end algorithmic advances. To add new modeling languages or algorithms, researchers need only to develop a translator to/from the new intermediate language, and will then be able to integrate each advance with the full state-of-the-art in model checking. This tutorial will include an overview of the model checking intermediate language semantics and demonstrations of (provably correct) translators to and from that representation.

MiniZinc for Formal Methods

Peter J. Stuckey

Monash University

Melbourne, Australia

Peter.Stuckey@monash.edu

Abstract—MiniZinc is a free and open-source constraint modeling language, designed for solving discrete optimisation problems. You can use MiniZinc to model constraint satisfaction and optimization problems in a high-level, solver-independent way, taking advantage of a large library of pre-defined constraints encapsulating different combinatorial substructures of the problem. Your model is then compiled into FlatZinc, a solver input language that is understood by a wide range of solvers including leading CP solvers such as OR-tools from Google, and CP-optimiser from IBM, and leading MIP solvers such as Gurobi and Cplex. MiniZinc is useful for Formal Reasoning problems where we reason about a bounded size problem on discrete objects (including integers). In this tutorial we will give an introduction to MiniZinc focusing, at least in the latter part, on where it can be applied to Formal Methods problems. Formal methods modelling through MiniZinc leads to different ways to model things such as state progression, and can take advantage of combinatorial substructures that occur in such problems, such as injective mappings. Overall MiniZinc gives an alternate, and often highly competitive, approach to using SMT for answering some kinds of formal reasoning questions.

Local Search and Its Application in CDCL/CDCL(T) solvers for SAT/SMT

Shaowei Cai

Chinese Academy of Sciences

Beijing, China

caisw@ios.ac.cn

Abstract—Modern SAT solvers are based on a paradigm named conflict driven clause learning (CDCL), and CDCL(T) remains the main method for SMT. Local search is an important alternative for satisfiable instances, which has witnessed significant progress in SAT, and has begun to show promising results in SMT. Furthermore, recent techniques integrating local search into CDCL have brought significant improvements, and local search is widely used in state of the art CDCL solvers as an important component. Similar results have also been observed in CDCL(T). This talk will introduce state of the art local search methods for SAT and SMT, and also present the recent techniques of combining local search and CDCL/CDCL(T).

NASA's core Flight System Framework Overview

David Swartwout


NASA


Houston, TX, USA

dave.c.swartwout@nasa.gov

Abstract—The core Flight System (cFS) is a platform and project independent reusable software framework and set of reusable software applications. There are three key aspects to the cFS architecture: a dynamic run-time environment, layered software, and a component-based design. It is the combination of these key aspects that makes it suitable for reuse on any number of flight projects and/or embedded software systems at a significant cost and schedule savings. This tutorial will give a brief overview of the architecture and demonstrate a simple app development and deployment.

The FMCAD 2023 Student Forum

Mikolas Janota 
 Czech Technical University
 Prague, Czechia
 mikolas.janota@gmail.com

Nina Narodytska 
 VMware Research
 Palo Alto, USA
 n.narodytska@gmail.com

Abstract—The Student Forum at the International Conference on Formal Methods in Computer-Aided Design (FMCAD) gives undergraduate and graduate students the opportunity to introduce their research to the Formal Methods community and receive feedback. In 2023, the event took place in Ames, Iowa, USA. Eighteen students were invited to give a short talk and present a poster of their work.

Since 2013, the FMCAD Student Forum provides a platform for undergraduate and graduate students at any career stage to present their research to the audience of the FMCAD conference. The 2023 edition of the FMCAD Student Forum follows the tradition of its predecessors, which took place in:

- Portland, Oregon, USA in 2013 [1]
- Lausanne, Switzerland in 2014 [2]
- Austin, Texas in 2015 [3] and 2018 [4]
- Mountain View, California, USA in 2016 [5]
- Vienna, Austria in 2017 [6]
- San Jose, California, USA in 2019 [7]
- Virtual in 2020 [8] and 2021 [9]
- Trento, Italy in 2022 [10]

FMCAD 2023 hosted the eleventh edition of the Student Forum. Graduate and undergraduate students were invited to submit two-page reports of their current research and ongoing work in the scope of the FMCAD conference. There were 16 submissions to the forum and all of them were accepted. The Student Forum program committee reviews were based on the overall quality, novelty of the work, its potential impact on the Formal Methods community, as well as the potential positive impact on the student to have the opportunity to participate in the forum. The accepted submissions covered a wide range of topics relevant to the FMCAD community, from foundational aspects of automated reasoning, to analysis and verification of software, hardware, and neural networks, as well as applications of formal methods to confidential computing, security and chemistry. Each submission received at least 2 reviews. The following contributions have been accepted¹:

- Arijit Shaw: *Towards Building A Scalable Bit-vector Model Counter*
- Rachel Cleaveland: *MemGlue: An Update-Based Cache Coherence Protocol for Heterogeneous Hardware*
- Guy Amir: *Finding Formal Explanations of Reactive DNNs*

- Landon Taylor: *Enhancing Property-Directed Reachability for Chemical Reaction Networks*
- Benjamin Valpey: *Formal Analysis of Tensor Core Math via SMT*
- Samuel Coward: *Datapath Optimization, Analysis and Verification: E-Graphs Can Do It All*
- Joseph Tafese: *Btor2MLIR: A Format for Hardware Verification*
- Sophie Andrews and Matthew Sotoudeh: *Incremental Bounded Model Checking as Program Source Changes*
- Cayden Codel: *Modifying the DDFW Local Search Algorithm*
- Viansa Schmulbach: *Ordering Interventions for Hardware Security*
- Áron Ricardo Perez-Lopez and Akshay Srivatsan: *Mixed-Signal Verification via Digital Simulation*
- Raya Elsaleh: *On Facilitating the Development of Neural Networks Verifiers*
- Muhammad Usama Sardar: *Formal Specification and Verification of Attestation in Confidential Computing*
- Charles Pert: *Synthesizing Omega-Regular Expressions from Omega-automata*
- Tephilla Prince: *Two-dimensional Bounded Model Checking for Petri Nets*
- Ankit Shukla: *Simplifying Dependency Quantified Boolean Formulas with Biclique Cover Transformations*

We formed a program committee to cover a wide range of topics so students could receive expert feedback on their work. The 2023 FMCAD Student Forum program committee consisted of Mikolas Janota (co-chair), Nina Narodytska (co-chair), Haniel Barbosa, Jaroslav Bendík, Armin Biere, Nikolaj Bjørner, Martin Blicha, Martin Brain, Rayna Dimitrova, Katalin Fazekas, Mathias Fleury, Isabel Garcia, Arie Gurfinkel, Martin Jonas, Daniela Kaufmann, Konstantin Korovin, Corina Pasareanu, Mathias Preiner, Giles Reger, Andrew Reynolds, Karem Sakallah, Mark Santolucito, Carsten Sinz, Nestan Tsiskaridze, and Florian Zuleger.

We would like to thank the organizers of FMCAD, as well as the FMCAD Student Forum program committee, who have made the FMCAD Student Forum possible. We would like to thank FMCAD and NSF for providing travel support to students. Additionally, we are grateful to the student authors and their research mentors who have contributed their excellent work to the program.

¹Only student authors listed for brevity.

REFERENCES

- [1] T. Wahl, “The FMCAD graduate student forum,” in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 16–17. [Online]. Available: <https://doi.org/10.1109/FMCAD.2013.7035523>
- [2] R. Piskac, “The FMCAD 2014 graduate student forum,” in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 2014, p. 13. [Online]. Available: <https://doi.org/10.1109/FMCAD.2014.6987589>
- [3] G. Weissenbacher, “The FMCAD 2015 graduate student forum,” in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, R. Kaivola and T. Wahl, Eds. IEEE, 2015, p. 8. [Online]. Available: <https://doi.org/10.1109/FMCAD.2015.7542246>
- [4] D. Jovanović and A. Reynolds, “The FMCAD 2018 graduate student forum,” in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 1–1, <https://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD18/student-forum/>.
- [5] H. Hojjat, “The FMCAD 2016 graduate student forum,” in *Formal Methods in Computer-Aided Design (FMCAD), 2016*. IEEE, 2016, pp. 8–8, <https://fmcad.forsyte.at/FMCAD16/student-forum.html>.
- [6] K. Heljanko, “The FMCAD 2017 graduate student forum,” in *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, 2017, pp. 10–10, <https://fmcad.org/FMCAD17/student-forum/>.
- [7] G. Fedyukovich, “The FMCAD 2019 student forum,” in *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2019, pp. 1–1, <https://fmcad.forsyte.at/FMCAD19/student-forum/>.
- [8] P. Schrammel, “The FMCAD 2020 student forum,” in *2020 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2020, pp. 1–1, <https://fmcad.forsyte.at/FMCAD20/student-forum/>.
- [9] M. Santolucito, “The FMCAD 2021 student forum,” in *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2021, pp. 1–1, https://fmcad.org/FMCAD21/student_forum/.
- [10] M. Preiner, “The FMCAD 2023 student forum,” in *2022 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2022, pp. 1–1, https://fmcad.org/FMCAD22/student_forum/.

Formally Explaining Neural Networks within Reactive Systems

Shahaf Bassan^{1,*}, Guy Amir^{1,*}, Davide Corsi², Idan Refaeli¹, and Guy Katz¹

¹The Hebrew University of Jerusalem, Jerusalem, Israel, {shahaf, guyam, idan0610, guykatz}@cs.huji.ac.il

²University of Verona, Verona, Italy, davide.corsi@univr.it

Abstract—Deep neural networks (DNNs) are increasingly being used as controllers in reactive systems. However, DNNs are highly opaque, which renders it difficult to explain and justify their actions. To mitigate this issue, there has been a surge of interest in explainable AI (XAI) techniques, capable of pinpointing the input features that caused the DNN to act as it did. Existing XAI techniques typically face two limitations: (i) they are heuristic, and do not provide formal guarantees that the explanations are correct; and (ii) they often apply to “one-shot” systems, where the DNN is invoked independently of past invocations, as opposed to reactive systems. Here, we begin bridging this gap, and propose a formal DNN-verification-based XAI technique for reasoning about multi-step, reactive systems. We suggest methods for efficiently calculating succinct explanations, by exploiting the system’s transition constraints in order to curtail the search space explored by the underlying verifier. We evaluate our approach on two popular benchmarks from the domain of automated navigation; and observe that our methods allow the efficient computation of minimal and minimum explanations, significantly outperforming the state of the art. We also demonstrate that our methods produce formal explanations that are more reliable than competing, non-verification-based XAI techniques.

I. INTRODUCTION

Deep neural networks (DNNs) [56] are used in numerous key domains, such as computer vision [54], natural language processing [24], computational biology [9], and more [23]. However, despite their tremendous success, DNNs remain “black boxes”, uninterpretable by humans. This issue is concerning, as DNNs are prone to critical errors [19], [96] and unexpected behaviors [10], [28].

DNN opacity has prompted significant research on explainable AI (XAI) techniques [62], [77], [78], aimed at explaining the decisions made by DNNs, in order to increase their trustworthiness and reliability. Modern XAI methods are useful and scalable, but they are typically heuristic; i.e., there is no provable guarantee that the produced explanation is correct [20], [45]. This hinders the applicability of these approaches to critical systems, where regulatory bars are high [66].

These limitations provide ample motivation for *formally* explaining DNN decisions [20], [33], [39], [66]. And indeed, the formal verification community has suggested harnessing recent developments in DNN verification [13], [22], [26], [29], [36], [67], [69]–[71], [81], [86], [91], [92] to produce

provable explanations for DNNs [17], [39], [44]. Typically, such approaches consider a particular input to the DNN, and return a subset of its features that caused the DNN to classify the input as it did. These subsets are called *abductive explanations*, *prime implicants* or *PI-explanations* [17], [44], [84]. This line of work constitutes a promising step towards more reliable XAI; but so far, existing work has focused on explaining decisions of “one-shot” DNNs, such as image and tabular data classifiers [17], [43], [44], and has not addressed more complex systems.

Modern DNNs are often used as controllers within elaborate reactive systems, where a DNN’s decisions affect its future invocations. A prime example is *deep reinforcement learning* (DRL) [59], where DNNs learn control policies for complex systems [11], [18], [57], [63], [72], [85], [95]. Explaining the decisions of DRL agents (XRL) [32], [50], [64], [74] is an important domain within XAI; but here too, modern XRL techniques are heuristic, and do not provide formally correct explanations.

In this work, we make a first attempt at formally defining abductive explanations for *multi-step decision processes*. We propose novel methods for computing such explanations and supply the theoretical groundwork for justifying the soundness of these methods. Our framework is model-agnostic, and could be applied to diverse kinds of models; but here, we focus on DNNs, where producing abductive explanations is known to be quite challenging [14], [17], [44]. With DNNs, our technique allows us to reduce the number of times a network has to be unrolled, circumventing a potential exponential blow-up in runtime; and also allows us to exploit the reactive system’s transition constraints, as well as the DNN’s sensitivity to small input perturbations, to curtail the search space even further.

For evaluation purposes, we implemented our approach as a proof-of-concept tool, which is publicly available as an artifact accompanying this paper [16]. We used this tool to automatically generate explanations for two popular DRL benchmarks: a navigation system on an abstract, two-dimensional grid, and a real-world robotic navigation system. Our evaluation demonstrates that our methods significantly outperform state-of-the-art, rigorous methods for generating abductive explanations, both in terms of efficiency and in the size of the explanation generated. When comparing our approach to modern, heuristic-based XAI approaches, our

* Both authors contributed equally.

explanations were found to be significantly more precise. We regard these results as strong evidence of the usefulness of applying verification in the context of XAI.

The rest of this paper is organized as follows: Sec. II contains background on DNNs, their verification, and their formal explainability. Sec. III contains our definitions for formal abductive explanations and contrastive examples for reactive systems. In Sec. IV we propose different methods for computing such abductive explanations. We then evaluate these approaches in Sec. V, followed by a discussion of related work in Sec. VI; and we conclude in Sec. VII.

II. BACKGROUND

DNNs. Deep neural networks (DNNs) [56] are directed, layered graphs, whose nodes are referred to as *neurons*. They propagate data from the first (*input*) layer, through intermediate (*hidden*) layers, and finally onto an *output* layer. A DNN's output is calculated by assigning values (representing input *features*) to the input layer, and then iteratively calculating the neurons' values in subsequent layers. In classification, each output neuron corresponds to a *class*, and the input is classified as the class matching the greatest output. Fig. 1 depicts a toy DNN. The input layer has three neurons and is followed by a weighted-sum layer that calculates an affine transformation of the input values. For example, given input $V_1 = [1, 1, 1]^T$, the second layer evaluates to $V_2 = [7, 8, 11]^T$. This is followed by a ReLU layer, which applies the $\text{ReLU}(x) = \max(0, x)$ function to each value in the previous layer, resulting in $V_3 = [7, 8, 11]^T$. The output layer computes the weighted sum $V_4 = [15, -4]^T$. Because the first output neuron has the greatest value, V_1 is classified as the output class corresponding to that neuron.

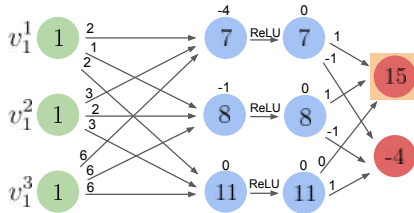


Fig. 1: A toy DNN.

DNN Verification. We define a DNN verification query as a tuple $\langle P, N, Q \rangle$, where N is a DNN that maps an input vector x to an output vector $y = N(x)$, P is a predicate over x , and Q is a predicate over y [51]. A DNN verifier needs to answer whether there exists some input x' that satisfies $P(x') \wedge Q(N(x'))$ (a SAT result) or not (an UNSAT result). It is common to express P and Q in the logic of real arithmetic [61]. The problem of verifying DNNs is known to be NP-Complete [51].

Formal Explanations for Classification DNNs. A classification problem is a tuple $\langle F, D, K, N \rangle$, where (i) $F = \{1, \dots, m\}$

is the feature set; (ii) $D = \{D_1, D_2, \dots, D_m\}$ are the domains of individual features, and the entire feature space is $\mathbb{F} = (D_1 \times D_2 \times \dots \times D_m)$; (iii) $K = \{c_1, c_2, \dots, c_n\}$ represents the set of all classes; and (iv) $N: \mathbb{F} \rightarrow K$ is the classification function, represented by a neural network. A *classification instance* is a pair (v, c) , where $v \in \mathbb{F}$, $c \in K$, and $c = N(v)$. Intuitively, this means that N maps the input v to class c .

Formally explaining the instance (v, c) entails determining *why* v is classified as c . An *explanation* (also known as an *abductive explanation*) is defined as a subset of features, $E \subseteq F$, such that fixing these features to their values in v guarantees that the input is classified as c , regardless of features in $F \setminus E$. The features *not* part of the explanation are “free” to take on any arbitrary value, but cannot affect the classification. Formally, given an input $v = (v_1, \dots, v_m) \in \mathbb{F}$ classified by the neural network to $N(v) = c$, we define an explanation as a subset of features $E \subseteq F$, such that:

$$\forall x \in \mathbb{F}. \bigwedge_{i \in E} (x_i = v_i) \rightarrow (N(x) = c) \quad (1)$$

We demonstrate formal explanations using the running example from Fig. 1. For simplicity, assume that each input can only take the values 0 or 1. Fig. 2 shows that the set $\{v_1^1, v_1^2\}$ is an explanation for the input vector $V_1 = [1, 1, 1]^T$: setting the first two features in V_1 to 1 ensures that the classification is unchanged, regardless of the values the third feature takes.

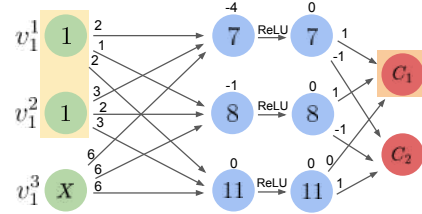


Fig. 2: $\{v_1^1, v_1^2\}$ is an explanation for input $V_1 = [1, 1, 1]^T$.

A candidate explanation E can be verified through a verification query $\langle P, N, Q \rangle = \langle E = v, N, Q_{\neg c} \rangle$, where $E = v$ means that all of the features in E are set to their corresponding values in v , and $Q_{\neg c}$ implies that the classification of this query is *not* c . If this query is UNSAT, then E is a valid explanation for the instance (v, c) .

It is straightforward to show that the set of all features is a trivial explanation. However, smaller explanations typically provide more meaningful information regarding the decision of the classifier; and we thus focus on finding *minimal* and *minimum* explanations. A *minimal explanation* is an explanation $E \subseteq F$ that ceases to be an explanation if any of its features are removed:

$$\begin{aligned} & (\forall x \in \mathbb{F}. \bigwedge_{i \in E} (x_i = v_i) \rightarrow (N(x) = c)) \wedge \\ & (\forall j \in E. \exists y \in \mathbb{F}. \bigwedge_{i \in E \setminus j} (y_i = v_i) \wedge (N(y) \neq c)) \end{aligned} \quad (2)$$

A minimal explanation for our running example, $\{v_1^1, v_1^2\}$, is depicted in Fig. 15 in the extended version of this paper [15].

A *minimum explanation* is a subset $E \subseteq F$ which is a minimal explanation of minimum size; i.e., there is no other minimal explanation $E' \neq E$ such that $|E'| < |E|$. Fig. 16 in the extended version of this paper [15] shows that $\{v_1^3\}$ is a minimal explanation of minimal cardinality, and is hence a minimum explanation in our example.

Contrastive Examples. We define a contrastive example (also known as a *contrastive explanation (CXP)*) as a subset of features $C \subseteq F$, whose alteration may cause the classification of v to change. More formally:

$$\exists x \in \mathbb{F}. \bigwedge_{i \in F \setminus C} (x_i = v_i) \wedge (N(x) \neq c) \quad (3)$$

A contrastive example for our running example appears in Fig. 3.

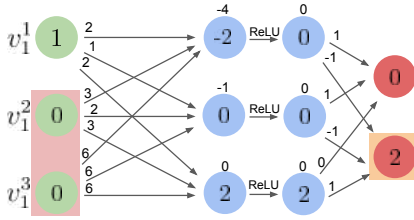


Fig. 3: $\{v_1^2, v_1^3\}$ is a contrastive example for $V_1 = [1, 1, 1]^T$.

Checking whether C is a contrastive example can be performed using the query $\langle P, N, Q \rangle = \langle (F \setminus C) = v, N, Q_{\neg c} \rangle$: C is contrastive iff the quest is SAT. Any set containing a contrastive example is contrastive, and so we consider only contrastive examples that are minimal, i.e., which do not contain any smaller contrastive examples.

Contrastive examples have an important property: every explanation contains at least one element from every contrastive example [17], [43]. This can be used for showing that a *minimum hitting set* (MHS; see [15]) of all contrastive examples is a minimum explanation [41], [76]. In addition, there exists a duality between contrastive examples and explanations [43], [47]: minimal hitting sets of all contrastive examples are minimal explanations, and minimal hitting sets of all explanations are minimal contrastive examples. This relation can be proved by reducing explanations and contrastive examples to minimal unsatisfiable sets and minimal correction sets, respectively, where this duality is known to hold [43]. Calculating an MHS is NP-hard, but can be performed in practice using modern MaxSAT or MILP solvers [38], [58]. The duality is thus useful since computing contrastive examples and calculating their MHS is often more efficient than directly computing minimum explanations [17], [43], [44].

III. K-STEP FORMAL EXPLANATIONS

A reactive system is a tuple $R = \langle S, A, I, T \rangle$, where S is a set of states, A is a set of actions, I is a predicate over the states of S that indicates initial states, and $T \subseteq S \times A \times S$ is a transition relation. In our context, a reactive system has an associated neural network $N : S \rightarrow A$. A k -step execution \mathcal{E} of

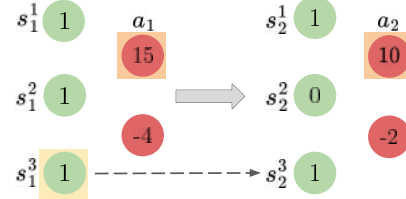


Fig. 4: $(\{s^3\}, \emptyset)$ is a (minimum) multi-step explanation for \mathcal{E} .

R is a sequence of k states (s_1, \dots, s_k) , such that $I(s_1)$ holds, and for all $1 \leq i \leq k-1$ it holds that $T(s_i, N(s_i), s_{i+1})$. We use $\mathcal{E}_S = (s_1, \dots, s_k)$ to denote the sequence of k states visited in \mathcal{E} , and $\mathcal{E}_A = (a_1, \dots, a_k)$ to denote the sequence of k actions selected in these states. More broadly, a reactive system can be considered as a deterministic, finite-state transducer Mealy automaton [82]. Our goal is to better understand \mathcal{E} , by finding abductive explanations and contrastive examples that explain why N selected the actions in \mathcal{E}_A .

K-Step Abductive Explanations. Informally, we define an explanation E for a k -step execution \mathcal{E} as a subset of features of each of the visited states in \mathcal{E}_S , such that fixing these features (while freeing all other features) is sufficient for forcing the DNN to select the actions in \mathcal{E}_A . More formally, $E = (E_1, \dots, E_k)$, such that $\forall x_1, x_2, \dots, x_k \in \mathbb{F}$,

$$\left(\bigwedge_{i=1}^{k-1} T(x_i, N(x_i), x_{i+1}) \wedge \bigwedge_{i=1}^k \bigwedge_{j \in E_i} (x_i^j = s_i^j) \right) \rightarrow \bigwedge_{i=1}^k N(x_i) = a_i \quad (4)$$

We continue with our running example. Consider the transition relation $T = \{(s, a, s') \mid s^3 = s'^3\}$; i.e., we can transition from state s to state s' provided that the third input neuron has the same value in both states, regardless of the action selected in s . Observe the 2-step execution $\mathcal{E} : s_1 = (1, 1, 1) \xrightarrow{c_1} s_2 = (1, 0, 1) \xrightarrow{c_1}$, depicted in Fig. 4 (for simplicity, we omit the network's hidden neurons), and suppose we wish to explain $\mathcal{E}_A = \{c_1, c_1\}$. Because $\{s^3\}$ is an explanation for the first step, and because fixing s_1^3 also fixes the value of s_2^3 , it follows that fixing s_1^3 is sufficient to guarantee that action c_1 is selected twice — i.e., $(\{s^3\}, \emptyset)$ is a multi-step explanation for \mathcal{E} .

Given a candidate k -step explanation, we can check its validity by encoding Eq. 4 as a DNN verification query. This is achieved by *unrolling* the network N for k subsequent steps; i.e., by encoding a network that is k times larger than N , with input and output vectors that are k times larger than the original. We must also encode the transition relation T as a set of constraints involving the input values, to mimic k time-steps within a single feed-forward pass. We use $N_{[i]}$ to denote an unrolling of the neural network N for i steps, for $1 \leq i \leq k$.

Using the unrolled network $N_{[k]}$, we encode the negation of Eq. 4 as the query $\langle P, N, Q \rangle = \langle E = \mathcal{E}_S, N_{[k]}, Q_{\neg \mathcal{E}_A} \rangle$, where $E = \mathcal{E}_S$ means that we restrict the features in each subset $E_i \in E$ to their corresponding values in s_i ; and $Q_{\neg \mathcal{E}_A}$ indicates

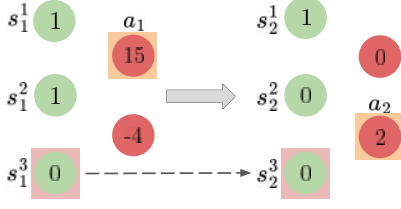


Fig. 5: $(\{s^3\}, \{s^3\})$ is a multi-step contrastive example for \mathcal{E} .

that in some step i , an action that is not a_i was selected by the DNN. An UNSAT result for this query indicates that E is an explanation for \mathcal{E} , because fixing E 's features to their values forces the given sequence of actions to occur.

We can naturally define a *minimal* k -step explanation as a k -step explanation that ceases to be a k -step explanation when we remove any of its features. A *minimum* k -step explanation is a minimal k -step explanation of the lowest possible cardinality; i.e., there does not exist a k -step explanation $E' = (E'_1, E'_2, \dots, E'_k)$ such that $\sum_{i=1}^k |E'_i| < \sum_{i=1}^k |E_i|$.

K-Step Contrastive Examples. A contrastive example C for an execution \mathcal{E} is a subset of features whose alteration can cause the selection of an action not in \mathcal{E}_A . A k -step contrastive example is depicted in Fig. 5: altering the features s_1^1 and s_2^3 may cause action c_2 to be chosen instead of c_1 in the second step. Formally, C is an ordered set of (possibly empty) subsets $C = (C_1, C_2, \dots, C_k)$, such that $C_i \subseteq F$, and for which $\exists x_1, x_2, \dots, x_k \in \mathbb{F}$ such that

$$\left(\bigwedge_{i=1}^{k-1} T(x_i, N(x_i), x_{i+1}) \right) \wedge \left(\bigwedge_{i=1}^k \bigwedge_{j \in F \setminus C_i} (x_i^j = s_i^j) \right) \wedge \left(\bigvee_{i=1}^k N(x_i) \neq a_i \right) \quad (5)$$

Similarly to multi-step explanations, C is a multi-step contrastive example iff the verification query: $\langle P, N, Q \rangle = \langle (F \setminus C_1, F \setminus C_2, \dots, F \setminus C_k) = \mathcal{E}_S, N_{[k]}, Q_{\neg \mathcal{E}_A} \rangle$ is SAT.

IV. COMPUTING FORMAL K-STEP EXPLANATIONS

We now propose four different methods for computing formal k -step explanations, focusing on *minimal* and *minimum* explanations. All four methods use an underlying DNN verifier to check candidate explanations, but differ in how they enumerate different explanation candidates until ultimately converging to an answer. We begin with the more straightforward methods.

Method 1: A Single, K-Sized Step. The first method is to encode the negation of Eq. 4 by unrolling all k steps of the network, as described in Sec. III. This transforms the problem into explaining a non-reactive, single-step system (e.g., a “one-shot” classifier). We can then use any existing abductive explanation algorithm for explaining the unrolled DNN (e.g., [17], [43], [44]).

This method is likely to produce small explanation sets but is extremely inefficient. Encoding $N_{[k]}$ results in an input

space roughly k times the size of any single-step encoding. Such an unrolling for our running example is depicted in Fig. 6. Due to the NP-completeness of DNN verification, this may cause an exponential growth in the verification time of each query. Since finding minimal explanations requires a linear number of queries (and for minimum explanations — a worst-case exponential number), this may cause a substantial increase in runtime.

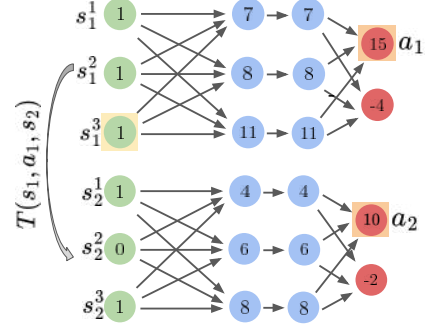


Fig. 6: Finding explanations using a 2-step unrolling.

Method 2: Combining Independent, Single-Step Explanations. Here, we dismantle any k -step execution into k individual steps. Then, we *independently* compute an explanation for each step, using any existing algorithm, and without taking the transition relation into account. Finally, we concatenate these explanations to form a multi-step explanation. Fixing the features of the explanation in each step ensures that the ensuing action remains the same, guaranteeing the soundness of the combined explanation.

The downside of this method is that the resulting E need not be minimal or minimum, even if its constituent E_i explanations are minimal or minimum themselves; see Fig. 7. In this instance, finding a minimum explanation for each step results in the 2-step explanation $(\{s^3\}, \{s^3\})$, which is *not minimal* — even though its components are minimum explanations for their respective steps. The reason for this phenomenon is that this method ignores the transition constraints and information flow across time-steps. This can result in larger and less meaningful explanations, as we later show in Sec. V.

Method 3: Incremental Explanation Enumeration. We now suggest a scheme that takes into consideration the transition

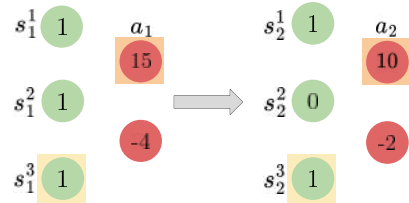


Fig. 7: Explaining each step individually.

constraints between steps (unlike Method 2), but which encodes the verification queries for validating explanations in a more efficient manner than Method 1. The scheme relies on the following lemma:

Lemma 1. *Let $E = (E_1, E_2, \dots, E_k)$ be a k -step explanation for execution \mathcal{E} , and let $1 \leq i \leq k$ such that $\forall j > i$ it holds that $E_j = F$. Let E' be the set obtained by removing a set of features $F' \subseteq E_i$ from E_i , i.e., $E' = (E_1, \dots, E_{i-1}, E_i \setminus F', E_{i+1}, \dots, E_k)$. In this case, fixing the features in E' prevents any changes in the first $i - 1$ actions (a_1, \dots, a_{i-1}) ; and if any of the last $k - i + 1$ actions (a_i, \dots, a_k) change, then a_i must also change.*

A proof appears in the extended version of this paper [15]. The lemma states that “breaking” an explanation E of \mathcal{E} at some step i (by removing features from the i ’th step), given that the features in steps $i + 1, \dots, k$ are fixed, causes a_i to change before any other action. In this scenario, we can determine whether E explains \mathcal{E} using a simplified verification query: we can check whether (E_1, \dots, E_i) explains the first i steps of \mathcal{E} , regardless of steps $i + 1, \dots, k$. If so, then a_i cannot change; and from Lemma 1, no action in \mathcal{E}_A can change, and (E_1, \dots, E_k) is an explanation for \mathcal{E} . Otherwise, E allows an action in \mathcal{E}_A to change, and it does not explain \mathcal{E} . We can leverage this property to efficiently enumerate candidates as part of a search for a minimal/minimum explanation for \mathcal{E} , as explained next.

Finding Minimal Explanations with Method 3. A common approach for finding minimal explanations for a “one-shot” classification instance is via a greedy algorithm, which dispatches a linear number of queries to the underlying verifier [44]. Such an algorithm can start with the explanation set to be the entire feature space, and then iteratively attempt to remove features. If removing a feature allows misclassification, the algorithm keeps it as part of the explanation; otherwise, it removes the feature and continues. A pseudo-code for this approach appears in Alg. 1.

Algorithm 1 Greedy-Minimal-Explanation

Input N (DNN), F (N ’s features), v (values), c (predicted class)

```

1: Explanation  $\leftarrow F$ 
2: for each  $f \in F$  do
3:   if verify  $((\text{Explanation} \setminus \{f\}) = v, N, Q_{-c})$  is UNSAT
   then
4:     Explanation  $\leftarrow \text{Explanation} \setminus \{f\}$ 
5: return Explanation

```

We suggest performing a similar process for explaining \mathcal{E} . We start by fixing all features in all states of \mathcal{E} to their values; i.e., we start with $E = (E_1, \dots, E_k)$ where $E_i = F$ for all i , and then perform the following steps:

First, we iteratively remove individual features from E_1 , each time checking whether the modified E remains an explanation for \mathcal{E} . Since all features in steps $2, \dots, k$ are fixed,

it follows from Lemma 1 that checking whether the modified E explains \mathcal{E} is equivalent to checking whether the modified E_1 explains the selection of a_1 . Thus, we perform a process that is identical to the one in the greedy Alg. 1 for finding a minimal explanation for a “one-shot” classification DNN. At the end of this phase, we are left with $E = (E_1, \dots, E_k)$ where $E_i = F$ for all $i > 1$ and E_1 was reduced by removing features from it. We keep all current features in E fixed for the following steps.

Second, we begin to iteratively remove features from E_2 , each time checking whether the modified E still explains \mathcal{E} . Since the features in steps $3, \dots, k$ are entirely fixed, it suffices (from Lemma 1) to check whether the modified (E_1, E_2) explains the selection of the first two actions (a_1, a_2) of \mathcal{E}_A . This is performed by checking whether

$$(\forall x_1, x_2 \in \mathbb{F}. \quad T(x_1, a_1, x_2) \wedge \bigwedge_{j \in E_1} (x_1^j = s_1^j) \wedge \bigwedge_{j \in E_2} (x_2^j = s_2^j)) \rightarrow N(x_2) = a_2 \quad (6)$$

We do not need to require that $N(x_1) = a_1$ (as in Method 1) — this is guaranteed by Lemma 1. This is significant, because it exempts us from encoding the neural network twice as part of the verification query. We denote the negation of Eq. 6 for validating (E_1, E_2) as: $\langle P, N, Q \rangle = \langle (E_1, E_2) = \mathcal{E}_{S_{[2]}}, N, Q_{-a_2} \rangle$.

Third, we continue this iterative process for all k steps of \mathcal{E} , and find the minimal explanation for each step separately. In step i , for each query we encode i transitions and check whether the modified E still explains the first i steps of \mathcal{E} (by encoding $\langle (E_1, \dots, E_i) = \mathcal{E}_{S_{[i]}}, N, Q_{-a_i} \rangle$), which *does not* require encoding the DNN i times. The correctness of each step follows directly from Lemma 1.

The pseudo-code for this process appears in Alg. 2. The minimality of the resulting explanation holds because removing any feature from this explanation would allow the action in that step to change (since minimality is maintained in each step of the algorithm). An example of the first two iterations of this process on our running example appears in Fig. 8: in the first iteration, we attempt to remove features from the first step, until converging to an explanation E_1 . In the second iteration, while the features in E_1 remain fixed to their values, we encode the constraints of the transition relation $T(s_1, a_1, s_2)$ between the first two steps, and dispatch queries to verify candidate explanations for the second step — until converging to a minimal explanation (E_1, E_2) . In this case, $E_2 = \emptyset$, and $(\{s^3\}, \emptyset)$ is a valid explanation for the 2-step execution, since fixing the value of s_1^3 determines the value of s_2^3 — which forces the selection of a_2 in the second step.

We emphasize that incrementally enumerating candidate explanations for a k -step execution in this way is preferable to simply finding a minimal explanation by encoding verification queries that encompass all k -steps, à la Method 1: (i) in each iteration, we dispatch a verification query involving only a single invocation of the DNN, thus circumventing the linear growth in the network’s size — which causes an exponential worst-case increase in verification times; and (ii) in each

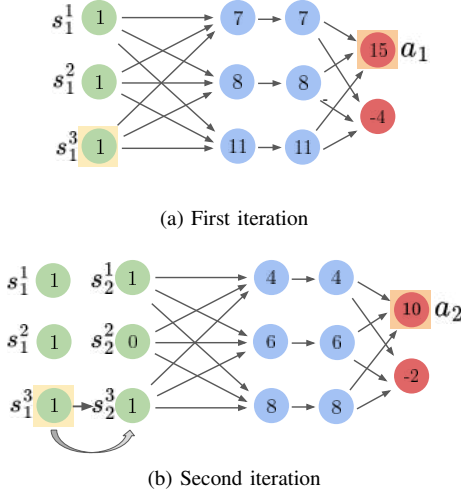


Fig. 8: Running Method 3 for finding minimal explanations, for two iterations.

iteration, we do not need to encode the entire set of k disjuncts (from the negation of Eq. 4), since we only need to validate a_i on the i 'th iteration, and not all actions of \mathcal{E}_A .

Algorithm 2 Incremental-Minimal-Explanation-Enumeration

Input N (DNN), F (N 's features), \mathcal{E} (execution of length k to explain)

- 1: Explanation $\leftarrow (E_1, \dots, E_k)$ where $E_i = F$ for all $1 \leq i \leq k$
 - 2: **for each** $i \in \{1, \dots, k\}$ and $f \in E_i$ **do**
 - 3: **if** verify $((E_1, \dots, E_i \setminus f) = \mathcal{E}_{S_{[i]}}, N, Q_{-a_i})$ is UNSAT **then**
 - 4: $E_i \leftarrow E_i \setminus f$
 - 5: **return** Explanation
-

Finding Minimum Explanations with Method 3. We can also use our proposed enumeration to efficiently find *minimum* explanations, using a recursive approach. In each step $i = 1, \dots, k$, we iterate over all the possible explanations, each time considering a candidate explanation and recursively invoking the procedure for step $i + 1$. In this way, we iterate over all the possible multi-step explanation candidates and can return the smallest one that we find. This process is described in Alg. 3.

Finding a minimum explanation in this manner is superior to using Method 1, for the same reasons noted before. In addition, the exponential blowup here is in the number of explanations in each step, and not in the entire number of features in each step — which is substantially smaller in many cases. Nevertheless, as the method advances through steps, it is expected to be significantly harder to iterate over all the candidate explanations. We discuss more efficient ways for finding global minimum explanations in Method 4.

Algorithm 3 Incremental-Minimum-Explanation-Enumeration

Input N (DNN), F (N 's features), \mathcal{E} (execution to explain)
 \triangleright **Global Variables**

- 1: AllExplanations \leftarrow ALL-EXPLANATION-
 - 2: RECURSIVE-SEARCH($\emptyset, 1$)
 - 3: **return** $E \in$ AllExplanations such that E is with minimum cardinality
-

Algorithm 4 All-Explanation-Recursive-Search

Input E (explanation), i (step number)

- 1: **if** $i = k$ **then**
 - 2: **return** E
 - 3: AllExplanations $\leftarrow \emptyset$
 - 4: **for each** subset F' of F **do**
 - 5: **if** verify $(E \cdot (F') = \mathcal{E}_{S_{[i]}}, N, Q_{-a_i})$ is UNSAT **then**
 - 6: Explanations \leftarrow All-Explanation-
 - 7: Recursive-Search($E \cdot (F')$, $i+1$)
 - 8: AllExplanations \leftarrow AllExplanations \cup Explanations
 - 9: **return** AllExplanations
-

Method 4: Multi-Step Contrastive Example Enumeration.

As mentioned earlier, a common approach for finding minimum explanations is to find all contrastive examples, and then calculate their minimum hitting set (MHS). Because DNNs tend to be sensitive to small input perturbations [87], small contrastive examples are often easy to find, and this can expedite the process significantly [17]. When performing this procedure on a multi-step execution \mathcal{E} , we show that it is possible to enumerate contrastive example candidates in a more efficient manner than simply using the encoding from Method 1.

Lemma 2. Let \mathcal{E} be a k -step execution, and let $C = (C_1, \dots, C_k)$ be a minimal contrastive example for \mathcal{E} ; i.e., altering the features in C can cause at least one action in \mathcal{E}_A to change. Let $1 \leq i \leq k$ denote the index of the first action a_i that can be changed by features in C . It holds that: $C_i \neq \emptyset$; $C_j = \emptyset$ for all $j > i$; and if there exists some $l < i$ such that $C_l \neq \emptyset$, then all sets $\{C_l, C_{l+1}, \dots, C_i\}$ are not empty.

The lemma gives rise to the following scheme. We examine some contrastive example C' of a set of subsequent steps of \mathcal{E} . For simplicity, we discuss the case where $C' = (C'_i)$ involves only a single step i ; but the technique generalizes to subsets of steps, as well. Such a C'_i can be found using a “one-shot” verification query on step i , without encoding the transition relation or unrolling the network. Our goal is to use C' to find many contrastive examples for \mathcal{E} , and use them in computing the MHS. We observe that there are three possible cases:

- 1) $C = (\emptyset, \dots, \emptyset, C'_i, \emptyset, \dots, \emptyset)$ already constitutes a contrastive example for \mathcal{E} . In this case, we say that $C' = (C'_i)$ is an *independent contrastive example*.
- 2) The features in C'_i can cause a skew from \mathcal{E} only

when features from preceding steps $l, \dots, i-1$ (for some $l < i$) are also altered. In this case, we say that C' is a *dependent contrastive example*, and that it depends on steps $l, \dots, i-1$; and together, the features from all these steps form the contrastive example $C = (\emptyset, \dots, \emptyset, C_l, \dots, C_{i-1}, C'_i, \emptyset, \dots, \emptyset)$ for \mathcal{E} .

- 3) C' is a *spurious contrastive example*: the first $i-1$ steps in \mathcal{E} , and the constraints that the transition relation imposes, prevent the features freed by C'_i from causing any action besides a_i to be selected in step i .

Fig. 9 illustrates the three cases. The first case is identical to the one from Fig. 5, where $(\{s^3\})$ is a dependent contrastive example of the second step, which depends on the previous step and is part of a larger contrastive example: $(\{s^3\}, \{s^3\})$. In the second case, assume that T requires that $s_3^1 + s_3^2 \neq 1$ for any feasible transition. Thus, the assignment for s_3^2 which may cause the second action in the sequence to change is not reachable from the previous step, and hence $(\{s^3\})$ is a spurious contrastive example of the second step. In the third case, assume that T allows all transitions, and hence $(\{s^3\})$ is an independent contrastive example for the second step; and so $(\emptyset, \{s^3\})$ is a contrastive example of the entire execution.

It follows from Lemma 2 that one of these three cases must always apply. We next explain how verification can be used to classify each contrastive example of a subset of steps into one of these three categories. If C' is independent, it can be used as-is in computing the MHS; and if it is spurious, it should be ignored. In the case where C' is dependent, our goal is to find all multi-step contrastive examples that contain it, for the purpose of computing the MHS. We next describe a recursive algorithm, termed *reverse incremental enumeration* (RIE), that achieves this.

Reverse Incremental Enumeration. Given a contrastive example C' containing features from a set of subsequent steps of \mathcal{E} , we propose to classify it into one of the three categories by iteratively dispatching queries that check the reachability of C' from the previous steps of the sequence. We execute this procedure by recursively enumerating contrastive examples in previous steps. For simplicity, we assume again that $C' = (C'_i)$ is a single-step contrastive example of step i .

- 1) For checking whether C' is an independent contrastive example of \mathcal{E} , we set $C_{i-1} = \emptyset$ and $C_i = C'_i$, and check whether $C = (C_{i-1}, C_i)$ is a contrastive example for steps $i-1$ and i . This is achieved by dispatching the following query: $\exists x_{i-1}, x_i \in \mathbb{F}$ such that:

$$T(x_{i-1}, N(x_{i-1}), x_i) \wedge \left(\bigwedge_{l=1}^i \bigwedge_{j \in F \setminus C_l} (x_l^j = s_l^j) \right) \wedge (N(x_i) \neq a_i) \quad (7)$$

If the verifier returns SAT, C'_i is independent of step $i-1$, and hence independent of all steps $1, \dots, i-1$. Hence, C' is an independent contrastive example of \mathcal{E} .

- 2) If the query from Eq. 7 returns UNSAT, we now attempt to decide whether C' is dependent. We achieve this through additional verification queries, again setting $C_i = C'_i$, but

now setting C_{i-1} to a *non empty* set of features — once for every possible set of features, separately. We again generate a query using the encoding from Eq. 7, and if the verifier returns SAT it follows that C' is dependent on step $i-1$, and that $C'' = (C_{i-1}, C_i)$ is a contrastive example for steps $i-1$ and i . We recursively continue with this enumeration process, to determine whether C'' is independent, dependent of step $i-2$, or a spurious contrastive example.

- 3) In case the previous phases determine that C' is neither independent nor part of a larger contrastive example, we conclude that it is spurious.

An example of a single reverse incremental enumeration step on a contrastive example C' in our running example is depicted in Fig. 10, and its recursive call is shown in Alg. 5 (Cxps denotes the set of all multi-step contrastive examples containing the initial C').

Algorithm 5 Reverse Incremental Enumeration (RIE)

Input i (starting index), j (reversed index), $C' = (C'_j, \dots, C'_i)$

- 1: **if** $j=1$ **then**
 - 2: **return** $C' \triangleright C'$ is trivially independent of steps $j < 1$
 - 3: **if** $(\emptyset, C'_j, \dots, C'_i)$ is a contrastive example of steps $j-1 \dots i$ **then**
 - 4: **return** $(C_l \mid \forall 1 \leq l \leq j-1, C_l = \emptyset) \cdot C' \triangleright C'$ is independent of step $j-1$
 - 5: Cxps $\leftarrow \emptyset$
 - 6: **for each** subset C_f of F **do**
 - 7: **if** (C_f, C'_j, \dots, C'_i) is a contrastive example of steps $j-1 \dots i$ **then**
 - 8: Cxps \leftarrow Cxps \cup RIE($i, j-1, C_f$) $\triangleright C'$ is dependent of step $j-1$
 - 9: **return** Cxps \triangleright if Cxps is empty, C' is spurious
-

Using reverse incremental enumeration, we can find all multi-step contrastive examples of \mathcal{E} :

- 1) First, we find all contrastive examples for the first step of \mathcal{E} . This is again the same as finding contrastive examples of a “one-shot” classification problem, and can be performed efficiently [17], via Alg. 7. We first enumerate all contrastive examples of size 1 (i.e., contrastive *singletons*); then all contrastive examples of size 2 that do not contain contrastive singletons within them; and then continue this process for all $1 \leq i \leq |F|$ (“skipping” all non-minimal cases).
- 2) Next, we search for all contrastive examples for the second step of \mathcal{E} , in the same manner. We perform a reverse incremental enumeration on each contrastive example found, obtaining all contrastive examples for steps 1 and 2.
- 3) We continue iteratively, each time visiting a new step i and reversely enumerating all contrastive examples that affect steps $1, \dots, i$. We stop when we reach the final step, $i = k$.

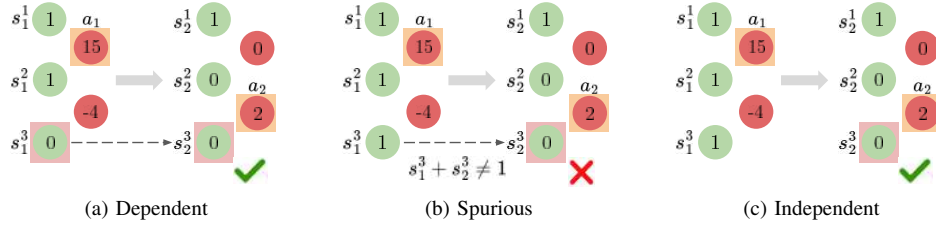


Fig. 9: $(\{s^3\})$ as a dependent, spurious and independent contrastive example.

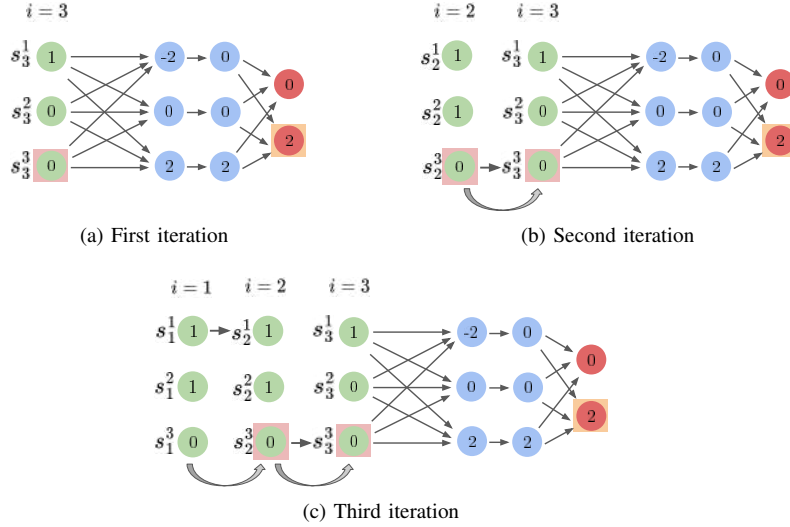


Fig. 10: An illustration of reverse incremental enumeration. We start with a single-step contrastive example, $C'_3 = \{s^3\}$ for the third step of the execution. In the second iteration, we find that (C'_3) is dependent on the previous step, and that $(\{s^3\}, \{s^3\})$ constitutes a contrastive example for steps 2 and 3. In the third iteration, $(\{s^3\}, \{s^3\})$ is found to be independent of the first step, and hence $(\emptyset, \{s^3\}, \{s^3\})$ is a contrastive example for \mathcal{E} .

The full enumeration process for finding all contrastive examples of \mathcal{E} is described fully in Alg. 6, which invokes Alg. 7.

Algorithm 6 Enumerate-All-Cxps

Input N (DNN), F (N 's features), \mathcal{E} (execution to explain)

▷ **Global Variables**

- 1: $\text{Cxps} \leftarrow \emptyset$
 - 2: **for each** $i \in \{1, \dots, k\}$ **do**
 - 3: $\text{CxpCandidates} \leftarrow \text{ENUMERATE-ALL-CXPS-IN-}$
 - 4: $\text{SINGLE-STEP}(i)$
 - 5: **for each** $\text{Cxp} \in \text{CxpCandidates}$ **do**
 - 6: $\text{Cxps} \leftarrow \text{Cxps} \cup \text{RIE}((\text{Cxp}), i, i)$
 - 7: **return** Cxps
-

We also make the following observation: we can further expedite the enumeration process by discarding sets that contain contrastive examples within them since we are specifically searching for minimal contrastive examples. For instance, in the given example in Fig. 10, if we find $(\emptyset, s^1, \emptyset)$ as a contrastive example for the entire multi-step instance, we no longer need to consider sets in step 2 that contain s^1 when

Algorithm 7 Enumerate-All-Cxps-In-Single-Step

Input N (DNN), F (N 's features), \mathcal{E} (execution to explain), i (step number)

- 1: $\text{Cxps} \leftarrow \emptyset$ ▷ denotes the set of all contrastive examples
 - 2: **for each** $1 \leq i \leq |F|$ **do**
 - 3: **for each** subset c of F of length i not containing sets from Cxps **do**
 - 4: **if** $\text{verify}(F \setminus c = s_i, N, Q_{-a_i})$ is SAT **then**
 - 5: $\text{Cxps} \leftarrow \text{Cxps} \cup c$
 - 6: **return** Cxps
-

iterating in reverse from step 3 to step 2. Our evaluation shows that this approach can significantly improve performance as the increasing number of contrastive examples found in previous steps greatly reduces the search space.

Of course, our approach's worst-case complexity is still exponential in the number of steps, k , because each dependent contrastive example requires a recursive call that potentially enumerates all contrastive examples for the previous step. However, the number of recursive iterations is limited by

the dependency between steps. For instance, if contrastive examples in step i are only dependent on step $i - 1$ and not on step $i - 2$, the recursive iterations will be limited to 2. Additionally, skipping the verification of candidates containing contrastive examples found in previous steps can also significantly reduce runtime.

V. EVALUATION

Implementation and Setup. We created a proof-of-concept implementation of all aforementioned approaches and benchmarks [16]. To search for explanations, our tool [16] dispatches verification queries using a backend DNN verifier (we use *Marabou* [52], previously employed in additional studies [2]–[7], [21], [75], although other engines may also be used). The queries encode the architecture of the DNN in question, the transition constraints between consecutive steps of the reactive system, and the candidate explanation or contrastive example being checked. Calculating the MHS, when relevant, was done using RC-2, a MaxSAT-based tool of the PySat toolkit [42].

Benchmarks. We trained DRL agents for two well-known reactive system benchmarks: GridWorld [88] and TurtleBot [89] (see Fig. 11). GridWorld involves an agent moving in a 2D grid, while TurtleBot is a real-world robotic navigation platform. These benchmarks have been extensively studied in the DRL literature. GridWorld has 8 input features per state, including agent coordinates, target coordinates, and sensor readings for obstacle detection. The agent can take 4 possible actions: UP, DOWN, LEFT, or RIGHT. TurtleBot has 9 input features per state, including lidar sensor readings, target distance, and target angle. The agent has 3 possible actions: LEFT, RIGHT, or FORWARD. We trained our DRL agents with the state-of-the-art PPO algorithm [79]. Additional details appear in the extended version of this paper [15].

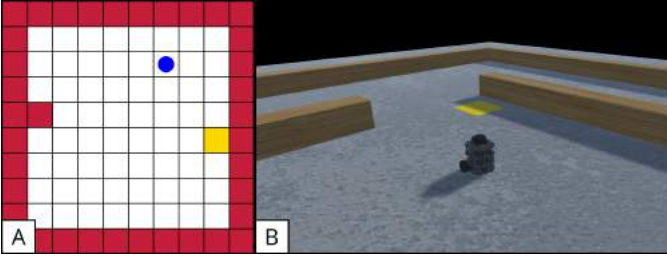


Fig. 11: Benchmarks: (A) GridWorld; and (B) TurtleBot.

Generating Executions. We generated 200 unique multi-step executions of our two benchmarks: 100 GridWorld executions (using 10 agents, each producing 10 unique executions of lengths $6 \leq k \leq 14$), and 100 TurtleBot executions (using 100 agents, each producing a single execution of length $6 \leq k \leq 8$). Next, from each k -step execution, we generated k unique sub-executions, each representing the first i steps of the original execution ($1 \leq i \leq k$). This resulted in a total of 931 GridWorld executions and 647 Turtlebot executions. We used

TABLE I: *GridWorld*: columns from left to right: experiment type, method name (and number), time and size of returned explanation (out of experiments that terminated), and the percent of solved instances (the rest timed out). The bold highlighting indicates the method that generated the explanation with the optimal size.

setting	experiment	time (s)	size			solved (%)
		avg.	min	avg.	max	
minimal (local)	one-shot (1)	304	5	33	112	98
	independent (2)	1	5	34	97	99.9
	incremental (3)	1	5	18	78	99.7
minimum (global)	one-shot (1)	405	5	14	32	29.8
	independent (2)	4	5	35	98	98.3
	incremental (3)	1,396	5	7	9	17.9
	reversed (4)	39	5	7	16	99.7

these executions to assess the different methods for finding minimal and minimum explanations. Each experiment ran with a timeout value of $3 \cdot i$ hours, where i is the execution’s length.

Experiments. We begin by comparing the performance of the four methods discussed in Sec. IV: (i) encoding the entire network as a “one-shot” query; (ii) computing individual explanations for each step; (iii) incrementally enumerating explanations; and (iv) reversely enumerating contrastive examples and calculating their MHS. We note that we use Methods 1–3 to generate both minimal and minimum explanations, whereas Method 4 is only used to generate minimum explanations. To generate minimum explanations using the “one-shot” encodings of Methods 1 and 2, we use the state-of-the-art approach of Ignatiev et al. [44]. We use two common criteria for comparison [17], [43], [44]: the *size* of the generated explanations (small explanations tend to be more meaningful), and the overall runtime and timeout ratios.

Results. Results for the GridWorld benchmark are presented in Table I. These results clearly indicate that Method 2 (generating explanations in independent steps) was significantly faster in all experiments, but generated drastically larger explanations — about two times larger when searching for a *minimal* explanation, and about five times larger for a *minimum* explanation, on average. This is not surprising; as noted earlier, the explanations produced by such an approach do not take the transition constraints into account, and hence, may be quite large. In addition, we note again that this approach does not guarantee the minimality of the combined explanation, even when combining minimal/minimum explanations for each step. The corresponding results for TurtleBot appear in our extended paper [15], and also demonstrate similar outcomes.

When comparing the three approaches that can guarantee minimal explanations, the incremental enumeration approach (Method 3) is clearly more efficient than the “one-shot” scheme (running for about 1 second compared to above 5 minutes, on average, across all solved instances), as depicted in Fig. 12. For the minimum explanation comparison, the results show that the reversed-enumeration-based strategy (Method 4) ran significantly faster than all other methods that can

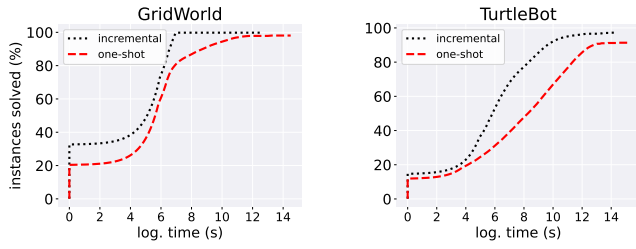


Fig. 12: *Minimal explanation*: number of solved instances depending on (accumulative) time, for the methods that guarantee minimality.

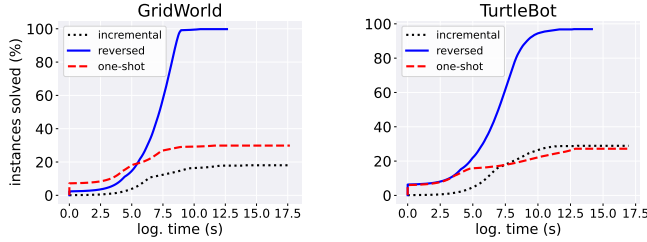


Fig. 13: *Minimum explanation*: number of solved instances depending on (accumulative) time, for the methods that guarantee minimality.

find guaranteed minimum explanations: on average, it ran for 39 seconds, while the other methods ran for more than 6 and 23 minutes. In addition, out of all methods guaranteed to produce a minimum explanation, experiments that ran with the “reversed” strategy hit significantly fewer timeouts. The “reversed” strategy outperforms the competing methods significantly, on both benchmarks (see Fig. 13).

Next, we analyzed the strategies at a higher resolution — focusing on a *step-wise* level comparison, i.e., on analyzing how the length of the execution affected runtime. The results (see Figs. 17-20 in the extended version of this paper [15]) demonstrate the drastic performance gain of our “reversed” strategy as k increases: this strategy can efficiently find explanations for longer executions, while the competing “one-shot” strategy fails. This again is not surprising: when dealing with large numbers of steps, the transition function, the unrolling of the network, and the underlying enumeration scheme become more taxing on the underlying verifier. A full analysis of both benchmarks, and all explanation types, appears in [15].

Explanation Example. We provide a visual example of an instance from our GridWorld experiment identified as a minimum explanation. The results (depicted in Fig. 14) include a minimum explanation for an execution of 8 steps. They show the following meaningful insight: fixing part of the agent’s location sensors at the initial step, and a single sensor in the sixth step, is sufficient for forcing the agent to move along the original path, regardless of any other sensor reading.

Comparison to Heuristic XAI Methods. We also compared our results to popular, non-verification-based, heuristic XAI

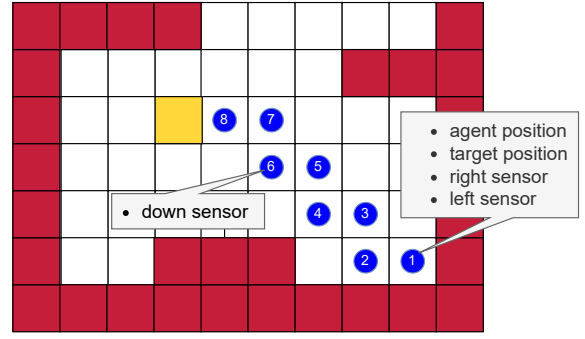


Fig. 14: *GridWorld*: a 5-sized explanation for an 8-step execution. The steps are numbered (in blue circles), the target is the yellow square, and the obstacles are depicted in red.

methods. Although these methods proved scalable, they often returned unsound explanations when compared to our approach. For additional details, see [15].

VI. RELATED WORK

This work joins recent efforts on utilizing formal verification to explain the decisions of ML models [17], [25], [44], [55], [83], [84], [93]. Prior studies primarily focused on formally explaining *classification* over various domains [17], [44], [44], [45], [55], [93] or specific model types [35], [40], [46], [48], [65], while others explored alternative ways of defining explanations over classification tasks [8], [34], [49], [55], [68], [73], [90], [93].

Methods closer to our own have focused on formally explaining DNNs [17], [37], [44], [55], [93], where the problem is known to be complex [44], [60]. This work relies on the rapid development of DNN verification [1], [12], [13], [27], [30], [51], [53], [94]. There has also been ample work on heuristic XAI [31], [62], [77], [78], [80], including approaches for explaining the decisions of reinforcement-learning-based reactive systems (XRL) [32], [50], [64], [74]. However, these methods do not provide formal guarantees.

VII. CONCLUSION

Although DNNs are used extensively within reactive systems, they remain “black-box” models, uninterpretable to humans. We seek to mitigate this concern by producing formal explanations for executions of reactive systems. As far as we are aware, we are the first to provide a formal basis of explanations in this context, and to suggest methods for efficiently producing such explanations — significantly outperforming the competing approaches. We also note that our approach is agnostic to the type of reactive system, and can be generalized beyond DRL systems, to any k -step reactive DNN system (including RNNs, LSTMs, GRUs, etc.). Moving forward, a main extension could be scaling our method, beyond the simple DRLs evaluated here, to larger systems of higher complexity. Another interesting extension could include evaluating the attribution of the hidden-layer features, rather than just the input features.

Acknowledgments. The work of Bassan, Amir, Refaeli, and Katz was partially supported by the Israel Science Foundation (grant number 683/18). The work of Amir was supported by a scholarship from the Clore Israel Foundation. The work of Corsi was partially supported by the “Dipartimenti di Eccellenza 2018-2022” project, funded by the Italian Ministry of Education, Universities, and Research (MIUR).

REFERENCES

- [1] A. Albarghouthi. *Introduction to Neural Network Verification*. verified-deeplearning.com, 2021.
- [2] G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Verifying Learning-Based Robotic Navigation Systems. In *Proc. 29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 607–627, 2023.
- [3] G. Amir, Z. Freund, G. Katz, E. Mandelbaum, and I. Refaeli. veriFIRE: Verifying an Industrial, Learning-Based Wildfire Detection System. In *Proc. 25th Int. Symposium on Formal Methods (FM)*, pages 648–656, 2023.
- [4] G. Amir, O. Maayan, T. Zelazny, G. Katz, and M. Schapira. Verifying Generalization in Deep Learning. In *Proc. 35th Int. Conf. on Computer Aided Verification (CAV)*, pages 438–455, 2023.
- [5] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.
- [6] G. Amir, H. Wu, C. Barrett, and G. Katz. An SMT-Based Approach for Verifying Binarized Neural Networks. In *Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 203–222, 2021.
- [7] G. Amir, T. Zelazny, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 27–37, 2022.
- [8] G. Anderson, S. Pailoor, I. Dillig, and S. Chaudhuri. Optimization and Abstraction: a Synergistic Approach for Analyzing Neural Network Robustness. In *Proc. 40th ACM SIGPLAN Conf. on Programming Languages Design and Implementations (PLDI)*, pages 731–744, 2019.
- [9] C. Angermueller, T. Pärnamaa, L. Parts, and O. Stegle. Deep Learning for Computational Biology. *Molecular Systems Biology*, 12(7):878, 2016.
- [10] J. Angwin, J. Larson, S. Mattu, and L. Kirchner. Machine Bias. *Ethics of Data and Analytics*, pages 254–264, 2016.
- [11] S. Aradi. Survey of Deep Reinforcement Learning for Motion Planning of Autonomous Vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 2020.
- [12] G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Könighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.
- [13] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks and its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1249–1264, 2019.
- [14] P. Barceló, M. Monet, J. Pérez, and B. Subercaseaux. Model Interpretability through the Lens of Computational Complexity. In *Proc. 33rd Conf. on Neural Information Processing Systems (NeurIPS)*, 2020.
- [15] S. Bassan, G. Amir, D. Corsi, I. Refaeli, and G. Katz. Formally Explaining Neural Networks within Reactive Systems, 2023. Technical Report. <https://arxiv.org/abs/2308.00143>.
- [16] S. Bassan, G. Amir, D. Corsi, I. Refaeli, and G. Katz. Formally Explaining Neural Networks within Reactive Systems: Artifact, 2023. <https://zenodo.org/record/8197762>.
- [17] S. Bassan and G. Katz. Towards Formal Approximated Minimal Explanations of Neural Networks. In *Proc. 29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 187–207, 2023.
- [18] L. Brunke, M. Greeff, A. Hall, Z. Yuan, S. Zhou, J. Panerati, and A. Schoellig. Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning. *Annual Review of Control, Robotics, and Autonomous Systems*, 5:411–444, 2022.
- [19] CACM Staff. A Case Against Mission-Critical Applications of Machine Learning. *Communications of the ACM*, 62(8):9–9, 2019.
- [20] O. Camburu, E. Giunchiglia, J. Foerster, T. Lukasiewicz, and P. Blunsom. Can I Trust the Explainer? Verifying Post-Hoc Explanatory Methods, 2019. Technical Report. <http://arxiv.org/abs/1910.02065>.
- [21] M. Casadio, E. Komendantskaya, M. Daggit, W. Kokke, G. Katz, G. Amir, and I. Refaeli. Neural Network Robustness as a Verification Property: A Principled Case Study. In *Proc. 34th Int. Conf. on Computer Aided Verification (CAV)*, pages 219–231, 2022.
- [22] D. Corsi, E. Marchesini, and A. Farinelli. Formal Verification of Neural Networks for Safety-Critical Tasks in Deep Reinforcement Learning. In *Proc. 37th Int. Conf. on Uncertainty in Artificial Intelligence (UAI)*, 2021.
- [23] D. Corsi, L. Marzari, A. Pore, A. Farinelli, A. Casals, P. Fiorini, and D. Dall’Alba. Constrained Reinforcement Learning and Formal Verification for Safe Colonoscopy Navigation. In *Proc. IEEE Int. Conf. on Intelligent Robots and Systems (IROS)*, 2023.
- [24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding, 2018. Technical Report. <https://arxiv.org/abs/1810.04805>.
- [25] T. Fel, M. Ducoffe, D. Vigouroux, R. Cadène, M. Capelle, C. Nicodème, and T. Serre. Don’t Lie to Me! Robust and Efficient Explainability with Verified Perturbation Analysis, 2022. Technical Report. <https://arxiv.org/abs/2202.07728>.
- [26] T. Gehr, M. Mirman, D. Drachler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
- [27] C. Geng, N. Le, X. Xu, Z. Wang, A. Gurfinkel, and X. Si. Toward Reliable Neural Specifications, 2022. Technical Report. <https://arxiv.org/abs/2210.16114>.
- [28] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. <http://arxiv.org/abs/1412.6572>.
- [29] D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-Driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th. Int. Symp. on on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
- [30] D. Guidotti, L. Pulina, and A. Tacchella. pyNeVer: A Framework for Learning and Verification of Neural Networks. In *Proc. 19th. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 357–363, 2021.
- [31] D. Gunning, M. Stefik, J. Choi, T. Miller, S. Stumpf, and G.-Z. Yang. XAI—Explainable Artificial Intelligence. *Science Robotics*, 4(37):eaay7120, 2019.
- [32] A. Heuillet, F. Couthouis, and N. Díaz-Rodríguez. Explainability in Deep Reinforcement Learning. *Knowledge-Based Systems*, 214:106685, 2021.
- [33] R. Hoffman, S. Mueller, G. Klein, and J. Litman. Metrics for Explainable AI: Challenges and Prospects, 2018. Technical Report. <https://arxiv.org/abs/1812.04608>.
- [34] X. Huang, M. Cooper, A. Morgado, J. Planes, and J. Marques-Silva. Feature Necessity & Relevancy in ML Classifier Explanations. In *Proc. 29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 167–186, 2023.
- [35] X. Huang, Y. Izza, A. Ignatiev, and J. Marques-Silva. On Efficiently Explaining Graph-Based Classifiers, 2021. Technical Report. <https://arxiv.org/abs/2106.01350>.
- [36] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
- [37] X. Huang and J. Marques-Silva. From Robustness to Explainability and Back Again, 2023. Technical Report. <https://arxiv.org/abs/2306.03048>.
- [38] IBM. The CPLEX optimizer, 2018.
- [39] A. Ignatiev. Towards Trustable Explainable AI. In *Proc. 29th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 5154–5158, 2020.
- [40] A. Ignatiev and J. Marques-Silva. SAT-Based Rigorous Explanations for Decision Lists. In *Proc. 24th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, pages 251–269, 2021.
- [41] A. Ignatiev, A. Morgado, and J. Marques-Silva. Propositional Abduction with Implicit Hitting Sets, 2016. Technical Report. <http://arxiv.org/abs/1604.08229>.
- [42] A. Ignatiev, A. Morgado, and J. Marques-Silva. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *Proc. 21st Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, pages 428–437, 2018.

- [43] A. Ignatiev, N. Narodytska, N. Asher, and J. Marques-Silva. From Contrastive to Abductive Explanations and Back Again. In *Proc. 19th Int. Conf. of the Italian Association for Artificial Intelligence (AlxIA)*, pages 335–355, 2020.
- [44] A. Ignatiev, N. Narodytska, and J. Marques-Silva. Abduction-Based Explanations for Machine Learning Models. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, pages 1511–1519, 2019.
- [45] A. Ignatiev, N. Narodytska, and J. Marques-Silva. On Validating, Repairing and Refining Heuristic ML Explanations, 2019. Technical Report. <http://arxiv.org/abs/1907.02509>.
- [46] A. Ignatiev, F. Pereira, N. Narodytska, and J. Marques-Silva. A SAT-Based Approach to Learn Explainable Decision Sets. In *Proc. 9th Int. Joint Conf. on Automated Reasoning (IJCAR)*, pages 627–645, 2018.
- [47] A. Ignatiev, A. Previti, M. Liffiton, and J. Marques-Silva. Smallest MUS Extraction with Minimal Hitting Set Dualization. In *Proc. 21st Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 173–182, 2015.
- [48] Y. Izza, A. Ignatiev, and J. Marques-Silva. On Explaining Decision Trees, 2020. Technical Report. <http://arxiv.org/abs/2010.11034>.
- [49] Y. Izza, A. Ignatiev, N. Narodytska, M. Cooper, and J. Marques-Silva. Efficient Explanations with Relevant Sets, 2021. Technical Report. <http://arxiv.org/abs/2106.00546>.
- [50] Z. Juozapaitis, A. Koul, A. Fern, M. Erwig, and F. Doshi-Velez. Explainable Reinforcement Learning via Reward Decomposition. In *Proc. IJCAI/ECAP Workshop on Explainable Artificial Intelligence*, 2019.
- [51] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
- [52] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.
- [53] B. Könighofer, F. Lorber, N. Jansen, and R. Bloem. Shield Synthesis for Reinforcement Learning. In *Proc. Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 290–306, 2020.
- [54] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Proc. 30rd Conf. on Neural Information Processing Systems (NeurIPS)*, 2017.
- [55] E. La Malfa, A. Zbrzezny, R. Michémore, N. Paoletti, and M. Kwiatkowska. On Guaranteed Optimal Robust Explanations for NLP Models, 2021. Technical Report. <https://arxiv.org/abs/2105.03640>.
- [56] Y. LeCun, Y. Bengio, and G. Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.
- [57] A. Lekharu, K. Moulai, A. Sur, and A. Sarkar. Deep Learning Based Prediction Model for Adaptive Video Streaming. In *Proc. Int. Conf. on Communication Systems & NETWORKS (COMSNETS)*, pages 152–159, 2020.
- [58] C. Li and F. Many. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, pages 903–927. IOS Press, 2021.
- [59] Y. Li. Deep Reinforcement Learning: An Overview, 2017. Technical Report. <http://arxiv.org/abs/1701.07274>.
- [60] P. Liberatore. Redundancy in Logic I: CNF Propositional Formulae. *Artificial Intelligence*, 163(2):203–232, 2005.
- [61] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. Algorithms for Verifying Deep Neural Networks, 2020. Technical Report. <http://arxiv.org/abs/1903.06758>.
- [62] S. Lundberg and S.-I. Lee. A Unified Approach to Interpreting Model Predictions. In *Proc. 31st Conf. on Neural Information Processing Systems (NeurIPS)*, 2017.
- [63] N. Luong, D. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. Kim. Applications of Deep Reinforcement Learning in Communications and Networking: A Survey. *IEEE Communications Surveys & Tutorials*, 21(4):3133–3174, 2019.
- [64] P. Madumal, T. Miller, L. Sonenberg, and F. Vetere. Explainable Reinforcement Learning through a Causal Lens. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 2493–2500, 2020.
- [65] J. Marques-Silva, T. Gerspacher, M. Cooper, A. Ignatiev, and N. Narodytska. Explaining Naive Bayes and Other Linear Classifiers with Polynomial Time and Delay. In *Proc. 33rd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 20590–20600, 2020.
- [66] J. Marques-Silva and A. Ignatiev. Delivering Trustworthy AI through formal XAI. In *Proc. 36th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 3806–3814, 2022.
- [67] L. Marzari, D. Corsi, F. Cicalese, and A. Farinelli. The #DNN-Verification Problem: Counting Unsafe Inputs for Deep Neural Networks. In *Proc. 32nd Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2023.
- [68] K. McMillan. Bayesian Interpolants as Explanations for Neural Inferences, 2020. Technical Report. <https://arxiv.org/abs/2004.04198>.
- [69] M. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations, 2021. Technical Report. <https://arxiv.org/abs/2103.03638>.
- [70] T. Okudono, M. Waga, T. Sekiyama, and I. Hasuo. Weighted Automata Extraction from Recurrent Neural Networks via Regression on State Spaces. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 5037–5044, 2020.
- [71] E. Polgreen, R. Abboud, and D. Kroening. Counterexample Guided Neural Synthesis, 2020. Technical Report. <https://arxiv.org/abs/2001.09245>.
- [72] A. Pore, D. Corsi, E. Marchesini, D. Dall’Alba, A. Casals, A. Farinelli, and P. Fiorini. Safe Reinforcement Learning using Formal Verification for Tissue Retraction in Autonomous Robotic-Assisted Surgery. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2021.
- [73] P. Prabhakar and Z. Afzal. Abstraction Based Output Range Analysis for Neural Networks, 2020. Technical Report. <https://arxiv.org/abs/2007.09527>.
- [74] E. Puiutta and E. Veith. Explainable Reinforcement Learning: A Survey. In *Proc. Int. Cross-Domain Conf. for Machine Learning and Knowledge Extraction (CD-MAKE)*, pages 77–95, 2020.
- [75] I. Refaeli and G. Katz. Minimal Multi-Layer Modifications of Deep Neural Networks. In *Proc. 5th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS)*, 2022.
- [76] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.
- [77] M. Ribeiro, S. Singh, and C. Guestrin. “Why should I Trust You?” Explaining the Predictions of any Classifier. In *Proc. 22nd Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1135–1144, 2016.
- [78] M. Ribeiro, S. Singh, and C. Guestrin. Anchors: High-Precision Model-Agnostic Explanations. In *Proc. 32nd AAAI Conf. on Artificial Intelligence (AAAI)*, 2018.
- [79] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms, 2017. Technical Report. <http://arxiv.org/abs/1707.06347>.
- [80] R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-Cam: Visual Explanations from Deep Networks via Gradient-Based Localization. In *Proc. 20th IEEE Int. Conf. on Computer Vision (ICCV)*, pages 618–626, 2017.
- [81] S. Seshia, A. Desai, T. Dreossi, D. Fremont, S. Ghosh, E. Kim, S. Shivakumar, M. Vazquez-Chanlatte, and X. Yue. Formal Specification for Deep Neural Networks. In *Proc. 16th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 20–34, 2018.
- [82] M. Shahbaz and R. Groz. Inferring Mealy Machines. In *Proc. Conf. on Formal Methods (FM)*, pages 207–222, 2009.
- [83] W. Shi, A. Shih, A. Darwiche, and A. Choi. On Tractable Representations of Binary Neural Networks, 2020. Technical Report. <http://arxiv.org/abs/2004.02082>.
- [84] A. Shih, A. Choi, and A. Darwiche. A Symbolic Approach to Explaining Bayesian Network Classifiers, 2018. Technical Report. <http://arxiv.org/abs/1805.03364>.
- [85] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the Game of Go Without Human Knowledge. *nature*, 550(7676):354–359, 2017.
- [86] M. Sotoudeh and A. Thakur. Correcting Deep Neural Networks with Small, Generalizing Patches. In *Proc. Workshop on Safety and Robustness in Decision Making*, 2019.
- [87] J. Su, D. Vargas, and K. Sakurai. One Pixel Attack for Fooling Deep Neural Networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019.
- [88] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

- [89] L. Tai, G. Paolo, and M. Liu. Virtual-to-Real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2017.
- [90] S. Waeldechen, J. Macdonald, S. Hauch, and G. Kutyniok. The Computational Complexity of Understanding Binary Classifier Decisions. *Journal of Artificial Intelligence Research*, 70:351–387, 2021.
- [91] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter. Beta-Crown: Efficient Bound Propagation with Per-Neuron Split Constraints for Neural Network Robustness Verification. In *Proc. 34th Conf. on Neural Information Processing Systems (NeurIPS)*, volume 34, pages 29909–29921, 2021.
- [92] H. Wu, A. Ozdemir, A. Zeljić, A. Irfan, K. Julian, D. Gopinath, S. Fouladi, G. Katz, C. Păsăreanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 128–137, 2020.
- [93] M. Wu, H. Wu, and C. Barrett. VeriX: Towards Verified Explainability of Deep Neural Networks, 2022. Technical Report. <https://arxiv.org/abs/2212.01051>.
- [94] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.
- [95] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An End-to-End Automatic Cloud Database Tuning System using Deep Reinforcement Learning. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 415–432, 2019.
- [96] Z. Zhou and L. Sun. Metamorphic Testing of Driverless Cars. *Communications of the ACM*, 62(3):61–67, 2019.

Lightweight Online Learning for Sets of Related Problems in Automated Reasoning

Haoze Wu
Stanford University
Stanford, CA, USA
haozewu@stanford.edu

Christopher Hahn
Stanford University
Stanford, CA, USA
hahn@cs.stanford.edu

Florian Lonsing
Unaffiliated
Linz, Austria
fml@florianlonsing.com

Makai Mann
MIT Lincoln Laboratory
Lexington, MA, USA
makai.mann@ll.mit.edu

Raghuram Ramanujan
Davidson College
Davidson, NC, USA
raramanujan@davidson.edu

Clark Barrett
Stanford University
Stanford, CA, USA
barrett@cs.stanford.edu

Abstract—We present Self-Driven Strategy Learning (SDSL), a lightweight online learning methodology for automated reasoning tasks that involve solving a set of related problems. SDSL does not require offline training, but instead automatically constructs a dataset while solving earlier problems. It fits a machine learning model to this data which is then used to adjust the solving strategy for later problems. We formally define the approach as a set of abstract transition rules. We describe a concrete instance of the SDSL calculus which uses conditional sampling for generating data and random forests as the underlying machine learning model. We implement the approach on top of the KISSAT solver and show that the combination of KISSAT+SDSL certifies larger bounds and finds more counter-examples than other state-of-the-art bounded model checking approaches on benchmarks obtained from the latest Hardware Model Checking Competition.

I. INTRODUCTION

Many automated reasoning tasks involve solving a set of related problems that share common structure. For example, in Bounded Model Checking [1], [2], one repeatedly checks deeper and deeper unrolls of a transition system for a property violation. In iterative (e.g., counter-example-guided) abstraction refinement [3], one verifies a condition on an increasingly precise model of a system. And in symbolic execution [4], one analyzes the possible outcomes of a program on symbolic inputs by incrementally adding path conditions. Often, a fixed, predetermined high-level solving strategy (e.g., the choice of a solver and its parameter settings) is used in this iterative solving process. However, given the structural similarity within the set of problems, a natural question is: *can we leverage information gathered while solving earlier problems to adjust the solving strategy for later problems on the fly?*

Adapting high-level solving strategies for particular problem distributions, a practice often termed *meta-algorithmic design* [5], is already a well-established technique. Automated configuration techniques [6], which optimize an algorithm's performance on a given set of problems, are widely used among practitioners. Per-instance algorithm selection techniques (e.g., SATzilla [7]) train machine learning models to predict a suitable strategy for a given problem based on its

structural characteristics. More recently, attempts to improve constraint solving using deep learning also generally follow the paradigm of choosing a particular problem distribution (which can be either broad, such as main-track benchmarks from SAT competitions [8], or narrow, such as graph coloring problems [9]), gathering training data using instances in that distribution, and learning a strategy over the data.

A shared, and arguably undesirable, trait of the aforementioned approaches is that they all involve an *offline phase*, in which significant time and (often manual) effort are required to obtain an optimized solving strategy that can be used on new unseen problems. While the cost of the offline phase might be justified by the potential performance gain in the long run, the very distinction between an offline phase and an online phase already makes the reasoning less *automated*.

Our first observation is that for an automated reasoning procedure whose execution involves solving a set S of related problems, it is possible to move the meta-algorithmic design online, as part of the solving, by narrowing the scope of *problem distribution* all the way down to S itself. More concretely, we propose to solve some of the problems in S not just once, but multiple times, each time using a different solving strategy from a space of candidates. The strategies used for solving later problems are selected based on information recorded during the multiple runs (e.g., using lightweight machine learning techniques). We present this general method, which we term **Self-Driven Strategy Learning (SDSL)**, as a set of transition rules, which can be used to model different ways of carrying out on-the-fly meta-algorithmic design.

Though there are many possible ways to instantiate SDSL, we focus on a strategy space consisting of one fixed solver whose parameters are allowed to vary. One obvious method for exploring this strategy space involves choosing the first few problems, optimizing the parameter settings for them with a standard tuning procedure, and then using the optimized strategy for future problems. However, a drawback of this approach is that it only operates on a fixed set of problems and cannot explicitly take into account possible relationships

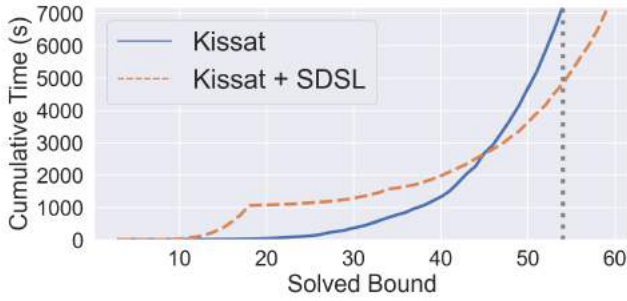


Fig. 1. Executions of Bounded Model Checking with and without SDSL on a hardware model checking benchmark (`arb.n2.w128.d64`). The trajectories show the cumulative wall-clock time required to certify a bound.

between this set of problems and later problem instances.

To allow for such flexibility, our second observation is that a tuning procedure can be viewed, not as a way to select a specific solving strategy, but instead as a means of creating a dataset, where each data point is a pair consisting of a particular solving strategy and a particular problem in the problem set. A machine learning model is trained on this dataset to predict the effect of a given solving strategy on a given problem. The model can then be used as an oracle to select the solving strategy for future problems.

We apply our methodology to a case study of Bounded Model Checking (BMC) problems. We study different SDSL instantiations and compare against existing model checkers. On satisfiable and unsolved bitvector benchmarks from the latest Hardware Model Checking Competition [10], our approach consistently boosts the performance of a BMC-procedure built on top of the KISSAT SAT solver [11]. Additionally, it compares favorably against state-of-the-art open-source model checkers AVR [12] and PONO [13], contributing several unique solutions and speeding up many more. A preview on a single benchmark is shown in Fig. 1. We see that SDSL invests time learning a good solving strategy in the beginning, which results in better performance when solving later problems.

We summarize our main contributions as follows:

- 1) we propose to move meta-algorithmic design online as part of solving a set of related problems;
- 2) we propose a general methodology called Self-Driven Strategy Learning and present it formally as a set of transition rules; and
- 3) we implement our approach and apply it to Bounded Model Checking problems, where it shows significant improvement over other state-of-the-art approaches.

The rest of the paper is organized as follows. After a discussion of related work in Sec. II, we first define a basic calculus for iteratively solving a set of related problems in Sec. III. Next, SDSL is presented as an extension of this calculus with additional rules for data collection, learning, and strategy updates in Sec. IV. We explore the design space of SDSL in Sec. V, discussing how to sample training data and which machine learning models to use. In Sec. VI, we describe in detail the instantiation of SDSL for Bounded Model

Checking. In Sec. VII we present experimental results on Bounded Model Checking problems, and finally, we conclude with an account of current limitations and future directions in Sec. VIII.

II. RELATED WORK

Our approach is inspired and informed by several existing lines of work.

Incremental solving: A well-established paradigm for exploiting structural similarity is *incremental solving* [14] in which each new query to a solver can be made by modifying the most recent formulas asserted in the previous query without resetting the solver. SDSL is an orthogonal approach for leveraging structural similarity and may be preferable in cases where incremental solving is not beneficial or not supported.

In principle, the two can be combined. A straightforward way would be to switch to incremental solving mode after fixing a solving strategy. A tighter combination would require updating strategies *between* incremental invocations, something that current solvers typically do not allow. In case one wants to both switch solvers on the fly and leverage incrementality, proof-transfer techniques such as solver state migration [15] are likely needed. For our particular BMC case study, we found that the direct use of an incremental SAT/SMT solver has mixed effects on performance (see the extended version [16] of the paper). This suggests that it might be worth revisiting BMC-specific incremental solving techniques such as conflict clause shifting [17] before investigating the interplay between SDSL and incremental solving in BMC, which we leave for future work.

Automated Configuration: Our work is motivated by the success of offline meta-algorithmic design approaches such as automated configuration [6], [18], [19] and per-instance algorithm selection [7], [20], [21]. Automated configuration focuses on finding (near) optimal parameter settings of an algorithm for a fixed set of problems, using either local search or performance prediction techniques [22]–[24]. Per-instance algorithm selection techniques were among the first to utilize machine learning to improve constraint solving. The idea is to train an oracle to predict the performance (e.g., runtime) of a set of candidate algorithms on a formula based on its structural characteristics. SDSL differs from both approaches primarily in that it moves meta-algorithmic design *online*.

ML for AR: Machine Learning has been applied in multiple ways to expedite a variety of automated reasoning tasks, including satisfiability checking [8], [9], [25]–[31], Mixed-Integer Convex Programming [32], [33], program/function synthesis [34]–[36], and symbolic execution [37], [38].

While most existing techniques require an offline training phase, the general idea of using online learning also appears in previous work. The Conflict-Driven Clause Learning paradigm itself can be viewed as online learning. In the MapleSAT solver [27], [28], branching is formulated as a multi-armed bandit problem where the estimated reward of each arm (i.e., variable) is maintained and updated throughout the solving. This reinforcement learning interpretation of a dynamic

$$\begin{array}{c}
\frac{i < \mathbf{K} \quad \text{check}(f_i, v) = \text{UNSAT}}{i, v \Rightarrow i + 1, v} \text{ (Next)} \\
\\
\frac{i = \mathbf{K} \quad \text{check}(f_i, v) = \text{UNSAT}}{i, v \Rightarrow \text{FAIL}} \text{ (Failure)} \\
\\
\frac{\text{check}(f_i, v) = \text{SAT}}{i, v \Rightarrow \text{SUCCESS}} \text{ (Success)}
\end{array}$$

Fig. 2. Transition rules for solving a set of related problems. The starting configuration is $\langle 1, v \rangle$.

branching heuristic is perhaps inspired by the study [39] of the popular VSIDS [40] branching heuristic and its later variants [41], which also track a score for each variable during the solving. In contrast to this direction of online learning, SDSL operates on a set of related problems rather than on a single instance. Moreover, SDSL focuses on selecting from a set of existing strategies rather than inventing new ones.

To conclude this section, we remark that in practice, offline learning, “in-solver” online learning, and SDSL could be combined to solve a set of related problems. For example, one could choose to use SAT/SMT solvers with built-in learning components, set the initial solving strategy using offline learning, and then use SDSL to further customize the strategy online. The exploration of such combinations is beyond the scope of this paper, but is a promising future direction.

III. SOLVING SETS OF RELATED PROBLEMS

In this section, we present a simple calculus, *SR*, for iteratively solving a set of related problems. Let $\mathcal{F} = \{f_1, \dots, f_{\mathbf{K}}\}$ be a set of \mathbf{K} related formulas. Assume we have a function $\text{check} : \mathcal{F} \times \mathcal{V} \rightarrow \{\text{SAT}, \text{UNSAT}\}$, which takes as input a formula $f \in \mathcal{F}$ and a solving strategy v (from a set \mathcal{V} called the *strategy space*) and returns either SAT (satisfiable) or UNSAT (unsatisfiable). Additionally, assume that we can stop once any formula is SAT.

The rules of the basic *SR* calculus are shown in Fig. 2. The rules operate over a *configuration*, which is either one of the distinguished symbols $\{\text{SUCCESS}, \text{FAIL}\}$ or a tuple $\langle i, v \rangle$, where $i \in [1, \mathbf{K}]$ is the current formula index and $v \in \mathcal{V}$ is the current solving strategy. The rules describe the conditions under which a certain configuration can transform into another configuration. The **Next** rule says that if the current formula is unsatisfiable and the maximal index \mathbf{K} has not been reached yet, then the current index will be increased. On the other hand, if the current formula is unsatisfiable and it is the last formula, the **Failure** rule transitions the system to the FAIL configuration. The **Success** rule states that SUCCESS can be reached when the current formula is satisfiable.¹

An *SR-execution* is a sequence of configurations that respect the rules in *SR*. Note that no rule updates the solving strategy v . We augment the calculus with strategy updates next.

¹The conditions for progress and termination are inspired by BMC but are applicable in other settings when solving related problems.

$$\begin{array}{c}
\frac{v_s \in \mathcal{V} \quad j \leq i \quad c = \text{cost}(f_j, v_s)}{i, v, D, T \Rightarrow i, v, D \cup \{\langle v_s, j, c \rangle\}, T} \text{ (Collect)} \\
\\
\frac{T' = \text{fit}(D)}{i, v, D, T \Rightarrow i, v, D, T'} \text{ (Train)} \\
\\
\frac{\mathcal{V}_s \subseteq \mathcal{V} \quad v' = \arg \min_{v_s \in \mathcal{V}_s} T(v_s, i)}{i, v, D, T \Rightarrow i, v', D, T} \text{ (Strategize)}
\end{array}$$

Fig. 3. Additional transition rules for Self-Driven Strategy Learning.

Given two configurations C, C' , we use $C \vdash C'$ to denote C can transition (in one or more steps) to C' . We state the following two propositions which are straightforward to verify.

Proposition 1 (Soundness and Completeness): \mathcal{F} contains a satisfiable formula if and only if $\langle 1, v \rangle \vdash \text{SUCCESS}$.

Proposition 2 (Termination): There exist no infinite *SR*-executions.

IV. SELF-DRIVEN STRATEGY LEARNING

A. Informal Presentation

Self-Driven Strategy Learning (SDSL) attempts to learn, on the fly, which solving strategy among a set of candidates \mathcal{V} to use for each formula in \mathcal{F} . Learning is based on data gathered during solving. To obtain the data, we occasionally solve a formula multiple times, each time with a different strategy in \mathcal{V} . For a strategy v_s , we record its effect, $c \in \mathbb{R}$, when solving f_i by creating a data point $\langle v_s, i, c \rangle$, where c is a measure of the cost of the strategy. For example, c could be the total run time required to solve f_i .

Given such a dataset, an oracle $T : \mathcal{V} \times [1, \mathbf{K}] \mapsto \mathbb{R}$ is trained to predict the cost of a given strategy when run on a given formula. When solving a new formula, we select the one that T predicts will be most effective, and as more data is collected with each call to **check**, T is updated.

An essential characteristic of SDSL is that the training data is gathered for a specific, and *a priori* unknown, set of formulas in an *online* and *automatic* manner, as part of the solving process. This approach has two challenging implications. The learning process must not incur a large overhead; otherwise, insufficient time is left for actual solving. Additionally, the choice of \mathcal{V} is crucial as it must be large enough to contain good candidate strategies but also not too large to explore. We address these challenges in Sections V and VI, respectively.

B. Formal presentation

Formally, we present SDSL as an extension of *SR*. Configurations are as in *SR* except that tuple configurations $\langle i, v, D, T \rangle$ have two additional components (assumed to be left unchanged by rules in *SR*): $D \in \mathcal{P}(\mathcal{V} \times [1, \mathbf{K}] \times \mathbb{R})$ is a dataset, each of whose members records the result of running a single strategy on a single formula; and $T : \mathcal{V} \times [1, \mathbf{K}] \mapsto \mathbb{R}$ is an oracle (e.g., a machine learning model) that predicts the cost of a strategy on a formula. Initially, D is empty, and T is arbitrary (e.g., always return 0). The additional transition rules of SDSL are described in Fig. 3.

The **Collect** rule samples a strategy $v_s \in \mathcal{V}$, evaluates its cost when solving f_j , and augments D with this new data. The rule is parameterized by a function $\text{cost} : \mathcal{F} \times \mathcal{V} \mapsto \mathbb{R}$. The **Train** rule updates the oracle T with a new one trained on the current dataset D . It is parameterized by a machine learning algorithm **fit** (e.g., k-NN, tree ensemble, deep learning, etc.). Finally, the **Strategize** rule updates the current strategy by sampling a set of strategies \mathcal{V}_s from \mathcal{V} and choosing the one with the best predicted cost for the current index i . The extended calculus is still sound and complete (i.e., Proposition 1 still holds). Since the added rules can effectively be applied at any time, Proposition 2 only holds if we allow only a finite number of applications of the new rules.

Note that in the **Collect** rule, the results of solving the formula f_j are discarded, as f_j must have been solved already in some previous application of the **Next** rule. It is possible to extend the SR calculus to allow UNKNOWN results from **check**, but the completeness property would be lost.

A reasonable strategy for applying SDSL rules is as follows:

- 1) After every application of **Next**, issue one *learning epoch*; that is, apply **Collect** m times on the current problem, then apply **Train**;
- 2) If **Train** has been applied at least once, apply **Strategize** whenever i is updated;
- 3) If the *estimated learning time* exceeds some threshold n , override the first policy and do not issue any more learning epochs;
- 4) Terminate whenever **Success** or **Failure** applies.

The estimated learning time is calculated as the time spent on learning so far plus $m \cdot t$, where t is the runtime of solving the current problem using the current strategy. If $|\mathcal{V}|$ is small, it may be reasonable to use $m = |\mathcal{V}|$ and try each strategy from \mathcal{V} . In the more general scenario where $m \ll |\mathcal{V}|$, the choice of which samples to use impacts the quality of the dataset. We discuss this choice and present a conditional sampling procedure in Sec. V-A. The purpose of restricting the training time in step three is to ensure that training does not dominate the total time taken. This simple criterion for when to stop learning already works reasonably well in practice. We leave the exploration of more sophisticated heuristics to future work.

V. DESIGN SPACE IN SDSL

This section discusses the design space in the implementation of SDSL and proposes solutions to the following questions: 1) How should training data be sampled? 2) Which machine learning model and training algorithm should be used? The solutions we propose focus on the case where a strategy is simply a set of values for a specific set of solver parameters. In this case, the strategy space is the cartesian product of p sub-strategy spaces, each representing a single parameter: $\mathcal{V} = \mathcal{V}_1 \times \dots \times \mathcal{V}_p$. The set of possible values for each parameter can vary (e.g., parameter values could be Booleans, strings, or numbers), but for now we assume each \mathcal{V}_i is finite.

A. Gathering Informative Training Data

To make the most informed decision, we could try all candidate strategies on all previously considered problems, but this is infeasible when $|\mathcal{V}|$ is large. In the following, we consider the scenario where m samples are drawn from a strategy space \mathcal{V} , where $m \ll |\mathcal{V}|$.

In this restrictive setting, we need to ensure our dataset contains a sufficient number of low-cost strategies (if there are any). Sampling uniformly is unlikely to achieve this goal because in practice, many or most candidate strategies could have high cost. For this reason, we propose to *explicitly* favor low-cost strategies in the sampling process. One way to do this is by using Markov-Chain Monte-Carlo (MCMC) sampling, which in our setting can be used to generate a sequence of solving strategies with the desirable property that in the limit, strategies with the lowest cost are most frequently drawn. A popular MCMC method is the Metropolis-Hastings (M-H) Algorithm [42], instantiated in the context of SDSL as follows:

- 1) Choose a current strategy v ;
- 2) Propose to replace the current strategy with a new one v' , which comes from a *proposal distribution* $q(v'|v)$;
- 3) If $\text{cost}(f, v') \leq \text{cost}(f, v)$, accept v' as the current strategy;
- 4) Otherwise, accept v' as the current strategy with some probability $a(v \rightarrow v')$ (e.g., a probability inversely proportional to the increase in cost);
- 5) Go to step 2.

This process is repeated until m samples are drawn. Importantly, under this scheme, a proposal that results in lower cost is always accepted, while a proposal that does not may still be accepted. This means that the algorithm greedily moves to a better strategy whenever possible, but also has a means for escaping local minima. In our implementation, the acceptance probability is computed using a common method [43] described as follows. We first transform $\text{cost}(f, v)$ into a probability distribution $p(v)$:

$$p(v) \propto \exp(-\beta \cdot \text{cost}(f, v)) ,$$

where $\beta > 0$ is a configurable parameter. The acceptance probability is then computed as:

$$\begin{aligned} a(v \rightarrow v') &= \min \left(1, \frac{p(v')}{p(v)} \right) \\ &= \min (1, \exp(\beta \cdot (c - c'))) , \end{aligned}$$

where $c = \text{cost}(f, v)$ and $c' = \text{cost}(f, v')$. Under this acceptance probability, the larger that c' is compared to c , the lower the probability to accept. On the other hand, the larger β is, the more reluctant we are to move to a worse proposal.

To ensure the aforementioned convergence property of MCMC in the limit, the proposal distribution must be both *symmetric* and *ergodic*.² For discrete search spaces, a common proposal distribution is the symmetric random walk, which

²A proposal distribution q is symmetric if $q(v'|v) = q(v|v')$ for any $v, v' \in \mathcal{V}$ and is ergodic if there is a non-zero probability of reaching a strategy $v \in \mathcal{V}$ from any other strategy $v' \in \mathcal{V}$ in a finite number of steps.

moves to one of the *neighbors* of the current sample with equal probability. For our strategy space, we define the neighbors of a strategy as all strategies for which exactly k parameter values are different. We use $k = 1$ in our implementation.

Note that MCMC sampling can be used not only in the data collection process, but also in the **Strategize** rule (i.e., to choose $\mathcal{V}_s \subseteq \mathcal{V}$). Since in this case we use the machine learning model as an oracle of *cost* (which is much cheaper than calling a solver), a larger sample size is affordable.

The sampling scheme presented above largely coincides with many local search approaches used in the automated configuration literature [5]. Borrowing more insights from that literature and devising more sophisticated sampling schemes are interesting directions for future work.

B. Lightweight Online Learning

In the online setting, the machine learning model must generalize from sparse data in limited time. This means the model needs to be both robust against outliers and efficient to train. Training a neural network from scratch, for example, is likely unsuitable, because it requires large amounts of data and, depending on the architecture, could be costly to train. On the other hand, lightweight ensemble models, which consist of a set of sub-models with different strengths and weaknesses, are well-suited for SDSL.

Our data is what is often called *tabular data*, that is, it can be represented as a table with rows and columns, where each row corresponds to a sample, and each column corresponds to a feature. When the strategy space consists of parameter settings, each sample has $p + 2$ features: the p parameters, the problem index, and the cost. Tree-based ensemble methods such as *random forests* are generally considered to be a good match for such data [44].

A random forest consists of a set of B *regression trees* $\{f_1, \dots, f_B\}$. Each tree is trained independently by sampling data from the data set D . A regression tree makes a prediction by following a path from the root node to a leaf node, based on the values of the input features, and returning the cost associated with the leaf node (which generally is the average of the costs of the training points that map to that leaf node). The predictions of a random forest f are computed by averaging the predictions of the individual trees in f .

A random forest is both efficient to train and efficient for prediction [45]. The time complexity of training a random forest with B trees is $O(B \cdot m \cdot n \cdot \log m)$, where m is the number of data points and n is the number of input features. The inference time complexity for a random forest is $O(B \cdot n)$.

Many machine learning algorithms are themselves parameterized and the performance of the model depends on a good choice of the hyperparameters. For a tree-based algorithm like random forest, an important hyperparameter is the maximal depth allowed for each individual regression tree: too shallow, and the model's prediction will be inaccurate; too deep, and

the model might overfit to outliers.³ The standard way to find suitable values of the hyperparameters is via (cross-)validation: split the data into a training set and a validation set, train models with different hyperparameters on the training set, evaluate them on the validation set, and pick the best one. However, in the SDSL setting where data is already sparse, validation is less feasible because it is hard to make sure that both the training set and the validation set are representative of the input space. Instead, we propose the following pragmatic heuristic: start by training a random forest with shallow trees and then retrain with incrementally deeper trees as needed until the training score is high enough.

VI. CASE STUDY: BOUNDED MODEL CHECKING

Bounded Model Checking (BMC) [1], [2], [17] is a well-known technique for checking whether a property P holds along bounded executions of a given system M . The algorithm starts by checking all executions of length k ; if no counter-example is found, k is increased and the system checked until either a counter-example is found, the problem becomes intractable, or some upper bound on k is exceeded.

BMC is useful in practice for at least two reasons. First, it is often the most efficient way to find counter-examples (if they exist) when trying to prove that a system has a particular property. Second, when techniques capable of providing a full (i.e. unbounded) proof fail (which is often the case in practice), BMC still establishes a certain confidence in the system by providing formal guarantees for bounded executions. The larger the *certified bound*, the stronger the guarantee.

A basic BMC formula for checking whether a property P holds for a system M along executions of length k is:

$$I_0 \wedge \bigwedge_{i=0}^{k-1} \rho(i, i+1) \wedge \left(\bigvee_{i=0}^k \neg P_i \right) ,$$

where I_0 represents the initial state of M , $\rho(i, i+1)$ represents how the system evolves in a single step, and P_i represents the property at step i in the execution. This formula is satisfiable iff there is an execution of length less than or equal to k such that the property P does not hold at the end of the execution.

In practice, when the bound is increased, additional constraints are added stating that previously checked states are safe (in order to prune the search space). For example, suppose the check for bound k' is unsatisfiable. To check bound $k > k'$, we use the following formula:

$$bmc(k', k) := I_0 \wedge \bigwedge_{i=0}^{k-1} \rho(i, i+1) \wedge \bigwedge_{i=0}^{k'} P_i \wedge \left(\bigvee_{i=k'+1}^k \neg P_i \right) .$$

We use BMC as a case study for our approach. For a given system and property, we solve the following set of problems:

$$\mathcal{F} = \{bmc(k - s, k) \mid k = i \cdot s, 1 \leq i \leq \mathbf{K}\} ,$$

³Instead of tuning the tree depth while fixing the number of trees, one could alternatively grow deep trees and tune the number of trees (to be large enough). However, this makes training and prediction much more costly.

where s is the *step size*. We focus on hardware model checking problems where the set of formulas to solve is in the theory of bitvectors [46]. We use standard techniques to encode the bitvector problems as Boolean satisfiability (SAT) problems [47]. Thus, \mathcal{F} is a set of Boolean formulas, and we can implement **check** using an off-the-shelf SAT solver. We use the state-of-the-art KISSAT SAT solver [11].

In the following, we discuss the choice of the cost function **cost** and the strategy space \mathcal{V} for this case study.

A. Choosing the Cost Function

One plausible cost function for a strategy v and a formula f_i is the ratio of the runtime to that of some default strategy v_0 , i.e., if the runtime is t with strategy v and t_0 with v_0 , then $\text{cost}(f_i, v) = \frac{t}{t_0}$. While this definition works in practice, the use of runtime makes SDSL’s behavior non-deterministic across different runs. This is undesirable for many reasons, including experimental reproducibility. Therefore, we instead use the *number of conflicts* generated by the SAT solver, which is accepted as a good proxy for runtime [48]. Given the same parameter settings, the number of conflicts generated by KISSAT on the same problem is deterministic.

B. Choosing the strategy space

As discussed in Sec. IV-B, the choice of the strategy space is crucial to the effectiveness of SDSL. KISSAT has over 90 configurable parameters, so considering all of them is impractical. One plausible approach is to rely on expert/domain knowledge and empirical studies to identify a reasonable set of parameters to consider. We follow this approach to define two strategy spaces for KISSAT.

The first one, \mathcal{V}_{exp} (Table I), is based on a study by Dutertre [49] on the effect of SAT solver parameters on bitvector problems.⁴ We allow two possible values for each parameter, the default one and an alternative one. For options that were found to be beneficial in [49], we include the corresponding parameter in KISSAT and for non-Boolean parameters, we set the alternative value to be more aggressive;⁵ for Boolean parameters, we simply set the alternative value to be the opposite of the default. In total, \mathcal{V}_{exp} contains 8192 (2^{13}) possible parameter settings.

The second strategy space, \mathcal{V}_{dev} (Tab. II), is based on suggestions made by the developer of KISSAT.⁶ It contains significantly fewer possible parameter settings (216).

⁴We consider all options considered in Table 2 of [49], except four: “lucky” and “walk” control procedures that find satisfying assignments independent of the main search; “scan-index” is not available in KISSAT; and “compacting” is a data-structure optimization that we do not believe has strong correlations with the number of conflicts. Noting that [49] does not consider any options related to branching, we additionally consider the *bumpreasonsrate* parameter, which controls the eagerness of reason-side literal bumping [27] and reportedly [11], [50] has significant impact on SAT Competition benchmarks.

⁵The alternative values are selected as follows: **int* parameters are divided by 10; **lim* parameters are divided by 100; and **effort* parameters are doubled. This works well in practice, and in further testing, setting the parameters to other reasonable values did not significantly alter the overall results. In the future, it might be advisable to obtain expert knowledge also on the specific values of the parameters.

⁶See <https://github.com/arminbiere/kissat/issues/25>

TABLE I
THE STRATEGY SPACE \mathcal{V}_{exp} BASED ON [49].

KISSAT option	default	alternative
<i>and</i>	1	0
<i>bumpreasonsrate</i>	10	1
<i>chrono</i>	1	0
<i>eliminateint</i>	500	50
<i>eliminateocclim</i>	2000	20
<i>forwardeffort</i>	100	200
<i>ifthenelse</i>	1	0
<i>probeint</i>	100	10
<i>rephaseint</i>	1000	100
<i>stable</i>	1	0
<i>substituteeffort</i>	10	20
<i>subsumeocclim</i>	1000	10
<i>vivifyeffort</i>	100	200

TABLE II
THE STRATEGY SPACE \mathcal{V}_{dev} .

KISSAT option	default	alternative(s)
<i>chrono</i>	1	0
<i>phase</i>	1	0
<i>stable</i>	1	0, 2
<i>target</i>	1	0, 2
<i>tier1</i>	2	1
<i>tier2</i>	6	3, 9

Designing principled ways to automatically construct the strategy space (e.g., using techniques for assessing parameter importance [51]) is an important direction for future work.

C. Implementation

We implemented an SDSL-based BMC procedure in PYTHON3.⁷ Our prototype takes as input a model checking problem in the BTOR/BTOR2 format [52], [53] and can run BMC on that input with or without SDSL. We implemented SDSL following the strategy described in Sec. IV-B. The BMC step size and the maximal bound are also command-line arguments. Additional input arguments include:

- 1) \mathcal{V} : path to a CSV file representation of the strategy space (e.g., Tabs. I and II);
- 2) n : the time budget for the learning epochs (see Sec. IV-B), by default 15% of the total time limit;
- 3) m : the number of samples to draw per learning epoch, by default 100;
- 4) The number of samples to draw in the **Strategize** rule, by default 500;
- 5) The number of trees in the random forest, by default 50;
- 6) The initial tree depth, by default a third of the number of parameters in \mathcal{V} ;
- 7) The random seed, by default 0.

The default values are used in all experiments unless otherwise specified.

The formula $bmc(k', k)$ is generated online by first creating a bitvector formula using the PONO Model Checker [13],

⁷Available at <https://github.com/anwu1219/sdsl/>

then bit-blasting it into a SAT formula using the BOOLECTOR solver [54]. The versions of the solvers are reported in the extended version [16] of the paper. Our prototype does not leverage incrementality for reasons discussed in Sec. II. We use the Scikit-Learn machine learning library [55] for training the Random Forest. Apart from the number of trees and the depth of the trees, we use the default hyperparameters of Scikit-Learn’s Random Forest module. The prototype runs on one thread, though the sampling, training, and inference are in principle parallelizable.

VII. EXPERIMENTAL EVALUATION

We consider the bitvector track benchmarks from the latest Hardware Model Checking Competition (HWMCC) [10]. We omit all unsatisfiable benchmarks, since these are not solvable using BMC. What remain are 65 benchmarks that were reported to be satisfiable during the competition and 24 benchmarks that were unsolved during the competition. All experiments are performed on a cluster equipped with Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz running Ubuntu 20.04. Each job is given one physical core and 8 GB memory.

A. Unrolling the unsolved benchmarks

In the first experiment, we focus on the 24 unsolved benchmarks. For each benchmark, the goal is to either find a property violation or to prove that the property holds for as large a bound as possible. A CPU time limit of 2 hours is given for each benchmark. We consider two BMC step sizes: 1 and 10.⁸ For each step size, we run as baselines our KISSAT-based BMC implementation without SDSL (denoted KISSAT) and the BMC engine of PONO (denoted PONO), which makes incremental calls to BOOLECTOR to solve bitvector queries.

1) *Performance of SDSL using the strategy space in Tab. I:* We first evaluate KISSAT + SDSL_{exp}, the SDSL-extended BMC procedure using \mathcal{V}_{exp} (Tab. I) as the strategy space. The results are shown in Tab. III. We report the largest *solved* (i.e., certified or falsified) bound k for each configuration. For KISSAT + SDSL_{exp} and KISSAT, we also show the total time to solve all formulas up until the largest commonly solved bound ($t_{c.s.}$). For KISSAT + SDSL_{exp}, this includes the time spent on learning. We further report the number of learning epochs (ep) and the time spent on learning (t_{learn}) for KISSAT + SDSL_{exp}. Graphic illustrations in the style of Fig. 1 and the duration of each training epoch are presented in the extended version [16] of the paper.

When the BMC step size is 1, KISSAT + SDSL_{exp} is able to certify larger bounds compared with the baseline configurations on 22 out of the 24 benchmarks, with an average bound increase of 3.9 (52.6 – 48.7). This improvement is highly non-trivial, considering that to reach a larger bound, KISSAT + SDSL_{exp} needs to 1) certify all the formulas up to the baseline bound; 2) spend time (on average 975 seconds) learning a solving strategy; and 3) solve an additional set of harder formulas with the remaining time, one for each increase in the

bound size. Comparing $t_{c.s.}$ sheds further light on the performance gain enabled by SDSL: on average, KISSAT + SDSL_{exp} is $1.3\times$ faster ($\frac{6354}{4811}$) on the set of commonly solved problems. The fraction of $t_{c.s.}$ that KISSAT + SDSL_{exp} spends on actual solving (not including learning) is 3836 seconds (4811 – 975). Thus, on average a $1.7\times$ speedup ($\frac{6354}{3836}$) is achieved in the sheer performance of the SAT solver. Upon closer examination, the learning time is dominated by the data collection, with actual training and inference only taking 2.1% of t_{learn} on average.

When using BMC step size 10, both the KISSAT-based baseline and KISSAT + SDSL_{exp} find counter-examples on 8 benchmarks (highlighted in red). In all but one of those benchmarks, KISSAT + SDSL_{exp} finds counter-examples faster. Additionally, KISSAT + SDSL_{exp} certifies a larger bound than KISSAT on 4 benchmarks. For the remaining 12 benchmarks, the two configurations certify the same bounds, but KISSAT + SDSL_{exp} reduces the runtime on only 3 of them. One explanation for this is that the number of affordable learning epochs is significantly smaller when the step size is 10 due to the increased hardness of individual formulas. As a result, fewer strategies are considered. For example, on `arb.n2.w128.d64`, a total of 871 unique solving strategies are evaluated when the step size is 1, whereas only 175 strategies are evaluated when the step size is 10. Nonetheless, overall, KISSAT + SDSL_{exp} is still $1.3\times$ faster ($\frac{3712}{2927}$) at certifying the same bounds.

It is important to note that using step size 10 does not necessarily lead to a larger certified bound. Take `circ.w128.d128` for example: KISSAT + SDSL_{exp} can unroll to an execution length of 46 with step size 1 while only unrolling to 30 with step size 10. This also applies to an incremental solver like PONO, which certifies an execution length of 39 with step size 1 versus 20 with step size 10. This suggests that the optimal step size varies in practice.

2) *Performance of SDSL using the strategy space in Tab. II:* We repeat the same experiment for the other SDSL configuration KISSAT + SDSL_{dev}, which uses the smaller strategy space \mathcal{V}_{dev} (Tab. II). The result is shown in Tab. IV. To summarize, KISSAT + SDSL_{dev} still boosts the performance of KISSAT though the overall gain is less. For step size 1, the average solved bound by KISSAT + SDSL_{dev} is 49.4 compared to 48.7 by KISSAT. The overall reduction in $t_{c.s.}$ is not significant (2.8%) though the reduction in the pure solving time (computed by subtracting t_{learn} from $t_{c.s.}$) is still clear (16.5%). For step size 10, KISSAT + SDSL_{exp} and KISSAT unroll to the same bound on each instance, but it takes KISSAT + SDSL_{exp} 12.7% less time to get there. It is not too surprising that the performance gain resulting from KISSAT + SDSL_{dev} is smaller than from KISSAT + SDSL_{exp}. The smaller strategy space has far fewer strategy options and might simply not contain a better strategy than the default one.

In the extended version [16] of the paper, we also consider two additional SDSL configurations. One includes all boolean flags in the strategy space; the other uses local-search-based tuning instead of machine learning to pick the solving strategy. Both configurations perform worse than the KISSAT-based

⁸The value of 10 is chosen based on a study by Lonsing [56].

TABLE III

EVALUATION OF KISSAT + SDSL_{exp} ON BV BENCHMARKS OF HARDWARE MODEL CHECKING COMPETITION 2020 THAT WERE NOT SOLVED DURING THE COMPETITION. *ep* IS THE NUMBER OF TRAINING EPOCHS. *t_{learn}* IS THE TIME SPENT ON DATA COLLECTION, TRAINING, AND INFERENCE. *t_{c.s.}* IS THE CUMULATIVE TIME (*t_{learn}* INCLUDED) TO SOLVE ALL THE FORMULAS UP UNTIL THE LARGEST BOUND COMMONLY SOLVED BY KISSAT + SDSL_{exp} AND KISSAT. *k* IS THE LARGEST SOLVED BOUND WITHIN 2 HOURS AND IS **highlighted** IF A VIOLATION IS FOUND (I.E., THE BENCHMARK IS SOLVED).

Benchmark	step size = 1							step size = 10						
	KISSAT + SDSL _{exp}				KISSAT			KISSAT + SDSL _{exp}				KISSAT		PONO
	<i>ep</i>	<i>t_{learn}</i>	<i>t_{c.s.}</i>	<i>k</i>	<i>t_{c.s.}</i>	<i>k</i>	<i>k</i>	<i>ep</i>	<i>t_{learn}</i>	<i>t_{c.s.}</i>	<i>k</i>	<i>t_{c.s.}</i>	<i>k</i>	<i>k</i>
arb.n2.w128.d64	10	1040	4816	59	7198	54	45	2	751	3676	70	4940	70	50
arb.n2.w64.d64	9	936	4431	59	6515	53	48	2	638	3985	70	4224	70	50
arb.n2.w8.d128	10	1031	5599	54	6795	52	45	2	1127	3932	60	5528	60	50
arb.n3.w16.d128	9	937	4898	58	7118	53	48	2	807	3653	70	5792	60	60
arb.n3.w64.d128	10	1062	4856	58	7132	53	44	1	84	3427	60	5120	60	50
arb.n3.w64.d64	10	985	4809	58	7055	53	46	2	919	3736	70	5018	70	50
arb.n3.w8.d128	9	983	4760	58	6996	53	46	2	906	3227	60	6017	60	50
arb.n4.w128.d64	9	963	6143	54	6107	53	46	2	1086	3704	70	5627	70	50
arb.n4.w16.d64	10	984	4768	57	6599	53	49	2	580	3349	70	5231	70	70
arb.n4.w8.d64	10	1081	5863	54	6659	52	48	2	617	3177	70	4563	70	50
arb.n5.w128.d64	9	1105	5095	57	6731	53	47	1	139	4204	70	5587	70	50
circ.w128.d128	7	845	5746	46	6549	44	39	1	536	2348	30	1743	30	20
circ.w128.d64	8	887	6071	47	6452	46	39	1	747	2688	30	1962	30	20
circ.w16.d128	11	967	4997	58	7179	54	47	2	876	5778	50	4978	50	40
circ.w64.d128	9	870	5409	51	6895	48	44	1	232	4575	40	4294	40	30
dspf.p22	4	1078	3741	31	5654	28	27	1	285	289	20	90	20	20
pgm.3.prop5	10	1019	7128	128	5663	133	131	2	618	6844	190	6052	190	170
picor.AX.nom.p2	2	929	3307	16	3636	15	14	1	117	279	20	151	20	20
picor.pcregs-p0	5	992	4020	32	6402	30	30	0	0	83	20	85	20	20
picor.pcregs-p2	5	840	6149	30	5003	31	31	0	0	89	20	91	20	20
shift.w128.d64	7	858	4098	27	5393	25	21	1	582	678	30	353	20	20
shift.w16.d128	9	1084	3020	45	5817	39	28	2	864	1731	50	2606	40	20
shift.w32.d128	8	821	2778	43	6273	35	27	1	216	2656	30	2424	30	20
zipversa.p03	5	1110	2961	82	6683	59	45	2	1182	2138	110	6622	90	330
Mean	8.1	975	4811	52.6	6354	48.7	43.1	1.5	580	2927	57.5	3712	55.4	55.4

BMC baseline. It is worth noting that local-search-based tuning does also result in speedup in *t_{c.s.}* and improves upon KISSAT on 16 of the 24 instances. However, the performance gain is less significant compared to KISSAT + SDSL_{dev} and, on certain benchmarks, tuning landed on parameter settings that drastically harm the performance. This suggests that using empirical performance models can be more robust than direct tuning in our setting.

B. Mini Hardware Model Checking Competition

We evaluate KISSAT + SDSL_{exp} with step size 10 on all the satisfiable and unsolved bitvector benchmarks from HWMCC. As in the competition, we use a time limit of 1 hour for this experiment. We consider the basic KISSAT-based BMC procedure (also using step size 10) as a baseline. In addition, we perform an apples-to-oranges comparison to two algorithm portfolios, one of PONO and the other of the AVR model checker [12], which was the winner of the most recent competition.⁹ We use the competition portfolio of AVR, which consists of 16 single-threaded solving modes. The PONO portfolio contains 13 single-threaded modes selected by the developers

of PONO. Each mode can construct counter-examples. The AVR portfolio contains two BMC modes, both with step size 5. The PONO portfolio contains 1 BMC mode, with step size 11.

The number of solved instances and the total time on solved instances are shown in Tab. V. To study the complementarity of the configurations, we also report the number of unique solutions and the performance of a virtual best configuration. Results on individual benchmarks are reported in the extended version [16] of the paper.

KISSAT + SDSL_{exp} solves all the instances solved by KISSAT plus 7 more, suggesting that while SDSL might create overhead for easy instances, this overhead is overcome by benefits in the long run. Impressively, those 7 problems are also not solved by the AVR and PONO portfolios. This suggests that including an SDSL-driven BMC procedure in a model checking algorithm portfolio can be beneficial.

C. Ablation studies of training budget and model architecture

To study the effect of dataset size and model accuracy, we select one benchmark `picor.pcregs-p0`, and vary the learning budget (in seconds) in the set {180, 360, 720, 1080, 1440} and the decision tree depth from

⁹We hope to also compare with the bit-level solver ABC [57] but have no information about the commands and version used for the competition. We have contacted the ABC team and will include such results after hearing back.

TABLE IV
EVALUATION OF KISSAT + SDSL_{dev}. THE SETUP IS THE SAME AS TAB. III

Benchmark	step size = 1						step size = 10					
	KISSAT + SDSL _{dev}			KISSAT			KISSAT + SDSL _{dev}			KISSAT		
	<i>ep</i>	<i>t_{learn}</i>	<i>t_{c.s.}</i>	<i>k</i>	<i>t_{c.s.}</i>	<i>k</i>	<i>ep</i>	<i>t_{learn}</i>	<i>t_{c.s.}</i>	<i>k</i>	<i>t_{c.s.}</i>	<i>k</i>
arb.n2.w128.d64	11	861	6714	54	7198	54	2	631	3720	70	4940	70
arb.n2.w64.d64	10	896	6158	54	6515	53	2	575	4981	70	4224	70
arb.n2.w8.d128	11	1006	6291	53	6795	52	2	684	3674	60	5528	60
arb.n3.w16.d128	9	824	6061	54	7118	53	2	635	3905	60	5792	60
arb.n3.w64.d128	10	966	6417	54	7132	53	2	735	3937	60	5120	60
arb.n3.w64.d64	11	1053	5452	55	7055	53	2	598	5619	70	5018	70
arb.n3.w8.d128	9	857	5780	55	6996	53	2	635	3703	60	6017	60
arb.n4.w128.d64	10	901	5859	55	6107	53	2	702	6712	70	5627	70
arb.n4.w16.d64	10	898	5626	55	6599	53	2	462	3266	70	5231	70
arb.n4.w8.d64	11	973	4749	57	6659	52	2	532	3774	70	4563	70
arb.n5.w128.d64	10	869	5672	55	6731	53	1	92	5283	70	5587	70
circ.w128.d128	8	694	6948	43	5879	44	1	400	2158	30	1743	30
circ.w128.d64	8	829	6650	45	5803	46	1	788	2663	30	1962	30
circ.w16.d128	12	969	6961	54	7179	54	2	433	4927	50	4978	50
circ.w64.d128	10	713	6759	48	6895	48	1	179	4361	40	4294	40
dsfp.p22	4	959	4148	31	5654	28	1	185	177	20	90	20
pgm.3.prop5	16	920	6933	133	6996	133	3	557	5664	190	6052	190
picor.AX.nom.p2	2	590	3540	15	3636	15	1	59	191	20	151	20
picor.pcregs-p0	6	873	6573	28	3337	30	0	0	86	20	85	20
picor.pcregs-p2	5	646	6045	28	2786	31	0	0	86	20	91	20
shift.w128.d64	8	666	5965	25	5393	25	1	1392	601	20	353	20
shift.w16.d128	9	940	5209	40	5817	39	1	139	2058	40	2606	40
shift.w32.d128	8	796	5551	36	6273	35	1	217	2615	30	2424	30
zipversa.p03	2	460	7037	59	6683	59	1	268	3650	90	6622	90
Mean	8.8	840	5962	49.4	6134	48.7	1.5	454	3242	55.4	3712	55.4

TABLE V
COMPARISON WITH TWO ALGORITHM PORTFOLIOS ON SATISFIABLE AND UNSOLVED BV HWMCC BENCHMARKS (89 IN TOTAL).

Config.	Threads	Slv.	Time	Unique
KISSAT + SDSL _{exp}	1	68	27362	7
KISSAT	1	61	6358	0
AVR PORTFOLIO	16	48	12113	2
PONO PORTFOLIO	13	63	10723	0
VIRTUAL BEST	31	72	24700	–

1 to 10.¹⁰ We consider all 50 combinations of the two. For each combination, we run KISSAT + SDSL_{exp} (step size 1, time limit 2 hours) 12 times, each time with a different random seed (0...11); we show the median certified bound in the top half of Fig. 4 and the average training score (R^2 score, the larger the better) in the last learning epoch in the bottom half. The largest bound certified by KISSAT without SDSL is 30.

Noticeably, on this instance, improvements in the certified bound are achieved when the depth of the tree is at least 4 and the learning budget is at least 1080 seconds. This suggests that both a sufficient amount of training data and an accurate model are necessary for SDSL to work in practice. If not enough data

¹⁰For this experiment we use a fixed tree depth instead of the dynamic one described in Sec. V-B.

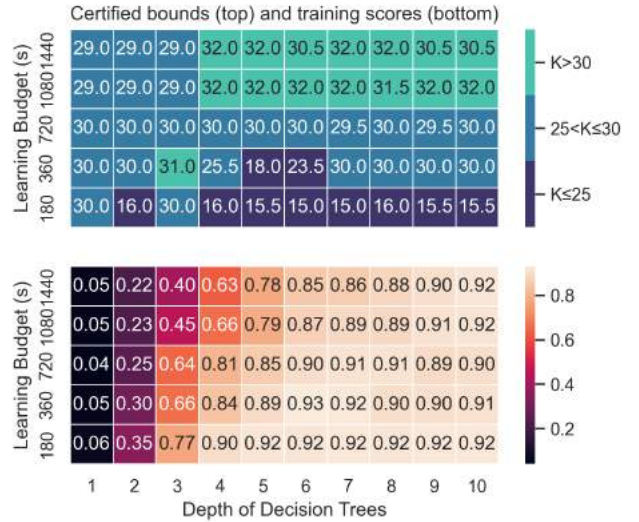


Fig. 4. Varying the learning budget and tree depth on `picor.pcregs-p0`.

is collected (bottom right), the machine learning model cannot extrapolate well to new problem instances. On the other hand, if the machine learning model is not accurate enough (top left), the strategy it suggests can also be misleading. Determining the optimal learning budget on a per-benchmark basis is a topic worth studying in the future.

VIII. CONCLUSION, LIMITATIONS, AND FUTURE WORK

We introduced Self-Driven Strategy Learning, a conceptually simple, easy-to-implement online learning approach for solving sets of related problems in automated reasoning. We presented the methodology formally as a set of transition rules and instantiated it in the context of Bounded Model Checking. Our experiments show that equipping a BMC-procedure with SDSL results in a significant performance boost, both in terms of certified bounds and solved instances, when comparing against state-of-the-art open-source model checkers.

One thing to consider when applying SDSL is that a good return on investment in learning depends on a sensible *a priori* choice of the strategy space. Another limitation is that when the problem set is small, gathering sufficient training data can be challenging. An intriguing question is whether a problem can be decomposed into sub-problems automatically in order to obtain sufficient data. Other future directions include alternative orders of applying the SDSL rules, applying SDSL to other automated reasoning tasks (e.g., symbolic execution, max-satisfiability, iterative abstraction refinement), and combining SDSL with offline learning and incremental solving.

Acknowledgments: We thank the anonymous reviewers for their careful reviews and constructive feedback. This work was supported in part by the National Science Foundation (grant 1269248) and by the Stanford Center for Automated Reasoning. Additionally, the NASA University Leadership initiative (grant #80NSSC20M0163) provided funds to assist the authors with their research, but this article solely reflects the opinions and conclusions of its authors and not any NASA entity.

REFERENCES

- [1] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded model checking using satisfiability solving,” *Formal methods in system design*, vol. 19, pp. 7–34, 2001.
- [2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.
- [3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 2000, pp. 154–169.
- [4] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [5] H. H. Hoos, F. Hutter, and K. Leyton-Brown, “Automated configuration and selection of sat solvers,” in *Handbook of Satisfiability*. IOS Press, 2021, pp. 481–507.
- [6] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu, “Boosting verification by automatic tuning of decision procedures,” in *Formal Methods in Computer Aided Design (FMCAD’07)*. IEEE, 2007, pp. 27–34.
- [7] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Satzilla: portfolio-based algorithm selection for sat,” *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008.
- [8] D. Selsam and N. Bjørner, “Guiding high-performance sat solvers with unsat-core predictions,” in *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*. Springer, 2019, pp. 336–353.
- [9] E. Yolcu and B. Póczos, “Learning local search heuristics for boolean satisfiability,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [10] M. Preiner, A. Biere, and N. Froleyks, “Hardware model checking competition 2020,” 2020.
- [11] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020,” in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.
- [12] A. Goel and K. Sakallah, “Avr: abstractly verifying reachability,” in *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part I 26*. Springer, 2020, pp. 413–422.
- [13] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett, “Pono: a flexible and extensible smt-based model checker,” in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer, 2021, pp. 461–474.
- [14] J. N. Hooker, “Solving the incremental satisfiability problem,” *The Journal of Logic Programming*, vol. 15, no. 1-2, pp. 177–186, 1993.
- [15] A. Biere, M. S. Chowdhury, M. J. Heule, B. Kiesl, and M. W. Whalen, “Migrating solver state,” in *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
- [16] H. Wu, C. Hahn, F. Lonsing, M. Mann, R. Ramanujan, and C. Barrett, “Lightweight online learning for sets of related problems in automated reasoning [extended version],” *arXiv preprint arXiv:2305.11087*, 2023.
- [17] O. Strichman, “Accelerating bounded model checking of safety properties,” *Formal Methods in System Design*, vol. 24, pp. 5–24, 2004.
- [18] F. Hutter, H. H. Hoos, and T. Stützle, “Automatic algorithm configuration based on local search,” in *Aaai*, vol. 7, 2007, pp. 1152–1157.
- [19] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, “Satenstein: Automatically building local search sat solvers from components,” *Artificial Intelligence*, vol. 232, pp. 20–42, 2016.
- [20] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, “Algorithm selection for SMT,” *Int. J. Softw. Tools Technol. Transf.*, vol. 25, no. 2, pp. 219–239, 2023. [Online]. Available: <https://doi.org/10.1007/s10009-023-00696-0>
- [21] L. Xu, H. Hoos, and K. Leyton-Brown, “Hydra: Automatically configuring algorithms for portfolio-based selection,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, 2010, pp. 210–216.
- [22] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, “Algorithm runtime prediction: Methods & evaluation,” *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.
- [23] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, K. Tierney *et al.*, “Model-based genetic algorithms for algorithm configuration,” in *IJCAI*, 2015, pp. 733–739.
- [24] K. Leyton-Brown, E. Nudelman, and Y. Shoham, “Empirical hardness models: Methodology and a case study on combinatorial auctions,” *Journal of the ACM (JACM)*, vol. 56, no. 4, pp. 1–52, 2009.
- [25] M. Balunovic, P. Bielik, and M. Vechev, “Learning to solve smt formulas,” *Advances in Neural Information Processing Systems*, vol. 31, 2018.
- [26] C. Hahn, F. Schmitt, J. U. Kreber, M. N. Rabe, and B. Finkbeiner, “Teaching temporal logics to neural networks,” in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: <https://openreview.net/forum?id=dOcQK-f4byz>
- [27] J. H. Liang, V. Ganesh, P. Poupard, and K. Czarnecki, “Learning rate based branching heuristic for SAT solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140. [Online]. Available: https://doi.org/10.1007/978-3-319-40970-2_9
- [28] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh, “Machine learning-based restart policy for cdcl sat solvers,” in *Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21*. Springer, 2018, pp. 94–110.

- [29] H. Wu, "Improving sat-solving with machine learning," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 2017, pp. 787–788.
- [30] H. Wu and R. Ramanujan, "Learning to generate industrial sat instances," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 10, no. 1, 2019, pp. 206–207.
- [31] J. You, H. Wu, C. Barrett, R. Ramanujan, and J. Leskovec, "G2sat: learning to generate sat formulas," *Advances in neural information processing systems*, vol. 32, 2019.
- [32] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, R. Addanki, T. Hapuarachchi, T. Keck, J. Keeling, P. Kohli, I. Ktena, Y. Li, O. Vinyals, and Y. Zwols, "Solving mixed integer programs using neural networks," *CoRR*, vol. abs/2012.13349, 2020. [Online]. Available: <https://arxiv.org/abs/2012.13349>
- [33] D. Bertsimas and B. Stellato, "The voice of optimization," *Machine Learning*, vol. 110, no. 2, pp. 249–277, Feb 2021. [Online]. Available: <https://doi.org/10.1007/s10994-020-05893-5>
- [34] P. Golia, S. Roy, and K. S. Meel, "Manthan: A data-driven approach for boolean function synthesis," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*. Springer, 2020, pp. 611–633.
- [35] P. Golia, F. Slivovsky, S. Roy, and K. S. Meel, "Engineering an efficient boolean functional synthesis engine," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.
- [36] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016.
- [37] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, "Learning to accelerate symbolic execution via code transformation," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- [38] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2526–2540. [Online]. Available: <https://doi.org/10.1145/3460120.3484813>
- [39] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, "Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers," in *Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings 11*. Springer, 2015, pp. 225–241.
- [40] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 530–535.
- [41] A. Biere and A. Fröhlich, "Evaluating cdcl variable scoring schemes," in *Theory and Applications of Satisfiability Testing—SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer, 2015, pp. 405–422.
- [42] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The american statistician*, vol. 49, no. 4, pp. 327–335, 1995.
- [43] R. E. Kass, B. P. Carlin, A. Gelman, and R. M. Neal, "Markov chain monte carlo in practice: a roundtable discussion," *The American Statistician*, vol. 52, no. 2, pp. 93–100, 1998.
- [44] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.
- [45] G. Louppe, "Understanding random forests: From theory to practice," *arXiv preprint arXiv:1407.7502*, 2014.
- [46] C. Barrett, A. Stump, and C. Tinelli, "The satisfiability modulo theories library (smt-lib). www," *SMT-LIB. org*, vol. 15, pp. 18–52, 2010.
- [47] D. Kroening and O. Strichman, *Decision procedures*. Springer, 2016.
- [48] F. Beskyd and P. Surynek, "Domain dependent parameter setting in sat solver using machine learning techniques," in *Agents and Artificial Intelligence: 14th International Conference, ICAART 2022, Virtual Event, February 3–5, 2022, Revised Selected Papers*. Springer, 2023, pp. 169–200.
- [49] B. Dutertre, "An empirical evaluation of sat solvers on bit-vector problems," in *SMT*, 2020, pp. 15–25.
- [50] A. Biere, "Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018," *Proceedings of SAT Competition*, vol. 14, pp. 316–336, 2017.
- [51] F. Hutter, H. Hoos, and K. Leyton-Brown, "An efficient approach for assessing hyperparameter importance," in *International conference on machine learning*. PMLR, 2014, pp. 754–762.
- [52] R. Brummayer, A. Biere, and F. Lonsing, "Btor: bit-precise modelling of word-level problems for model checking," in *Proceedings of the joint workshops of the 6th international workshop on satisfiability modulo theories and 1st international workshop on bit-precise reasoning*, 2008, pp. 33–38.
- [53] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2, btormc and boolector 3.0," in *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Springer, 2018, pp. 587–595.
- [54] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014. [Online]. Available: <https://doi.org/10.3233/sat190101>
- [55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al., "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.
- [56] F. Lonsing, "Pono: An smt-based model checker," in *Center for Automated Reasoning Workshop*. Stanford, CA, 2022. [Online]. Available: <http://www.florianlonsing.com/talks/Lonsing-CentaurRetreat-2022-talk.pdf>
- [57] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.

DelBugV: Delta-Debugging Neural Network Verifiers

Raya Elsaleh and Guy Katz

The Hebrew University of Jerusalem, Jerusalem, Israel

Email: {rayae,guykatz}@cs.huji.ac.il

Abstract—Deep neural networks (DNNs) are becoming a key component in diverse systems across the board. However, despite their success, they often err miserably; and this has triggered significant interest in formally verifying them. Unfortunately, DNN verifiers are intricate tools, and are themselves susceptible to soundness bugs. Due to the complexity of DNN verifiers, as well as the sizes of the DNNs being verified, debugging such errors is a daunting task. Here, we present a novel tool, named DELBUGV, that uses automated *delta debugging* techniques on DNN verifiers. Given a malfunctioning DNN verifier and a correct verifier as a point of reference (or, in some cases, just a single, malfunctioning verifier), DELBUGV can produce much simpler DNN verification instances that still trigger undesired behavior — greatly facilitating the task of debugging the faulty verifier. Our tool is modular and extensible, and can easily be enhanced with additional network simplification methods and strategies. For evaluation purposes, we ran DELBUGV on 4 DNN verification engines, which were observed to produce incorrect results at the 2021 neural network verification competition (VNN-COMP’21). We were able to simplify many of the verification queries that trigger these faulty behaviors, by as much as 99%. We regard our work as a step towards the ultimate goal of producing reliable and trustworthy DNN-based software.

I. INTRODUCTION

Deep neural networks (DNNs) [21] are software artifacts that are generated automatically, through the generalization of a finite set of examples. These artifacts have been shown to outdo manually crafted software in a variety of key domains, such as natural language processing [19], [25], [37], image recognition [25], [59], protein folding [26], [41], and many others. However, this impressive success comes at a price: unlike traditional software, DNNs are opaque artifacts, and are incomprehensible to humans. This poses a serious challenge when it comes to certifying, modifying, extending, repairing or reasoning about them [22], [27], [32].

In an effort to address these issues, the formal methods community has taken up an interest in *DNN verification* [27], [30], [45]: automated techniques that can determine whether a DNN satisfies a prescribed specification, and provide a counter-example if it does not. DNN verification technology has been making great strides, and its applicability has been demonstrated in various domains [2], [3], [18], [30], [33]. In fact, this technology has progressed to a point where DNN verifiers themselves have become quite complex, and consequently error-prone; especially as they often perform delicate arithmetic operations that can introduce bugs into the verification process [30]. Thus, it is not surprising that various bugs have been observed in these tools [29]. For example,

in the VNN-COMP’21 competition [9], various verifiers have been shown to disagree on the result of multiple verification queries (each query is comprised of a neural network and a property to be checked), or produce incorrect counter-examples — indicating bugs in those verifiers. Moreover, many verifiers are still under development, with new and experimental features being introduced, possibly allowing the introduction of new bugs, as well. An inability to trust the results of DNN verifiers could undermine the benefits of DNN verification technology, and clearly needs to be addressed.

Here, we propose to mitigate this issue by adopting known techniques from related fields (e.g., SMT solving [12]) — specifically, that of *delta debugging*. The idea is to leverage the fact that DNN verification is at a point where many verification tools are available, and to allow engineers to readily compare the results produced by their verification tool to those produced by others, in order to identify and correct bugs. When a verification query that triggers some bug in a verifier is detected, we can initiate an automated process that repeatedly and incrementally *simplifies* the verification query. After each simplification step, we can check that the verifier in question still disagrees with the remaining, *oracle* verifiers, until reaching the simplest verification query that we can find. If this final query is much simpler than the original, it will be that much easier for engineers to debug their tools, eventually improving their overall soundness.

We present a new tool, DELBUGV (**Delta deBugging Neural Network Verifiers**), that takes as input a verification query, a malfunctioning DNN verifier that errs on the given verification query, and an oracle DNN verifier. Within DELBUGV, we implement a set of operations for simplifying the neural network of the given verification query into a network with fewer layers and fewer neurons. We empirically design a strategy that applies these operations sequentially in an order that produces much simpler verification queries. In some cases, when the malfunctioning DNN verifier produces a faulty counter-example, DELBUGV can run in *single solver* mode – without an oracle verifier, where the query is repeatedly simplified as long as the malfunctioning DNN verifier continues to produce incorrect counter-examples.

For evaluation, we tested DELBUGV on 4 DNN verifiers “suspected” of errors, per the results of VNN-COMP’21 [9]: Marabou [32], NNV [50]–[53], [60], NeuralVerification.jl(NV.jl) [36], and nenum [7], [8], [50], [51]. We ran DELBUGV on queries where pairs of these verifiers

disagreed. Our evaluation demonstrates that DELBUGV could reduce the size of the error-triggering queries by an average of 96.8%, and by as much as 99% in some cases, resulting in very simple neural networks. We believe that these results highlight the significant potential of our tool and approach.

The rest of the paper is organized as follows. In Sec. II we provide the necessary background on DNNs and their verification. Next, in Sec. III we describe the design of DELBUGV, focusing on its algorithm and network simplification methods and the strategy we use to apply those methods. The implementation and evaluation of DELBUGV are discussed in Sec. IV. This is followed by a discussion of related work in Sec. V, and we conclude in Sec. VI.

II. BACKGROUND

Neural Networks. A *neural network* is a directed acyclic graph in which the nodes, called neurons, are organized in layers l^0, l^1, \dots, l^n . l^0 is called the input layer, l^n the output layer, and layers l^1, \dots, l^{n-1} are called hidden layers. Each hidden layer has an associated non-linear *activation function*. In feed-forward networks, which are our subject matter here, neurons in layer l^i have edges connecting them only to neurons in the next layer, layer l^{i+1} .

Each neuron in the network (except the ones in the input layer) has a bias value, and each edge has a weight. The biases and weights belonging to neurons in layer l^i are organized into a vector B^i and a matrix W^i , respectively. The j, j' -th entry of W^i is the weight assigned to the edge out-going from the j' -th neuron in layer l^{i-1} and entering the j -th neuron in layer l^i . For a *fully connected* layer, W^i is a full matrix; whereas for a *convolutional* layer, W^i is very sparse, and has a specific structure (discussed later).

An input to neural network \mathcal{N} is a vector I of values of the neurons in the input layer, and it produces an output vector $\mathcal{N}(I)$ which is the values of the neurons in the output layer. We denote the values of neurons in layer l^i , prior to applying the activation function, by $\mathcal{N}^i(I)$; and the values after applying the activation function by $\mathcal{N}^{a^i}(I)$. The values of the neurons are evaluated according to the rules:

$$\begin{aligned} \mathcal{N}^{l^0}(I) &= I, & \mathcal{N}^{l^i}(I) &= W^i \mathcal{N}^{a^{i-1}}(I) + B^i, \\ \mathcal{N}^{a^i}(I) &= \text{Act}^i(\mathcal{N}^{l^i}(I)) \end{aligned}$$

where Act^i is the activation function associated with layer l_i .

We define the size of a neural network to be the total number of neurons in the graph (including the neurons in the input and output layers) and denote it by $|\mathcal{N}|$. The automated training (i.e., selection of weights and biases) of neural networks is beyond our scope here; see, e.g., [21].

Fig. 1 depicts a neural network, \mathcal{N}_e , with a single input, a single output, and 2 hidden layers with 3 neurons in each. It uses the ReLU activation function, $\text{ReLU}(x) = \max(0, x)$. The bias of each neuron is listed above it, and weights are

listed over the edges (zero values are omitted). In matrix representation, the weights and biases are:

$$\begin{aligned} W^1 &= \begin{bmatrix} -5 \\ -0.5 \\ -1 \end{bmatrix}, B^1 = \begin{bmatrix} 10 \\ -2.5 \\ 7 \end{bmatrix}, W^2 = \begin{bmatrix} 0.8 & -1 & -2 \\ 0 & 0.5 & 0 \\ 2 & 0.5 & -1 \end{bmatrix}, \\ B^2 &= \begin{bmatrix} 8 \\ 2 \\ 0 \end{bmatrix}, W^3 = \begin{bmatrix} 0.25 \\ 2 \\ 0.5 \end{bmatrix}^T, B^3 = [0] \end{aligned}$$

\mathcal{N}_e is of size 8 (every l_j^i and r_j^i pair in the figure are counted as one neuron; we split them only for visualization purposes), and has 4 layers. The figure also demonstrates an evaluation of the network, for the input $x = 5$. The assignment of each node is listed below it; and we can see that the produced output in this case is $y = 5$.

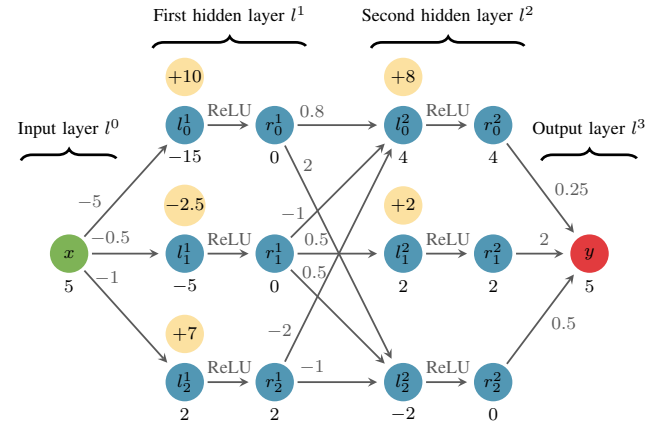


Fig. 1: An example of a neural network, \mathcal{N}_e , with ReLU activation functions.

Convolutional Neural Networks. A *convolutional neural network* is a neural network with one or more convolutional layers (typically, these are the first layers of the network). The parameters of a convolutional layer include the height h and width w of images in the input; the kernel size k ; the stride size s ; the padding size p ; the input channels c_i ; the output channels c_o ; the kernel weights W , given as a tensor of dimensions $(c_o \times c_i \times k \times k)$; and the biases, B , organized in an array of length c_o . We assume for simplicity that the kernel size, padding size, and stride size are equal along all axes, although this is not a limitation of our approach.

The convolutional layer filters its input, which is a $(c_i \times h \times w)$ -dimensional matrix, using the above parameters and outputs a multidimensional matrix which represents feature maps. For additional information on how a convolutional layer computes its output, see [21]. Note that convolutional layers are comprised strictly of linear operations.

Neural Network Verification. A *property* \mathcal{P} is a set of constraints on the inputs and outputs of the neural network. These constraints give rise to an input region $I(\mathcal{P})$ and an output region $O(\mathcal{P})$. Verifying \mathcal{P} , with respect to some neural

network, entails determining whether there exists an input in $I(\mathcal{P})$ that the neural network maps to an output in $O(\mathcal{P})$ (the SAT case), or not (the UNSAT case). Typically, \mathcal{P} is specified so that $O(\mathcal{P})$ represents *undesirable* behavior, and so an UNSAT result indicates that the system is correct. $\mathcal{P}_e = (5 \leq x \leq 10) \wedge (5 \leq y \leq 10)$ is an example of a property of \mathcal{N}_e in Fig. 1.

A *neural network verifier* takes in a verification query (a neural network and a property) and attempts to automatically verify it. When successful, it returns a SAT or UNSAT answer; otherwise, it can return ERROR, or TIMEOUT. When a neural network verifier returns SAT, it also returns an input that proves the satisfiability of the query. Given a verifier \mathcal{V} and a verification query $Q = (\mathcal{N}, \mathcal{P})$, we denote by $\mathcal{V}(Q) \in \{\text{SAT}, \text{UNSAT}, \text{ERROR}, \text{TIMEOUT}\}$ the answer of \mathcal{V} on Q . If $\mathcal{V}(Q) = \text{SAT}$, we denote by $\mathcal{V}_w(Q) \in I(\mathcal{P})$ the satisfying assignment (the witness) returned by the verifier.

Continuing with our running example, given a sound neural network verifier \mathcal{V}_e and the verification query $Q_e = (\mathcal{N}_e, \mathcal{P}_e)$, $\mathcal{V}_e(Q_e) = \text{SAT}$ and a valid witness is $(\mathcal{V}_e)_w(Q_e) = (5)$, since $\mathcal{N}_e((5)) = (5) \in O(\mathcal{P}_e)$.

Neural network verification is complex, both theoretically and practically [30]; and modern tools apply sophisticated techniques to verify large networks [1]. These techniques are typically theoretically sound, but implementation bugs can cause verifiers to produce incorrect results. These bugs are easier to track and correct if the problem manifests for queries with small networks.

In a situation where two verifiers disagree on the satisfiability of a given query, at least one of them must answer SAT and provide a satisfying assignment. We evaluate the neural network on that assignment, and determine whether it indeed satisfies the property at hand. If so, we conclude that the other verifier, which returned UNSAT, is faulty; otherwise, if the satisfying assignment is incorrect, we determine that the verifier that answered SAT is faulty. The remaining verifier then takes the role of the oracle verifier.

III. DELBUGV: DELTA-DEBUGGING VERIFICATION QUERIES

A. General Flow

Applying *delta-debugging* techniques means automatically simplifying an input x that triggers a bug in the system into a simpler input, x' , that also triggers a bug [40]. x' can often trigger the bug faster, thus reducing overall debugging time; and also trigger fewer code lines that are unrelated to the bug, allowing engineers to more easily identify its root cause. In our setting, given a verification query $Q = (\mathcal{N}, \mathcal{P})$ that triggers a bug in a neural network verifier, we seek to generate another query $Q' = (\mathcal{N}', \mathcal{P})$, with a much smaller (simplified) neural network: $|\mathcal{N}'| < |\mathcal{N}|$. The motivation for focusing on the neural network, and not on the verification conditions, is that common verification conditions are typically already quite simple [56], whereas neural network sizes have a crucial effect on verifier performance [30].

The general delta debugging framework that our tool follows appears as Alg. 1. The inputs to the process are a faulty verifier \mathcal{V} , an oracle verifier \mathcal{V}_O , and a verification query $Q = (\mathcal{N}, \mathcal{P})$. The algorithm maintains a candidate result neural network \mathcal{N}_r that triggers a bug in \mathcal{V} and make it produce an incorrect answer, and whose size is iteratively decreased. In each iteration, the algorithm invokes Alg. 2 to attempt simplifying \mathcal{N}_r . The process terminates when Alg. 2 states that it cannot simplify \mathcal{N}_r any further, or when a timeout limit is exceeded. Finally, it returns the verification query with the smallest \mathcal{N}_r it achieved.

Algorithm 1 Reduce Verification Query

Input: $\mathcal{V}, \mathcal{V}_O, Q = (\mathcal{N}, \mathcal{P})$

// Faulty Verifier, Oracle Verifier, Verification query

Output: Q_r *// A simplified query*

- 1: $\mathcal{N}_r \leftarrow \mathcal{N}$
 - 2: $\text{progressMade} \leftarrow \text{True}$
 - 3: **while** $\text{noTimeout}() \wedge \text{progressMade}$ **do**
 - 4: $\mathcal{N}_r \leftarrow \mathcal{N}$
 - 5: $\text{progressMade}, \mathcal{N} \leftarrow \text{Simplify}(\mathcal{V}, \mathcal{V}_O, Q)$
 - 6: **return** $(\mathcal{N}_r, \mathcal{P})$
-

Alg. 2 takes in the same arguments as Alg. 1, and its goal is to perform one successful simplification step on \mathcal{N} , from a pool of potential steps. The algorithm heuristically chooses a sequence of simplification steps to attempt (Line 1), and then performs them, one by one, until one is successful. We propose several simplification steps in Sec. III-B. Specifying the order according to which these simplification steps are attempted (Line 1) is key, and different strategies may result in different simplified networks — we propose one such strategy in Sec. III-B.

Algorithm 2 Simplify

Input: $\mathcal{V}, \mathcal{V}_O, Q = (\mathcal{N}, \mathcal{P})$

// Faulty Verifier, Oracle Verifier, Verification query

Output: True/False, Q_r *// Whether the query was simplified, and the simplified query*

- 1: $\text{Attempts} = (M_0, M_1, \dots) \leftarrow \text{attemptsBySimplificationStrategy}(\mathcal{N})$
 - 2: **while** $\text{Attempts} \neq \emptyset$ **do**
 - 3: $M_i \leftarrow \text{Attempts.pop}()$
 - 4: $\mathcal{N}_r \leftarrow M_i(\mathcal{N})$
 - 5: **if** $\text{successSimplification}(\mathcal{V}, \mathcal{V}_O, (\mathcal{N}_r, \mathcal{P}))$ **then**
 - 6: **return** True, \mathcal{N}_r
 - 7: **return** False, \mathcal{N}
-

Line 5 of Alg. 2 invokes Alg. 3 to check whether the simplification step attempted succeeded or not. To do so, Alg. 3 first checks whether \mathcal{V} answers SAT, but returns an incorrect counter-example. If so, this candidate should clearly be kept. Otherwise, the algorithm checks whether \mathcal{V} and \mathcal{V}_O disagree in their verdicts; if so, it returns True. In all other

cases, i.e. where one of the verifiers times out, or when there is no basis for comparison (one of the verifiers returned an error), the algorithm returns False, and an alternative simplification step in Alg. 2 is attempted.

Algorithm 3 *successSimplification*

Input: $\mathcal{V}, \mathcal{V}_O, Q = (\mathcal{N}, \mathcal{P})$

// Faulty Verifier, Oracle Verifier, Verification query

Output: True/False *// Was the query successfully simplified?*

```

1: if  $\mathcal{V}(\mathcal{N}, \mathcal{P}) = \text{SAT} \wedge \mathcal{V}_O(Q) \notin I(\mathcal{P})$  then
2:   return True
3: if  $\mathcal{V}(\mathcal{N}, \mathcal{P}) = \text{SAT} \wedge \mathcal{N}(\mathcal{V}_O(Q)) \notin O(\mathcal{P})$  then
4:   return True
5: if  $\mathcal{V}(\mathcal{N}, \mathcal{P}), \mathcal{V}_O(\mathcal{N}, \mathcal{P}) \in \{\text{SAT}, \text{UNSAT}\}$ 
    $\wedge \mathcal{V}(\mathcal{N}, \mathcal{P}) \neq \mathcal{V}_O(\mathcal{N}, \mathcal{P})$  then
6:   return True
7: return False

```

One possible risk when using Alg. 1 is a “flip” between the two verifiers. This can happen when initially, \mathcal{V}_O produces a correct answer and \mathcal{V} does not; but after a simplification step, \mathcal{V} starts producing the correct answer and \mathcal{V}_O starts producing an incorrect answer. This situation is unlikely: the simplification steps we propose later make local modifications to the network, and are consequently far more likely to continue to trigger the same bug in \mathcal{V} than to trigger a new one in \mathcal{V}_O . Still, this concern can be mitigated even further by using multiple oracle verifiers, and ensuring that they all agree amongst themselves while \mathcal{V} dissents. Even though this design does not completely prevent a “flip” scenario, it makes it highly unlikely.

Single Verifier Mode. Our approach could also be applied to delta-debug a single verifier that returns incorrect satisfying assignments, without using an oracle. As we explain in Sec. III-B, the simplification methods we apply require the returned satisfying assignment from either the faulty or the oracle verifier; and thus, if the faulty verifier returns an incorrect satisfying assignment for the query at hand, we can drop the oracle verifier. This is achieved by removing the last “if” condition from Alg. 3 and removing the oracle verifier \mathcal{V}_O from the inputs. Note, however, that if the faulty verifier returns an UNSAT answer, an oracle verifier is always needed.

B. Simplification Methods

A core component of Alg. 1 is the selection of simplification strategy to apply (Line 1 in Alg. 2). We now describe our pool of neural network simplification methods, and the strategy that we suggest for selecting among them. The goal of all the simplification methods we propose here is to reduce neural network sizes, while keeping the network’s behavior (i.e., its outputs) similar to that of the original; especially on the counter-example provided by either the faulty verifier or the oracle verifier. Note that a single simplification method can often be applied multiple times, in different ways, using different input parameters.

Method 1: linearizing piecewise-linear activation functions between fully-connected layers. In general, the presence of activation functions is a major source of complexity in the verification process of neural networks: they render the problem NP-complete, require complex mechanisms for linearly approximating them, and often entail case-splitting that slows down the verifiers [30], [39], [57]. Thus, in order to simplify the neural network, we propose to eliminate such activation functions, by *fixing them to a single linear segment*, effectively replacing them with linear constraints. This procedure is performed on an entire layer at a time; which, in turn, creates a sequence of consecutive purely linear layers that can then be merged into a single linear layer, reducing the overall number of layers and neurons in the network.

In choosing the linear segment to which each function is fixed, we propose to use the counter-example I provided by either the faulty verifier or the oracle verifier. The output of the new linear segment we choose, with respect to I , will match the output of the activation function on I .

For simplicity, we focus here on the ReLU activation function ($\text{ReLU}(x) = \max(x, 0)$), although the technique is applicable to any piecewise-linear function. Intuitively, in such cases we propose to replace *active* ReLUs ($x \geq 0$) by the identity function, and *inactive* ReLUs ($x < 0$) by zero. More formally, observe two consecutive layers, l^t and l^{t+1} , in the neural network \mathcal{N} , where layer l^t has a ReLU activation function. We construct an alternative layer, l^a , to replace both l^t and l^{t+1} . l^a inherits the activation function of l^{t+1} . The weights W^a and the biases B^a of l^a are calculated as:

$$W^a = W^{t+1} W' W^t$$

$$B^a = W^{t+1} W' B^t + B^{t+1}$$

where

$$W'_{i,j} = \begin{cases} 1 & i = j \wedge (N_Q^{l^t}(I))_i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Here W' is the new linear segment replacing the activation function ReLU. Finally, the obtained simplified network \mathcal{N}_r is the network \mathcal{N} where layers l^t and l^{t+1} are deleted and replaced with l^a .

Fig. 2 depicts the result of applying this method on layers l^2 and l^3 from Fig. 1, using the assignment $I_e = (5)$. Fig. 2a depicts the layers selected for merging; and Fig. 2b depicts the resulting neural network. Notice that $\mathcal{N}_e^{l^2}(I_e) = (4, 2, -2)$, meaning that only the ReLUs in neurons l_0^2 and l_1^2 are active. Thus, these ReLUs are replaced by the identity function, whereas the inactive ReLU of l_2^2 is replaced by 0. After this step, layers l^2 and l^3 perform only linear operations, and are merged into a single layer.

Method 2: linearizing piecewise-linear activation functions between convolutional layers. In this method, a convolutional layer is combined with the layer following it (either a fully connected layer or a convolutional one), and replaced by a single, fully connected layer.

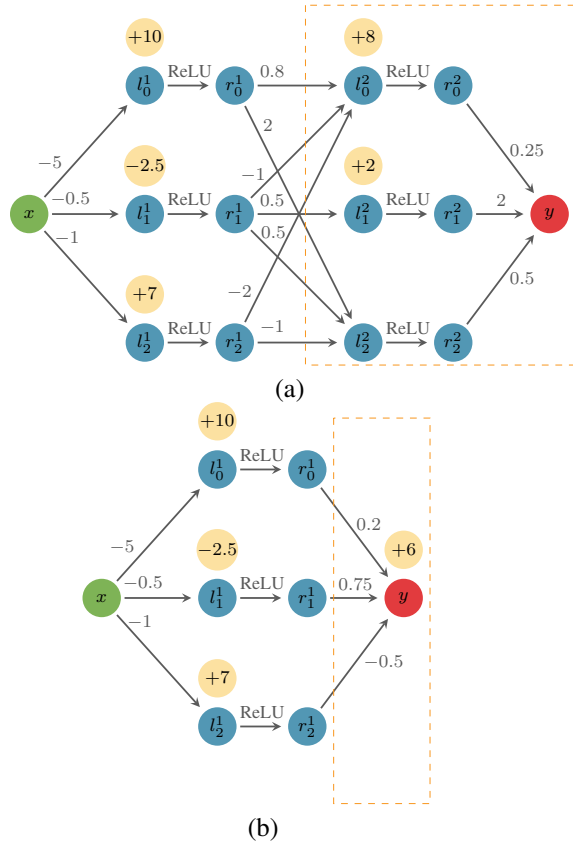


Fig. 2: \mathcal{N}_e with layers l^2 and l^3 selected in orange (a), and then merged (b).

For simplicity, we focus here on the case where the second layer is fully connected. More formally, observe two consecutive layers, l^t and l^{t+1} in \mathcal{N} , where l^t is a convolutional layer and l^{t+1} is a fully connected layer. Our goal is to construct an alternative layer, l^a , that will replace l^t and l^{t+1} . Since a convolutional layer is a particular case of fully connected layer, we construct l^a by first converting the convolutional layer l^t into a fully connected one, denoted l^c ; then linearizing the activation functions, as in *Method 1*; and finally, combining the two layers into one. This method may add edges to the network, and could potentially cause the network size to temporarily increase. However, when this method is used in conjunction with the remaining methods, it sets the network up for additional simplification, which ultimately results in a much smaller network.

Denote by W^t and W^{t+1} the matrices representing the weights of layer l^t and l^{t+1} respectively, and by B^t and B^{t+1} the vectors representing their respective biases. To transform a convolutional layer into a fully connecting one, we calculate the weights, W^c , and the biases, B^c , of the fully connected layer replacing the convolutional one, according to the conventional layer parameters. First, we turn its input and output from a multidimensional tensors into 1-dimensional vectors. The height and width (dimensions) of the feature maps

in the convolutional layer's output are: h_o, w_o where

$$h_o = \left\lfloor \frac{h + 2p - k}{s} \right\rfloor + 1, \quad w_o = \left\lfloor \frac{w + 2p - k}{s} \right\rfloor + 1.$$

The convolutional layer's output contains c_o feature maps, i.e., the dimensions of the output are $(c_o \times h_o \times w_o)$. Thus, the dimensions of W^c are $(c_o h_o w_o \times c_i h w)$. W^c is a sparse matrix. To calculate the value of the i, j -th entry in W^c , we first compute the following values:

$$\begin{aligned} c'_i &= \left\lfloor \frac{j}{hw} \right\rfloor, \quad c'_o = \left\lfloor \frac{i}{h_o w_o} \right\rfloor, \\ i' &= \left\lfloor \frac{i - c_i hw}{w} \right\rfloor - \left(\left\lfloor \frac{j - c_o h_o w_o}{w_o} \right\rfloor \cdot s - p \right) \\ j' &= ((i - c_i hw) \bmod w) - (((j - c_o h_o w_o) \bmod w_o) \cdot s - p) \end{aligned}$$

c'_i and c'_o are the input and output channels that the i, j -th entry should be associated with. i' and j' are the indices in the kernel that should match to the i, j -th entry. The weight matrix W^c is given by:

$$W^c_{i,j} = \begin{cases} W^t_{c'_i, c'_o, i', j'} & 0 \leq i' \wedge j' < k \\ W^c_{i,j} = 0 & \text{otherwise} \end{cases}$$

Finally,

$$B^c_i = B^t_{\left\lfloor \frac{i}{h_o w_o} \right\rfloor}$$

According to this construction of W^c and B^c , they will have the same functionality as the convolutional operation they replace (assuming no floating-point or numerical errors). This step may temporarily increase the number of edges in the network (the number of neurons remains fixed); but this is required to prepare for the minimization step.

The next step is to linearize the ReLU. This is done in a similar manner to the linearization in the previous method, from which we get W' . Next, we construct the weights W^a and the biases B^a of the alternative layer l^a :

$$\begin{aligned} W^a &= W^{t+1} W' W^c \\ B^a &= W^{t+1} W' B^c + B^{t+1} \end{aligned}$$

And the activation function assigned to the new layer l_a is the same as the one assigned to layer l_{t+1} . Finally, the simplified neural network \mathcal{N}_r is the network \mathcal{N} , where layers l_t and l_{t+1} are deleted and replaced with l_a .

In case l^{t+1} is also a convolutional layer, we convert it to a fully connected layer, as we did with l^t ; and the remainder of the process is unchanged.

Method 3: merging neurons. In this method, we seek to merge a pair of neurons in the same layer into a single neuron, thus decreasing the neural network size by one. Of course, this entails selecting the weights of this new neuron's incoming and outgoing edges, as well as its bias. Our motivation is to cause the merged neuron to produce values close to those of the original neurons, and consequently cause little changes in the neural network's eventual output. We present first the technical

process of merging neurons, and later discuss *which* pairs of neurons should be merged.

We focus again on the case where the activation function is ReLU. We first use the counter-example I (returned by either the faulty verifier or the oracle verifier) to check whether the activation functions of the neurons being merged have the same phase — i.e., if they are both active, or both inactive. If they have the same phase, we compute the merged neuron's weights and biases using the original neurons' weights and biases. Specifically, the weight of each edge incoming to the merged neuron is the mean of the original incoming edge weights, and the neuron's bias is the mean of the original neurons' biases; whereas the weights of its outgoing edges are the weighted sum, according to I , of the original outgoing edge weights. We choose a weighted sum, instead of a simple sum, to ensure that the neurons in the following layer obtain values similar to their original ones with respect to I ; and also to preserve the network's behavior. In case one of the neurons is active and the other is inactive, we simply delete the inactive one, since it does not contribute to the following layer's neuron values (with respect to I).

Formally, given a neural network, \mathcal{N} , two successive layers in it, l^t and l^{t+1} , and two neurons indices $b < c$, we construct two alternative layers l^a and l^{a+1} that will replace l^t and l^{t+1} respectively. l^a and l^{a+1} inherit the activation functions of l^t and l^{t+1} respectively. If the ReLUs of the neurons b and c in layer l^t have the same phases: $(\mathcal{N}^{l^t}(I))_b, (\mathcal{N}^{l^t}(I))_c > 0$ or $(\mathcal{N}^{l^t}(I))_b, (\mathcal{N}^{l^t}(I))_c < 0$, the weights and the biases $W^a, W^{a+1}, B^a, B^{a+1}$ of the alternative layers are calculated as follows:

$$B_i^a = \begin{cases} B_i^t & i < b \vee b < i < c \\ \frac{B_b^t + B_c^t}{2} & i = b \\ B_{i+1}^t & c \leq i \end{cases}$$

$$B^{a+1} = B^{t+1}$$

$$W_{i,j}^a = \begin{cases} W_{i,j}^t & i < b \vee b < i < c \\ \frac{W_{b,j}^t + W_{c,j}^t}{2} & i = b \\ W_{i+1,j}^t & c \leq i \end{cases}$$

$$W_{i,j}^{a+1} = \begin{cases} W_{i,j}^{t+1} & j < b \vee b < j < c \\ 2 \cdot \frac{(W_{i,b}^{t+1}(\mathcal{N}^{l^{t+1}}(I))_b + W_{i,c}^{t+1}(\mathcal{N}^{l^{t+1}}(I))_c)}{(\mathcal{N}^{l^{t+1}}(I))_b + (\mathcal{N}^{l^{t+1}}(I))_c} & j = b \\ W_{i,j+1}^{t+1} & c \leq j \end{cases}$$

Otherwise, if the ReLUs of the neurons b and c in layer l^t have different phases: $(\mathcal{N}^{l^t}(I))_b > 0 \wedge (\mathcal{N}^{l^t}(I))_c < 0$ (assume w.l.o.g. that the c -th neuron is the inactive one), the weights and biases $W^a, W^{a+1}, B^a, B^{a+1}$ of the alternative

layers are calculated as follows:

$$B_i^a = \begin{cases} B_i^t & i < c \\ B_{i+1}^t & c \leq i \end{cases}, \quad B^{a+1} = B^{t+1}$$

$$W_{i,j}^a = \begin{cases} W_{i,j}^t & i < c \\ W_{i+1,j}^t & c \leq i \end{cases}, \quad W_{i,j}^{a+1} = \begin{cases} W_{i,j}^{t+1} & j < c \\ W_{i,j+1}^{t+1} & c \leq j \end{cases}$$

Finally, the obtained simplified neural network \mathcal{N}_r , is the network \mathcal{N} where layers l^t and l^{t+1} are replaced with l^a and l^{a+1} respectively. This method can be applied repeatedly, to reduce the network size even further.

An example of applying this method on the pair of neurons l_0^2 and l_1^2 in \mathcal{N}_e from Fig. 1 using the assignment $I_e = (5)$ appears in Fig. 3. Fig. 3a shows the neurons selected for merging, and Fig. 3b shows the result of the merge.

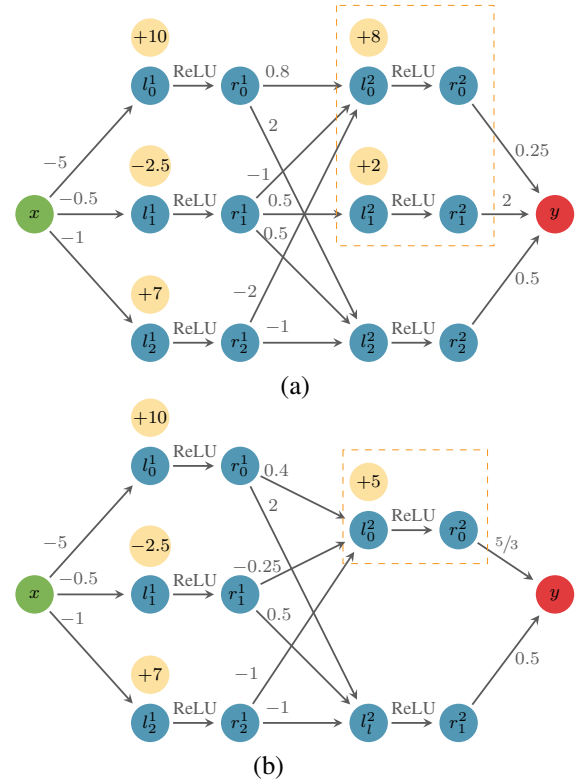


Fig. 3: \mathcal{N}_e with neurons l_0^2 and l_1^2 selected in orange (a), and then merged (b).

Choosing which pair of neurons to merge is crucial for the success of this method. Every two neurons in the same layer are valid candidates; however, some pairs are more likely to succeed than others by resulting in a simplified neural network that behaves similarly to the original. We consider the following possible approaches for prioritizing between the pairs: (1) an arbitrary ordering; (2) prioritizing pairs with neurons that are assigned similar values (prior to the activation function), when the network is evaluated on assignment I . The motivation is that merging such pairs is expected to have smaller effect on the overall functionality of the neural network; (3) prioritizing pairs of neurons whose

ReLU are inactive when evaluated on I . The motivation is that inactive neurons may have little effect on the bug at hand. This approach can be combined with Approach 2 to prioritize pairs with similar values after categorizing them by the status of the ReLUs; (4) prioritizing pairs of neurons with positive values with respect to I . This approach, too, can be combined with Approach 2; and (5) prioritizing pairs of neurons with negative values, and then pairs with positive values, with respect to I . This approach is a combination of Approaches 3 and 4, and again uses Approach 2 for internal prioritization within each category.

Strategy for applying the simplification rules. Within Alg. 1, the simplification steps mentioned above can be invoked in any order. We propose to attempt methods that significantly reduce the neural network size first, in order to reduce verification times. We empirically observed that this is achieved by the following strategy: first, attempt to linearize and merge convolutional layers (*Method 2*). Second, attempt to linearize and merge fully connected layers (*Method 1*) — starting with the output layer, and working backwards towards the input layer. Finally, merge neurons (*Method 3*) according to Approach 5. However, our implementation is highly customizable, and users can configure it to use any other strategy, according to the task at hand.

To illustrate, applying our proposed strategy to \mathcal{N}_e from Fig. 1, with respect to the assignment $I_e = (5)$ in which $\mathcal{N}_e^1(I_e) = (-15, -5, 2)$ and $\mathcal{N}_e^{l^2}(I_e) = (4, 2, -2)$, would result in attempting the simplification methods in the following order: (1) merge the layers l^2 and l^3 ; (2) merge the layers l^1 and l^2 ; (3) merge the pair of neurons l_0^1, l_1^1 ; (4) merge the pair of neurons l_1^2, l_2^2 ; (5) merge the pair of neurons l_0^2, l_2^2 ; (6) merge the pair of neurons l_1^1, l_2^1 ; and then, (7) merge the pair of neurons l_0^1, l_2^1 . These steps are attempted, in order, until one succeeds; after which the strategy is reapplied to the simplified network, and so on.

IV. IMPLEMENTATION AND EVALUATION

We designed our tool, DELBUGV, to be compatible with the standard input format used in the VNN-COMP competition [9], in which verification queries are encoded using the *VNN-LIB* format [11]; and which, in turn, relies on the *Open Neural Network Exchange (ONNX)* format. This facilitated integrating DELBUGV with the various verifiers. DELBUGV is implemented in Python, and contains classes that wrap objects of these formats. The tool has a modular design that allows applying our proposed minimization methods in any order desired. The source code can be found at <https://github.com/Raya5/DelBugV>.

VNN-COMP’21 included 12 participating neural network verifiers, and these were tested on a set of verification queries. We began by extracting from the VNN-COMP’21 results pairs of dissenting verifiers, and the verification queries that triggered these discrepancies. Each such triple (two verifiers and a query) constitutes an input to DELBUGV. This extraction led us to target the following verifiers: (1) Marabou [32];

(2) NNV [50]–[53], [60]; (3) NeuralVerification.jl (NV.jl) [36]; and (4) nnenum [7], [8], [50], [51]. In the experiments described next, we used the same versions of these verifiers that were used in VNN-COMP’21.

Neuron Merging and Prioritization Approaches. For our first experiment, we set out to determine which of the neuron-pair prioritization schemes described as part of *Method 3* in Sec. III-B is the most successful. We measured success along two parameters: the size of the simplified network obtained, and by the percentage of successful merging steps along the way. We tested our algorithm on 5 input triples, involving networks of size 310 each. Using only *Method 3*, we ran DELBUGV with each of the prioritization schemes, and counted for each scheme the number of merging steps performed and the number of the steps that succeeded on all of the 5 input triples. Table. I shows the results of this comparison: the second column indicates, for every approach, the percentage of the successful steps out of all the steps tried, aggregated for all 5 benchmarks.

TABLE I: Comparing neurons merging approaches (Method 3) by size reduction and successful merges.

	Successful merges (%)	Average Reduction (%)
Approach 1	37.2%	96.0%
Approach 2	68.4%	95.9%
Approach 3	71.6%	96.0%
Approach 4	62.9%	95.8%
Approach 5	75.9%	96.0%

Looking at the average reduction sizes, the results indicate that all 5 approaches were able to achieve a similar reduction in size, with a slight advantage to approaches 1, 3 and 5. However, the number of successful merges varied significantly — from Approach 1, in which only 37.2% of the merge steps were successful, and up to 75.9% for Approach 5 (in bold). These results thus indicate that Approach 5 is the most efficient among the considered approaches, and so we used it as our default strategy for Method 3 in the subsequent experiments.

Linearizing ReLU Activations. In *Method 1* and *Method 2* in Sec. III-B, we proposed to linearize activation functions, and then merge them with the previous and following layers. These methods can be applied to any piecewise-linear activation function in the network. The order in which they are applied is customizable. In this experiment, we set out to compare linearizing ReLUs in ascending order (from input layer towards output layer), and in descending order (from output towards input). Table II shows the results of this experiment.

Every row in the table corresponds to an input triple to DELBUGV (two disagreeing verifiers and a verification query that they disagreed on), and the two simplification approaches that were attempted.

For each such experiment, the second column indicates the number of simplification steps tried, until DELBUGV reached saturation (there were no additional steps to try). The third column indicates the number of the successful steps out of all

TABLE II: Comparing linearizing layers approaches by successful steps. * indicates the existence of a convolutional layer.

	Linearizing approach	No. of steps	No. of successful steps	Successful steps %	Neuron reduction %
1.	Ascending	6	6	100.0%	96.7%
	Descending	6	6	100.0%	96.7%
2.	Ascending	6	6	100.0%	96.7%
	Descending	6	6	100.0%	96.7%
3.	Ascending	6	6	100.0%	96.7%
	Descending	6	6	100.0%	96.7%
4.	Ascending	6	0	0.0%	0.0%
	Descending	6	0	0.0%	0.0%
5.	Ascending	12	5	41.6%	80.6%
	Descending	9	6	66.6%	96.7%
6.	Ascending	3	2	66.6%	39.2%
	Descending	2	2	100.0%	39.2%
7.	Ascending	3*	2*	66.6%	65.8%
	Descending	2*	1	50.0%	0.0%

the steps. In column four, the percentage of successful steps out of all steps is shown; we use this column to compare the approaches. We mark in bold the leading approach for every triple. The final column shows the reduction percentage in the neural network size. When one of the approaches was clearly superior, the entry appears in bold.

To analyze the results, observe, e.g., the 5th experiment in Table II. The results imply that when using the ascending approach, 12 linearizing and merging steps were made, until the network could not be simplified further with either *Method 1* or *Method 2*. Of these 12 steps, 5 were successful — and consequently, the simplified network has 5 fewer layers than the original. In contrast, with the descending approach only 9 steps were made until the network could not be simplified further, 6 of which were successful. Consequently, the simplified network in this case has 6 fewer layers compared to the original.

The results indicate that linearizing in descending order slightly outperforms linearizing in ascending order, although the gap is not very significant. We believe that the results depend also on both the functionality of the verifier and the values of the network as well. Meaning, they can vary between the benchmarks. The neural network in the last row included a convolutional layer, and, according to the results, linearizing it in ascending order performed better. After investigating this query further, we noticed that in the ascending order approach, the convolutional layer was merged into a fully connected one; whereas the descending approach did not succeed in removing or merging any convolutional layers. We thus conclude that, for a convolutional network, it is advisable to apply *Method 2* before applying *Method 1*.

An interesting phenomenon in both our methods is that they have an overall high reduction percentage for most of the strategies. The strategies mainly differ in the number of steps taken to reach this reduction. This phenomenon implies that the simplification methods are overall effective, but may be time-consuming. Thus, using strategies that entail dispatching fewer verification queries using the verifiers is more productive.

Delta Debugging Discrepancies from VNN-COMP’21. For our final experiment, we considered 13 triples of verifiers, oracle verifiers, and verification queries. Of these triples, 11 contained DNNs from the ACAS-Xu family [30], 1 was a DNN from the MNIST DNNs [35], and 1 was a DNN from the Oval21 benchmark [9]. The DNNs from the ACAS-Xu family had 8 layers: 6 inner fully-connected layers with 50 neurons in each and 5 neurons in both the input and output layer — 310 neurons in total in each network. The MNIST DNN contained 4 fully-connected layers with 784 neurons in the input layer, 256 neurons in each of the hidden layers and 10 neurons in the output layer — 1306 neurons in total. The Oval21 DNN is a convolutional neural network with 5 layers. Its input layer contains 3072 neurons (which represent $3 \times 32 \times 32$ images). Those neurons are processed by the first two convolutional hidden layers to 4096 neurons and then to 2048 neurons. The following hidden layer is a fully-connected one with 100 neurons followed by the output layer which contains 10 neurons — in total, the Oval21 DNN has 9326 neurons. It is worth mentioning that the properties we used (taken from VNN-COMP) separately state the input specifications and output specifications; however, our methods do not require such a distinction. Further, although the VNN-COMP specifications contains primarily linear constraints, our method can also handle relational properties.

Using the optimal configuration of our tool as previously discussed, we applied the full-blown delta-debugging algorithm to all of our 13 benchmarks. The results appear in Table. III. Every row in the table represents a triple, and the first two columns indicate the number of neurons in the original network, and the number of remaining neurons after delta debugging was applied. The next two columns indicate the number of layers in the original and reduced networks; and the final column indicates the percentage of neurons that were removed.

TABLE III: Delta-debugging using our algorithm. * indicates the existence of a convolutional layer.

Neurons		Layers		Reduction percentage
In Original	In reduced	In original	In reduced	
310	6	8	2	98%
310	7	8	2	97%
310	6	8	2	98%
310	12	8	8	96%
310	6	8	2	98%
9326	12	5*	3	99%
1306	11	4	2	99%
310	10	8	3	96%
310	6	8	2	98%
310	10	8	4	96%
310	10	8	4	96%
310	9	8	4	97%
310	13	8	6	95%

Overall, the algorithm performed exceedingly well, reducing the network sizes by an average of 96.8% (!); and, in some cases, causing a size decrease of 99%, from a neural network with 1306 neurons and 4 layers to just 11 neurons and 2 layers (an input layer and an output layer, without any activation

functions). The minimal decrease observed was 95%, from 310 neurons to 13. We regard these results as a very strong indication of the usefulness of delta debugging in the context of DNN verification. Further analyzing the results, we observe that the ReLU linearization simplification rule was responsible for an average of 66% of the size reduction, whereas the remaining two rules were responsible for an average of 34% — indicating that the ReLU linearization simplification rule is the main workhorse of our approach at its current configuration.

V. RELATED WORK

With the increasing pervasiveness of DNNs, the verification community has been devoting growing efforts to verifying them. Numerous approaches have been proposed, including SMT-based approaches [23], [30]–[32], [48], [58], approaches based on LP or MILP solvers [14], [16], [49], reachability-based approaches [38], [60], abstraction and abstract-interpretation based approaches [5], [18], [24], [27], [39], [44], [46], [57], synthesis-based approaches [33], [42], run-time optimization [4], [6], quantitative verification [10], verification of recurrent networks [28], [62], and many others. These approaches, in turn, have been used in numerous application domains [15], [17], [20], [47], [54], [55], [61]. Given the scope of these efforts, and the number of available tools, it is not surprising that bugs are abundant, and that engineers are in need of efficient debugging tools.

To the best of our knowledge, no previous work has applied delta debugging in the context of DNN verification, although similar approaches have been shown successful in the related domains of SMT [12], [40] and SAT [13] solving. Related efforts have attempted to reduce DNN sizes, with the purpose of producing smaller-but-equivalent networks, or networks smaller with respect to a particular verification property of interest [5], [34], [43], [44]. In the future, principles from these approaches could be integrated as simplification strategies within our delta-debugging approach.

VI. CONCLUSION

In this paper, we presented the DELBUGV tool for automatically reducing the size of a verification query with respect to an erroneous neural network verifier. We focused on delta-debugging techniques, and proposed multiple minimization methods for reducing neural network sizes. These techniques attempt to simplify the neural network in question, while modifying it as little as possible. We also suggested a strategy for the order in which to apply those methods. We demonstrated the effectiveness of DELBUGV on actual benchmarks from the VNN-COMP’21 competition, and were able to significantly simplify them. We regard this work as another step towards more sound tools for DNN verification.

Moving forward, we aim to continue this line of work in several directions. One direction we plan to pursue is to extend our evaluation, by considering more diverse sets of benchmarks and comparing our approach also to existing, general-purpose delta debuggers. Another direction is to extend our pool of neural network simplification methods, for

example by supporting also activation functions that are not piecewise linear (e.g., Sigmoids). Additionally, we want to theoretically analyze our simplification methods. Such analysis will potentially benefit us in reducing the need for oracle verifiers.


Acknowledgements. This work was partially supported by the Binational Science Foundation (grant number 2021769).

REFERENCES


- [1] A. Albarghouthi. *Introduction to Neural Network Verification*. verified-deeplearning.com, 2021.
- [2] G. Amir, Z. Freund, G. Katz, E. Mandelbaum, and I. Refaeli. veriFIRE: Verifying an Industrial, Learning-Based Wildfire Detection System. In *Proc. 25th Int. Symposium on Formal Methods (FM)*, 2023.
- [3] G. Amir, T. Zelazny, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 27–37, 2022.
- [4] G. Anderson, S. Pailoor, I. Dillig, and S. Chaudhuri. Optimization and Abstraction: a Synergistic Approach for Analyzing Neural Network Robustness. In *Proc. 40th ACM SIGPLAN Conf. on Programming Languages Design and Implementations (PLDI)*, pages 731–744, 2019.
- [5] P. Ashok, V. Hashemi, J. Kretinsky, and S. Mohr. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proc. 18th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 92–107, 2020.
- [6] G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Könighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.
- [7] B. Bak. nenum: Verification of Relu Neural Networks with Optimized Abstraction Refinement. In *Proc. 13th NASA Formal Methods Symposium (NFM)*, pages 19–36, 2021.
- [8] S. Bak. Execution-Guided Overapproximation (EGO) for Improving Scalability of Neural Network Verification. In *Proc. 3rd Int. Workshop on Verification of Neural Networks (VNN)*, 2020.
- [9] S. Bak, C. Liu, and T. Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results, 2021. Technical Report. <http://arxiv.org/abs/2109.00498>.
- [10] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks and its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1249–1264, 2019.
- [11] C. Barrett, G. Katz, D. Guidotti, L. Pulina, N. Narodytska, and A. Tacchella. The Verification of Neural Networks Library (VNN-LIB), 2019. www.vnnlib.org.
- [12] R. Brummayer and A. Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proc. 7th Int. Workshop on Satisfiability Modulo Theories (SMT)*, 2009.
- [13] R. Brummayer, F. Lonsing, and A. Biere. Automated Testing and Debugging of SAT and QBF Solvers. In *Proc. 13th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, pages 44–57, 2010.
- [14] R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. Mudigonda. A Unified View of Piecewise Linear Neural Network Verification. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 4795–4804, 2018.
- [15] G. Dong, J. Sun, J. Wang, X. Wang, and T. Dai. Towards Repairing Neural Networks Correctly, 2020. Technical Report. <http://arxiv.org/abs/2012.01872>.
- [16] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.
- [17] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 305–318, 2021.
- [18] T. Gehr, M. Mirman, D. Drachler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

- [19] Y. Goldberg. A Primer on Neural Network Models for Natural Language Processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.
- [20] B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In *Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 260–278, 2020.
- [21] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
- [22] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. <http://arxiv.org/abs/1412.6572>.
- [23] D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th. Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
- [24] E. Goubault, S. Palumby, S. Putot, L. Rustenholz, and S. Sankaranarayanan. Static Analysis of ReLU Neural Networks with Tropical Polyhedra. In *Proc. 28th Int. Symposium on Static Analysis (SAS)*, pages 166–190, 2021.
- [25] J. Guo, H. He, T. He, L. Lausen, M. Li, H. Lin, X. Shi, C. Wang, J. Xie, S. Zha, et al. GluonCV and GluonNLP: Deep Learning in Computer Vision and Natural Language Processing. *Journal of Machine Learning Research*, 21(23):1–7, 2020.
- [26] J. Hou, B. Adhikari, and J. Cheng. DeepSF: Deep Convolutional Neural Network for Mapping Protein Sequences to Folds. *Bioinformatics*, 34(8):1295–1303, 2018.
- [27] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
- [28] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 57–74, 2020.
- [29] K. Jia and M. Rinard. Exploiting Verified Neural Networks via Floating Point Numerical Error, 2020. Technical Report. <http://arxiv.org/abs/2003.03021>.
- [30] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
- [31] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks, 2021.
- [32] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.
- [33] B. Könighofer, F. Lorber, N. Jansen, and R. Bloem. Shield Synthesis for Reinforcement Learning. In *Proc. Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 290–306, 2020.
- [34] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 183–192, 2021.
- [35] Y. LeCun. The MNIST Database of Handwritten Digits, 1998. <http://yann.lecun.com/exdb/mnist/>.
- [36] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. Algorithms for Verifying Deep Neural Networks, 2020. Technical Report. <http://arxiv.org/abs/1903.06758>.
- [37] X. Liu, P. He, W. Chen, and J. Gao. Multi-Task Deep Neural Networks for Natural Language Understanding, 2019. Technical Report. <http://arxiv.org/abs/1901.11504>.
- [38] A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. <http://arxiv.org/abs/1706.07351>.
- [39] M. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations. In *Proc. 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2022.
- [40] A. Niemetz, M. Preiner, and C. Barrett. Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers. In *Proc. 34th Int. Conf. on Computer Aided Verification (CAV)*, pages 92–106, 2022.
- [41] F. Noé, G. De Fabritiis, and C. Clementi. Machine Learning for Protein Folding and Dynamics. *Current Opinion in Structural Biology*, 60:77–84, 2020.
- [42] E. Polgreen, R. Abboud, and D. Kroening. Counterexample Guided Neural Synthesis, 2020. Technical Report. <https://arxiv.org/abs/2001.09245>.
- [43] P. Prabhakar. Bisimulations for Neural Network Reduction. In *Proc. 23rd Int. Conf. Verification on Model Checking, and Abstract Interpretation (VMCAI)*, pages 285–300, 2022.
- [44] P. Prabhakar and Z. Afzal. Abstraction Based Output Range Analysis for Neural Networks, 2020. Technical Report. <https://arxiv.org/abs/2007.09527>.
- [45] L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.
- [46] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.
- [47] M. Sotoudeh and A. Thakur. Correcting Deep Neural Networks with Small, Generalizing Patches. In *Workshop on Safety and Robustness in Decision Making*, 2019.
- [48] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU Networks. *Journal of Machine Learning*, pages 1–28, 2021.
- [49] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, 2017. Technical Report. <http://arxiv.org/abs/1711.07356>.
- [50] H.-D. Tran, S. Bak, W. Xiang, and T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 18–42, 2020.
- [51] H.-D. Tran, D. Manzanar Lopez, P. Musau, X. Yang, L. Nguyen, W. Xiang, and T. Johnson. Star-Based Reachability Analysis of Deep Neural Networks. In *Proc. Int. Symposium on Formal Methods (FM)*, pages 670–686, 2019.
- [52] H.-D. Tran, P. Musau, D. Lopez, X. Yang, L. Nguyen, W. Xiang, and T. Johnson. Parallelizable Reachability Analysis Algorithms for Feed-Forward Neural Networks. In *Proc. 7th Int. Workshop on Formal Methods in Software Engineering (FormalSE)*, pages 31–40, 2019.
- [53] H.-D. Tran, X. Yang, D. Lopez, P. Musau, L. Nguyen, W. Xiang, S. Bak, and T. Johnson. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems, 2020. Technical Report. <http://arxiv.org/abs/2004.05519>.
- [54] C. Urban, M. Christakis, V. Wüstholtz, and F. Zhang. Perfectly Parallel Fairness Certification of Neural Networks. In *Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–30, 2020.
- [55] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. Păsăreanu. NNrepair: Constraint-based Repair of Neural Network Classifiers, 2021. Technical Report. <http://arxiv.org/abs/2103.12535>.
- [56] International Verification of Neural Networks Competition (VNN-COMP), 2020. <https://sites.google.com/view/vnn20/vnncomp>.
- [57] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and Z. Kolter. Beta-CROWN: Efficient Bound Propagation with Per-Neuron Split Constraints for Complete and Incomplete Neural Network Verification. In *Proc. 35th Conf. on Neural Information Processing Systems (NeurIPS)*, 2021.
- [58] H. Wu, A. Ozdemir, A. Zeljić, A. Irfan, K. Julian, D. Gopinath, S. Fouladi, G. Katz, C. Păsăreanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 128–137, 2020.
- [59] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep Image: Scaling up Image Recognition. Technical Report. <http://arxiv.org/abs/1501.02876>.
- [60] W. Xiang, H. Tran, and T. Johnson. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 2018.
- [61] X. Yang, T. Yamaguchi, H.-D. Tran, B. Hoxha, T. Johnson, and D. Prokhorov. Neural Network Repair with Reachability Analysis, 2021. Technical Report. <https://arxiv.org/abs/2108.04214>.
- [62] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.

Towards Compositional Hardware Model Checking Certification

Emily Yu* 
emily.yu2019@gmail.com

Nils Froleyks* 
nils.froleyks@jku.at

Armin Biere† 
biere@cs.uni-freiburg.de

Keijo Heljanko†§ 
keijo.heljanko@helsinki.fi

*Johannes Kepler University, Linz, Austria

†University of Freiburg, Freiburg, Germany

‡Helsinki Institute for Information Technology and

§University of Helsinki, Helsinki, Finland

Abstract—In this paper, we revisit and formalize temporal decomposition, as one of the most basic, widely-used and effective preprocessing techniques in hardware model checking. The main contribution is a certification framework for hardware model checking using temporal decomposition. Our approach enables generation of a single inductive invariant in a compositional way using inductive invariant certificates provided by existing certifying model checkers on the result of preprocessing a model through temporal decomposition. We implement and evaluate the method on hardware model checking competition benchmarks. The experiments confirm the effectiveness of temporal decomposition. The proposed certification approach makes it feasible to generate a generic proof for model checking and preprocessing.

I. INTRODUCTION

The study of compositional reasoning for safety-critical systems can be traced back to a few decades ago [1]. Compositional model checking breaks the model checking procedure down into several smaller problems, thus enabling faster and more efficient verification. For example, preprocessing techniques are widely used in current industry in combination with standard model checking algorithms.

Among these, temporal decomposition [2] is considered important in industrial hardware model checking [3], [4]. It computes an over-approximation of reachable states to efficiently find a set of transient signals that stabilize to their constant values during any possible execution. A sequence of transformations is applied to the design under verification, including time-shifting it from the reset states and elimination of transient logic. The verification problem thus consists of verifying that the property holds within the time frame the design has been shifted, and that the transformed circuit is safe. For the latter an existing model checker that can provide certificates is employed, such as k -induction [5], symbolic model checking using BDDs [6], and IC3/PDR [7].

While progress in verification using compositional reasoning continues, the number of certifiable approaches are limited [8], [9]. One central objective of verification is to develop a *standardised* method to generate machine-checkable proofs for certifying model checking [9]. This is especially crucial in safety-critical industrial environments, as a faulty processor

design can be extremely costly for a hardware manufacturer. Even though a number of single-engine model checkers are able to generate proofs, a key difficulty in this area is to produce a single generic certificate for complex verification pipelines. Furthermore, as in SAT solving [10]–[12], proof generation becomes rather involved when using preprocessing techniques, as the proofs from the preprocessors need to be lifted to proofs for the original model checking problem. This problem is exacerbated by the fact that the certificate given by a model checker can be more complex than a simple inductive invariant [8], [9]. For these reasons there is currently no viable industrial-strength model checker that provides certificates.

In this paper, we make a contribution toward this direction by revisiting temporal decomposition and developing a novel, practical, compositional framework for certifying the model checking result of the base engine and the employed preprocessing technique in a single proof. The distinguishing feature of our approach is to generate a *single witness circuit* as a certificate for the entire verification procedure, while related work [13], [14] relies on conceptually more complex deductive frameworks or has not been applied to industrial relevant hardware model checking problems that we are targeting. The approach in [15] also uses a deductive proof system for temporal decomposition that requires multiple independent parts to be checked, whereas our goal is to design a format such that only one witness circuit is checked, instead of a multi-stage proof. In theory, a single semantically simple proof format can be much easier to check by both untrusted and formally verified certificate checkers. Proof formats and checkers which follow the actual reasoning more closely will require to be adapted for every new technique used in the model checker. In contrast to [15], [16] that the underlying certificate provided by base model checkers is an inductive invariant, a witness circuit is more general. Moreover, different from [13]–[15], [17] as well as [18] which aims at handling more expressiveness, our work focuses on certifying *safety properties* which is the most prominent part in hardware model checking competitions and arguably in an industrial setting too. Additionally others have focused on either verifying model checkers themselves or lifting it directly to theorem proving [19]–[21]. Fully verifying model checkers can be a heavy task, as an update in the

Funded by FWF project W1255-N23, the LIT AI Lab funded by the State of Upper Austria, and Academy of Finland project 336092.

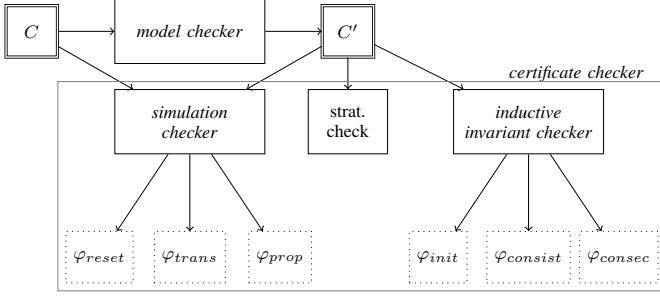


Fig. 1: An outline of the certification flow where C' is the certificate generated from the model checker (formally defined in Def. 12) and C is the original circuit. The certificate checker consists of three components and internally generates six SAT formulas and calls an underlying SAT solver.

optimization techniques often requires the entire procedure to be verified again. Lastly, similar problems have also been addressed in the context of software verification [22], [23].

The resulting certificate in our framework can be checked via six simple SAT checks and a polynomial check following the certification flow established in [8] as illustrated in Fig. 1, thus reducing trusting a PSPACE hard model checking flow into verifying a simple certificate circuit in co-NP. Our compositional approach breaks the formal proofs into manageable parts while enabling more certifiable techniques. For demonstration we focus on the k -induction algorithm as the base engine for the transformed circuit that provides a more complex certificate, as well as a BDD-based model checker. The reason is that our proposed flow requires model checkers to provide certificates for models with reset functions, which is technically rather involved to add to existing complex multi-engine model checkers, but this is a feature we want to encourage to be implemented.

Based on this, we have implemented a prototype certifying hardware model checker CHMC that performs temporal decomposition and provides a certificate that can be verified using a SAT solver. We also present two optimisations for detecting transient logic in a circuit. The experiments show our tool is able to solve 29 (out of 818) additional instances enabling temporal decomposition with the k -induction base engine, and an additional of 39 instances using the BDD backend. Our method can produce certificates for all instances solved by the model checker, and effectively verify them.

II. BACKGROUND

This paper extends the set of certifiable model checking techniques thus adopts the certificate format from [8], [9].

We assume a set of Boolean variables \mathcal{V} . A literal l is either a variable $l \in \mathcal{V}$ or its negation. We denote $\mathbb{B}(\mathcal{V})$ as the set of all Boolean functions over \mathcal{V} conveniently represented and with formulas (Boolean expressions). A *cube* is a non-contradictory set of literals. For $f \in \mathbb{B}(\mathcal{V})$ we write $f|_l$ to denote the formula after replacing all occurrences of l in f with \top and all occurrences of $\neg l$ with \perp . This naturally

extends to cubes (interpreted as conjunction of literals). We also write $\text{vars}(f)$ to denote the variables that occur in f .

In the following we use a symbolic representation of hardware circuits matching the notation of [8] summarised below. This definition has the advantage of being highly compatible with the AIGER format [24] used for hardware model checking in practice. Note that we use “ \Rightarrow ” for semantic implication, “ \rightarrow ” for syntactic implication and “ \equiv ” for semantic equivalence cf. [9].

Definition 1 (Circuit [9]). A circuit is a tuple $C = (I, L, R, F, P)$ where I is a set of Boolean input variables, L is a set of Boolean latch variables, $R = \{r_l(L) \in \mathbb{B}(L) \mid l \in L\}$ is a set of reset functions, $F = \{f_l(I, L) \in \mathbb{B}(I, L) \mid l \in L\}$ is a set of transition functions, and $P(I, L) \in \mathbb{B}(I, L)$ is the property formula.

We write $R(L') = \bigwedge_{l \in L'} (l \simeq r_l(L))$ for some $L' \subseteq L$, where “ \simeq ” is used for syntactic equivalence [25]. Furthermore L_i denotes a copy of L in the temporal direction, thus $U_m = \bigwedge_{i \in [0, m)} (L_{i+1} \simeq F(I_i, L_i))$ denotes an unrolling of length m .

Definition 2 (Stratified circuit [8]). Given a circuit $C = (I, L, R, F, P)$ with $R = \{r_l \mid l \in L\}$. The dependency graph G_R has latch variables L as nodes and contains a directed edge (a, b) from a to b iff $a \in \text{vars}(r_b)$ and $r_b \neq b$. The circuit is stratified iff G_R is acyclic.

Definition 3 (Stratified simulation [8]). Given two stratified circuits $C = (I, L, R, F, P)$ and $C' = (I, L', R', F', P')$ where $L \subseteq L'$. The circuit C is simulated by C' iff: (i) $r_l(L) \equiv r'_l(L')$ for $l \in L$; (ii) $f_l(I, L) \equiv f'_l(I, L')$ for $l \in L$; and (iii) $P'(I, L') \Rightarrow P(I, L)$.

Intuitively, a circuit is stratified if its reset functions are acyclic. The stratification assumption allows the designed certificate in [8] to be checked by simple SAT checks instead of having a one-alternation QBF check as in [9]. The stratified simulation relation can be therefore verified via three SAT checks as stated in the definition above and one polynomial time check for stratification of reset functions of C' .

Definition 4 (k -induction [9]). Given a circuit C , P is k -inductive in C iff: (i) $U_{k-1} \wedge R(L_0) \Rightarrow \bigwedge_{i \in [0, k)} P(I_i, L_i)$; and (ii) $U_k \wedge \bigwedge_{i \in [0, k)} P(I_i, L_i) \Rightarrow P(I_k, L_k)$.

We make use of k -induction [5] formulated as a combination of BMC check and a consecution check. It generalizes the concept of checking an *inductive invariant* which is equivalent to 1-induction where the invariant is simply the property.

III. OVERVIEW

Temporal decomposition helps to simplify model checking by removing parts of the circuit that are only needed for initialisation. Using it as a preprocessing technique, the model checking problem is decomposed into smaller sub-problems.

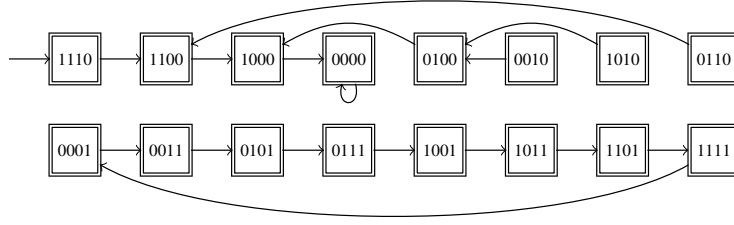


Fig. 2: State space of a 4-bit shift counter.

A series of transformation is applied to the circuit by time-shifting it and removing transient signals found via an over-approximation of the reachable states. We begin with an example to show that it can be quite useful and complementary to model checking techniques such as k -induction.

Example 1 (Shift Counter). Figure 2 shows the state transition diagram of the considered circuit, i.e., a shift counter over 4 bits, with an initial state 1110, where the least significant bit controls the operational mode and never changes. When this bit is set to 0, a left shift is performed; if it is set to 1, the other bits operate as a three-bit binary counter.

Consider the property that at least one bit is zero. As the diagram shows, it is 8-inductive (as the single bad state with all bits one is not reachable, but the longest path only having it as last state has 8 states). In the more general case, with an arbitrary large number of bits n , the inductive depth k is exponentially large (2^{n-1}) in n , for a k -induction-based model checker. The size of the certificate for k -induction using only the approach presented in [8] without temporal decomposition would then be in $\mathcal{O}(k \cdot n)$ and thus exponential too.

However, if we consider only the reachable part of the state space, all bits are transient signals as they all eventually stabilise to zero. Therefore we can use temporal decomposition to simplify the design by time-shifting it and removing transient logic. The time-shifting depth (later we call it the *duration*) is linear in n as it grows linearly in the number of bits. In this particular scenario, the model checking problem for the terminal part becomes trivial. We only need a BMC check for the initial $n - 1$ time frames. This leaves us with the problem: how to certify model checking with temporal decomposition?

The overall certification approach we propose is outlined in Fig. 3. For this example suppose we have a set of transient signals found that stabilise to constant values with duration 3. In practice the time-shifting namely *circuit-forwarding* of a design is implemented implicitly by computing the successor states originating from reset, however, with the objective of producing elegant and compositional proofs, we construct intermediate circuits forwarded by one transition only.

At each forwarding, the circuit gets unrolled by one time step from reset. For the sequence of circuits C_0, \dots, C_3 , a bounded model checker is used to verify that all initial states are good states. The last forwarded circuit along the pipeline (C_3) is simplified with transient elimination to obtain the factor circuit C'_3 , which is given to a base model checker (e.g., k -

induction, BDD or IC3/PDR) that also produces a certificate (i.e., a witness circuit [8]). We now construct a composite witness circuit certifying both the preprocessing algorithm for the transients and the safety property. We then build a sequence of backward witness circuits while adding the BMC check for the initial states each time. At each step W_i is a witness circuit of C_i . In the end, we get W_0 as the final witness circuit with an inductive invariant for the entire procedure.

The final witness can be checked by an external proof checker [8]. If the check passes, the original circuit is guaranteed to be safe. It is thus not necessary to trust the correctness of the presented framework nor its implementation. The formal proofs provided in the following sections serve to show that if the original circuit is safe, a valid certificate can be produced.

IV. TEMPORAL DECOMPOSITION

As one of our contributions we revisit temporal decomposition by defining a precise formalism and proving its correctness, which is an improvement over the theory in [2]. We show that our method is complete and will provide a valid certificate whenever temporal decomposition is employed.

In practice, temporal decomposition uses ternary simulation [26] to find transient signals [2]. As generalization we define *cube semantics* and the notion of *cube simulation*, that subsumes ternary simulation as well as other optimisations. We make use of this in our implementation (see Section VI).

Definition 5 (Cube simulation). Given a circuit $C = (I, L, R, F, P)$, then a cube simulation $c_0, \dots, c_\delta, \dots, c_{\delta+\omega}$ for $\delta, \omega \in \mathbb{N}$ is a sequence of cubes over latch variables L such that,

- $R(L) \Rightarrow c_0$.
- For $i \in [0, \delta + \omega)$ we require validity of $c_i \wedge (L' \simeq F(I, L)) \Rightarrow c'_{i+1}$ (L' and c'_{i+1} denote primed copies).
- In addition, it is called a cube lasso iff $c_{\delta+\omega} \wedge (L' \simeq F(I, L)) \Rightarrow c'_\delta$.

Note that for $\delta = 0$ and $\omega = 0$, it would simply be a self-loop. This symbolic definition of cube simulation could be implemented directly by a symbolic engine. However, ternary simulation is more appropriate for industrial benchmarks [2].

For brevity we omit (the natural) formal definitions here, but remark that a “bounded version” of cube simulation is subsumed by model checking and in particular ternary simulation does not yield more transients than those produced by Boolean constraint propagation in the SAT solver. As a result, for SAT based bounded model checkers (and therefore

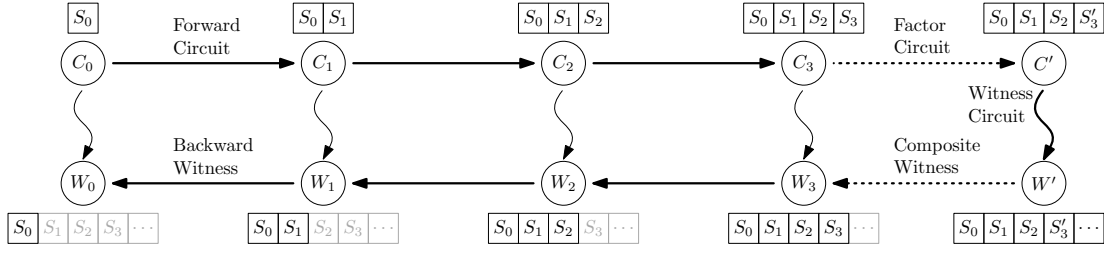


Fig. 3: An outline of our certification framework (for duration of 3).

k -induction-based model checkers on satisfiable instances), there is hardly any gain using temporal decomposition.

Proposition 1. *Bounded model checking subsumes bounded cube simulation.*

Under cube simulation, transient signals can be found by identifying those that stay constant from a certain point onwards along a cube lasso.

Definition 6 (Transients via cube lasso). *Given a circuit C and a cube lasso $c_0, \dots, c_\delta, \dots, c_{\delta+\omega}$. A set of transient signals T is a cube over latches that satisfies $c_i \Rightarrow \bigwedge_{l \in T} l$ for $i \in [\delta, \delta+\omega]$.*

For the purpose of temporal decomposition, we could always time-shift the circuit by δ , however, it can make model checking of the final circuit easier to reduce this value as much as possible. For a set of transients found via cube lasso, we formally define the *unstable duration* of a circuit. It is the smallest value for which the last transient becomes constant.

Definition 7 (Unstable duration). *Given a cube lasso $c_0, \dots, c_\delta, \dots, c_{\delta+\omega}$ and a set of transients T , the unstable duration $d \leq \delta$ is the lowest index that satisfies $c_i \Rightarrow \bigwedge_{l \in T} l$ for $i \in [d, \delta + \omega]$.*

For the rest of the paper, we simply refer to it as duration.

By taking the disjunction of all cubes from the duration onwards, we get an over-approximation of all the reachable states in the time-shifted circuit.

Definition 8 (Cube loop invariant). *Given a cube lasso $c_0, \dots, c_\delta, \dots, c_{\delta+\omega}$, and a set of transients T with a duration d . The cube loop invariant ϕ_T is defined as:*

$$\phi_T = \bigvee_{i \in [d, \delta+\omega]} c_i.$$

An immediate observation is that the cube loop invariant is simply the inductive invariant that implies the stabilised transients in the time-shifted circuit by the duration. We formalise this in the following lemma.

Lemma 1. *Given a circuit C , a cube lasso $c_0, \dots, c_d, \dots, c_\delta, \dots, c_{\delta+\omega}$, and a set of transients T with a duration d . Let C' be the resulting circuit of applying circuit forwarding to C iteratively d times. The cube loop invariant is an inductive invariant of C' for $\bigwedge_{l \in T} l$.*

To formalize “time shifting” we introduce the notion of *stable variables* which have the same values throughout the entire execution and do not appear in any other transition function nor in the property.

Definition 9 (Stable variable). *Given a circuit $C = (I, L, R, F, P)$, the set $\Lambda \subseteq L$ is a set of stable variables if for all $l \in \Lambda$, the following holds: (i) $f_l = l$; (ii) for all $l' \in L \setminus \{l\}$, $l \notin \text{vars}(f_{l'})$; and (iii) $l \notin \text{vars}(P)$.*

This simple purely syntactic definition of stable variables is the weakest condition that allows us to avoid a spurious exponential blowup during circuit forwarding (Def. 10). Note that here identifying stable variables is a simple polynomial time check. Replacing it with stronger conditions, such as cone-of-influence reduction [27], after every forward circuit construction would potentially yield a smaller certificate, which we leave to future work.

A forward circuit is constructed based on a given circuit by copying the set of active variables (i.e., non-stable), and updating the resets of the original active variables to one transition ahead using oracles (uninitialised latches) instead of inputs. The formal definition is given below.

Definition 10 (Forward circuit). *Given a circuit $C = (I, L, R, F, P)$ with a set of stable variables $\Lambda \subseteq L$ (we refer to $A = L \setminus \Lambda$ as the set of active variables). The forward circuit $C' = (I', L', R', F', P)$ is defined as follows:*

- $L' = L \cup I' \cup A'$ where I' and A' are copies of I and A .
- $R' = \{r'_l \mid l \in L'\}$:
 - For $l \in \Lambda \cup A'$, $r'_l = r_l(I, \Lambda \cup A')$.
 - For $l \in I'$, $r'_l = l$.
 - For $l \in A$, $r'_l = f_l(I', A')$.
- $F' = \{f'_l \mid l \in L'\}$:
 - For $l \in L$, $f'_l = f_l(I, L)$.
 - For $l \in I' \cup A'$, $f'_l = l$.

An alternative to Def. 10 is to simply copy all latches when forwarding a circuit, however, the resulting circuit would be exponential in the duration when doing so iteratively. In the following we show that the forward circuit is stratified

Lemma 2. *Given a stratified circuit C . Its forward circuit C' is also stratified.*

Proof. Based on the reset function definition in Def. 10, the reset functions of $\Lambda \cup A'$ are the same as in $R(L)$, which

are acyclic. The variables in I' are uninitialised, thus do not depend on the rest of the variables. As $R'(A)$ only uses variables distinct from A , we conclude C' is stratified. \square

For the forward circuit, we proceed to simplify it by removing the set of transients that have been found. The resulting simplified circuit (namely *factor circuit*) is formally defined in the following definition.

Definition 11 (Factor circuit). *Given a circuit $C = (I, L, R, F, P)$ and a set of transients T from cube simulation. Its factor circuit is a tuple $C|_T = (I, L', R', F', P')$ such that,*

- $L' = L \setminus \text{vars}(T)$.
- $R' = \{r_l|_T \mid l \in L'\}$.
- $F' = \{f_l|_T \mid l \in L'\}$.
- $P' = P|_T$.

In the following proposition, we state that the conjunction of the factor property $P|_T$ and $\bigwedge_{l \in T} l$ (i.e., the transients are constant) implies the original property. The same follows for reset and transition functions.

Proposition 2. $(P|_T \wedge \bigwedge_{l \in T} l) \Rightarrow P$, $(R|_T \wedge \bigwedge_{l \in T} l) \Rightarrow R$, and $(F|_T \wedge \bigwedge_{l \in T} l) \Rightarrow F$.

An important observation we make is that the inductive depth of the property does not increase after temporal decomposition, which is later confirmed in our experimental evaluation. However, we have already seen in Example II an exponential reduction in the inductive depth. We formally prove it and summarise it in the following theorem, which follows directly from Lemmas 3, and 4.

Theorem 1. *Given a circuit C where the property is k -inductive. Let $C|_T$ be the circuit resulting from temporal decomposition of C . Its property is k -inductive.*

Lemma 3. *Given a circuit $C = (I, L, R, F, P)$ where the property is k -inductive. Let $C' = (I, L', R', F', P')$ be the factor circuit. P' is k -inductive in C' .*

Proof. We do a proof by contradiction by assuming P' is not k -inductive in C' . First we consider the case where the BMC check fails in C' (i.e., $U'_{k-1} \wedge R'(L'_0) \wedge \neg \bigwedge_{i \in [i, k]} P'(I_i, L'_i)$ has a satisfying assignment). By Lemma 1, the transients stay constant in C , and by Propositions 2, we can construct a satisfying assignment for $U_{k-1} \wedge R(L_0) \wedge \neg \bigwedge_{i \in [i, k]} P(I_i, L_i)$.

This contradicts our assumption. The second scenario where the consecution check fails in C' follows the same logic. \square

Lemma 4. *Given a circuit $C = (I, L, R, F, P)$ where the property is k -inductive. Let $C' = (I, L', R', F', P)$ be the forward circuit. P is k -inductive in C' .*

Proof. We provide a proof by contradiction assuming P is not k -inductive in C' . Similarly we assume $R'(L'_0) \wedge U'_{k-1} \wedge \neg \bigwedge_{i \in [0, k]} P(I_i, L_i)$ has a satisfying assignment. By Def. 10 the

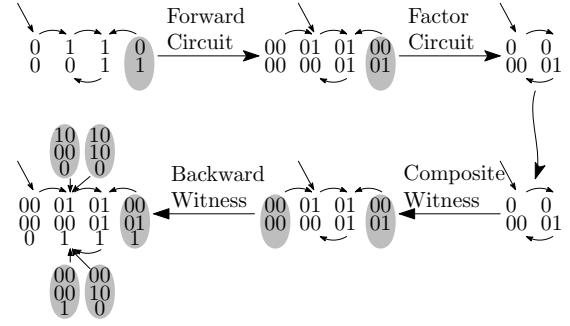


Fig. 4: An illustration of the overall certification flow for the delayed clock example. Initial states are marked with an additional arrow; bad states are marked gray. In the original circuit (top left), the lower bit is oscillating, and the upper bit is the enabler bit.

same satisfies $R(L_0) \wedge U_k \wedge \neg \bigwedge_{i \in [1, k]} P(I_i, L_i)$. This implies that a bad state is reachable from the initial states thus contradiction. We then consider the consecution check fails in C' such that $U'_k \wedge \bigwedge_{i \in [0, k)} P(I_i, L_i) \wedge \neg P(I_k, L_k)$ has a satisfying assignment. By Def. 10 the transition function stays the same for the common latches thus the same assignment satisfies the formula $U_k \wedge \bigwedge_{i \in [0, k)} P(I_i, L_i) \wedge \neg P(I_k, L_k)$. Therefore we have reached a contradiction. \square

V. CERTIFICATION

In this section, we present a compositional certification framework that is complete. Along the model checking procedure with temporal decomposition, a certificate can be automatically generated by the model checker following the format defined in this section.

Example 2. *Consider the scenario of a delayed clock (Fig. 4). The clock has one bit (the bottom bit in the diagram) that oscillates between zero and one after the enabler bit (first bit) is set to one. There is only one initial state where the clock is set to zero and enabler bit not set. A state is bad if the clock is high without being enabled.*

After preprocessing, a base model checker is called for verifying the factor circuit (In Fig. 4 the property of the factor circuit is already an inductive invariant thus the factor circuit is simply the certificate). It is required to provide a certificate that is then used to build the final certificate. We give a formal definition of the general certificate format below.

Definition 12 (Witness circuit). *Given a circuit C , a witness circuit $W = (I, M, S, G, Q)$ of C satisfies the following:*

- W simulates C under stratified simulation relation.
- Q is an inductive invariant in W .

Since the method is compositional, with the witness circuit of the factor circuit produced from the model checker, we combine it with the loop invariant for cube lasso to construct a composite witness circuit that certifies both.

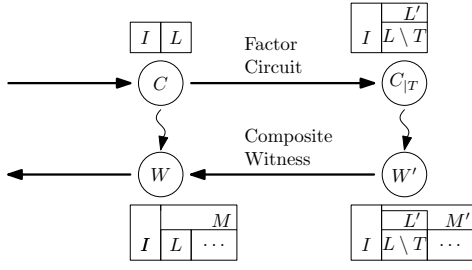


Fig. 5: Constructing composite witness circuit.

Definition 13 (Composite witness circuit). *Given a stratified circuit $C = (I, L, R, F, P)$ with the cube loop invariant ϕ_T for the set of transients T , and its factor circuit $C|_T = (I, L', R', F', P')$ with its witness circuit $W' = (I, M', S', G', Q')$. The composite witness circuit $W = (I, M, S, G, Q)$ is constructed as follows:*

- $M = L \cup (M' \setminus L')$.
- $S = \{s_l \mid l \in M\}$:
 - For $l \in L$, $s_l = r_l$;
 - For $l \in M' \setminus L'$, $s_l = s'_l$.
- $G = \{g_l \mid l \in M\}$:
 - For $l \in L$, $g_l = f_l$;
 - For $l \in M' \setminus L'$, $g_l = g'_l$.
- $Q = \phi_T(L) \wedge Q'(I, M')$.

Intuitively, in the composite witness circuit we add back the previously eliminated transients. The new property simply combines the cube loop invariant from cube simulation and the property of the witness circuit of the factor circuit, which is an inductive invariant in the composite witness circuit.

Theorem 2. *Given a stratified circuit C , its factor circuit $C|_T$ with an inductive invariant ϕ_T , a witness circuit W' of $C|_T$, and the composite witness circuit W . Then W is a witness circuit of C .*

Proof. We show that W simulates C . The factor circuit $C|_T$ is obviously stratified as C is stratified by Def. 11. The witness circuit W' is also stratified. The reset functions of L remain the same in W , and $R(L)$ do not contain latches in $M' \setminus L'$. Thus W is stratified. Based on Def. 13, the common latches $L \subseteq M$ and inputs are the same in both circuits. As the reset functions and transition functions of L remain the same, this satisfies the reset check and transition check. Since W' is a witness of $C|_T$, we have $Q' \Rightarrow P'$, where $P' \equiv P|_T$ by Def. 11. Since ϕ_T is an inductive invariant for transients and by Lemma 1, we have $\phi_T \Rightarrow \bigwedge_{l \in T} l$. By Proposition 2, we have $Q \Rightarrow P$. Therefore W simulates C .

We then show the BMC check passes ($S(M) \Rightarrow Q(I, M)$) by assuming that $S(M)$ and thereby $R(L) \wedge S'(M' \setminus L')$ holds, and shall proceed to the conclusion $\phi_T(L) \wedge Q'(I, M')$, by Def. 13. We begin with $R(L)$ and may deduce $\phi_T(L)$ immediately since Lemma 1 applies to C and the circuit is in its reset state.

From this point on, our attention will be on the subset of L that is free of transients, L' by Def. 11. We have $R(L') = R'(L')$ again by Def. 11, and then $R'(L') = S'(L')$ since W' simulates $C|_T$, which by Def. 3 implies the resets to be the same. We combine this with the second half of our initial assumption to arrive at $S'(M')$. The witness circuit W' is therefore at reset, and its inductive invariant $Q'(I, M')$ holds. We conclude with $Q(I, M)$.

We make the observation that $G(I, L) \equiv F(I, L)$, $F(I, L \setminus T) \equiv F'(I, L \setminus T) \equiv G'(I, L \setminus T)$ by Def. 12 and Def. 11. The latter together with Def. 13 gives us $G(I, M') \equiv G'(I, M')$. Assume a fixed satisfying assignment for $V_1 \wedge Q(I_0, M_0)$, where V_1 is the unrolling of length 1 in W . By the observation above, this also satisfies U_1 and V'_1 which are the unrollings of C and W' respectively. By Def. 13 the assignment satisfies $\phi_T(L_0) \wedge Q'(I_0, M'_0)$ which are inductive invariants in C and W' respectively. We thus have $\phi_T(L_1) \wedge Q'(I'_1, M'_1)$. As we have shown the inductiveness of Q with the BMC check and consecution check, together with the simulation relation, we conclude W is a witness circuit of C . \square

We now introduce *backward witness circuit* built based on a given witness circuit for one backward step, as illustrated in Fig. 6. Intuitively, at each backward step, the backward witness circuit certifies the BMC check for the initial states of the corresponding circuit C while maintaining and delaying the behaviours of the given witness circuit W' by one step.

This is achieved by using one bit b which becomes constant true exactly one step after initialisation. At reset, W has the same values as C , with the additional latches holding the reset values from W' ; once b is set, it operates as W' . Recall that when constructing the forward circuit, an additional copy of inputs and active latches is added for copying the initial values of those in the given circuit. Since input values are non-deterministic, this needs to be recovered when constructing W such that the inputs are copied to ensure matching values.

Definition 14 (Backward witness circuit). *Given a stratified circuit $C = (I, L, R, F, P)$, and its forward circuit $C' = (I, L', R', F', P)$. Let $W' = (I, M', S', G', Q')$ be the witness circuit C' . The backward witness circuit $W = (I, M, S, G, Q)$ is defined as follows:*

- 1) $M = M' \cup \{b\}$.
- 2) $S = \{s_l \mid l \in M\}$ such that:
 - For $l \in L$, $s_l = r_l$.
 - For $l \in M' \setminus L$, $s_l = s'_l$.
 - $s_b = \perp$.
- 3) $G = \{g_l \mid l \in M\}$ such that:
 - For $l \in L$, $g_l = f_l$.
 - For $l \in M' \setminus L$, $g_l = \text{ite}(b, g'_l, s'_l)$.
 - For $l' \in L' \setminus L$, $g_{l'} = \text{ite}(b, l', l)$ where l is the literal with the same index in $I \cup A$ as l' in $I' \cup A'$ ($= L' \setminus L$).
 - $g_b = \top$.
- 4) $Q = \bigwedge_{i \in [0,3]} q_i$, where

- $q_0 = P(I, L)$.
- $q_1 = b \rightarrow Q'(I, M')$.
- $q_2 = \neg b \rightarrow R(L)$.
- $q_3 = \neg b \rightarrow S'(M' \setminus L)$.

We can thus obtain a witness circuit that certifies (i) the property holds at reset for forward circuits and (ii) the transformed property holds in the factor circuit. In an iterative manner eventually we can construct a witness circuit for the entire pipeline illustrated in Fig. 3. Here the property consists of four subproperties: q_0 is the original property; q_1 states that the property from W' needs to hold when b is set; q_2 states that b is false only at initialisation; and q_3 states that all latches except those in L need to hold the reset values as in W' .

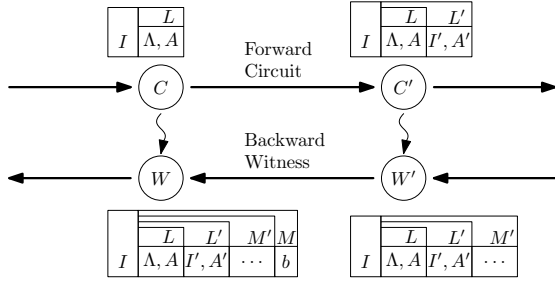


Fig. 6: An illustration of the latch contents of circuits. C is a time-shifted circuit with latches of Λ (stable variables) and A (active variables). We obtain C' by adding a copy of I' and A' which are used for unrolling the initial states. W' is the witness circuit simulating C' . In addition to the common latches L' , M' can also include a set of unknown latches (potentially generated from the model checker). W is the backward witness circuit, containing all latches from M' and an additional bit b used for an initialisation step.

By Def. 12 the backward witness needs to pass the six SAT checks for the stratified simulation relation and the witness inductiveness, as well as the stratification check of its reset functions. We summarise this in the following theorem.

Theorem 3. *Given a stratified circuit C where the property holds in all reset states, the forward circuit C' with its witness circuit W' , and the backward witness circuit W as defined in Def. 14. Then W is a witness circuit of C .*

Proof. First we show W is stratified. By Def. 14, $S(L) \equiv R(L)$, therefore stratified and does not depend on latches outside L , and $S(\{b\})$ is independent of other variables. Furthermore, for the rest of latches $M' \setminus L$, the reset functions are the same as in W' therefore stratified. We conclude that W is stratified. By Def. 14, $L \subseteq M$ and the inputs I stay the same in W . Based on Def. 14, for the common latches L , their reset function and transition function are the same as in C , and $q_0 \equiv P$. Therefore W simulates C .

We proceed to prove that the BMC check passes in W , i.e., $S(M) \Rightarrow Q(I, M)$. Since b is false at reset, q_1 is trivially satisfied. The reset also directly implies $R(L)$ and $S'(M' \setminus L)$, which gives us q_2 and q_3 . We have q_0 since the property holds

at reset in C . We then move on to prove the consecution check passes ($V_1 \wedge Q(I_0, M_0) \Rightarrow Q(I_1, M_1)$). We consider two cases based on the value of b_0 and begin with the case $b \equiv \top$. By Def. 14, since b always transitions to true, $q_2(I_1, L_1)$ and $q_3(I_1, L_1)$ are trivial. Additionally, $q_1(I_1, M_1)$ implies $Q'(I_1, M'_1)$ which by Def. 12 implies $P(I_1, L_1)$, as C and C' share the same property. Thus we only need to show $Q'(I_1, M'_1)$ holds after one transition to satisfy $q_1(I_1, M_1)$. Consider a fixed satisfying assignment for $V_1 \wedge Q(I_0, M_0)$ and b is true. We show that the same assignment satisfies V'_1 (the unrolling of W'). For latches in $M' \setminus L'$ this follows directly from the definition. By Def. 12 and Def. 10 L has the same transition function in all 4 circuits (Fig. 6). The latches in $L' \setminus L$ stay constant which matches Def. 10. The rest follows from Def. 12. With this, $q_1(I_0, M'_0)$ gives us $Q'(I_0, M'_0)$ which is an inductive invariant in W' . $Q'(I_1, M'_1)$ follows.

Now consider a satisfying assignment where b is false thereby $R(L_0)$ and $S'(M'_0 \setminus L_0)$. By Def. 10, for the latches in L the transition function matches the reset function in C' , thus we get $R'(L_1)$. The latches in $L' \setminus L$ copy the values of $I \cup A$. By Def. 10 $R'(I'_1)$ holds trivially and $R'(A'_1)$ follows from $R(L_0)$. We get $R'(L'_1 \setminus L_1)$. Together with the previous result we have $R'(L'_1)$, which by Def. 3 yields $S'(L'_1)$. The reset for the rest of the latches $M' \setminus L'$ follows directly from Def. 14 and we get $S'(M'_1)$. Since the inductive invariant of W' holds in its reset we conclude with $Q'(I_1, M'_1)$. \square

The above concludes the description and formal proof of our certification framework. To perform temporal decomposition on a given circuit under verification, we use cube simulation to find a set of transient signals and determine the duration d . The original circuit is then time-shifted by constructing forward circuits iteratively d times. After eliminating transient signals we obtain a simplified circuit that is then given to a base model checker for verifying the simplified safety property while producing a witness circuit. We construct a final witness circuit by building backward witness circuit d times again in an iterative manner. This final witness circuit serves as a single certificate for both preprocessing and backend model checking. It can be easily verified by an external certifier.

VI. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We implemented the method proposed in Section IV and V in the certifying model checker CHMC [28]. Our model checker CHMC supports two configurations for the backend solver performing certification on simplified models (i.e., factor circuits) after preprocessing: k -induction and BDD. As factor circuits use reset functions, we extended the k -induction-based open-source model checker MCAIGER [29] to support reset functions and also extended the tool presented by the authors of [8] to generate witness circuits. For the BDD-based configuration, we used the simple BDD-based model checker AIGTRAV developed by one of the authors of this paper at JKU Linz which already supports reset functions. We reencoded the convergent BDD to an AIG encoding the membership in the set of reachable states, which is simply the inductive invariant. The witness circuit is the factor

circuit having the inductive invariant as the new property. Our model checker CHMC then performs witness composition and backwinding to generate the final witness verified by CERTIFAIGER++ [30].

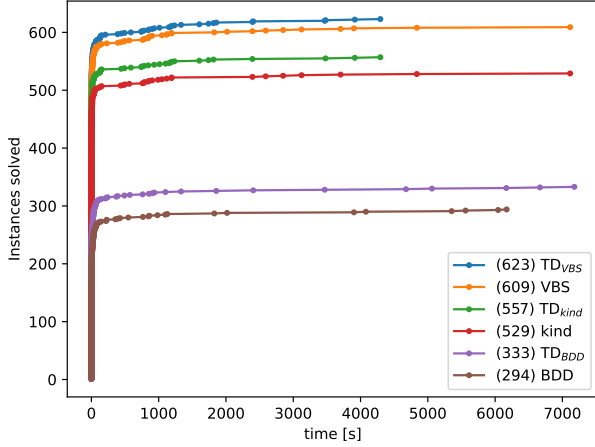


Fig. 7: Comparison for temporal decomposition with k -induction (TD_{kind}) and BDDs (TD_{BDD}) and the base engines on HWMCC10 instances. We show the results of the best performing solver for each instance with temporal decomposition (TD_{VBS}) as well as the best performing solver without it (VBS).

In principle our approach naturally works for any model checker that produces a certificate (e.g., IC3/PDR, interpolation), however, currently the available implementations for these do not support reset functions nor do they follow the pre-defined certificate format. It seems non-trivial to do so for certain model checkers. Note that a simple constraint-based construction to eliminate reset functions does not solve the problem, as the produced certificate needs to be lifted for the overall certification. It is however natural to modify symbolic model checkers to support reset functions, which we strongly encourage for extending the certification capability to a larger set of model checking tools.

State-of-the-art model checkers such as ABC [3] use additional preprocessing techniques and are multi-engines for efficient verification, which are hard to separate from each other thus making it difficult to certify a complete model checker with a multitude of preprocessing algorithms. However, we have shown how the certification of an important preprocessing technique, namely temporal decomposition, can be certified in a compositional manner. We believe that similar compositional certification techniques can be identified for other preprocessing techniques. In particular the approach used here can be used for any other preprocessing technique subsumed by cube simulation.

To increase trust in our tool we generated 42 million random circuits [24] and checked all produced certificates. We used the HWMCC'10 benchmarks [31] and additionally scaled the shift counter example for further evaluation. All experiments were conducted in parallel on 32 nodes of a cluster. Each node has

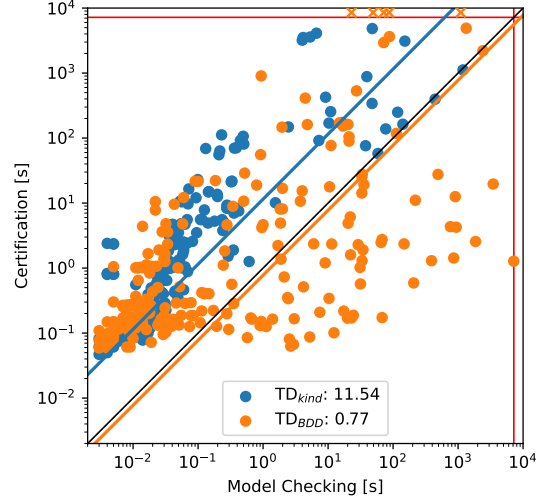


Fig. 8: Certification vs. model checking time.

access to two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled) and 128 GB main memory. We allocate 8 instances to every node with a timeout of 2 hours and memory limit of 16 GB per instance.

We studied the impact of the preprocessing technique on k -induction and BDD-based model checking. The preprocessing (ternary simulation and circuit transformation) terminated within 200ms on all instances. Fig. 7 displays the number of instances solved over time by the four configurations. We observe that CHMC with k -induction (TD_{kind}) outperforms the rest. It verified a total of 557 instances, among which 235 are UNSAT. In particular, compared with base MCAIGER (kind), we gain 29 UNSAT instances. We further inspected the inductive depths of the original circuits and their simplified ones; the results match the claim of Theorem 1. In our benchmark set k was reduced for 19 instances. As for the BDD configuration (TD_{BDD}), CHMC performed well on SAT cases, solving an additional of 39 instances. These results clearly demonstrate that simplifying transient logic in circuits can be beneficial for model checking.

We now proceed to evaluate the certification framework on both k -induction and BDD configurations. Table I summarises the results obtained on certifying the HWMCC10 benchmarks, where we display a subset of interesting instances (sorted by witness size). CHMC was able to produce certificates for all 235 unsatisfiable instances that it was able to model-check. For the k -induction variant, all certificates were successfully verified, with a mean ratio of certification time and model checking time of 11.54, which can be further inspected in Fig. 8. This shows the practicality of our method.

On average, duration d tends to be small, which can be partially explained as the starting sequence in designs playing a role. On the BDD variant, the average ratio of certification and model checking time is 0.77. However, 5 instances experienced time-outs when verifying the certificates produced by

TABLE I: Columns report cycle length (ω), stem length of cube lasso (δ), duration (d), no. transients found (τ), inductive depth of factor circuits (k), original model size ($\#_M^K$), model checking time ($t_M(s)$), certification time ($t_C(s)$), and certificate size ($\#_M^K$). All circuit sizes ($\#_M^K$) are measured in no. thousands of gates including latches and inputs. The certification time is the total of generation time and SAT solving time. We report three mean values on different subsets of benchmarks. Mean TD_{BDD} and TD_{kind} are computed over 223 and 235 instances respectively, where certification checks terminated. Mean TD_{\cap} concerns the intersection of both sets, from which we display 20 instances that produced the biggest certificate sizes for TD_{kind} . Additionally, we list 5 instances at the bottom for which the BDD-based algorithm succeeded in model checking the simplified circuit, but produced certificates that could not be checked within the time limit.

	Decomposition				Model		TD_{kind}			TD_{BDD}		
	ω	δ	d	τ	k	$\#_M^K$	t_M	t_C	$\#_M^K$	t_M	t_C	$\#_M^K$
Mean $TD_{BDD}(223)$	1.9	5.8	0.3	4.9	1.7	4	–	–	–	100.29	77.71	58
Mean $TD_{kind}(235)$	2.1	8	0.9	46.3	4.1	7	10.40	120.06	93	–	–	–
Mean $TD_{\cap}(157)$	2.2	5.7	0.4	5.1	1.7	4	0.19	46.40	35	65.33	46.69	36
pj2002	1	4	2	2	9	37	4.13	3522.82	1757	33.09	25.56	129
pj2003	1	4	2	2	9	37	4.00	3203.89	1757	33.57	27.57	129
cmuperiodic	1	1	1	1	96	2	16.41	173.04	480	0.05	0.52	5
mentorbm1p09	1	39	1	122	2	36	0.45	92.00	331	0.18	22.17	123
mentorbm1or	1	39	1	122	1	36	0.32	21.24	123	0.10	21.56	123
nusmvreactorp4	1	1	1	1	13	1	0.20	12.59	85	7177.65	1.28	15
neclafp5001	1	10	10	21	0	2	0.05	1.34	79	0.03	1.37	79
neclafp5002	1	10	10	21	0	2	0.05	1.34	79	0.03	1.40	79
bobsynthand	1	38	1	1	0	19	0.12	7.85	73	0.02	7.79	73
139464p0	1	4	1	2	0	21	0.09	12.05	45	0.06	12.16	45
bj08amba4g5	1	6	0	0	3	14	0.13	69.07	42	219.68	11.06	14
139463p0	1	4	1	2	0	15	0.07	9.72	33	0.04	9.60	33
bj08amba3g62	1	4	0	0	3	10	0.10	22.99	31	1.85	4.92	10
139454p0	1	4	1	2	0	13	0.06	4.73	30	0.06	4.78	30
139462p0	1	4	1	2	0	10	0.06	4.82	23	0.03	4.81	23
139453p0	1	4	1	2	0	9	0.05	3.51	21	0.03	3.58	21
bj08amba5g82	1	5	0	3	0	20	0.07	16.49	20	1.98	16.85	20
139444p0	1	4	1	2	0	8	0.05	3.60	19	0.03	3.62	19
pdtvisvsal6a04	5	7	0	0	2	7	0.05	2.57	18	1.95	147.20	258
pdtvisvsal6a00	5	7	0	0	2	7	0.05	2.62	18	0.02	1.12	7
pdtswvtma6x4p3	1	7	0	0	44	2	77.00	138.26	138	71.75	to	13534
pdtswvtma6x4p2	1	7	0	0	37	2	37.69	76.23	116	86.86	to	13327
nusmvreactorp3	1	1	1	1	4	1	0.04	2.30	24	1100.38	to	6588
pdtvisvsar04	5	7	0	0	2	2	0.02	0.61	8	22.97	to	11074
pdtvisminmax2	1	3	0	0	2	1	0.02	0.14	2	49.29	to	2776

TABLE II: Comparison of kind and TD_{kind} for the *Shift Counter*. The n column reports no. of bits. Circuits sizes ($\#_M$) measured in number of gates. Compared with TD_{kind} , as n increases the model checking quickly becomes hard for the k -induction only engine and it experiences time-outs (to), while TD_{kind} was still able to solve the instances within reasonable time.

n	Decomposition				Model		kind			TD_{kind}		
	ω	δ	d	τ	k	$\#_M$	t_M	t_C	$\#_M$	t_M	t_C	$\#_M$
2	1	1	1	2	2	4	0.003	0.051	66	0.011	0.053	23
3	1	2	2	3	4	13	0.003	0.053	250	0.010	0.057	79
4	1	3	3	4	8	22	0.004	0.066	739	0.010	0.057	159
5	1	4	4	5	16	31	0.005	0.093	1974	0.010	0.061	263
6	1	5	5	6	32	40	0.008	0.165	5045	0.013	0.069	391
7	1	6	6	7	64	49	0.031	0.361	12764	0.014	0.069	543
8	1	7	7	8	128	58	0.111	0.987	32883	0.010	0.080	719
9	1	8	8	9	256	67	0.461	3.460	88618	0.011	0.086	919
10	1	9	9	10	512	76	2.409	14.830	255649	0.011	0.084	1143
11	1	10	10	11	1024	85	13.711	59.410	799128	0.012	0.105	1391
12	1	11	11	12	2048	94	86.877	164.968	2698130	0.013	0.108	1663
13	1	12	12	13	4096	103	498.395	501.894	9693060	0.014	0.125	1959
14	1	13	13	14	8192	112	2686.540	2217.410	36368300	0.011	0.108	2279
1000	1	999	999	1000	2 ⁹⁹⁹	8986	to	to	to	5.902	1809	11996000

TABLE III: Optimised ternary simulation of three configurations: base version, counter stagnation, cube subsumption with counter stagnation. The number of latches is denoted by $\#_L$ ($\#_M$ is the total number of gates including inputs and latches), and t is time taken (seconds) by ternary simulation. Highlighted numbers indicate the best performing variant for each instance.

	$\#_M$	$\#_L$	ternary simulation				counter stagnation				cube subsumption			
			ω	δ	τ	t	ω	δ	τ	t	ω	δ	τ	t
bob12m04	269935	43950	2	9	199	0.06	2	9	199	0.07	2	6	153	0.05
6s376r	113207	4708	131072	262160	145	766.34	1	8198	116	16.30	1	2056	116	4.20
bob12m15	2855	448	12288	10379	133	1.62	12288	10379	133	1.60	3	4108	12	0.40
bobsmnut1	6301	644	1024	259	107	0.21	1024	259	107	0.19	1024	81	106	0.18
shift1add	174	27	1	262145	20	1.20	1	262145	20	1.25	1	2056	1	0.01
6s47	4950	815	to	to	to	to	2	262167	8	35.63	2	2085	6	0.30
6s100	763775	97598	to	to	to	to	1	2053	36	37.53	1	2053	36	41.48
6s107	25447	1568	to	to	to	to	1	32799	746	16.93	1	2079	746	1.18
6s149	112623	12781	to	to	to	to	1	20497	4	47.81	1	5137	4	12.88
6s342rb ₁₂₂	394016	56838	to	to	to	to	to	to	to	to	192	46080	1894	354.60
6s202b41	604375	68881	to	to	to	to	1	4136	8574	45.71	1	2088	8574	28.35
6s204b16	237048	28986	to	to	to	to	1	4136	4034	19.33	1	2088	4034	13.51
6s205b20	603985	68842	to	to	to	to	1	4136	8727	46.56	1	2088	8727	28.08
6s355rb ₈₇₄₀	179445	15091	to	to	to	to	1	8466	221	37.07	1	3346	221	15.51
6s400rb ₇₈₁₉	180067	14665	to	to	to	to	1	8466	221	36.96	1	2322	221	11.00
cucnt128	1272	128	to	to	to	to	to	to	to	to	1	61440	0	1.82
cucnt32	312	32	to	to	to	to	to	to	to	to	1	1054	0	0.01

the BDD backend. Indeed, the BDD algorithm can produce large certificates due to the nature of using the exact set of reachable states as the inductive invariant.

We further investigate the shift counter example by scaling the number of bits (n). The results obtained are reported in Table II. As expected, we found that temporal decomposition significantly simplifies model checking, whereas MCAIGER quickly experiences timeouts. Note that in this particular example, the value of k simply decreases to 0 on the simplified model which becomes trivial after temporal decomposition.

Optimised Ternary Simulation To find transient signals we initially implemented a base version of ternary simulation. We now present two optimisations, also subsumed by cube simulation: (i) *Counter stagnation*: if the current cube size does not decrease in a few thousand steps, we observe a *stagnation* in the search. We thus remove a random latch that has flipped sign between the last two cubes. In case we remove a latch that is part of a large binary counter in the design, all more significant bits will be removed in a linear number of simulation steps. This technique can therefore achieve an exponential reduction in simulation steps. (ii) *Cube subsumption*: the cube lasso can terminate in a cube which implies a previous cube under the transition relation (Def. 5). Ternary simulation on the other hand utilises a hash map to terminate when an exact match of a cube is visited a second time. With cube subsumption it terminates when the current cube is a (not necessarily proper) superset of a previously encountered cube. For this we implement a forward-subsumption algorithm utilising a one-watch-literal data structure [32].

We compare it to two configurations with different levels of optimisation on 20,815 instances including all HWMCC benchmarks (2007-2020) [33]. The optimised versions can miss transients however it does not happen often. Table III displays the instances where the number of transients differ among three configurations. Worth mentioning is that cube

subsumption together with counter stagnation did not time out on any instances of the entire benchmark set.

VII. CONCLUSION


While certification for model checking has been studied for decades, the number of certifiable approaches are still limited. In this paper we revisited the popular preprocessing technique temporal decomposition by formally defining it and proving its correctness. We further present a certification framework in a compositional manner, that builds a single witness circuit. The approach was evaluated on a wide range of benchmarks. Note that our framework requires model checkers to allow reset functions, that is a feature we would like to encourage existing model checkers to implement. Experimental results show that our approach is quite effective in practice.


In the future we plan to investigate other preprocessing techniques such as retiming [34], phase abstraction [35], as well as stabilizing constraints [15], [36]. Furthermore, our simple generic model checking certificate makes it appealing to work on verified proof checkers too.


REFERENCES

- [1] E. M. Clarke, D. E. Long, and K. L. McMillan, “Compositional model checking,” in *LICS*. IEEE Computer Society, 1989, pp. 353–362.
- [2] M. L. Case, H. Mony, J. Baumgartner, and R. Kanzelman, “Enhanced verification by temporal decomposition,” in *FMCAD*. IEEE, 2009, pp. 17–24.
- [3] R. K. Brayton and A. Mishchenko, “ABC: An academic industrial-strength verification tool,” in *CAV*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 24–40.
- [4] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, “The nuXmv symbolic model checker,” in *CAV*, ser. Lecture Notes in Computer Science, vol. 8559. Springer, 2014, pp. 334–342.
- [5] M. Sheeran, S. Singh, and G. Stålmarck, “Checking safety properties using induction and a SAT-solver,” in *FMCAD*, ser. Lecture Notes in Computer Science, vol. 1954. Springer, 2000, pp. 108–125.
- [6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, “Symbolic model checking: 10²⁰ states and beyond,” in *LICS*. IEEE Computer Society, 1990, pp. 428–439.
- [7] A. R. Bradley, “SAT-based model checking without unrolling,” in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 6538. Springer, 2011, pp. 70–87.
- [8] E. Yu, N. Froleys, A. Biere, and K. Heljanko, “Stratified certification for k-induction,” in *FMCAD*. IEEE, 2022, pp. 59–64.
- [9] E. Yu, A. Biere, and K. Heljanko, “Progress in certifying hardware model checking results,” in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 12760. Springer, 2021, pp. 363–386.
- [10] M. J. H. Heule, “Proofs of unsatisfiability,” in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 635–668. [Online]. Available: <https://doi.org/10.3233/FAIA200998>
- [11] M. J. Heule and A. Biere, “Proofs for satisfiability problems,” in *All about Proofs, Proofs for all*, B. W. Paleo and D. Delahaye, Eds. College Publications, 2015, ch. 1.
- [12] M. Järvisalo, M. Heule, and A. Biere, “Inprocessing rules,” in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370. [Online]. Available: https://doi.org/10.1007/978-3-642-31365-3_28
- [13] S. Eriksson, G. Röger, and M. Helmert, “Unsolvability certificates for classical planning,” in *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, L. Barbulescu, J. Frank, Mausam, and S. F. Smith, Eds. AAAI Press, 2017, pp. 88–97.
- [14] K. S. Namjoshi, “Certifying model checkers,” in *CAV*, ser. Lecture Notes in Computer Science, vol. 2102. Springer, 2001, pp. 2–13.
- [15] A. Griggio, M. Roveri, and S. Tonetta, “Certifying proofs for SAT-based model checking,” *Formal Methods Syst. Des.*, vol. 57, no. 2, pp. 178–210, 2021.
- [16] —, “Certifying proofs for LTL model checking,” in *FMCAD*. IEEE, 2018, pp. 1–9.
- [17] A. Abuin, A. Bolotov, U. Díaz-de-Cerio, M. Hermo, and P. Lucio, “Towards certified model checking for PLTL using one-pass tableaux,” in *TIME*, ser. LIPIcs, vol. 147. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 12:1–12:18.
- [18] T. Kuismin and K. Heljanko, “Increasing confidence in liveness model checking results with proofs,” in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 8244. Springer, 2013, pp. 32–43.
- [19] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J. Smaus, “A fully verified executable LTL model checker,” in *CAV*, ser. Lecture Notes in Computer Science, vol. 8044. Springer, 2013, pp. 463–478.
- [20] —, “A fully verified executable LTL model checker,” *Arch. Formal Proofs*, vol. 2014, 2014.
- [21] S. Wimmer and J. von Mutius, “Verified certification of reachability checking for timed automata,” in *TACAS (1)*, ser. Lecture Notes in Computer Science, vol. 12078. Springer, 2020, pp. 425–443.
- [22] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann, “Correctness witnesses: exchanging verification results between verifiers,” in *SIGSOFT FSE*. ACM, 2016, pp. 326–337.
- [23] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig, “Verification witnesses,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 57:1–57:69, 2022.
- [24] A. Biere, K. Heljanko, and S. Wieringa, “AIGER 1.9 and beyond,” Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 11/2, 2011.
- [25] A. Degtyarev and A. Voronkov, “Equality reasoning in sequent-based calculi,” in *Handbook of Automated Reasoning (in 2 volumes)*, J. A. Robinson and A. Voronkov, Eds. Elsevier and MIT Press, 2001, pp. 611–706.
- [26] C. H. Seger and R. E. Bryant, “Formal verification by symbolic evaluation of partially-ordered trajectories,” *Formal Methods Syst. Des.*, vol. 6, no. 2, pp. 147–189, 1995.
- [27] E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, and H. Veith, *Model checking, 2nd Edition*. MIT Press, 2018. [Online]. Available: <https://mitpress.mit.edu/books/model-checking-second-edition>
- [28] CHMC, “CHMC,” 2023, <http://fmv.jku.at/chmc>.
- [29] A. Biere and R. Brummayer, “Consistency checking of all different constraints over bit-vectors within a SAT solver,” in *FMCAD*. IEEE, 2008, pp. 1–4.
- [30] Certifaiger, “Certifaiger,” 2021, <http://fmv.jku.at/certifaiger>.
- [31] A. Biere and K. Claessen, “Hardware model checking competition 2010,” 2010, <http://fmv.jku.at/hwmc2010/>.
- [32] L. Zhang, “On subsumption removal and on-the-fly CNF simplification,” in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 482–489. [Online]. Available: https://doi.org/10.1007/11499107_42
- [33] M. Preiner, A. Biere, and N. Froleys, “Hardware model checking competition 2020,” 2020.
- [34] A. Kuehlmann and J. Baumgartner, “Transformation-based verification using generalized retiming,” in *CAV*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Springer, 2001, pp. 104–117.
- [35] P. Bjesse and J. H. Kukula, “Automatic generalized phase abstraction for formal verification,” in *ICCAD*. IEEE Computer Society, 2005, pp. 1076–1082.
- [36] K. Claessen and N. Sörensson, “A liveness checking algorithm that counts,” in *FMCAD*. IEEE, 2012, pp. 52–59.

BTOR2MLIR: A Format and Toolchain for Hardware Verification

Joseph Tafese 
 University of Waterloo
 Waterloo, Canada
 jetafese@uwaterloo.ca

Isabel Garcia-Contreras 
 University of Waterloo
 Waterloo, Canada
 igarcia@uwaterloo.ca

Arie Gurfinkel 
 University of Waterloo
 Waterloo, Canada
 agurfink@uwaterloo.ca

Abstract—Formats for representing and manipulating verification problems are extremely important for supporting the ecosystem of tools, developers, and practitioners. A good format allows representing many different types of problems, has a strong toolchain for manipulating and translating problems, and can grow with the community. In the world of hardware verification, and, specifically, the Hardware Model Checking Competition (HWMCC), the BTOR2 format has emerged as the dominating format. It is supported by BTOR2TOOLS, verification tools, and Verilog design tools like Yosys. In this paper, we present an alternative format and toolchain, called BTOR2MLIR, based on the recent MLIR framework. The advantage of BTOR2MLIR is in reusing existing components from a mature compiler infrastructure, including parsers, text and binary formats, converters to a variety of intermediate representations, and executable semantics of LLVM. We hope that the format and our tooling will lead to rapid prototyping of verification and related tools for hardware verification.

I. INTRODUCTION

Hardware Verification has been one of the biggest drivers of formal verification research [1], with a history that spans many breakthroughs. The developments in this field have thrived through organized events such as the Hardware Model Checking Competition (HWMCC) [2] which has run since 2011. BTOR2 [3] has emerged as the dominating format in this competition. BTOR2 has been translated into several languages, for example, Constrained Horn Clauses (CHCs)¹² and LLVM-IR³ to make use of existing verification techniques. Universality, however, was not an objective of these projects, and thus, for these translations, be it to CHCs or to LLVM-IR, similar tasks had to be replicated.

During the past decade, the LLVM project [4] has dedicated significant effort to universality. One such effort is MLIR [5], a project that proposes a generic intermediate representation with operations and types common to many programming languages. MLIR was designed to be easily extensible, by providing tools to build new intermediate representations (IR) as dialects of the base MLIR. This eases the creation of new compilers, circumventing the need to re-implement core technologies and optimizations. Extensibility and scalability are what MLIR strives for, making it a great candidate for the

creation of new tools and formats that represent many types of problems and have strong tool support for manipulating and translating problems.

During the same time, with the rise of LLVM as a compiler infrastructure, many software verification tools have been built for LLVM-IR programs. Existing tools tackle this hard problem in many ways. For example, dynamic verification is implemented in LIBFUZZER [6], a fuzzer, and KLEE [7], a symbolic execution engine; SMT-based static verification is implemented in SEAHORN [8] both as Bounded and Unbounded Model Checking; and CLAM [9] static analysis that analyzes LLVM-IR statically using abstract interpretation.

This paper contributes BTOR2MLIR, a format and toolchain for hardware verification. It is built on MLIR to incorporate advances and best practices in compiler infrastructure, compiler design, and the maturity of LLVM. At its core, BTOR2MLIR provides an intermediate representation for BTOR2 as an MLIR dialect. This dialect has an encoding very close to BTOR2 and preserves BTOR2's semantics. This design not only facilitates the creation of a new format for hardware verification but also simplifies the extension of this format to support future targets by using MLIR for all intermediate representations. For example, BTOR2MLIR can be used to generate LLVM-IR from our custom MLIR dialect. The value of this approach is quite evident in CIRCT [10], an open-source project, that applies this design to tackle the inconsistency and usability concerns that plague tools in the Electronic Design Automation industry. Although it has a different goal than BTOR2MLIR, both projects draw great benefit from adapting the benefits of an MLIR design to their respective fields.

As an added bonus, using BTOR2MLIR to generate LLVM-IR enables the reuse of established tools to apply software verification techniques to verify hardware circuits. To illustrate the usability of the toolchain, a new model checker is developed using SEAHORN. The results are compared to BTORMC [3], a hardware model checker provided by the creators of BTOR2.

The rest of the paper is organized as follows. Section II lays some background. Our format and toolchain, BTOR2MLIR, is described in Section III. We discuss its correctness in Section IV and evaluate the tool in Section V. We close with a note on related works in Section VI and conclude in

¹<https://github.com/zhanghongce/HWMCC19-in-CHC>

²<https://github.com/stepwise-alan/btor2chc>

³<https://github.com/stepwise-alan/btor2llvm>

II. BACKGROUND

BTOR2: BTOR2 [3] is a format for quantifier-free formulas and circuits over bitvectors and arrays, with SMT-LIB [11] semantics, that is used for hardware verification. BTOR2 files are often generated using tools like YOSYS [12], from the original design in a language like VERILOG [13]. A simple 4-bit counter is shown in Fig. 1. Its corresponding description, in VERILOG, is shown in Fig. 1b. The circuit updates its output at each step starting from 0 to its maximum value, 15. It also has the safety property that the output should not be equal to 15, shown by the assertion in Fig. 1b. The circuit together with the desired safety property are captured in BTOR2 in Fig. 1c. First, a bitvector of width 4 is defined as '1' in line 1. Sorts are used later when declaring registers and operations. For example, lines 2, 3, 5, and 8 refer to sort '1', respectively, by declaring '2' to be a zero bitvector (0000) (line 2), state `out` to be a register of sort '1' (line 3), '5' to be a one bitvector (0001) (line 5) and '8' to be bitvector of ones (1111) (line 8). On line 4, `out` is initialized with value '2'. On line 7, the transition function is defined (activated at each clock edge), by assigning the next state of `out` to the value `out` incremented by one (the result of line 6). Finally, a safety property is defined in line 11 with the keyword `bad`, requiring that the equality of line 10 does not hold. That is, the value of `out` is never 1111. Note that no clock is specified in Fig. 1c. In BTOR2 it is always assumed that there is one single clock, and the keyword `next` is used to declare how registers are updated after a clock cycle. For a register that has not been assigned a next value, it will get a new non-deterministic value or keep its initial value (if one was given).

BTORMC: BTORMC [3] is a bounded model checker (BMC) for BTOR2. BTORMC generates verification conditions as SMT formulas and uses BOOLECTOR [3] as an SMT solver. Based on the satisfiability result of the formula, BTORMC on our example tells us that the safety property is violated, as expected, since `out` does reach a state with value 1111.

MLIR: Multi-Level Intermediate Representation (MLIR) [5] is a project that was developed for TensorFlow [14] to address challenges faced by the compiler industry at large: modern languages end up creating their own high-level intermediate representation (IR) and the corresponding technologies. Furthermore, these domain-specific compilers have to be recreated for different compilation and optimization targets and do not easily share a common infrastructure or intermediate representations. To remedy this, MLIR facilitates the design and implementation of code generators, translators, and optimizers at different levels of abstraction and also across application domains, hardware targets, and execution environments.

Modern languages vary in the set of operations and types that they use, hence the need to create domain-specific high-level IRs. MLIR addresses this problem by making it easy for a user to define their own dialects. An MLIR dialect captures

the operations and types of a target language. It is created using TABLEGEN, a domain specific language for defining MLIR dialects. It is used to automatically generate code to manipulate the newly defined dialect including its Abstract Syntax Tree (AST) and parsing. MLIR tools and optimizations such as static single assignment, constant propagation, and dead-code elimination can be applied off the shelf to custom MLIR dialects. These capabilities make MLIR a reusable and extensible compiler infrastructure. One of its strengths is the builtin dialects it introduces, such as a BUILTIN, STANDARD, and LLVM dialects⁴, among others. These dialects make it possible to have a rich infrastructure for dialect conversion that enables a user to define pattern-based rewrites of operations from one dialect to another. For example, a dialect conversion pass is provided to convert operations in the STANDARD dialect to operations in the LLVM dialect. MLIR also provides an infrastructure for user-defined language translation passes. One such pass that is provided out of the box is a translation from LLVM dialect to LLVM-IR.

III. BTOR2MLIR

We present our tool, BTOR2MLIR, which contributes the BTOR DIALECT, and three modules on the existing MLIR infrastructure: a BTOR2 to BTOR DIALECT translation pass, a BTOR DIALECT to BTOR2 translation pass and a dialect conversion pass from BTOR DIALECT to LLVM dialect. Our tool has approximately 3900 lines of C++ code and 1200 lines of TABLEGEN. Fig. 2 shows the architecture of our tools with our contributions highlighted in green. BTOR2MLIR uses the original BTOR2 parser provided in BTOR2TOOLS [3], marked in blue, and MLIR builtin passes, marked in brown. BTOR2MLIR is open-sourced and publicly available on GitHub⁵.

We illustrate how each of the components of BTOR2MLIR works by translating a factorial circuit, shown in Fig. 3a, that is described in BTOR2. There are two safety properties, one per `bad` statement. Line 14 states that the loop counter, `i`, reaches 15. Line 19 states that the value of `factorial` is always even.

BTOR DIALECT: Our first contribution is the BTOR DIALECT, an MLIR dialect to represent BTOR2 circuits. Fig. 3b shows the BTOR DIALECT code corresponding to Fig. 3a. It represents the execution of the circuit using an MLIR function `main`. The control flow is explicit, using a standard MLIR representation of basic blocks with arguments and branches. The example has two basic blocks: an unnamed initial block (`bb0`) and a block `bb1`. Circuit initialization is modeled by instructions in `bb0`, and each cycle by instructions in `bb1`. Note that `bb1` has two predecessors: `bb0` for initialization and `bb1` for each cycle. Bitvector types are mapped to integer types (provided by MLIR), for example, `bitvec 4` becomes `i4`. Each operation in the BTOR DIALECT, prefixed with `btor`, models a specific BTOR2 operation. For example, `btor.mul`

⁴<https://github.com/llvm/llvm-project/tree/release/14.x/mlir/include/mlir/Dialect>

⁵<https://github.com/jetafese/btor2mlir/tree/llvm-14>

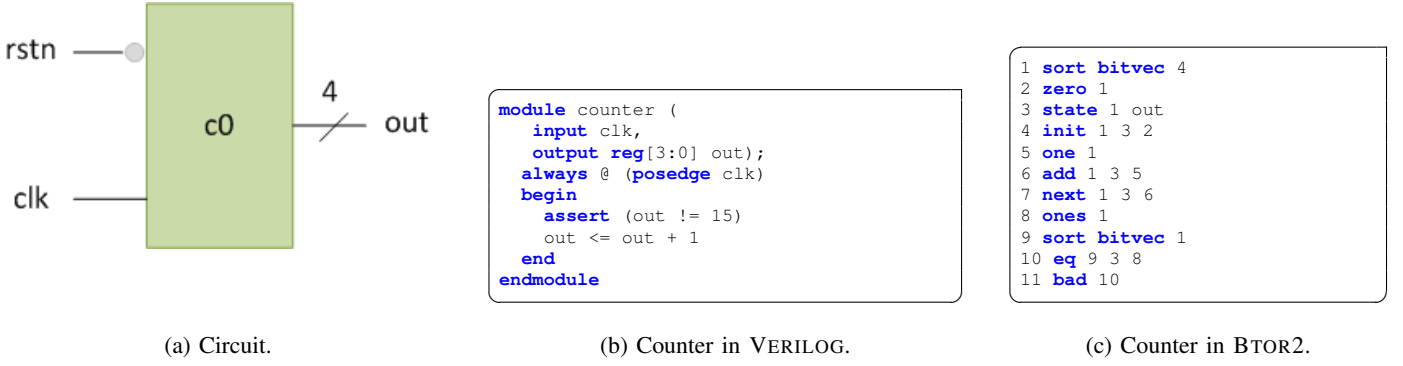


Fig. 1: 4-bit counter.

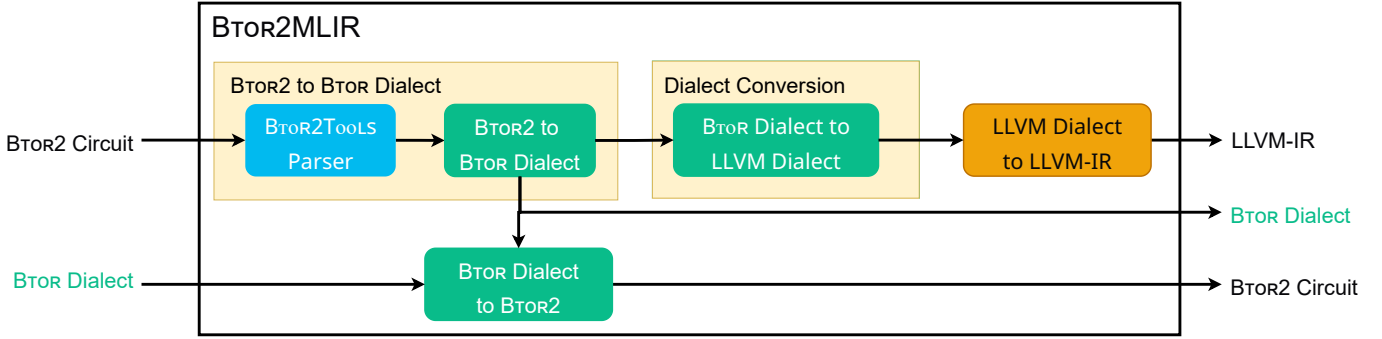


Fig. 2: BTOR2MLIR Architecture.

is `mul`, and `btor.slice` is `slice`. Safety properties such as `bad` are represented by `btor.assert_not`. Special operators such as `one`, `ones` and `constd` are represented by the `btor.constant` operation with the expected integer value. Boolean operators are represented by `btor.cmp`. For example, `eq` becomes `btor.cmp eq`.

Translating BTOR2 to BTOR DIALECT: BTOR2MLIR takes BTOR2 circuits as input, using BTOR2TOOLS to create a data structure for each BTOR2 line. Our pass then generates a program in BTOR DIALECT by constructing the appropriate MLIR AST. Each BTOR2 operator is mapped to a unique operation in BTOR DIALECT, a capability that is greatly simplified and enabled by the MLIR infrastructure.

A program in MLIR can be written using multiple dialects since the MLIR framework enables the interaction of multiple IRs. To enable this capability, MLIR provides dialects that are designed to serve as building blocks for more domain-specific dialects. We utilized the framework by building the BTOR DIALECT using the STANDARD and BUILTIN dialects. For example, we use the `module`, `func` and `bb` operations from BUILTIN. We utilize the `br` operation in the STANDARD dialect to enable interaction between the two basic blocks in Fig. 3b. This approach is consistent with the intended use of the STANDARD and BUILTIN dialects. It saves time and effort since we do not need to recreate operations that already exist in other dialects. Furthermore, MLIR provides a conversion pass from STANDARD dialect to LLVM dialect, making it worthwhile to build BTOR DIALECT on top of the BUILTIN

and STANDARD dialects.

Dialect conversion: The BTOR2MLIR conversion pass from BTOR DIALECT to LLVM dialect utilizes the MLIR infrastructure for pattern-based rewrites. It rewrites BTOR DIALECT operations into LLVM dialect operations. For most operations in BTOR DIALECT there exists a semantically equivalent operation in LLVM dialect. For example, `btor.constant` in Fig. 3b is converted to `llvm.mlir.constant` in LLVM dialect. For some operations, an equivalent in LLVM dialect does not exist, in these cases it is required to rewrite them into several LLVM operations (e.g., in `btor.slice`) and/or to modify the module structure (e.g., `btor.assert_not`). In LLVM dialect, `btor.slice` is replaced by a logical shift right, `llvm.lshr`, and a truncation operation, `llvm.trunc`. `btor.assert_not` is mapped to a new basic block in the LLVM dialect that has the `llvm.unreachable` operation. We split the basic block `bb1`, in Fig. 3b, by adding a conditional branch, `llvm.cond_br`, to direct control flow to the new block when the assertion is satisfied.

Translate LLVM Dialect to LLVM-IR: BTOR2MLIR uses a translation pass from LLVM dialect to LLVM-IR, provided by MLIR. Note the optimizations in the resulting LLVM-IR, shown in Fig. 3c, such as constant propagation and phi nodes.

IV. CORRECTNESS

When introducing a new tool or framework to the community, there is always a question of how polished it is. BTOR2MLIR builds on two mature frameworks:


```

1 sort bitvec 4
2 one 1
3 state 1 factorial
4 state 1 i
5 init 1 3 2
6 init 1 4 2
7 add 1 4 2
8 mul 1 3 4
9 next 1 4 7
10 next 1 3 8
11 ones 1
12 sort bitvec 1
13 eq 12 4 11
14 bad 13
15 slice 12 3 0 0
16 constd 1 3
17 ugt 12 4 16
18 and 12 17 15
19 bad 18

```

(a) Factorial in BTOR2.

```

module {
  func @main() {
    %0 = btor.constant 1 : i4
    br ^bb1(%0, %0 : i4, i4)
  ^bb1(%1: i4, %2: i4):
    %3 = btor.constant 1 : i4
    %4 = btor.add %2, %3 : i4
    %5 = btor.mul %1, %2 : i4
    %6 = btor.constant -1 : i4
    %7 = btor.cmp eq, %2, %6 : i4
    btor.assert_not(%7)
    %8 = btor.constant 0 : i4
    %9 = btor.constant 0 : i4
    %10 = btor.slice %1, %8, %9 : i4,
        i1
    %11 = btor.constant 3 : i4
    %12 = btor.cmp ugt, %2, %11 : i4
    %13 = btor.and %12, %10 : i1
    btor.assert_not(%13)
    br ^bb1(%5, %4 : i4, i4)
  }
}

```

(b) Factorial in BTOR DIALECT.

```

declare void @__VERIFIER_error()
define void @main() !dbg !3 {
  br label %1
1:
  ; preds = %14, %0
  %2 = phi i4 [%5,%14], [1,%0]
  %3 = phi i4 [%4,%14], [1,%0]
  %4 = add i4 %3, 1
  %5 = mul i4 %2, %3
  %6 = icmp eq i4 %3, -1
  %7 = xor i1 %6, true
  br i1 %7, label %8, label %15
8:
  ; preds = %1
  %9 = lshr i4 %2, 0
  %10 = trunc i4 %9 to i1
  %11 = icmp ugt i4 %3, 3
  %12 = and i1 %11, %10
  %13 = xor i1 %12, true
  br i1 %13, label %14, label %16
14:
  ; preds = %8
  br label %1
15:
  ; preds = %1
  call void @__VERIFIER_error()
  unreachable
16:
  ; preds = %8
  call void @__VERIFIER_error()
  unreachable
}

```

(c) Factorial in LLVM-IR.

Fig. 3: BTOR2 to BTOR DIALECT.

	original			roundtrip		
	time	safe/unsafe	TO	time	safe/unsafe	TO
bitvectors						
wolf/18D	157	34/0	2	168	34/0	2
wolf/19A	146	0/1	17	151	0/1	17
wolf/19B	2	3/0	0	2	3/0	0
wolf/19C	834	108/0	5	797	108/0	5
19/beem	278	9/2	4	280	10/2	3
19/goel	190	26/2	43	176	26/2	43
19/mann	4442	29/15	9	4751	30/15	8
20/mann	257	10/5	0	268	10/5	0
bitvectors + arrays						
wolf/18A	70	20/0	0	71	20/0	0
wolf/19B	2	2/3	0	2	2/3	0
19/mann	126	1/1	1	138	1/1	1
20/mann	18	3/3	0	18	3/3	0

TABLE I: Comparing round tripped files.

BTOR2TOOLS and MLIR. This is done not only because of the frameworks’ functionalities, but because they have been extensively reviewed, used, and tested. BTOR2TOOLS has been widely used in the hardware model-checking community since its introduction in 2018. MLIR builds on LLVM, a compiler framework that has been used and improved over numerous projects in the last two decades and is actively supported by industry.

Specifically, BTOR2MLIR uses the parser from BTOR2TOOLS to generate corresponding operations and functions in the BTOR DIALECT of MLIR. The BTOR DIALECT is written in TABLEGEN— an MLIR domain-

specific language for dialect creation. We show how our dialect and the class of binary operations are defined in Fig. 4a. For example, the `BtorBinaryOp` class defines a class of operations that have two arguments `lhs`, `rhs` and a result `res`. It also has a trait `SameOperandsAndResultType` to enforce that `lhs`, `rhs` and `res` have the same type. Finally, the class specifies how the default MLIR parsers and printers should handle such operations. We create our arithmetic operations as shown in Fig. 4b. We mark relevant operations as `Commutative`. Operation descriptions are not shown for simplicity. We ensure that each BTOR2 operator has a one-to-one mapping with an operation in the BTOR DIALECT so that the translation from BTOR2 to BTOR DIALECT is lossless and preserves BTOR2 semantics.

BTOR2MLIR relies on the optimization, folding, and canonicalization passes that MLIR provides in its translation from the LLVM Dialect in MLIR to LLVM-IR. MLIR also provides the mechanism for pattern-based rewrites which has helped us avoid the introduction of undefined behavior into the resulting LLVM-IR. We show an example of this in Fig. 5. MLIR allows us to identify which operations in the BTOR DIALECT we want to replace at the end of our conversion pass. A subset of such operations are shown in Fig. 5a. For each operation that has been identified, we provide a lowering that maps it to a legal operation in the LLVM dialect. We are able to use lowering patterns like `VectorConvertToLLVMPattern` from MLIR for common arithmetic and logical operations as shown in Fig. 5b.

We performed extensive testing using the HWMCC20 benchmark set to verify the correctness of BTOR2MLIR. This is the same benchmark set used to test [15]. The tests are run on a Linux machine with x86_64 architecture, using BTORMC


```

def Btor_Dialect : Dialect {
  ...
}

class BtorArithmeticOp<string mnemonic, list<Trait> traits = []> :
  Op<Btor_Dialect, mnemonic, traits>;

class BtorBinaryOp<string mnemonic, list<Trait> traits = []> :
  BtorArithmeticOp<mnemonic, !listconcat(traits
    [SameOperandsAndResultType])>,
  Arguments<(ins SignlessIntegerLike:$lhs,
    SignlessIntegerLike:$rhs)>,
  Results<(outs SignlessIntegerLike:$result)>
{
  let assemblyFormat = "$lhs `, ` $rhs attr-dict `:` type($result)";
}

```

(a) Creating BTOR DIALECT.

```

def AddOp : BtorBinaryOp<"add",
  [Commutative]> {
  ...
}

def SubOp : BtorBinaryOp<"sub"> {
  ...
}

def MulOp : BtorBinaryOp<"mul",
  [Commutative]> {
  ...
}

def UDivOp : BtorBinaryOp<"udiv"> {
  ...
}

```

(b) Creating Operations for BTOR DIALECT.

Fig. 4: Using TABLEGEN for Dialect Creation.

with an unroll bound of 20, a timeout of 300 seconds and memory limit of 65 GB. We present the results in Table I, where bitvector benchmarks categories are in the top half and bitvector + array benchmark categories are in the bottom half. All times in this table reflect solved instances and do not include timeouts. We do not show the time it takes to run BTOR2MLIR since the time is negligible. The results are grouped by competition contributor such that each row shows the time, instances solved (safe/unsafe) and timeouts (TO) for both the original and round-tripped circuits. For example, for the `wolf/18D` category, we can see that the original BTOR2 circuit solves 34 safe instances and 0 unsafe instances in 157 seconds, with 2 timeouts. The round-tripped circuit solves 34 safe instances and 0 unsafe instances in 168 seconds with two timeouts.

We can see that the safety properties in BTOR2 circuits are neither changed nor violated after being round-tripped by BTOR2MLIR. In two categories with only bitvectors, `19/beem` and `19/mann`, one more instance in each category is found safe after round trip, while the original circuit leads to a memout and timeout respectively. This gives us confidence that the translation to BTOR DIALECT, using the BTOR2TOOLS parser, is indeed correct. Then, we tested whether the same holds after translation to LLVM-IR. Through this method, we were able to ensure that BTOR2MLIR does not have errors when handling operations that are represented in the benchmark set. This approach is not complete, however, since it would not identify errors that might be in our implementation but are not exercised by the benchmarks we use. For example, BTOR2 expects that a division by zero would result in -1 , but there are no benchmarks that exercise this kind of division. We mitigate this by generating benchmarks for division, remainder, and modulus operators to ensure that the expected behavior of BTOR2 operators are represented in our test suite.

In the future, it is interesting to explore other translation validation and verification approaches. For example, it would be useful for BTOR2MLIR to produce a proof trail that

justifies all of the transformations that are performed by the tool. This, for example, might be possible to achieve by building on the work of [16], [17].

Limitations: BTOR2MLIR is able to round trip BTOR2 operators and their sorts. In LLVM-IR all BTOR2 operators and their sorts are supported as well, but not fairness and justice constraints.

V. EVALUATION

To evaluate BTOR2MLIR, we have built a prototype hardware model checker by connecting our tool with SEAHORN [8], a well-known model checker for C/C++ programs that works at the LLVM-IR level. It has recently been extended with a bit-precise Bounded Model Checking engine [18]. This BMC engine was evaluated in a recent case study [19] and we use the same configuration of SEAHORN in our evaluation.

The goal of our evaluation is to show that BTOR2MLIR makes it easy to connect hardware designs with LLVM-based verification engines. We did not expect the existing software engines to outperform dedicated hardware model checkers. However, we hope that this will enable further avenues of research. In the future, we plan to extend the framework to support other LLVM-based analysis tools, such as symbolic execution engine KLEE [7], and fuzzing framework [6].

For the evaluation, we have chosen the bitvector category of BTOR benchmarks from the most recent Hardware Model Checking Competition (HWMCC) [2]. We have excluded benchmarks with arrays since the export to LLVM-IR is not supported by SEAHORN in our experimental setup. All our experiments are run on a Linux machine with x86_64 architecture, with unroll bound of 20, a timeout of 300 seconds and memory limit of 65 GB. The results are presented in Table II, grouped by competition contributor. All times in this table reflect solved instances and do not include timeouts. We do not show the time it takes to run BTOR2MLIR since the time is negligible. In the rest of this section, we highlight some of the interesting findings.

We have run BTORMC on the same machine and exact same experimental setup (unroll bound and CPU and memory

```

void BtorToLLVMLoweringPass::runOnOperation() {
  LLVMConversionTarget target(getContext());
  RewritePatternSet patterns(getContext());
  LLVMTypeConverter converter(&getContext());
  mlir::btor::populateBtorToLLVMConversionPatterns(converter,
    patterns);

  ...
  /// binary operators
  // arithmetic
  target.addIllegalOp<btor::AddOp, btor::SubOp, btor::MulOp,
    btor::UDivOp...>();
  ...
}

```

(a) Identifying operations.

```

...
using AddOpLowering =
  VectorConvertToLLVMPattern<btor::AddOp, LLVM::AddOp>;
using SubOpLowering =
  VectorConvertToLLVMPattern<btor::SubOp, LLVM::SubOp>;
using MulOpLowering =
  VectorConvertToLLVMPattern<btor::MulOp, LLVM::MulOp>;
using UDivOpLowering =
  VectorConvertToLLVMPattern<btor::UDivOp, LLVM::UDivOp>;
...
void mlir::btor::populateBtorToLLVMConversionPatterns(
  LLVMTypeConverter &converter, RewritePatternSet
    &patterns) {
  patterns.add<
    AddOpLowering, SubOpLowering, MulOpLowering,
    UDivOpLowering, ...>(converter);
}
...

```

(b) Converting operations to LLVM-IR.

Fig. 5: Using Patter Based Rewriters in MLIR.

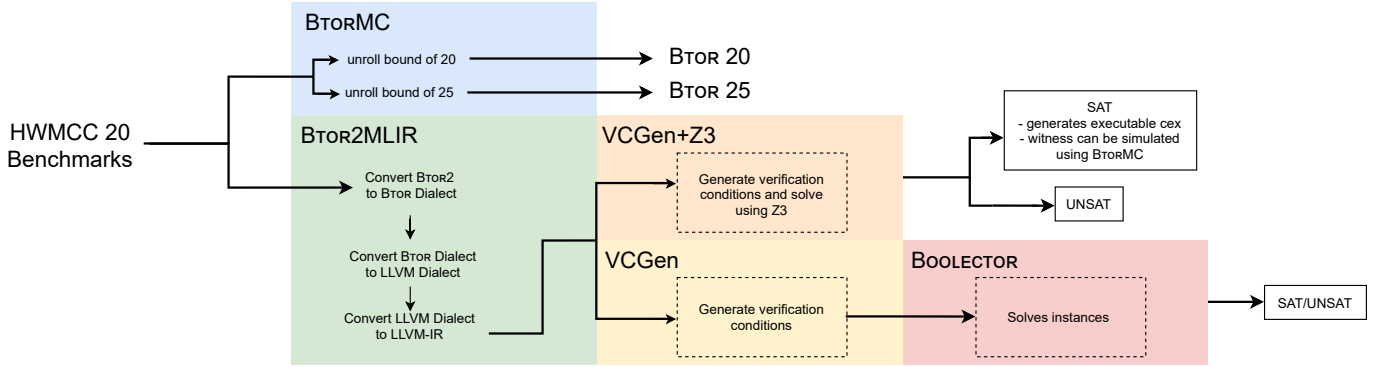


Fig. 6: Verification Strategies.

limits). We chose BTORMC because it is well integrated with the HWMCC environment and is specifically designed for BTOR2. The results of running BTORMC are shown in the first columns of BTORMC in Table II. For each category, we show the total time for all instances that are solved in that category, and the number of instances that are solved as safe, unsafe, and timed-out (TO), respectively. For example, the 20/mann category is solved in 257 seconds, 10 instances are safe, 5 are unsafe, and no instance has timed out. The performance of BTORMC is quite good across the board.

We evaluate the problems generated by BTOR2MLIR by

plugging them into SEAHORN. SEAHORN pre-processes programs before attempting to verify them. This includes, standard LLVM optimizations (i.e., -O3), loop unrolling and loop cutting are applied. We found that SEAHORN was able to, in some instances, remove the assertions in the LLVM-IR, meaning that the program was found to be safe statically, before invoking the BMC. The BMC also runs simplifications on the formulas that it sends to Z3, its default underlying SMT solver. The results for this run are shown in the Z3 columns of Table II. For example, the 20/mann category is solved in 94 seconds, 8 instances are safe, 5 are unsafe and 2 have

		BTORMC		SEAHORN		
		20	25	VCGen + Z3	VCGen	BTOR
wolf/18D	Time (s)	157	394	560	543	745
	Safe	34	34	29	-	34
	Unsafe	0	0	0	-	0
	TO	2	2	7	2	2
wolf/19A	Time (s)	146	106	-	-	-
	Safe	0	0	0	-	0
	Unsafe	1	1	0	-	0
	TO	17	17	18	18	18
wolf/19B	Time (s)	2	2	2	2	3
	Safe	3	3	3	-	3
	Unsafe	0	0	0	-	0
	TO	0	0	0	0	0
wolf/19C	Time (s)	834	1101	354	418	1085
	Safe	108	107	102	-	106
	Unsafe	0	0	0	-	0
	TO	5	6	11	2	7

		BTORMC		SEAHORN		
		20	25	VCGen + Z3	VCGen	BTOR
19/beem	Time (s)	278	251	309	35	85
	Safe	9	8	6	-	7
	Unsafe	2	2	2	-	2
	TO	4	5	7	4	6
19/goel	Time (s)	190	349	489	132	335
	Safe	26	25	25	-	28
	Unsafe	2	2	1	-	2
	TO	43	44	45	27	41
19/mann	Time (s)	4442	8674	3811	175	3015
	Safe	29	28	19	-	30
	Unsafe	15	15	14	-	14
	TO	9	10	20	2	9
20/mann	Time (s)	257	495	94	35	188
	Safe	10	10	8	-	9
	Unsafe	5	5	5	-	5
	TO	0	0	2	0	1

TABLE II: HWMCC20 Results.

timed out. The reported time does not include the instances that have timed out.

The aggregate time of SEAHORN on most of the categories is higher than that of BTORMC, often by a significant amount. We looked into this and found that SEAHORN treats the given bound as a lower bound, rather than an upper bound. That is, it ensures that it unrolls the programs to a depth of at least 20, but it may continue past that point. Taking this into account, we ran BTORMC with a bound of 25. The results are in the second columns of BTORMC in Table II. As expected, its aggregate times are higher than the run of BTORMC with bound 20. We notice, however, that it is slower than SEAHORN in the 19/mann category.

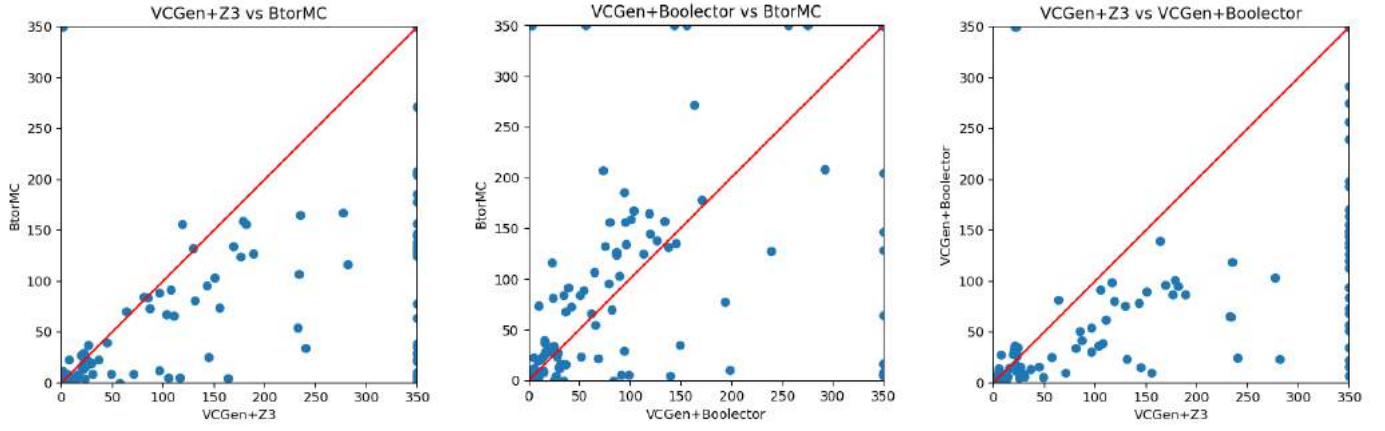
BOOLECTOR and Z3 are the SMT solvers used by BTORMC and SEAHORN respectively. Given that BOOLECTOR is optimized for BTOR2 circuits, we evaluated whether the SMT formulas generated by SEAHORN would be solved faster by BOOLECTOR. The results for generating SMT-LIB formulas using SEAHORN are presented in the VCGen column of Table II. The times are low for most categories except wolf/18D, wolf/19C, 19/goel and 19/mann. For example, it takes SEAHORN 175 seconds to generate the verification conditions for instances in the 19/mann category, with two timeouts. This includes the time it takes SEAHORN to print the SMT formulas to disk. We plug the resulting SMT formulas into BOOLECTOR and present the results in the BTOR columns of Table II. The results show that using SEAHORN to generate verification conditions and BOOLECTOR to solve these instances is often better than using BTORMC. For example, for category 19/mann, it takes 3015s for BOOLECTOR to solve 44 instances with 9 timeouts. Therefore, the total time for SEAHORN and BOOLECTOR (3190) represents the

	BTORMC		SEAHORN		
	20	25	VCGen+Z3	VCGen	BTOR
Time (s)	6 309	11 373	5 621	1 340	5 456
Safe	219	215	192	-	217
Unsafe	25	25	22	-	23
TO	80	84	110	55	84

TABLE III: Total results for each tool.

time it takes to translate, generate SMT formula and verify the 19/mann category. Note that two of the 9 timeouts in this category are attributed to the fact that SEAHORN has a timeout when generating verification conditions.

To get the big picture of how the different infrastructures performed, we collected the results over all categories in Table III. From this table, we can see that our hybrid pipeline combining BTOR2MLIR, SEAHORN, and BOOLECTOR solves 240 instances with 84 timeouts in 6796s (sum of VCGen and BTOR total times), which is very encouraging. We also present plots that compare the different pipelines that have been explored in Fig. 7. We set the time for all timeout instances to 350 seconds so that they are distinguished from instances that were solved close to the timeout threshold. First, we look at the performance of the hybrid pipeline that combines BTOR2MLIR, SEAHORN and its default SMT solver Z3 against BTORMC in Fig. 7a. Z3 does as well as BTORMC for most instances that are easy, however, it struggles when the problems are harder. This is not as clear from Table II since focuses on the number of timeouts and benchmarks solved. Second, we present the performance of BTORMC against the hybrid pipeline that combines BTOR2MLIR, SEAHORN and



(a) VCGen + Z3 vs BTORMC.

(b) VCGen + BOOLECTOR vs BTORMC.

(c) VCGen + Z3 vs VCGen + BOOLECTOR.

Fig. 7: Verification strategy comparison.

BOOLECTOR in Fig. 7b. We can see that there are more benchmarks that this pipeline solves faster than BTORMC. It is also clear that it solves more benchmarks than the Z3 configuration in Fig. 7a, as we would expect from Table III. Third, we compare the two hybrid pipelines in Fig. 7c. We can see that the configuration that uses SEAHORN to generate verification conditions and BOOLECTOR for solving easily outperforms the Z3 configuration.

VI. RELATED WORK

Translating BTOR2 circuits into other formats enables the application of different verification methods and techniques. The gains that can be made from applying one method of encoding over another could enable solving a class of benchmarks that are not solved with existing approaches.

BTOR2LLVM⁶ and BTOR2CHC⁷ are tools that convert BTOR2 circuits to programs in LLVM-IR and CHCs, respectively. These tools are developed in Python, in order to be light weight, but end up repeating shared functionality and tools since they lack a common infrastructure. Translated BTOR2 benchmarks⁸ have also been collected to facilitate research, but information of what tools were used to get the CHC format is not publicly available. While a collection of translated benchmarks is valuable, it is important that there are tools to do the translation on demand. This enables rapid prototyping in a way that saved benchmarks do not.

BTOR2C [15] is a recent tool that converts BTOR2 circuits to C programs. It has been used to facilitate the utilization of software analyzers by serving as a pre-processing step that bridges the gap between the world of software verification and hardware verification. There are limitations that arise, however, from differences in the semantics of BTOR2 and

C. An important limitation that C imposes on this project is the inability to represent arbitrary width bitvectors. This means that BTOR2 circuits which operate on bitvectors of width greater than 128 are not supported. These limitations, as well as BTOR2C lack of support for BTOR2 operators that have overflow detection are resolved by using LLVM-IR as the target language.

A common theme across these efforts is that they are not built on an architecture that can be easily extended. Each project aims to make it easier to utilize advances in formal verification, but they fail to offer a solution that does not require recreating components that already exist.

VII. CONCLUSION

In this paper, we present BTOR2MLIR — a new format and toolchain for hardware verification, based on the MLIR intermediate representation framework of the LLVM compiler infrastructure. Our goal is to open new doors for the research and applications of hardware verification by taking advantage of recent innovations in compiler construction technology. We believe that this project opens new avenues for exploring the application of existing verification and testing techniques developed for software to the hardware domain. As a proof of concept, we have connected BTOR2MLIR with the SEAHORN verification engine. While out-of-the-box, this gives acceptable performance, when combined with BOOLECTOR, a combination that is competitive against BTORMC. In the future, we plan to continue this line of research and explore applying testing and simulation technologies such as KLEE [7] and LIBFUZZER [6]. We also plan to generate formats for other verification techniques such as AIGER [20], Constrained Horn Clauses, and SMT-LIB.

REFERENCES

- [1] S. Malik, “Hardware verification: Techniques, methodology and solutions,” in *Tools and Algorithms for the Construction and Analysis of*

⁶<https://github.com/stepwise-alan/btor2llvm>

⁷<https://github.com/stepwise-alan/btor2chc>

⁸<https://github.com/zhanghongce/HWMCC19-in-CHC>

- Systems, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–1.
- [2] A. Biere, T. van Dijk, and K. Heljanko, “Hardware model checking competition 2017,” in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 9–9.
 - [3] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, “Btor2 , BtorMC and Boolector 3.0,” in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 587–595.
 - [4] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.
 - [5] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, “MLIR: A Compiler Infrastructure for the End of Moore’s Law,” 2020.
 - [6] K. Serebryany, “Continuous Fuzzing with libFuzzer and AddressSanitizer,” in *2016 IEEE Cybersecurity Development (SecDev)*, 2016, pp. 157–157.
 - [7] C. Cadar, D. Dunbar, and D. R. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *USENIX Symposium on Operating Systems Design and Implementation*, 2008.
 - [8] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, “The SeaHorn Verification Framework,” in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 343–361.
 - [9] A. Gurfinkel and J. A. Navas, “Abstract interpretation of LLVM with a region-based memory model,” in *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. Bloem, R. Dimitrova, C. Fan, and N. Sharygina, Eds., vol. 13124. Springer, 2021, pp. 122–144. [Online]. Available: https://doi.org/10.1007/978-3-030-95561-8_8
 - [10] S. Eldridge, P. Barua, A. Chapyzhenka, A. Izraelevitz, J. Koenig, C. Lattner, A. Lenharth, G. Leontiev, F. Schuiki, R. Sunder, A. Young, and R. Xia, “MLIR as Hardware Compiler Infrastructure,” in *Workshop on Open-Source EDA Technology (WOSET)*, 2021.
 - [11] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” www.SMT-LIB.org, 2016.
 - [12] C. Wolf, “Yosys open synthesis suite,” <https://yosyshq.net/yosys/>.
 - [13] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*. USA: Prentice-Hall, Inc., 1996.
 - [14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 265–283. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>
 - [15] D. Beyer, P.-C. Chien, and N.-Z. Lee, “Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator,” in *Proc. TACAS*, ser. LNCS 13994. Springer, 2023, pp. 1–21. [Online]. Available: <https://www.sosy-lab.org/research/btor2c/>
 - [16] S. Chatterjee, A. Mishchenko, R. K. Brayton, and A. Kuehlmann, “On resolution proofs for combinational equivalence,” in *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*. IEEE, 2007, pp. 600–605. [Online]. Available: <https://doi.org/10.1145/1278480.1278631>
 - [17] R. E. Bryant, “Tbuddy: A proof-generating BDD package,” in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds. IEEE, 2022, pp. 49–58. [Online]. Available: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_10
 - [18] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel, “Bounded Model Checking for LLVM,” in *Formal Methods in Computer Aided Design, FMCAD 2022*, 2022, p. 214.
 - [19] —, “Verifying verified code,” *Innov. Syst. Softw. Eng.*, vol. 18, no. 3, pp. 335–346, 2022. [Online]. Available: <https://doi.org/10.1007/s11334-022-00443-9>
 - [20] A. Biere, K. Heljanko, and S. Wieringa, “AIGER 1.9 and beyond,” Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 11/2, 2011.

Data-Driven Learning of Strong Conjunctive Invariants

Arkesh Thakkar

Department of Computer Science and Automation

Indian Institute of Science

Bengaluru, India

arkesh.thakkar14@gmail.com, arkeshkumart@iisc.ac.in

Deepak D'Souza

Department of Computer Science and Automation

Indian Institute of Science

Bengaluru, India

deepakd@iisc.ac.in

Abstract—Coming up with adequate inductive invariants is the key step in the deductive verification of programs. Moreover, coming up with *strong* inductive invariants is important in several application scenarios like compositional verification. Houdini-style approaches are popular techniques that learn the strongest conjunctive subset of a given set of constraints that is adequate to prove a given pre-post specification for a program. However, both the adequacy and strength of the learned invariants depend on the quality of the initial set of constraints. In this work, we propose a data-driven way of learning an initial set of constraints from sample runs of the program. We use a novel combination of linear regression for equality constraints and linear programming for inequality constraints. The evaluation of our technique shows that it is more efficient and leads to stronger adequate invariants than some of the state-of-the-art data-driven techniques.

I. INTRODUCTION

Finding likely invariants at points in a program from sample execution traces of the program has many applications. These include program understanding and specification mining [1], program testing [12], [22], and program verification [18], [21]. Our interest is mainly in the application of learning likely numerical invariants in the context of the Floyd-Hoare style verification of pre-post specifications for programs. Many existing techniques, like Houdini [8] and ICE-learning [6], [10] attempt to learn adequate inductive invariants from a set of base predicates. Such techniques could also benefit from a good base set of likely invariants.

In many applications, coming up with “tight” invariants that fit the sample data well while still generalizing well to unseen data or leading to adequate inductive invariants is important. Tight invariants are useful in proving strong postconditions, or in inferring strong postconditions for compositional verification (say of programs with sequential loops).

Prior works in data-driven invariant generation have typically used techniques like enumerative checking of a set of bounded templates [5], algebraic equality solving [16], [21], convex hulls [16], and neural network based learning [20], [23]. In this paper, we propose a technique for data-driven invariant learning based on classical linear regression for equality constraints, and linear programming for inequality constraints. We used linear regression along with feature selection to find a good set of equality invariants; feature selection gives us more optimal answers and avoids overfitting

the data. Our linear programming formulation for finding tight inequality invariants are both more efficient and produce stronger invariants in comparison to techniques based on convex hulls or neural networks.

We have evaluated our technique on a range of simple loop verification benchmarks from SVCOMP [2], by collecting sample data at the head of the loop and learning a set of candidate invariants. We then try to find a conjunctive subset of these invariants that is adequate to prove the postconditions using the Houdini algorithm. We did the same with three state-of-the-art invariant generation tools. We were able to find more adequate invariants than all the other tools (100 out of 104 programs), the strongest adequate invariant, in comparison with the other tools, in 87 programs, and the average time to generate the candidate set of invariants is 8.32 seconds.

The rest of the paper is organized as follows. We begin with an overview of our approach in Sec. II, followed by background material in Sec. III which gives the basics of Floyd-Hoare verification and the Houdini technique. Secs. IV, V and VI describe our technique to generate equality and inequality invariants, and the overall algorithm, respectively. Sec. VII describes the experimental evaluation of our approach. We discuss related work in Sec. VIII and conclude in Sec. IX.

II. OVERVIEW

Given a program P having a set of variables V , precondition pre and postcondition $post$ and a loop as shown in Fig. 1, we first collect the traces T of variables (i.e., the value that each variable holds) at different loop iterations by giving a random set of inputs that satisfy the pre condition. We also generate polynomial terms (i.e. non-linear terms) from the existing variables of P upto a user-given degree r . This helps us learn non-linear invariants as well, along with the linear invariants.

After collecting the trace data T and generating a polynomial degree, let the total number of dimensions (including new variables representing polynomial terms) be n . Considering each variable as a target variable and other $n - 1$ variables as independent variables, we apply feature selection techniques to get the features (i.e. variables among the $n - 1$ independent variables) that are closely related to the target variable. We then apply linear regression to find the equality relationships

among the target and closely related independent variables. Applying feature selection is important because if n is large, linear regression without feature selection could lead to overfitting and we may not be able to infer a generalized equality relationship between the target and independent variables.

Consider one of the programs named “knuth”, shown below, from the SVCOMP nla-digbench benchmark [2]. The program implements an algorithm that searches for a divisor for factorization.

```

unsigned n, a;
unsigned r, k, q, d, s, t;
d = a;
r = n % d;
t = 0;
k = n % (d - 2);
q = 4 * (n / (d - 2) - n / d);
s = sqrt(n);
while (1) {
    if (!(s >= d) && (r != 0)))
        break;
    if (2*r + q < k) {
        t = r;
        r = 2*r - k + q + d + 2;
        k = t;
        q = q + 4;
        d = d + 2;
    }
    else if ((2*r + q >= k)
    && (2*r + q < d + k + 2)) {
        t = r;
        r = 2*r - k + q;
        k = t;
        d = d + 2;
    }
    else if ((2*r + q >= k) &&
    (2*r + q >= d + k + 2) &&
    (2*r + q < 2*d + k + 4)) {
        t = r;
        r = 2*r - k + q - d - 2;
        k = t;
        q = q - 4;
        d = d + 2;
    }
    else {
        t = r;
        r = 2*r - k + q - 2*d - 4;
        k = t;
        q = q - 8;
        d = d + 2;
    }
}

```

A snapshot of collected traces is shown in the following table:

n	a	r	k	q	d	s	t
25	3	60	1	60	5	5	1
25	3	165	60	52	7	5	60
49	3	1	0	132	3	7	0
49	3	124	1	124	5	7	1
49	3	357	124	116	7	7	124
49	3	688	357	108	9	7	357
⋮							
19021	15	816146	792937	340	125	137	792937
19021	15	839441	816146	332	127	137	816146
19021	15	862810	839441	324	129	137	839441

TABLE I
VARIABLE VALUES IN TRACES OF EXAMPLE PROGRAM

The documented invariant for the program is

$$ddq - 2qd - 4rd + 4kd + 8r = 8n.$$

As it requires some degree 3 terms, generating terms up to degree 3 from the traces goes up to 164 dimensions. Without applying feature selection, linear regression leads to overfitting and doesn't give any equality invariant. Thus, feature selection is important for learning useful equality invariants. Furthermore, we use Linear Programming to help us learn tight inequality constraints in two and three dimensions.

As shown in Sec. VII, none of the three existing tools, namely Daikon, DIG, and GCLN, we experimented with were able to find an adequate invariant for this program, due to lack of effective feature selection techniques. In contrast, our tool LPGEN finds an adequate invariant for the above program, due to its use of feature selection techniques (details in Sec. IV).

III. BACKGROUND

A. Preliminaries

We will use \mathbb{Z} and \mathbb{R} to denote the set of integers and reals respectively. For an $m \times n$ matrix M over the reals, we will denote by M^T the transpose of M , which is the $n \times m$ matrix obtained by changing the rows of M into columns. For $m \times n$ matrices M and N , we will denote by $M + N$ (respectively $M - N$) the matrix obtained by the pointwise addition (respectively subtraction) of elements of M and N , and write $M \leq N$ to mean that each (i, j) -th element of M is less than or equal to the (i, j) -th element of N . For vectors $u = [u_1, \dots, u_n]$ and $v = [v_1, \dots, v_n]$, the inner product of u and v , denoted $u \cdot v$, is $\sum_{i=1}^n u_i v_i$. The L_2 norm of u , denoted $\|u\|$, is defined to be $\sqrt{(\sum_{i=1}^n u_i^2)}$, while the L_1 norm of u , denoted $\|u\|_1$, is defined to be $\sum_{i=1}^n |u_i|$.

We will be dealing with a quantifier-free logical language of constraints. Given a set of variables $V = \{x_1, \dots, x_n\}$, we define the following language of constraints over V . The terms t of this language will comprise variables from V , constants $l \in \mathbb{Z}$, and products and sums of terms. An (atomic) constraint over V is now of the form $t \sim t'$, where $\sim \in \{=, <, \leq, >, \geq\}$. In some cases we will also allow atomic constraints of the form $x \bmod l = d$ and $x \& l = m$, where x is an integer variable, and l and m are integers, representing that $x \bmod l$ is d and the bitwise-and of x and l is m ; and $\gcd(x, y) = \gcd(x', y')$ where x, y, x', y' are integer-valued variables in V , and \gcd denotes the greatest common divisor function. A constraint

over V is a boolean combination of atomic constraints over V .

A *valuation* u of variables in V is a map that assigns to each variable x in V a real value $u(x)$. Wherever convenient, we will represent such a valuation as a vector $[u(x_1), \dots, u(x_n)]$. A valuation u assigns a real value t^u to each term t over V , by inductively defining $l^u = l$, $x^u = u(x)$, and $(t + t')^u = t^u + (t')^u$, etc. We say u satisfies an atomic constraint $t \sim t'$ iff $t^u \sim (t')^u$. Finally, boolean combinations are handled in the expected manner. We write $u \models c$ to denote the fact that the valuation u satisfies the constraint c .

B. Verification Conditions for Simple Programs

We will consider programs with a simple loop structure, as shown in Fig. 1. The programs have a single loop, with a set of initialization statements S_1 , loop body statements S_2 , and post-loop statements S_3 . The statements in these blocks are assignments or if-then-else statements.

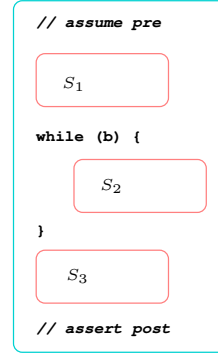
We will use standard Floyd-Hoare logic [9], [13] to specify and reason about the correctness of these programs. A state of a program is simply a valuation to its variables, and thus we can ask whether it satisfies a given constraint on the variables of the program. An “pre-post” specification for a program P over a set of variables V is pair of constraints $(pre, post)$ over V . We say P satisfies the specification $(pre, post)$ if every terminating execution of P that begins in a state satisfying pre , terminates in a state satisfying $post$.

A standard way to prove that a program P satisfies a specification $(pre, post)$ is to come up with a constraint inv over the variables of the program, satisfying the conditions (C1), (C2) and (C3) on the right in Fig. 1 (in the the sense that the implications are logically valid). In the conditions, we use the notation $\llbracket S \rrbracket$ to denote the standard logical semantics of the statement S over the set of variables V and $V' = \{x' \mid x \in V\}$, representing the states before and after the execution of S , respectively. For example, for an assignment statement $x := x + 1$ in a program with variables $V = \{x, y\}$, we have $\llbracket x := x + 1 \rrbracket$ to be $x' = x + 1 \wedge y' = y$. We also use the notation inv' to denote the constraint obtained by substituting x' for x in inv , for each variable $x \in V$. We call these conditions *verification conditions*, and they essentially require inv to be an *adequate inductive invariant* for P , with conditions (C1) and (C3) enforcing adequacy of inv w.r.t. the pre-post conditions, and condition (C2) enforcing inductiveness of inv .

C. Houdini Algorithm

We briefly outline the Houdini algorithm of [8] (see also [15]), which is useful in efficiently identifying an adequate inductive conjunctive subset of a given set of candidate invariants.

Let P be a program of the form shown in Fig. 1 over a set of variables V , and $(pre, post)$ be a given pre-post specification for P . Let \mathcal{C} be a finite set of constraints over V . Then the Houdini algorithm finds the strongest conjunctive subset of \mathcal{C} that is an adequate inductive invariant for P w.r.t. $(pre, post)$.



$$\begin{aligned}
(C1) \quad & pre \wedge \llbracket S_1 \rrbracket \implies inv' \\
(C2) \quad & inv \wedge b \wedge \llbracket S_2 \rrbracket \implies inv' \\
(C3) \quad & inv \wedge \neg b \wedge \llbracket S_3 \rrbracket \implies post'
\end{aligned}$$

Fig. 1. A simple loop program and corresponding VCs for inv to be an adequate inductive invariant

The algorithm proceeds as follows. As usual, we treat $\bigwedge \emptyset$ to be *true*.

- 1) Let \mathcal{C}_1 be the subset of \mathcal{C} containing all constraints $c \in \mathcal{C}$ such that $pre \wedge \llbracket S_1 \rrbracket \implies c'$.
- 2) Check if $\bigwedge \mathcal{C}_1$ satisfies condition (C3) (i.e. $\bigwedge \mathcal{C}_1 \wedge \neg b \wedge \llbracket S_3 \rrbracket \implies post'$). If not, return “No adequate invariant exists.”
- 3) Check if $\bigwedge \mathcal{C}_1$ satisfies condition (C2) (i.e. $\bigwedge \mathcal{C}_1 \wedge b \wedge \llbracket S_2 \rrbracket \implies \bigwedge \mathcal{C}_1'$). If so, return $\bigwedge \mathcal{C}_1$ as an adequate inductive invariant.

Else, let (u, v) be a counter-example, where $u \models b \wedge \bigwedge \mathcal{C}_1$ but $v' \not\models \bigwedge \mathcal{C}_1'$, where v is the state resulting from executing S_2 in u . Let \mathcal{C}_2 be the subset of \mathcal{C}_1 obtained by dropping constraints that are not satisfied by v . Thus $\mathcal{C}_2 = \mathcal{C}_1 - \{c \in \mathcal{C}_1 \mid v \not\models c\}$. Set \mathcal{C}_1 to \mathcal{C}_2 , and go back to Step 2.

IV. EQUALITY INVARIANTS USING LINEAR REGRESSION

Suppose we are interested in tracking the values of n different terms at the loop head in a program P . These terms could be variables of the program (like x and i) or product terms like $(x^2$ or $xi)$. Let us say we have collected m sample values. Then we can represent the values collected as a *data set* $D = \{d_1, \dots, d_m\}$, where each $d_i = [d_{i1}, \dots, d_{in}]$ represents the values of notional variables x_1, \dots, x_n standing for the terms we are interested in tracking. These notional variables are also called *features*. Let us fix such a data set D , with feature dimension $n \geq 2$ and number of samples $m \geq 1$, for the next couple of sections.

In this section, our goal is to learn a linear equality relationship between the notional variables, whenever one exists. To do this we consider each variable x as the *target* variable, and consider the remaining $n - 1$ variables as *independent* variables. The goal is to see if the target variable can be expressed as a “linear” combination of the independent variables. For example, if x_1 is our target variable, we would like to know if $x_1 = w_1 + w_2 x_2 + \dots + w_n x_n$, for some real-valued weights w_1, \dots, w_n . The technique of linear regression helps us to do this.

However, when the number of independent variables is large, linear regression may not give us good results as it could

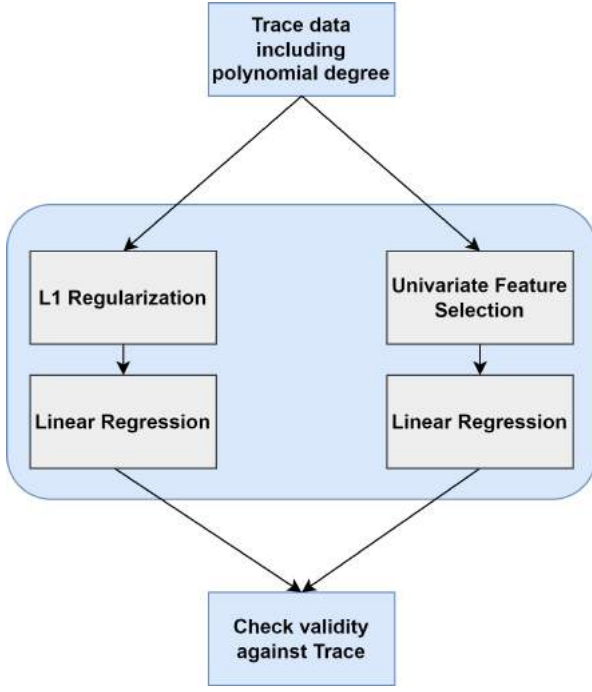


Fig. 2. Generating equality invariants

overfit the data in a dataset D and may fail to generalize. Hence we first find a subset of closely-related independent variables (this step is called *feature selection*), and then apply linear regression on this restricted set of variables.

In our proposed technique, we use two feature selection techniques, namely L1 regularization and univariate feature selection. We then perform two linear regression steps: once on the subset of features obtained by regularization and once more on the subset obtained by univariate feature selection. Finally, after obtaining the equality constraints, we check whether they satisfy all the data points and discard those that don't.

These steps are summarized in Fig. 2. In the rest of this section, we describe them in more detail.

A. Linear Regression

Given the data set $D = \{d_1, \dots, d_m\}$, we would like to know whether one of the variables (say x_1) bears a linear relationship to the remaining variables, in that there exist real values w_1, \dots, w_n such that for each i , d_{i1} equals (or is close to) $w_1 + w_2 d_{i2} + \dots + w_n d_{in}$. In the method of least-squares regression, for a given vector $W = [w_1, \dots, w_n]^T$, we define the *error* or *loss* it entails by

$$L(D, W) = \sum_{i=1}^m (d_{i1} - (w_1 + \sum_{k=2}^n w_k d_{ik}))^2. \quad (1)$$

We now ask for a weight vector W which minimizes $L(D, W)$.

In matrix form one can express $L(D, W)$ as $\|Y - XW\|^2$, where

$$X = \begin{bmatrix} 1 & d_{12} & d_{13} & \dots & d_{1n} \\ & \vdots & \vdots & & \vdots \\ 1 & d_{m2} & d_{m3} & \dots & d_{mn} \end{bmatrix}$$

and $Y = [d_{11}, \dots, d_{m1}]^T$. A standard result says that the value of W that minimizes $L(D, W)$ can be computed by the closed-form solution:

$$W = (X^T X)^{-1} X^T Y. \quad (2)$$

B. L1 Regularization

L1 regularization is a way of selecting the features that are correlated to the target variable. It seeks to penalize features that are not correlated to the target variable by making their coefficient weights zero.

Mathematically, in L1 regularization, we modify the loss function from Eq (1), by adding a term as follows:

$$L(D, W) = \|Y - XW\|^2 + 2\lambda \|W\|_1. \quad (3)$$

Here λ is a hyperparameter that controls the amount of regularization, and taking the L_1 -norm of W helps penalize outlier features by optimizing them towards 0. The value used for λ is 0.1 in LPGEN.

As the L_1 -norm is not differentiable at zero, its closed form does not exist, so the optimal value, i.e., minimal loss, is obtained by optimization techniques like gradient descent.

The features that are correlated to x_1 can now be taken to be those x_i 's for which the coefficient w_i is non-zero.

C. Univariate Feature Selection

Univariate feature selection is another technique for finding correlated features. It is typically faster than L1 regularization for large dimensions. Let us say we want to find features that are correlated to x_1 . We first run linear regression for the target variable x_1 by considering a single independent variable x_j at a time, with x_j ranging over x_2, \dots, x_n in turn. Let us say that for a particular variable x_j , linear regression learns the estimate $w_1 + w x_j$. We can now define the F -score of x_j , denoted $F(x_j)$, to be $MSR(x_j)/MSE(x_j)$ where $MSR(x_j)$ is the "mean sum of squares regression" of x_j , defined to be

$$MSR(x_j) = \sum_{i=1}^m (\bar{d} - (w_1 + w d_{ij}))^2$$

where $\bar{d} = (\sum_{i=1}^m d_{i1})/m$ is the mean of d_{11}, \dots, d_{m1} ; and $MSE(x_j)$ is the "mean sum of squares error" of x_j , defined to be

$$MSE(x_j) = \frac{\sum_{i=1}^m (d_{i1} - (w_1 + w d_{ij}))^2}{m - 2}.$$

The F -score is an indicator of how significantly a feature is related to the target variable. The greater the F -score, the greater is its significance.

We now compute the "top- k " features for x_1 as follows. Let \bar{f} be the mean F -score of the variables x_2, \dots, x_n . We now select those features x_j whose F -score is at least as much as \bar{f} , and return these as the features most correlated to x_1 .

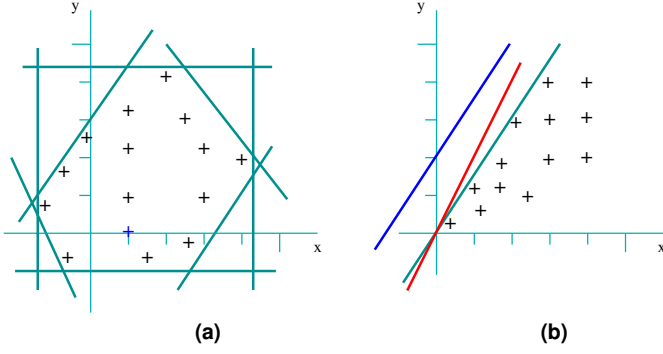


Fig. 3. (a) Structure of inequalities learnt and (b) optimizing slope and intercept

V. INEQUALITY INVARIANTS USING LINEAR PROGRAMMING

We now focus on learning *tight* inequality constraints that a given data set satisfies. We focus on inequalities involving two variables and three variables separately. In both cases, we formulate the problem in terms of an integer linear programming problem.

A. Two Dimensional Inequality Invariants

Our goal is to infer constraints of the form $w_1x + w_2y + b \geq 0$, for a given pair of variables x and y , which are satisfied by the data set D . We also want to find (heuristically) “tight” constraints. As illustrated in Fig. 3(a), we want to find (apart from the two horizontal and two vertical min/max bounds on x and y) four bounding lines corresponding to (a) w_1 being positive and w_2 being negative (the top-left line), (b) w_1 being negative and w_2 being positive (the bottom-right line), (c) w_1 being negative and w_2 being negative (the top-right line), and (d) w_1 being positive and w_2 being positive (the bottom-left line).

Let us focus on case (a) w.r.t. the data set in Fig. 3(b). Intuitively, we want to minimize the *slope* and the *y*-intercept of the line $w_1x + w_2y + b = 0$, subject to the constraint satisfying the data points. Thus, we want the cyan line ($3x - 2y + 0 \geq 0$) with slope $3/2$ and intercept 0 , in preference to the blue line ($3x - 2y + 2 \geq 0$) with slope $3/2$ and intercept 2 , and the red line ($2x - 1y + 0 \geq 0$) with slope 2 and intercept 0 .

Since we cannot formulate this directly as a linear programming problem, we use the following ILP formulation (with x and y corresponding to x_1 and x_2 respectively):

$$\text{minimize } 1000000 + 0.4w_1 + 0.49w_2 + 0.2b \quad (4)$$

subject to:

$$\begin{aligned} w_1d_{11} + w_2d_{12} + b &\geq 0 \\ &\dots \end{aligned} \quad (5)$$

$$\begin{aligned} w_1d_{m1} + w_2d_{m2} + b &\geq 0 \\ w_1, -w_2 &> 0 \end{aligned} \quad (6)$$

where w_1, w_2, b are the integer values we want to infer. We place the restriction that these values should be in the range $[-10^5, 10^5]$. The coefficients in the minimization formulas were based on experimental heuristics. The idea behind the choice of coefficients is shown in Figure 3(b). We want to minimize the slope and intercept to learn the cyan-colored line.

Let’s say we try to get w_1 and w_2 with equal weights in the minimization, i.e. Minimize $w_1 - w_2 + b$. Then it will give $2x - y + 0 \geq 0$ (corresponding to the red line), which is not as tight as the line $3x - 2y + 0 \geq 0$ (cyan line). Due to this reason, we gave a bit more weight to the negative coefficient than the positive one (i.e., 0.4 to w_1 , which is positive, and 0.49 to w_2 , which is negative). Apart from this, 1000000 is added to make the total addition positive, as the LP solver does not support negative minimization objectives.

Similarly, we formulate cases (b)–(d) by changing the objective function (4) as follows:

$$\text{case (b)} \quad 1000000 + 0.49w_1 + 0.4w_2 + 0.2b \quad (7)$$

$$\text{case (c)} \quad 1000000 - w_1 - w_2 + 0.2b \quad (8)$$

$$\text{case (d)} \quad 1000000 + w_1 + w_2 + 0.2b \quad (9)$$

The constraints each model is subject to are (5) and the corresponding version of (6).

We now ask an ILP solver to find optimal values of w_1, w_2, b for each of the four problem instances, giving us four inequality constraints.

This is repeated for each pair of variables.

B. Three Dimensional Inequality Invariants

Inequalities in three dimensions are inferred similar to the case of two dimensions. The goal is to infer constraints of the form $w_1x + w_2y + w_3z + b \geq 0$, for a given triplet of variables x, y and z , which are satisfied by the data set D . Also, we want to find (heuristically) “tight” constraints. Here apart from six min/max bounds for three variables, we find eight bounding lines corresponding to (a) w_1 being positive, w_2 being negative and w_3 being positive, (b) w_1 being positive, w_2 being positive and w_3 being negative, (c) w_1 being positive, w_2 being negative and w_3 being negative, (d) w_1 being negative, w_2 being positive and w_3 being positive, (e) w_1 being negative, w_2 being negative and w_3 being positive, (f) w_1 being negative, w_2 being positive and w_3 being negative, (g) w_1 being negative, w_2 being negative and w_3 being negative, and (h) w_1 being positive, w_2 being positive and w_3 being positive.

For obtaining tighter bounds in three dimensions, we use the following ILP formulation for case (a) (with x, y , and z corresponding to x_1, x_2 and x_3 respectively):

$$\text{minimize } 1000000 + 0.6w_1 + 0.2w_2 + 0.6w_3 + 0.2b \quad (10)$$

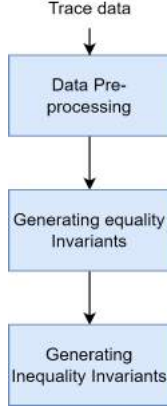


Fig. 4. Steps to generate invariant

subject to:

$$\begin{aligned} w_1 d_{11} + w_2 d_{12} + w_3 d_{13} + b &\geq 0 \\ &\dots \end{aligned} \quad (11)$$

$$\begin{aligned} w_1 d_{m1} + w_2 d_{m2} + w_3 d_{m3} + b &\geq 0 \\ w_1, -w_2, w_3 &> 0 \end{aligned} \quad (12)$$

where w_1, w_2, w_3, b are the integer values we want to infer. We place the restriction that these values should be in the range $[-10^5, 10^5]$.

Similarly, we formulate the cases (b)–(h) by changing the objective function Eq 10 as follows:

$$\text{case (b)} \quad 1000000 + 0.6w_1 + 0.6w_2 + 0.2w_3 + 0.2b \quad (13)$$

$$\text{case (c)} \quad 1000000 + 0.6w_1 - 0.2w_2 - 0.2w_3 + 0.2b \quad (14)$$

$$\text{case (d)} \quad 1000000 + 0.2w_1 + 0.6w_2 + 0.6w_3 + 0.2b \quad (15)$$

$$\text{case (e)} \quad 1000000 - 0.2w_1 - 0.2w_2 + 0.6w_3 + 0.2b \quad (16)$$

$$\text{case (f)} \quad 1000000 - 0.2w_1 + 0.6w_2 - 0.2w_3 + 0.2b \quad (17)$$

$$\text{case (g)} \quad 1000000 - w_1 - w_2 - w_3 + 0.2b \quad (18)$$

$$\text{case (h)} \quad 1000000 + w_1 + w_2 + w_3 + 0.2b \quad (19)$$

The constraints each model is subject to are (11) and the corresponding version of (12).

We now ask an ILP solver to find optimal values of w_1, w_2, w_3, b for each of the eight problem instances, giving us eight inequality constraints.

This is repeated for each triple of variables.

VI. OVERALL LPGEN PROCEDURE

We now describe how we put these techniques together in our tool LPGEN. The tool takes as input a data set (typically obtained by collecting execution traces at the head of the loop in a program), over a set of k named variables. We also supply a max-degree value r , which represents the max degree of terms the tool should consider in the invariants it learns. The tool finally outputs a set \mathcal{C} of candidate invariants. The main steps taken by the tool are shown in Fig. 4, and are described below in more detail.

a) *Data preprocessing*: Initialize \mathcal{C} to empty. Process the data set as follows:

- 1) If any variable x has a constant value l , add $x = l$ as an invariant to \mathcal{C} , and remove x from the data set. This reduces the computation time and chance of overfitting.
- 2) If any two variables x and y always have equal values, collect $x = y$ as an invariant and remove one of the variables from the data set.
- 3) For each remaining variable x , check if $x \bmod l$ (for some integer l) is constant, and add it as an invariant to \mathcal{C} . Also check if $x \& l$ is constant, and if so add it as an invariant to \mathcal{C} .
- 4) For all pairs of variables (x, y) and (w, z) check if $\gcd(x, y) = \gcd(w, z)$, and if so add it to \mathcal{C} .
- 5) Generate non-linear terms and data from the remaining variables upto r degree. For example, if the variables are (x, y) and $r = 2$, we get $\{x, y, x^2, xy, y^2\}$. The non-linear terms will help to find non-linear equality invariants. The number of terms from k variables of degree at most r is $n = \binom{k+r}{r} - 1$, which is now the dimension of our modified data set.
- 6) Finally, generate min and max bounds for each variable, and add them as invariants to \mathcal{C} .

b) *Learning Equalities*: We now apply the linear regression based technique described in Sec. IV on this data set. For each of the n variables, we first apply L1 regularization and univariate feature selection to find a set of correlated variables, followed by linear regression using these as the independent variables. This gives us $2n$ equality invariants. Each of these constraints is first checked to see if it satisfies the given data, and if so it is added to \mathcal{C} .

c) *Learning Inequalities*: Next we apply the techniques of Sec. V to obtain inequality constraints on pairs and triples of variables respectively. We considered only linear inequality learning, i.e., applying inequality learning as mentioned in Sec. V only on the original set of variables and not on the higher degree variables. This can be extended to non-linear inequality learning but at the cost of increased computation time and verification complexity. After inferring inequality invariants, they are checked for consistency with the data set before adding them to \mathcal{C} .

We now return the set \mathcal{C} as the set of candidate invariants.

VII. EXPERIMENTAL EVALUATION

We have implemented LPGEN in Python. We use the `scikit-learn` [19] library to perform feature selection and linear regression. We use the `PuLP` Python linear programming toolkit [14] for inferring inequality invariants.

The aims of our evaluation were threefold:

- (Adequacy) Whether the set of candidate invariants are sufficient to prove pre-post conditions for programs.
- (Strength) How strong are the adequate invariants?
- (Efficiency) How much time does it take to generate the invariants.

With these goals in mind, we evaluated the performance of LPGEN on a variety of pre-post verification benchmark suites

nla-digbench												
	Daikon			DIG			GCLN			LPGen		
Program	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)
cohenca_1	N	3.23	7.47	N	TO	0.06	0/5	74.13	75.10	Y	12.90	57.09
cohendiv_1	Y	1.43	4.36	N	TO	0.11	5/5	18.46	20.67	Y	8.01	17.75
cohendiv_2	Y	1.68	3.67	N	TO	0.06	5/5	26.61	28.22	Y	7.48	13.58
dijkstra_1	Y	1.10	1.50	N	TO	0.09	5/5	0.04	0.56	Y	0.17	0.40
dijkstra_2	N	1.33	2.23	N	TO	0.10	4/5	25.62	27.00	Y	4.66	24.35
divbin_1	Y	1.19	1.65	N	TO	0.05	5/5	17.58	18.17	Y	0.84	1.41
divbin_2	Y	1.42	3.92	N	TO	0.09	5/5	22.13	22.95	Y	4.70	7.49
egcd_1	Y	4.28	32.72	N	TO	0.07	0/5	66.75	67.32	Y	45.56	183.36
egcd2_1	Y	2.26	9.15	N	TO	0.10	0/5	18.17	24.28	Y	30.36	65.89
egcd2_2	Y	5.25	7.95	N	TO	0.11	0/5	50.23	50.34	Y	95.16	113.66
egcd3_1	Y	2.87	4.91	N	TO	0.06	1/5	27.21	30.84	Y	35.54	47.78
egcd3_2	Y	3.90	6.96	N	TO	0.11	0/5	24.94	25.14	Y	75.57	86.50
egcd3_3	Y	5.30	12.26	N	TO	0.07	0/5	32.46	32.72	Y	136.11	154.69
fermat1_1	N	1.46	3.27	Y	62.26	69.78	4/5	16.66	22.81	Y	4.31	11.31
fermat1_2	N	1.73	2.68	N	TO	0.05	4/5	16.94	17.38	Y	5.04	45.94
fermat1_3	N	1.71	2.28	N	TO	0.06	5/5	17.49	17.90	Y	4.52	10.02
fermat2_1	N	1.66	2.42	N	TO	0.10	4/5	16.57	21.95	Y	3.88	21.42
freire1_1	N	1.24	1.62	N	ERROR	0.06	5/5	27.78	28.11	Y	1.22	1.88
freire2_1	Y	5.80	7.04	N	ERROR	0.05	0/5	51.43	51.67	Y	18.11	29.81
geo1_1	Y	1.40	2.70	N	TO	0.12	5/5	24.75	25.65	Y	7.15	9.15
geo2_1	Y	1.48	3.51	N	TO	0.06	5/5	24.79	26.82	Y	7.21	9.68
geo3_1	N	2.94	7.76	N	TO	0.10	5/5	60.56	66.16	Y	34.57	36.57
hard_1	Y	1.36	2.98	N	TO	0.06	5/5	0.04	0.48	Y	2.60	4.04
hard_2	Y	1.63	12.01	N	TO	0.11	5/5	24.68	26.47	Y	8.58	25.29
knuth1_1	N	4.85	41.12	N	TO	0.07	0/5	42.64	42.68	Y	145.93	189.00
lcm1_1	Y	1.62	6.76	N	TO	0.06	0/5	17.53	19.20	Y	9.57	39.10
lcm1_2	Y	1.81	2.70	N	TO	0.06	5/5	28.98	34.27	Y	12.78	97.39
lcm1_3	Y	1.74	3.04	N	TO	0.06	5/5	19.18	24.12	Y	13.99	19.17
lcm2_1	Y	1.72	4.00	N	TO	0.06	2/5	19.46	19.90	Y	13.71	21.70
mannadiv_1	N	1.76	2.64	N	TO	0.05	5/5	25.21	25.73	Y	6.91	8.69
prod4br_1	N	2.01	4.98	N	TO	0.06	5/5	24.27	25.13	Y	16.68	24.07
prodbin_1	Y	1.39	22.64	Y	102.78	103.27	5/5	18.19	18.78	Y	6.47	24.24
ps2_1	Y	1.23	1.94	Y	6.21	6.98	5/5	17.84	18.57	Y	0.86	1.67
ps3_1	N	1.10	4.25	Y	6.49	7.07	5/5	17.40	28.07	Y	0.77	5.91
ps4_1	N	1.33	115.72	Y	8.38	9.71	5/5	19.82	25.01	Y	5.09	10.11
ps5_1	N	1.63	209.76	Y	33.37	33.96	0/5	19.02	19.53	Y	13.23	24.43
ps6_1	N	1.97	35.76	N	TO	0.28	0/5	19.33	21.35	Y	5.49	14.34
sqrt1_1	Y	1.51	2.66	N	154.06	154.31	3/5	34.23	35.65	Y	2.74	16.19

TABLE II
PERFORMANCE OF TOOLS ON THE NLA-DIGBENCH BENCHMARK

loop-invariants												
	Daikon			DIG			GCLN			LPGen		
Program	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)
bin-suffix-5	N	1.36	1.52	Y	0.45	3.59	0/5	0.02	0.43	Y	0.02	0.24
const	Y	1.11	1.39	Y	0.11	0.25	5/5	0.01	0.31	Y	0.01	0.11
eq1	Y	1.25	1.51	Y	6.10	6.57	5/5	2.55	3.51	Y	0.15	0.36
eq2	Y	1.49	1.77	Y	12.71	13.41	5/5	28.60	29.63	Y	0.29	0.60
even	N	1.15	1.26	Y	0.21	0.35	0/5	0.01	0.16	Y	0.01	0.17
linear-inequality-inv-a	N	1.34	1.91	N	23.86	24.05	0/5	1.53	2.61	Y	3.68	5.47
linear-inequality-inv-b	N	1.58	1.75	N	17.63	17.76	5/5	1.96	3.26	Y	3.54	5.18
mod4	N	1.18	1.28	Y	0.50	0.64	0/5	0.02	0.16	Y	0.01	0.18
odd	N	1.37	1.46	Y	0.32	0.50	0/5	0.02	0.16	Y	0.01	0.10

TABLE III
PERFORMANCE OF TOOLS ON THE LOOP-INVARIANTS BENCHMARK

from SVCOMP [2], and also did the same with three state-of-the-art data-driven invariant generation tools Daikon [5], DIG [16], and GCLN [23]. We first generate execution traces for each program by running them on manually provided inputs that satisfy the given precondition, and collect data at the head of the loop across iterations and runs. We then run the tools to generate the set of candidate invariants. We then use our implementation of the Houdini algorithm (using the Z3 solver [4]) to find the strongest conjunctive subset that is adequate to prove the programs.

We use SVCOMP benchmarks named `nla-digbench` (these typically need non-linear arithmetic invariants), `loop-invariants` (linear invariants) and `loop-zilu`

(mix of linear and non-linear invariants). Most of the programs are single loop programs. For the few programs containing multiple or nested loops, we make sub-problems out of these by considering the invariant of the outer loop as the *pre* for the inner loop. and the documented invariant of the inner loop as the post condition. The loop body is manually simplified in the case of nested loops for the inductiveness check. For the `nla-digbench` benchmark, the adequacy and inductiveness conditions are taken from the GCLN implementation [23]. For nested loops data is collected separately for each loop and it is treated as a separate problem for invariant inference. While running Houdini however, the loop body needs to be engineered carefully capturing the semantics of the loop for

loop-zilu												
Program	Daikon			DIG			GCLN			LPGen		
	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)	Adq	Inv (s)	Total (s)
benchmark01_conjunctive	N	1.11	1.23	Y	0.86	1.05	5/5	0.29	0.74	Y	0.02	0.12
benchmark02_linear	Y	1.20	1.49	Y	18.70	18.87	5/5	0.56	1.11	Y	0.73	1.14
benchmark02_linear_abstracted	Y	1.20	1.49	Y	18.70	18.87	5/5	0.56	1.11	Y	0.73	1.14
benchmark03_linear	Y	1.53	1.99	Y	21.41	22.26	5/5	28.46	29.28	Y	0.77	1.44
benchmark04_conjunctive	N	1.48	1.91	Y	43.14	43.43	0/5	1.71	2.16	Y	0.92	1.46
benchmark05_conjunctive	Y	1.41	1.71	Y	17.70	17.88	5/5	26.15	26.82	Y	0.77	1.30
benchmark06_conjunctive	Y	1.39	1.82	Y	85.33	85.75	2/5	30.34	31.23	Y	7.06	10.59
benchmark07_linear	N	1.33	1.54	N	3.46	3.84	0/5	24.96	25.42	Y	1.80	2.28
benchmark08_conjunctive	Y	1.44	1.72	Y	10.98	11.33	5/5	1.55	2.14	Y	0.70	1.21
benchmark09_conjunctive	Y	1.18	1.46	Y	1.41	1.58	5/5	0.56	1.02	Y	0.02	0.12
benchmark10_conjunctive	N	1.36	1.54	Y	1.78	1.95	5/5	0.51	0.81	Y	0.13	0.35
benchmark11_linear	Y	1.33	1.58	Y	4.58	4.74	5/5	0.56	0.97	Y	0.16	0.34
benchmark11_linear_abstracted	Y	1.33	1.58	Y	4.58	4.74	5/5	0.56	0.97	Y	0.16	0.34
benchmark12_linear	Y	1.23	1.63	Y	7.53	7.77	0/5	1.34	2.03	Y	10.69	11.09
benchmark13_conjunctive	Y	1.40	1.64	Y	10.88	11.07	5/5	1.57	2.46	Y	0.17	0.35
benchmark14_linear	N	1.25	1.38	Y	0.42	0.61	5/5	0.01	0.22	Y	0.01	0.11
benchmark15_conjunctive	Y	1.34	1.69	Y	22.30	22.49	5/5	28.37	28.86	Y	1.27	2.08
benchmark16_conjunctive	N	1.19	1.35	Y	3.28	3.95	0/5	25.46	25.74	Y	0.16	0.36
benchmark17_conjunctive	Y	1.46	1.85	Y	14.44	14.64	5/5	1.73	2.35	Y	0.21	0.42
benchmark18_conjunctive	Y	1.20	1.48	Y	13.33	13.81	5/5	1.73	2.40	Y	0.19	0.38
benchmark19_conjunctive	Y	1.23	1.53	Y	18.63	18.87	5/5	1.72	2.48	Y	0.24	0.44
benchmark20_conjunctive	Y	1.27	1.69	Y	25.93	26.12	5/5	1.73	2.28	Y	2.48	2.82
benchmark21_disjunctive	N	1.51	1.66	N	5.14	5.30	0/5	0.51	0.75	N	0.24	0.34
benchmark22_conjunctive	N	1.30	1.58	Y	1.83	2.08	0/5	0.47	0.65	Y	0.14	0.33
benchmark23_conjunctive	N	1.43	1.80	N	1.96	2.07	5/5	24.69	25.07	Y	0.14	0.36
benchmark24_conjunctive	N	1.50	1.81	Y	16.34	16.63	0/5	28.09	28.66	Y	0.90	1.45
benchmark25_linear	N	1.10	1.19	Y	0.29	0.48	5/5	0.01	0.33	Y	0.01	0.11
benchmark25_linear_abstracted	N	1.10	1.19	Y	0.29	0.48	5/5	0.01	0.33	Y	0.01	0.11
benchmark26_linear	Y	1.38	1.73	Y	4.71	4.87	5/5	0.50	0.74	Y	0.17	0.34
benchmark26_linear_abstracted	Y	1.38	1.73	Y	4.71	4.87	5/5	0.50	0.74	Y	0.17	0.34
benchmark27_linear	N	1.42	1.61	Y	26.51	26.70	0/5	27.74	28.03	Y	0.81	1.27
benchmark28_linear	Y	1.16	2.77	Y	3.40	4.04	5/5	0.57	0.90	Y	0.14	0.39
benchmark29_linear	N	1.25	1.42	N	4.27	4.32	0/5	0.48	0.66	Y	0.19	0.36
benchmark30_conjunctive	Y	1.08	1.47	Y	2.24	2.39	5/5	0.52	0.73	Y	0.02	0.12
benchmark31_disjunctive	N	1.25	1.39	Y	6.72	6.92	0/5	0.51	0.62	N	0.14	0.28
benchmark32_linear	Y	1.39	1.61	Y	0.12	0.37	5/5	21.39	21.52	Y	0.01	0.12
benchmark33_linear	N	1.35	1.47	Y	0.25	0.36	5/5	0.01	0.15	Y	0.01	0.11
benchmark34_conjunctive	Y	1.60	2.19	Y	33.46	33.78	5/5	30.93	31.25	Y	1.24	1.61
benchmark35_linear	N	1.04	1.17	Y	0.10	0.25	5/5	0.01	0.16	Y	0.01	0.12
benchmark36_conjunctive	N	1.30	1.42	Y	2.47	2.90	5/5	0.52	0.78	Y	0.01	0.11
benchmark37_conjunctive	N	1.30	1.46	Y	2.35	2.52	5/5	0.52	0.78	Y	0.02	0.11
benchmark38_conjunctive	Y	1.50	1.75	Y	3.73	3.88	5/5	26.50	26.71	Y	0.18	0.38
benchmark39_conjunctive	Y	1.15	1.49	Y	1.82	1.97	5/5	24.90	25.10	Y	0.16	0.37
benchmark40_polynomial	Y	1.56	1.83	N	4.96	5.01	0/5	0.53	0.72	Y	0.21	0.47
benchmark41_conjunctive	Y	1.42	1.72	Y	11.80	12.08	5/5	27.79	28.22	Y	0.17	0.38
benchmark42_conjunctive	Y	1.23	1.54	Y	11.01	11.30	5/5	27.45	27.90	Y	0.18	0.38
benchmark43_conjunctive	N	1.24	1.40	N	5.67	5.79	5/5	0.58	0.79	Y	0.18	0.40
benchmark44_disjunctive	N	1.18	1.32	Y	4.21	4.34	0/5	0.53	0.77	Y	0.16	0.32
benchmark45_disjunctive	N	1.42	1.51	N	3.57	3.80	0/5	0.58	0.71	N	0.18	0.35
benchmark46_disjunctive	N	1.49	1.65	N	22.59	22.71	0/5	1.85	2.12	N	0.96	1.43
benchmark47_linear	N	1.18	1.32	Y	4.63	4.76	0/5	0.59	0.85	Y	0.14	0.31
benchmark48_linear	N	1.36	1.58	Y	30.15	30.40	5/5	27.63	27.99	Y	10.80	11.30
benchmark49_linear	N	1.18	1.31	Y	38.28	38.46	0/5	1.66	1.83	Y	1.18	1.86
benchmark50_linear	N	1.39	1.49	Y	6.25	6.38	0/5	0.57	0.69	Y	0.18	0.35
benchmark51_polynomial	N	1.13	1.26	N	0.17	0.29	5/5	0.01	0.16	Y	0.01	0.11
benchmark52_polynomial	N	1.19	1.33	Y	0.22	0.40	5/5	0.01	0.20	Y	0.01	0.11
benchmark53_polynomial	Y	1.29	1.59	N	2.13	2.21	0/5	0.54	0.72	Y	0.19	0.44

TABLE IV
PERFORMANCE OF TOOLS ON THE LOOP-ZILU BENCHMARK

Benchmark	Total Programs	Without Feature Selection	With Feature Selection
nla-digbench	38	25	38
loop-zilu	57	53	53
loop-invariants	9	9	9

TABLE V
ADEQUATE INVARIANTS FOUND BY LPGEN WITH AND WITHOUT FEATURE SELECTION

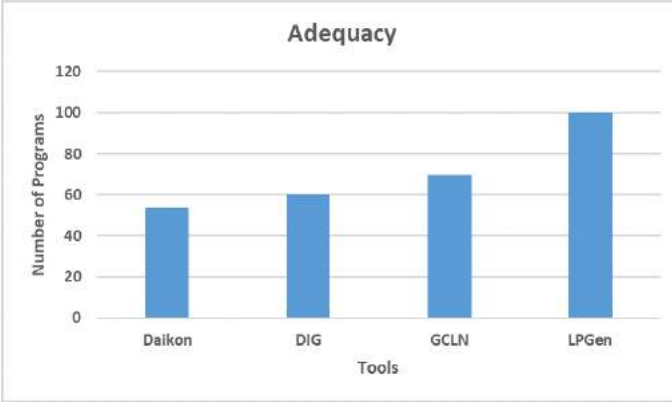


Fig. 5. Number of adequate invariants inferred by tools

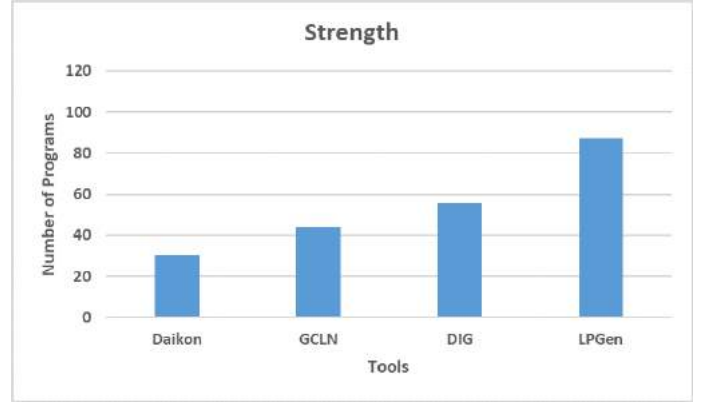


Fig. 6. Comparison of strength of invariants generated by each tool vs others

checking inductive conditions. Moreover, invariants involving gcd and bitwise-and are manually checked because Z3 does not support gcd and sometimes times out for bitwise operations. So we generated the gcd and bitwise-and during the procedure (i.e., pre-processing stage) but manually decided whether to include it in the conjunctive invariant returned by Houdini on the remaining candidate invariants.

Our experiments were performed on Google Colab, with a single CPU and 12 GB of RAM.

The results are summarized in Tables II, III and IV. The **Program** column specifies the name of the program along with the loop index, For example, “lcm1_2” represents the lcm1 program, and the loop index is 2 as it has 3 different loops within the program. The **Adq** column represents whether an adequate conjunctive subset was found or not. Column **Inv** gives the time in seconds taken to infer the set of candidate invariants, and **Total** gives the total time in seconds for inferring candidate invariants and running the Houdini algorithm.

In the **Adq** column “Y” represents the fact that an adequate invariant was found, and “N” represents the fact no adequate invariant was found. As GCLN infers invariants non-deterministically, its **Adq** column shows out of 5 runs (with different random seeds) how many times we were able to get an adequate invariant. “TO” in the **Inv** column means the tool didn’t compute any candidate invariants within 300 seconds. “ERROR” means that the tool crashes with some error.

Fig. 5 summarizes the total number of programs for which each tool inferred an adequate invariant. Out of 104 programs over the three benchmarks we got adequate invariants for 100 of them. The remaining four required disjunctive invariants which are out of the scope of our tool. GCLN was the closest with adequate invariants for 70 programs.

Fig. 6 summarizes the results of the strength of the adequate invariants learnt by each tool. For each tool, we check whether the adequate invariant it found *implies* that of the other tools. If so it earns one count. LPGEN generates the largest number of strong adequate invariants (87 programs). DIG was the next closest with 56 programs.

Table V summarizes the total number of programs for which adequate invariants were inferred by LPGEN, with and without using feature selection. When the number of features (i.e. dimension of the data) is high, as in the case of *nla-digbench* benchmarks, feature selection is important as without using it, LPGEN was able to come up with only 25 adequate invariants out of 38 programs. Both *loop-zilu* and *loop-invariants* benchmarks had small dimension of the data and thus got the same number of adequate invariants even without feature selection. But when the dimension increases, feature selection plays an important role, as in the case of *nla-digbench*. Among the two feature selection techniques, L1 Regularization is more effective, and out of 38 programs of *nla-digbench*, LPGEN found 37 adequate invariants by using only L1 regularization as a feature selection technique. Only 12 were found to be adequate out of 38 using only the univariate feature selection technique. By combining both, we got adequate invariants for all 38 programs in *nla-digbench*.

In terms of efficiency in generating invariants, the average time taken by LPGEN to infer candidate invariants is 8.32s which is around 5x slower than the average time of the fastest tool (Daikon) which is 1.56s. However we note that the total number of adequate invariants inferred by our tool is almost twice that of Daikon.

The LPGEN tool can be accessed at:
<https://github.com/Arkesh-Thakkar/LPGen>

VIII. RELATED WORK

We classify related work broadly according to whether they are white-box or black-box (depending on whether they use the source of the program or not).

There are several white-box invariant learning techniques that use the program text to come up with invariants that are sufficient to prove the correctness of the program. These include [7] and [3] which use constants and expressions from the program text to infer candidate invariants.

Among black-box techniques we consider two categories: techniques that use multi-round learning and techniques that

use single-shot learning. In the first category the ICE-learning line of work features prominently [6], [10], [11]. These techniques are able to learn both conjunctive and disjunctive invariants using decision-tree based learning. However they do not use execution traces of the program, and instead rely on counterexamples provided by a Teacher who has access to the program.

In the second category, where our work falls, the earliest and most prominent work is that of Daikon [5]. Daikon essentially checks whether 75 different types of invariant templates satisfy the trace data, and report those that do as candidate invariants.

Nguyen et al [16], [17] and Sharma et al [21] infer polynomial equalities from trace data, using algebraic equation solving techniques. [16] also infer polynomial inequality invariants using a convex polyhedron approach, and its relaxed version, octagonal inequalities, by creating a convex hull in two dimensions.

Finally, [23] and [20] proposed a novel neural architecture called GCLN and CLN architectures, respectively, to learn loop invariants. They use fuzzy logic based gates called T -norms and T -conorms representing the continuous versions of conjunction and disjunction. The GCLN tool uses a Piecewise Biased Quadratic Unit (PBQU) activation function for inferring inequality invariants.

All these works differ from ours in various aspects. Daikon is limited to generating equality invariants in only three dimensions and based on predefined templates. The DIG approach to find equality invariants with equation solvers which are costly for finding complex equality invariants, and finding inequality invariants by constructing convex hulls can be exponential in time complexity. The GCLN line of work is nondeterministic in nature due to random dropouts and lack of a proper feature selection technique. We overcome the above limitations as our tool allows equality invariants in n -dimensions and with proper feature selection techniques, namely L1 Regularization and univariate feature selection, we were able to find a good set of equality invariants. For inequalities, we use Linear Programming (LP) solvers for finding inequality invariants in two and three dimensions. LP solvers are much faster than constructing polyhedrons and with the optimization objective given in Sec.V, we were able to find tighter bounds.

IX. CONCLUSION

In this work we have presented a novel combination of techniques, including linear regression, feature selection, and linear optimization, to learn tight invariants from execution trace data. Our experimental evaluation shows that our tool is able to learn strong adequate conjunctive invariants for a variety of programs requiring linear and non-linear predicates, in a reasonably efficient manner.

Going ahead we would like to extend our work to learning disjunctive invariants, in both single shot and multi-round teacher-learner settings.

REFERENCES

- [1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In J. Launchbury and J. C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 4–16. ACM, 2002.
- [2] D. Beyer. Progress on software verification: SV-COMP 2022. In *Proc. TACAS (2)*, LNCS 13244, pages 375–402. Springer, 2022.
- [3] P. Darke, S. Prabhu, B. Chimdyalwar, A. Chauhan, S. Kumar, A. Basakchowdhury, R. Venkatesh, A. Datar, and R. K. Medicherla. VeriAbs: Verification by Abstraction and Test Generation - (Competition Contribution). In *TACAS, Part II*, volume 10806, pages 457–462, 2018.
- [4] L. De Moura and N. Björner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [5] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.
- [6] P. Ezudheen, D. Neider, D. D’Souza, P. Garg, and P. Madhusudan. Horn-ice learning for synthesizing invariants and contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):131:1–131:25, 2018.
- [7] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*, pages 170–178. IEEE, 2018.
- [8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. International Symposium of Formal Methods Europe (FME 2001), Berlin, Germany, 2001*, pages 500–517, 2001.
- [9] R. W. Floyd. Assigning Meanings to Programs. In J. Schwartz, editor, *Proc. Symposium on Applied Mathematics*, volume 19. American Mathematical Society, 1967.
- [10] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.
- [11] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices*, 51(1):499–512, 2016.
- [12] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In L. A. Clarke, L. Dillon, and W. F. Tichy, editors, *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 60–73. IEEE Computer Society, 2003.
- [13] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [14] S. Mitchell, M. OSullivan, and I. Dunning. Pulp: a linear programming toolkit for python. *The University of Auckland, Auckland, New Zealand*, 65, 2011.
- [15] D. Neider, S. Saha, P. Garg, and P. Madhusudan. Sorcar: Property-driven algorithms for learning conjunctive invariants. In B. E. Chang, editor, *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11822 of *Lecture Notes in Computer Science*, pages 323–346. Springer, 2019.
- [16] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 683–693. IEEE, 2012.
- [17] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–30, 2014.
- [18] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In P. G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSA 2002, Roma, Italy, July 22-24, 2002*, pages 229–239. ACM, 2002.
- [19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [20] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana. Cln2inv: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542*, 2019.

- [21] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*, pages 574–592. Springer, 2013.
- [22] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Autom. Softw. Eng.*, 13(3):345–371, 2006.
- [23] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 106–120, 2020.

Automating Cutoff-based Verification of Distributed Protocols

Shreesha G. Bhat and Kartik Nagar

Department of CSE, IIT Madras

Chennai, India

shreesha@iitm.ac.in, nagark@cse.iitm.ac.in

Abstract—Distributed protocols are generally parametric and are expected to work correctly on systems containing any number of nodes. Therefore, proving their correctness becomes an infinite state verification problem. The usual approach for verifying distributed protocols is to provide an inductive invariant that is strong enough to imply the safety property. But inductive invariants for even simple distributed protocols can be intricate and synthesizing them in an automated manner is a hard problem. In this work, we investigate an orthogonal cutoff-based technique for verifying distributed protocols. In a cutoff-based approach, one provides a finite-sized instance of the system which encompasses all possible modes of violation of the safety property. Analyzing such a cutoff instance for safety violations suffices to prove the correctness of the protocol for all instances. In this work, we formalize a simulation-based approach to check whether a given instance is a cutoff instance for protocols written in a general modelling language (RML) by identifying sufficient conditions which can be efficiently encoded in SMT. We propose simple static analyses to automatically synthesize the cutoff instance, simulation relation and other proof components, thus leading to a fully automated verification procedure. Finally we apply our technique on a number of protocols ranging from simple leader election and mutual exclusion protocols to complex quorum-based consensus protocols.

I. INTRODUCTION

Distributed protocols allow disparate nodes to work together towards completing a task, and form the backbone of today's distributed systems. These protocols are typically specified in a parametric fashion, which means they can be instantiated on a system with any number of nodes. The nodes communicate with each other through message passing, and these messages can be arbitrarily delayed or even lost. However, the distributed protocol is expected to work correctly under all such conditions. Here, correctness is typically defined in terms of a safety property which must be obeyed by every node at every step of the protocol. For example, the safety property of a distributed mutual exclusion protocol would say that two nodes should not be in their critical section at the same time. Since the protocols need to consider every possible network behavior, they are quite complex in nature. Verifying the correctness of distributed protocols then becomes highly important, but this problem is significantly complicated by the parametric nature of the protocol and the asynchronous, non-deterministic nature of the underlying network. Essentially, every possible instantiation of the protocol needs to be proven correct, and each such instantiation itself needs to consider a

large number of network behaviors. Further, there could be an infinite number of instantiations of the protocol.

Recent approaches ([1]–[6]) to verifying distributed protocols typically aim to find an inductive invariant, which is a property of the protocol state satisfied at every step of any protocol instance, which is inductive in nature and is stronger than the safety property. However, finding an inductive invariant is very hard, as conceptually, it should encompass all the complex logic that the protocol employs to maintain the safety property *under any abnormal network behavior in any instantiation*. In this work, we consider an alternative cutoff-based approach to protocol verification that cleanly separates the two problems of dealing with *arbitrary instantiations* and *arbitrary network behavior*. This approach requires a cutoff instance with a fixed, finite number of nodes whose correctness implies the correctness of any arbitrary protocol instance. Then, we only need to consider how the protocol maintains the safety property under arbitrary network behavior in the cutoff instance. Further, since the cutoff instance will have a constant, finite number of nodes, verifying its correctness becomes a finite state verification problem, which can be solved in an automated fashion.

In this paper, we focus on the problem of finding such a cutoff instance, and automatically showing that it is indeed a cutoff. The definition of a cutoff instance gives us the following characterization: *if there exists a violation of the safety property in any arbitrary protocol instance, then there should also exist a violation in the cutoff instance*. We automatically construct a cutoff instance which can simulate any violation of the safety property in any arbitrary protocol instance. While this seems like a tall order, we hypothesize that this problem is simpler due to two reasons: (i) a violation of the safety property typically involves only a small number of nodes (for example, a violation of the mutual exclusion property would only require *two* nodes to be in their critical section together), and further, the participation of other nodes of the system is either not required, or can be simulated by the violating nodes themselves, and (ii) most of the complex logic in the protocol implementation which ensures the absence of a violation can be side-stepped, since we are actually interested in simulating the presence of a violation.

While previous works have also attempted to use cutoff based approaches for verification ([7]–[10]), they have mostly been limited to either a restricted class of protocols [8] with

strong assumptions on the underlying network or a restricted class of specifications [9]. In this work, we consider a variety of protocols targeting different goals (consensus, mutual exclusion, key-value store, etc.) and do not make any assumptions about the underlying network. Our approach takes as input the protocol description written in the Relational Modeling Language (RML). We first develop a formalization of the cutoff approach which defines sufficient conditions for proving that a given protocol instance is a cutoff instance, which can be encoded using SMT. We then use our hypothesis concerning the simplicity of the cutoff instance to develop a static analysis based approach which directly synthesizes the cutoff instance from a violation of the safety property. Beginning from a state which violates the safety property, our analysis moves backwards to identify the necessary protocol actions and state components that could be involved in a violation. We then use the output of the static analysis to create a cutoff instance which faithfully simulates all the protocol actions and state components which could be involved in a violation. Finally, we apply our SMT encoding to check the correctness of the synthesized cutoff instance. We have implemented the proposed approach and applied it on 8 different distributed protocols, providing a fully automated cutoff-based proof of correctness for all of them.

To summarize, we make the following contributions:

- 1) We formalize the cutoff approach for distributed protocols written in the RML language, and identify sufficient conditions for proving the correctness of a cutoff instance.
- 2) We propose a simple static analysis-based approach to automatically synthesize from the protocol description, a cutoff instance and a simulation relation for proving the correctness of the cutoff.
- 3) We have implemented the approach in a prototype tool and have successfully verified 8 challenging protocols.

The rest of the paper is organized as follows: In §2, we illustrate the cutoff-based approach to protocol verification and our synthesis algorithm using an example. We formalize the cutoff approach for protocols written in RML in §3 and §4. Details of our synthesis algorithm are presented in §5. Experimental results are given in §6, followed by a discussion on related works and conclusion in §7.

II. MOTIVATING EXAMPLE: THE SHARDED KEY-VALUE STORE

A. Protocol Description

As a motivating example to demonstrate our technique, we consider the sharded key-value store protocol described in [1]. The protocol maintains key-value pairs distributed across a set of nodes. It implements a mechanism for nodes to *reshard* key-value pairs amongst one another in the presence of an unreliable network while maintaining the safety property that no two nodes should ever own a key simultaneously. A detailed pseudocode description of the protocol in the RML language [11] is provided below in Fig. 1.

Algorithm 1 The Sharded Key Value Store Protocol

```

1: type key, value, node, seqnum
2: relation table : node, key, value
3: relation transfer_msg : node, node, key, value, seqnum
4: relation ack_msg : node, node, seqnum
5: relation seqnum_sent : node, seqnum
6: relation unacked : node, node, key, value, seqnum
7: relation seqnum_rcvd : node, node, seqnum
8: init  $\forall n_1, n_2, k, v_1. \text{table}(n_1, k, v_1) \wedge \text{table}(n_2, k, v_2) \implies n_1 = n_2 \wedge v_1 = v_2 \triangleright$  All other relations are empty
9: action Reshard(n_old : node, n_new : node, k : key, v : value, s : seqnum)
10:   require table(n_old, k, v)  $\wedge \neg \text{seqnum\_sent}(s)$ 
11:   seqnum_sent(s)  $\leftarrow$  true
12:   table(n_old, k, v)  $\leftarrow$  false
13:   transfer_msg(n_old, n_new, k, v, s)  $\leftarrow$  true
14:   unacked(n_old, n_new, k, v, s)  $\leftarrow$  true
15: action DropTransferMsg(src : node, dst : node, k : key, v : value, s : seqnum)
16:   require transfer_msg(src, dst, k, v, s)
17:   transfer_msg(src, dst, k, v, s)  $\leftarrow$  false
18: action Retransmit(src : node, dst : node, k : key, v : value, s : seqnum)
19:   require unacked(src, dst, k, v, s)
20:   transfer_msg(src, dst, k, v, s)  $\leftarrow$  true
21: action RecvTransferMsg(src : node, dst : node, k : key, v : value, s : seqnum)
22:   require transfer_msg(src, dst, k, v, s)  $\wedge \neg \text{seqnum\_rcvd}(s)$ 
23:   seqnum_rcvd(s)  $\leftarrow$  true
24:   table(dst, k, v)  $\leftarrow$  true
25: action SendAck(src : node, dst : node, k : key, v : value, s : seqnum)
26:   require transfer_msg(src, dst, k, v, s)  $\wedge \text{seqnum\_rcvd}(s)$ 
27:   ack_msg(s)  $\leftarrow$  true
28: action DropAckMsg(src : node, dst : node, k : key, v : value, s : seqnum)
29:   require ack_msg(s)
30:   ack_msg(s)  $\leftarrow$  false
31: action RecvAckMsg(src : node, dst : node, k : key, v : value, s : seqnum)
32:   require ack_msg(s)
33:   unacked(src, dst, k, v, s)  $\leftarrow$  false
34: action Put(n : node, k : key, v : value)
35:   require  $\exists v'. \text{table}(n, k, v')$ 
36:   table(n, k, *)  $\leftarrow$  false
37:   table(n, k, v)  $\leftarrow$  true
38: safety  $\forall k, n_1, n_2, v_1, v_2, k. \text{table}(n_1, k, v_1) \wedge \text{table}(n_2, k, v_2) \implies n_1 = n_2 \wedge v_1 = v_2$ 

```

The protocol is described using a set of types, relations and actions. A type (or sort in RML terminology) is defined for nodes, keys, values and sequence numbers. The relations describe the state of the protocol and are defined over these sorts. In a step of the execution, any action can be fired provided that its guard (specified by the **require** keyword) is satisfied.

The relation *table*(*n, k, v*) indicates that the node *n* holds the key *k* with the value *v*. A Reshard action generates a *transfer_msg* from the key's current owner to its new owner. Transfer messages can be arbitrarily dropped (through the *DropTransferMsg* action), and hence the protocol employs

an acknowledgment mechanism, whereby the new owner needs to send an acknowledgment message upon receiving a *transfer_msg*, and the current owner will keep re-transmitting (through the Retransmit action) until it receives an acknowledgment. The acknowledgement message itself can be dropped and might require re-transmission. Since each *transfer_msg* message is tagged with a unique sequence number, the receiving node can ignore duplicate *transfer_msg*'s that arise from the re-transmission mechanism by marking the sequence number as received in line 24; the absence of which is used as a guard by RecvTransferMsg action. This prevents safety violations that can occur due to older transfer messages entering their out-of-date key value pair into the table of the destination node after it has already been re-sharded to some other node, or subsequent Put actions have occurred thereby altering the associated value.

B. Cutoff based Verification

The safety property for this protocol says that in all runs, we cannot have two different table entries for the same key. Intuitively, this is maintained at all times, because either a single node contains the key in its table, or the key is in-transit. The unique sequence number associated with a *transfer_msg* ensures that re-transmissions do not break the safety property. Prior works [1], [11] construct a complex inductive invariant which leverages the above observation to show the uniqueness of a number of state components, and ultimately implies the safety property. In this work, we take an orthogonal approach where we assume the existence of a hypothetical violation and focus on (1) identifying the key state components and actions of the protocol that contribute to this violation, and (2) simulating this violation by maintaining these state components in a fixed, small protocol instance. If the cutoff instance can be shown to simulate any violation of the safety property, proving the safety of the cutoff instance is sufficient to establish correctness for all instances of the protocol. This essentially formalizes the ‘small model’ property that has been empirically established by many prior works for bugs in concurrent and distributed systems. Note that while synthesizing the cutoff instance, we can completely ignore how the protocol blocks out potential scenarios where a violation can occur, which is one of the classical hurdles in crafting inductive invariants. For the sharded key-value store protocol, we show that a cutoff instance with 2 nodes can simulate all possible violations in arbitrary sized instances of the protocol (note that size refers to number of nodes).

C. Static Analysis

We employ a static analysis based approach on the protocol description to find out the relevant state components and actions that are necessary for simulating violations of the safety property. Consider a violation in an arbitrary size system L where we have two distinct nodes A_L, B_L and key K such that $table(A_L, K, V_1)$ and $table(B_L, K, V_2)$ hold. We are interested in collecting the relevant state components and actions that are responsible for this violating state of L . At

a very high level, our static analysis starts from the state components directly involved in the violation, and then finds actions which can set these state components. However, for these actions to be enabled, their guards will also need to be maintained. So the state components in the guards also now become relevant, and the above process continues until no new relevant actions or state components are found.

For the sharded key value store protocol, we start with the state components that are involved in the violation of the safety property as the initial set of relevant state components, $S = \{table(A_L, K, V_1), table(B_L, K, V_2)\}$. Consider the actions that set the clauses $table(A_L \langle B_L \rangle, K, V_1 \langle V_2 \rangle)$ (we use entries in brackets $\langle \rangle$ to succinctly represent both the clauses). We find that any action of the type $Put(A_L \langle B_L \rangle, K, V_1 \langle V_2 \rangle)$ and $RecvTransferMsg(*, A_L \langle B_L \rangle, K, V_1 \langle V_2 \rangle, *)$ can set these *table* entries, where $*$ represents any value. These are added to the set of relevant actions (denoted by A). Now we consider the components in the guards of these actions. For the *RecvTransferMsg* actions, the guard contains the clauses $\neg seqnum_recvd(*)$ and $transfer_msg(*, A_L \langle B_L \rangle, K, V_1 \langle V_2 \rangle, *)$. For the *Put* actions, we have $\exists v. table(A_L \langle B_L \rangle, K, v)$ as the guard clause. For the existential quantifier, we include $table(A_L \langle B_L \rangle, K, *)$ where the value entry is not restricted and therefore all such table entries are tracked as relevant. These entries are added to the set S .

In this way, we keep on collecting relevant actions and clauses, terminating in a fixed point after a few iterations. We also simplify the sets by noting that $*$ entries subsume other entries that contain specific values in that field. For example, if the S set contains an entry $table(A_L, K, V_1)$ and also an entry $table(A_L, K, *)$, the latter subsumes the former. On performing such reductions, we get the following fixed point sets S and A

$$\begin{aligned} S &= \{table(*, K, *), transfer_msg(*, *, K, *, *), \\ &\quad \neg seqnum_recvd(*), \neg seqnum_sent(*), \\ &\quad unacked(*, *, K, *, *)\} \\ A &= \{Put(*, K, *), RecvTransferMsg(*, *, K, *, *), \\ &\quad Reshard(*, *, K, *, *), Retransmit(*, *, K, *, *)\} \end{aligned}$$

Notice that though the protocol has 8 actions in total, the action set obtained from static analysis shows that only 4 of these actions are actually relevant in a violation. In particular, actions such as *DropTransferMsg* and *SendAck* are not required to simulate a violation. Intuitively, this is because these actions are not necessary to actually transfer a key from one node to another, which is needed for realizing a potential violation. Secondly, although the correctness of the protocol (that is, avoiding a violation) depends on a complex invariant involving uniqueness of a number of state components, we do not require any of that complexity to simulate a violation. The static analysis essentially ignores how exactly a violating state might have been obtained, but instead tries to trace the state components and actions that are essential for recreating the violation. For example, it is

possible that a transfer message may have been dropped by the network in a violating execution, and hence would need to be re-transmitted. However, the cutoff system need not drop the message in the first place (re-transmission is still required). Intuitively, if a violation occurs in L , by maintaining the state components in S and performing only the relevant actions in A , we can recreate the violation in the cutoff system C .

D. Simulation Relation & Lockstep

While the static analysis gives us the relevant state components and actions that need to be maintained in a cutoff system, we still need to prove that any violation in any protocol instance can be simulated by the cutoff instance. To show this, we establish a simulation between any arbitrary instance L and a cutoff instance C . The simulation is primarily governed by a *lockstep* which describes the action(s) taken by the cutoff instance C for every action in L . An action in L is simulated as zero or more actions in C . We also establish a *simulation relation* that holds inductively on the states of both L and C as they progress according to the lockstep. The simulation relation will be strong enough to show that at any step, a violation of the safety property in L will imply a violation in the state of C as well.

The main ingredients of the simulation relation and lockstep have already been identified via the static analysis, i.e. the relevant state components and corresponding actions required to reach a violating state. What remains is to map the relevant state components and actions of L to corresponding components of C . Such a mapping can be obtained by mapping nodes of L to their corresponding simulating node in C . Denoting the node mapping as $sim : \mathcal{D}_L \rightarrow \mathcal{D}_C$ (where \mathcal{D}_x represents the set of nodes in the instance x), the simulation relation maintains that relevant state components from the set S obtained from static analysis corresponding to any node $n \in \mathcal{D}_L$ in L match the corresponding state component of $sim(n)$ in C . The simulation relation does not say anything about the state components which are not relevant for the violation. Similarly, the lockstep ensures that whenever any action from A occurs in L , the corresponding action is triggered in C . The rest of the actions of L are ignored as they are not relevant to simulate the violation.

Specifically, for the sharded key value store protocol, let us denote the two nodes in the cutoff instance as A_C and B_C . Recall that A_L and B_L were nodes of the larger instance L which were involved in the violation. We have $sim(A_L) = A_C$ and $sim(B_L) = B_C$. We map the rest of the nodes to one of A_C or B_C , say B_C i.e. $\forall N \in \mathcal{D}_L. (N \neq A) \wedge (N \neq B) \implies sim(N) = B_C$. Intuitively, a node $N_C \in \mathcal{D}_C$ maintains the state and performs the actions for all the nodes $N_L \in \mathcal{D}_L$ such that $sim(N_L) = N_C$.

Applying the *sim* mapping on the relevant state components

S we get the following 5 clauses in the simulation relation:

- (1) $table_L(n, K, v) \implies table_C(sim(n), K, v)$
- (2) $unacked_L(n_1, n_2, K, v, s) \implies unacked_C(sim(n_1), sim(n_2), K, v, s)$
- (3) $\neg seqnum_sent_L(s) \implies \neg seqnum_sent_C(s)$
- (4) $\neg seqnum_recv_L(s) \implies \neg seqnum_recv_C(s)$
- (5) $transfer_msg_L(n_1, n_2, K, v, s) \implies transfer_msg_C(sim(n_1), sim(n_2), K, v, s)$

Here, we use rel_L and rel_C to denote the relation rel of the protocol for the instances L and C respectively and assume universal quantifiers over all lower-cased variables for each clause. Notice that the simulation relation ensures that any violation of safety property in the protocol state of the larger system (say $table_L(A_L, K, V_1)$ and $table_L(B_L, K, V_2)$) will result in a violation of the cutoff system. The lockstep defines the actions fired in the cutoff instance for actions of the larger instance, and ensures that the above simulation relation is maintained for every step of every execution. For actions not in the lockstep, no action is fired in the cutoff instance. Again, the *sim* mapping and the relevant actions A give the following lockstep:

- (1) $Put_L(n, K, c)$ **is simulated as** $Put_C(sim(N), K, V)$
- (2) $Reshard_L(n_1, n_2, K, v, s)$ **is simulated as** $Reshard_C(sim(n_1), sim(n_2), K, v, s)$
- (3) $Retransmit_L(n_1, n_2, K, v, s)$ **is simulated as** $Retransmit_C(sim(n_1), sim(n_2), K, v, s)$
- (4) $RecvTransferMsg_L(n_1, n_2, K, v, s)$ **is simulated as** $RecvTransferMsg_C(sim(n_1), sim(n_2), K, v, s)$

Now, we can show that the simulation relation holds inductively as the two instances L and C execute as-per the lockstep. This ensures that for every violating execution of the larger instance L , there exists a violating execution of C . By independently showing that C does not exhibit any violations (which is a much simpler problem, since it has only 2 nodes), we can infer the correctness of the protocol.

III. SETUP

We consider distributed protocols written in the Relational Modeling Language (RML) [11]. RML is a Turing-complete language, and has been used in many prior works related to distributed protocol verification. RML uses the notions of *relations* and *functions* as used in many-sorted first order logic to describe the state of a distributed protocol. Further, these can be defined over arbitrary domains, as specified by the protocol developer. Constraints on the initial state of the protocol, as well as the safety property can then be directly encoded as FOL formulae over the declared relations and functions.

The protocol description in RML $P = \langle D, R, F, \Psi, A, \Phi \rangle$ consists of a set of declarations (D, R, F) , axioms (Ψ) , actions (A) and a safety property (Φ) . The declarations define the vocabulary: D , R and F denote the set of domain names,

relation names and function names respectively (along with the relation and function signatures). The axioms (Ψ) are FOL formulae defined over the vocabulary which encode properties of the domains. Φ denotes the safety property, which is another FOL formula, while \mathbb{A} denotes the actions of the protocol.

Given the protocol description, we construct a labeled transition system modeling the execution of the protocol. The transition system $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}} = (\Sigma, \Sigma_0, \delta)$ is parameterized by a domain interpretation function \mathcal{I} which associates a finite domain of values with each domain name $d \in \mathbb{D}$. For the interpretation function \mathcal{I} to be valid, we require the domains in range of \mathcal{I} to satisfy all the axioms in Ψ . Each state $\sigma \in \Sigma$ is an interpretation of function and relation names in \mathbb{F} and \mathbb{R} to actual functions and relations over the domains defined by the interpretation function \mathcal{I} . That is, for a function signature $f : (d_1 \times \dots \times d_n) \rightarrow d$ in the protocol description, $\sigma(f)$ will be a function of the form $\mathcal{I}(d_1) \times \dots \times \mathcal{I}(d_n) \rightarrow \mathcal{I}(d)$. The same holds for a relation r in the description.

The RML protocol description also consists of a set of axioms Ψ_0 constraining the functions and relations in the initial state of the system. We define $\Sigma_0 = \{\sigma \in \Sigma \mid \sigma \models \Psi_0\}$ to be the set of states obeying the initialization axioms. Note that the notation $\sigma \models \Psi$ denotes the standard FOL definition of an interpretation (σ) being the model of an FOL formula (Ψ).

Transitions of $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}}$ will correspond to actions of the protocol. An action $a(\bar{v} : \bar{d}) = \langle g(\bar{v}), u(\bar{v}) \rangle$ is parameterized over a set of (typed) variable names (\bar{v}), and consists of two components: (i) an FOL formula g (also called the guard) which can contain free variables from \bar{v} , (ii) an FOL formula u which models the change in the protocol state, defined over unprimed and primed versions of the functions and relations of the protocol. If the current state of the protocol obeys the guard, then the state is updated atomically using the update formula. The transitions $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}}$ caused by the action a in the protocol are formally defined as follows:

$$\delta_a = \{(\sigma, a(\bar{x}), \sigma') \mid \exists \bar{x} \in \mathcal{I}(\bar{d}). \sigma \models g[\bar{x}/\bar{v}] \wedge \sigma, \sigma' \models u[\bar{x}/\bar{v}]\}$$

That is, for every valuation \bar{x} of the variables \bar{v} , there are transitions from states σ which obey the guard g to states σ' such that σ, σ' satisfy the update formula. The transition is labeled by the action name along with the actual parameters, i.e. $a(\bar{x})$. The complete set of transitions is obtained by considering the transition set of every action of the protocol: $\delta = \bigcup_{a \in \mathbb{A}} \delta_a$. Let δ^* denote the reflexive and transitive closure of δ .

The safety property Φ is defined as a FOL formulae using the declared domains, functions and relations. In this work, we assume that Φ only uses universal quantifiers. Hence, Φ has the form $\forall(\bar{x} : \bar{d}). \phi$. This assumption is consistent with prior works related to distributed protocol verification, and is not restrictive as almost all safety properties can be naturally expressed using just universal quantification.

A trace of $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}}$ is a sequence of states and transition labels of the form $\sigma_0 a_1 \sigma_1 a_2 \sigma_2 \dots a_n \sigma_n$ such that $\sigma_0 \in \Sigma_0$ and $(\sigma_i, a_{i+1}, \sigma_{i+1}) \in \delta$ for all $i, 0 \leq i \leq n-1$. Let $\mathcal{T}(\mathcal{A}_{\mathcal{I}}^{\mathbb{P}})$ denote

the set of traces of $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}}$. We use $\llbracket \mathcal{A}_{\mathcal{I}}^{\mathbb{P}} \rrbracket$ to denote the set of reachable states of $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}}$, i.e. $\llbracket \mathcal{A}_{\mathcal{I}}^{\mathbb{P}} \rrbracket = \{\sigma' \mid \sigma_0 \dots \sigma' \in \mathcal{T}(\mathcal{A}_{\mathcal{I}}^{\mathbb{P}})\}$. A transition system is safe if all of reachable states obey the safety property of the protocol:

Definition 1. Given a distributed protocol $\mathbb{P} = \langle \mathbb{D}, \mathbb{R}, \mathbb{F}, \Psi, \Phi, \mathbb{A} \rangle$, a valid interpretation of domains \mathcal{I} obeying Ψ , the transition system $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}}$ is **safe** if for every reachable state $\sigma \in \llbracket \mathcal{A}_{\mathcal{I}}^{\mathbb{P}} \rrbracket$, $\sigma \models \Phi$.

While $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}}$ will be a finite state system (because every domain defined by \mathcal{I} is finite), there can in general be infinite number of domains which satisfy the axioms Ψ of the protocol. For a distributed protocol to be safe, the transition system corresponding to every valid domain interpretation should be safe:

Definition 2. A distributed protocol $\mathbb{P} = \langle \mathbb{D}, \mathbb{R}, \mathbb{F}, \Psi, \Phi, \mathbb{A} \rangle$ is safe if for every valid domain interpretation function \mathcal{I} satisfying the axioms Ψ , $\mathcal{A}_{\mathcal{I}}^{\mathbb{P}}$ is safe.

IV. CUTOFF BASED VERIFICATION

Each valid interpretation of the domains of a protocol can be seen as a protocol instance. A typical example of a domain with infinite number of valid interpretations is the domain of nodes participating in a protocol. To prove that a protocol is correct, we would need to show its correctness for all possible protocol instances. In cutoff based verification, the idea is to only show correctness for a specific protocol instance called a cutoff instance. In the following, we now formalize cutoff based verification in our framework.

Definition 3. Given a distributed protocol \mathbb{P} , a **cutoff instance** \mathcal{C} is a valid interpretation of domains such that if $\mathcal{A}_{\mathcal{C}}^{\mathbb{P}}$ is safe, then for any valid interpretation \mathcal{L} , $\mathcal{A}_{\mathcal{L}}^{\mathbb{P}}$ is safe.

Theorem 1. For a distributed protocol \mathbb{P} , if \mathcal{C} is a cutoff instance, and $\mathcal{A}_{\mathcal{C}}^{\mathbb{P}}$ is safe, then the distributed protocol \mathbb{P} is safe.¹

Notice that the definition of a cutoff instance implies that if there exists a protocol instance with a violation of the safety property, then the cutoff instance will also have a violation of the safety property. In essence, the cutoff instance can simulate the violation of the safety property in any protocol instance. We use this characterization to propose three conditions which together imply that a protocol instance is a cutoff instance.

These conditions require a simulation relation between states of any arbitrary protocol instance and states of the cutoff instance. Suppose \mathcal{C} is the cutoff instance, resulting in the cutoff transition system $\mathcal{A}_{\mathcal{C}}^{\mathbb{P}} = (\Sigma^{\mathcal{C}}, \Sigma_0^{\mathcal{C}}, \delta_{\mathcal{C}})$. Let \mathcal{L} be some arbitrary protocol instance, resulting in the system $\mathcal{A}_{\mathcal{L}}^{\mathbb{P}} = (\Sigma^{\mathcal{L}}, \Sigma_0^{\mathcal{L}}, \delta_{\mathcal{L}})$. To ensure that \mathcal{C} is a cutoff instance, any trace of $\mathcal{A}_{\mathcal{L}}^{\mathbb{P}}$ which leads to a state violating the safety property should be simulated by a trace of $\mathcal{A}_{\mathcal{C}}^{\mathbb{P}}$ also leading to a state violating the safety property. Consider a relation

¹The proofs for the theorems are provided in the full version of our paper at <https://github.com/shreesha00/FMCAD.git>

$\gamma_{\mathcal{L}} \subseteq \Sigma^{\mathcal{C}} \times \Sigma^{\mathcal{L}}$. We formalize below the conditions which will ensure that \mathcal{C} is a cutoff instance.

$$\begin{aligned}\varphi_{init}(\gamma_{\mathcal{L}}) &\triangleq \forall \sigma_{\mathcal{L}} \in \Sigma_0^{\mathcal{L}}. \exists \sigma_{\mathcal{C}} \in \Sigma_0^{\mathcal{C}}. (\sigma_{\mathcal{L}}, \sigma_{\mathcal{C}}) \in \gamma_{\mathcal{L}} \\ \varphi_{step}(\gamma_{\mathcal{L}}) &\triangleq \forall \sigma_{\mathcal{L}}, \sigma'_{\mathcal{L}} \in \Sigma_{\mathcal{L}}. \forall \sigma_{\mathcal{C}} \in \Sigma_{\mathcal{C}}. \gamma_{\mathcal{L}}(\sigma_{\mathcal{L}}, \sigma_{\mathcal{C}}) \wedge (\sigma_{\mathcal{L}}, a, \sigma'_{\mathcal{L}}) \in \delta_{\mathcal{L}} \\ &\Rightarrow \exists \sigma'_{\mathcal{C}} \in \Sigma_{\mathcal{C}}. (\sigma_{\mathcal{C}}, \sigma'_{\mathcal{C}}) \in \delta_{\mathcal{C}}^* \wedge \gamma_{\mathcal{L}}(\sigma'_{\mathcal{L}}, \sigma'_{\mathcal{C}}) \\ \varphi_{safety}(\gamma_{\mathcal{L}}) &\triangleq \forall \sigma_{\mathcal{L}} \in \Sigma_{\mathcal{L}}. \forall \sigma_{\mathcal{C}} \in \Sigma_{\mathcal{C}}. \gamma_{\mathcal{L}}(\sigma_{\mathcal{L}}, \sigma_{\mathcal{C}}) \wedge \sigma_{\mathcal{L}} \models \neg \Phi \\ &\Rightarrow \sigma_{\mathcal{C}} \models \neg \Phi\end{aligned}$$

The init condition φ_{init} ensures that every initial state of $\mathcal{A}_{\mathcal{L}}^{\mathcal{P}}$ is related by $\gamma_{\mathcal{L}}$ to some initial state of $\mathcal{A}_{\mathcal{C}}^{\mathcal{P}}$. The step condition φ_{step} ensures that if states of the protocol instance \mathcal{L} and cutoff instance \mathcal{C} are related by $\gamma_{\mathcal{L}}$, then after a transition in $\mathcal{A}_{\mathcal{L}}^{\mathcal{P}}$, the new state of instance \mathcal{L} will continue to be related to a state of \mathcal{C} obtained after 0 or more transitions in $\mathcal{A}_{\mathcal{C}}^{\mathcal{P}}$. Finally, the safety condition φ_{safety} ensures that if a state in $\mathcal{A}_{\mathcal{L}}^{\mathcal{P}}$ violates the safety property (Φ), then its simulating state in $\mathcal{A}_{\mathcal{C}}^{\mathcal{P}}$ also violates the safety property. Together, these conditions ensure that any violating trace of any arbitrary protocol instance can be simulated by a violating trace of the cutoff instance.

Theorem 2. Given a distributed protocol \mathcal{P} and a valid interpretation \mathcal{C} , if for any valid interpretation \mathcal{L} , there exists a simulation relation $\gamma_{\mathcal{L}}$ such that $(\varphi_{init} \wedge \varphi_{step} \wedge \varphi_{safety})(\gamma_{\mathcal{L}})$, then \mathcal{C} is a cutoff instance of \mathcal{P} .

While the above conditions ensure that if the cutoff instance is safe, then any arbitrary protocol instance is also safe, we can further refine them based on the following observation: we only need to simulate till the first violation of the safety property, and hence, we can assume that the safety property holds in all states while simulating till the first violation. The refined step condition φ_{step}^{first} is defined as follows:

$$\begin{aligned}\varphi_{step}^{first}(\gamma_{\mathcal{L}}) &\triangleq \forall \sigma_{\mathcal{L}}, \sigma'_{\mathcal{L}} \in \Sigma_{\mathcal{L}}. \forall \sigma_{\mathcal{C}} \in \Sigma_{\mathcal{C}}. \gamma_{\mathcal{L}}(\sigma_{\mathcal{L}}, \sigma_{\mathcal{C}}) \wedge (\sigma_{\mathcal{L}}, a, \sigma'_{\mathcal{L}}) \in \delta_{\mathcal{L}} \wedge \\ &\Phi(\sigma_{\mathcal{L}}) \Rightarrow \exists \sigma'_{\mathcal{C}} \in \Sigma_{\mathcal{C}}. (\sigma_{\mathcal{C}}, \sigma'_{\mathcal{C}}) \in \delta_{\mathcal{C}}^* \wedge \gamma_{\mathcal{L}}(\sigma'_{\mathcal{L}}, \sigma'_{\mathcal{C}})\end{aligned}$$

Lemma 3. Given a distributed protocol \mathcal{P} and a valid interpretation \mathcal{C} , if for any arbitrary valid interpretation \mathcal{L} , there exists a simulation relation $\gamma_{\mathcal{L}}$ such that $(\varphi_{init} \wedge \varphi_{step}^{first} \wedge \varphi_{safety})(\gamma_{\mathcal{L}})$, then \mathcal{C} is a cutoff instance of \mathcal{P} .

If the protocol is not safe, then we can consider the first violation of the safety property in any arbitrary instance of the protocol. Since the cutoff instance can simulate this first violation, this would imply that the cutoff instance would also not be safe, thus proving the above lemma. We have found in our experiments that the refined conditions are often more effective in proving *cutoff-ness* of a protocol instance.

V. SYNTHESIZING THE CUTOFF INSTANCE

In this section, we describe our technique to synthesize the cutoff instance and the simulation relation from the protocol description.

Metadata component	Contents
$a \in P.actions$	$a.named_arguments : list(string)$ $a.guard_atoms : set((x, l, o))$ $a.body : set((x, l, o))$ where $x \in functions \cup relations$ $l \in list(named_arguments \cup \{*\})$ $o \in x.out \cup \{*\}$
$s \in P.sorts$	$string$ that corresponds to a type defined by the protocol
$r \in P.relations$	$r.args : list(sorts)$ $r.out = \mathbb{B}$
$f \in P.functions$	$f.args : list(sorts)$ $f.out \in sorts$

TABLE I: The protocol metadata structure P

A. Pre-processing & Notation

The protocol description in RML is statically pre-processed to obtain a metadata structure P which has actions, relations, sorts and functions denoted by $P.actions$, $P.sorts$, $P.relations$, $P.functions$. Refer to Table I for a formal description of the protocol metadata structure P and its components. Each action has a set of named arguments, guard atoms and a body. The guard atoms and the function body contain sets of triplets where each triplet contains: the function or relation under consideration, the named arguments of the action (or $*$) which are its arguments, an output value (or $*$). The output value indicates constraints that are expected on the relation/function in case of guard atoms; and the updated value of the relation/function entry in case of the body. In all cases, a $*$ represents that the corresponding entry cannot be determined statically and can therefore be unconstrained. As an example, the guard clause $table(n_old, k, v)$ for the Reshard action would be converted to the triple $(table, [n_old, k, v], true)$ and the update $seqnum_sent(s) \leftarrow true$ is converted to the triple $(seqnum_sent, [s], true)$.

An *instantiation* of an action a is a map from the named arguments of the action to values. A value of $*$ represents that the corresponding named argument can take any value. An *action invocation* is defined as a tuple (a, I) where $a \in P.actions$ and I is an instantiation of a . We define a *clause* as a triple (x, L, o) where $x \in P.relations \cup P.functions$, L is a list of values (some of which can be $*$) conforming to the types in $x.args$ and o is either a constant of type $x.out$ or $*$.

Referring back to our motivating example, an instantiation of the named arguments of the Reshard action would be

$$I = [n_old : *, n_new : a_L, k : K, v : *, s : *]$$

and correspondingly, an action invocation would be the tuple $(Reshard, I)$. Similarly, a clause on the *table* relation would be $(table, [a_L, K, *], true)$.

B. Static Analysis

Algorithm 4 contains our static analysis algorithm, which takes as input the protocol metadata structure P and an initial set of clauses S_{init} . S_{init} will be derived from the safety property of the protocol; more details are provided in §5.3. S_{init} contains the initial set of clauses relevant for preserving any violation of the safety property. We maintain two sets

S and A where S contains a set of clauses and A a set of action invocations. In each iteration, we consider all the new clauses added to the set S in the previous iteration (line 8). For each clause c , in line 9, we invoke $\text{ACTIONSTHATSET}(P, c)$ to obtain all the action invocations that potentially set the clause c . We then add the guards for all these action invocations to the set S in line 11. The while loop at line 5 terminates when no new clauses have been added in the previous iteration, thus indicating that we have reached a fixed point.

The function $\text{ACTIONSTHATSET}(P, c)$ takes as input the program P and a clause c to return a set of action invocations A which potentially set the clause c . The algorithm works by pattern matching. We iterate over actions and for each atomic update in the body of the action, we check if the atomic update tuple matches the tuple in the clause with respect to the function/relation it updates in line 5. The if condition in line 6 fails only if both the atomic update output and the clause output can be determined statically and they do not match each other.

As an example, assume that the if condition in line 5 passes i.e. both the atomic update and the clause refer to the same function/relation x i.e. $c.x = \text{at_update}.x = x$. If the clause output $c.o = *$ and $\text{at_update}.o = \text{true}$ then this means that we are interested in actions that potentially affect $x(c.L)$ in any way, and this atomic update therefore satisfies that requirement. Similarly, if $c.o = \text{true}$ and $\text{at_update}.o = *$, this means that we are interested in actions that set $x(c.L) = \text{true}$, but the value that the atomic update alters $x(\text{at_update}.l)$ cannot be determined statically. Therefore, conservatively, we assume that the atomic update could potentially alter it as required. But, if $c.o = \text{true}$ and $\text{at_update}.o = \text{false}$, then the if condition fails as the outputs can be determined statically but do not match.

In line 7 we create an instantiation of $a.\text{named_arguments}$ initialized to $*$. The PATTERNMATCH function considers the arguments of the update atom and the clause atom $\text{at_update}.l, c.L$ and checks for inconsistencies. For example, $\text{at_update}.l = (a, b, a)$ and $c.L = (1, 2, *)$ would pass the check whereas $\text{at_update}.l = (a, b, a)$ and $c.L = (1, 2, 3)$ would fail the check. If the pattern match succeeds, the for loop instantiates the named arguments in $\text{at_update}.l$ based on $c.L$. The tuple (a, I) now forms the action invocation which is added to the set of action invocations returned by the algorithm. The GUARDSFOR function returns the set of clauses involved in the guard for an action invocation. We iterate through all the guard atoms of the action in line 3. The for loop in lines 5-6 assigns concrete values to the named arguments in $g.l$ using the instantiation I provided in the action invocation. Then a clause tuple is created in line 7 and added to the list of clauses returned by the algorithm.

As an example of how these methods work, we refer back to sharded key value store example considered in §2. If $\text{ACTIONSTHATSET}(P, (\text{unacked}, [*, a_L, K, *, *], \text{true}))$ is invoked, then one of the actions returned by it would be the Reshard action, with the action invocation $(\text{Reshard}, [n_old : *, n_new : a_L, k : K, v : *, s : *])$ (this is because Reshard

sets *unacked* to *true* in Line-14, Algorithm 1). Similarly, if $\text{GUARDSFOR}(\text{Reshard}, [n_old : *, n_new : a_L, k : K, v : *, s : *])$ is invoked, the following set G is returned.

$$G = \{(\text{seqnum_sent}, [*, \text{false}], (\text{table}, [*, K, *], \text{true}))\}$$

C. Synthesizing the Cutoff Instance, Simulation Relation & Lockstep

Cutoff Instance. We start with the safety property Φ in the RML description. As described in §3, the safety property only contains universal quantifiers and hence is a formula of the form $\forall(\bar{x} : \bar{d}). \phi$. The size of the cutoff system is taken to be the number of universally quantified nodes in the safety property.

Obtaining S_{init} . Consider any arbitrary size instance L with \mathcal{D}_L denoting the set of nodes. To begin with the static analysis, we need to provide an initial set of clauses S_{init} as input along with the pre-processed protocol metadata structure P . To obtain S_{init} , we first negate the safety property and instantiate all the existentially quantified variables. We define $\mathcal{D}_L^v \subseteq \mathcal{D}_L$ the set of instantiated nodes or *violating nodes*. We then process the resulting FOL formula $\neg\phi$ to obtain the set of clauses involved in the formula.

As an example, consider the safety property for the Sharded Key Value store protocol from §2. We have

$$\begin{aligned} \forall N_1, N_2, K, V_1, V_2. \text{table}(N_1, K, V_1) \wedge \text{table}(N_2, K, V_2) \\ \implies N_1 = N_2 \wedge V_1 = V_2 \end{aligned}$$

As there are 2 quantifiers on nodes, the cutoff for the protocol is 2. Negating and instantiating $N_1 = a_L, N_2 = b_L, K = k, V_1 = v_1$ and $V_2 = v_2$, we get

$$\text{table}(a_L, k, v_1) \wedge \text{table}(b_L, k, v_2) \wedge (n_1 \neq n_2 \vee v_1 \neq v_2)$$

giving us the following set of clauses after processing

$$\{(\text{table}, [a_L, k, v_1], \text{true}), (\text{table}, [b_L, k, v_2], \text{true})\}$$

Synthesizing the Simulation Relation and Lockstep.

Having obtained S_{init} , we can now invoke $\text{STATICANALYSIS}(P, S_{init})$ to get the set of clauses S and set of action invocations A . We also have the cutoff instance C with its set of nodes \mathcal{D}_C . To define the lockstep and simulation relation, we map the nodes of the violating instance to nodes of the cutoff system. Such a mapping $\text{sim} : \mathcal{D}_L \rightarrow \mathcal{D}_C$ is defined as follows. Firstly, by construction, $|\mathcal{D}_L^v| = |\mathcal{D}_C|$ i.e., the number of nodes involved in the violation is the same as the number of nodes in the cutoff system. Consequently, we perform a one-to-one mapping of nodes from \mathcal{D}_L^v to \mathcal{D}_C . For the rest of the nodes $\mathcal{D}_L \setminus \mathcal{D}_L^v$ in the system L , we make the following observations:

- If S and A obtained from the static analysis do not have any components containing $*$ in any field of the node type, this implies that only actions and state components of the violating nodes are sufficient to simulate the violation. In such a case, there is no need to map nodes

Algorithm 2 ACTIONSTHATSET

Arguments: P the program, and a clause c **Returns:** A a set of action invocations

```

1: procedure ACTIONSTHATSET( $P, c$ )
2:    $A = \emptyset$ 
3:   for  $a \in P.actions$  do
4:     for  $at\_update = (x, l, o)$  in  $a.body$  do
5:       if  $at\_update.x == c.x$  then
6:         if  $\neg (c.o \neq * \text{ and } at\_update.o \neq * \text{ and } c.o \neq at\_update.o)$  then
7:           Create an instantiation  $I$  of  $a.named\_arguments$ , initialized to  $*$ ;
8:           if PATTERNMATCH( $at\_update.l, c.L$ ) then
9:             for  $i \in 1, len(at\_update.l)$  if  $at\_update.l[i] \neq *$  do
10:               $I[at\_update.l[i]] \leftarrow c.L[i]$ 
11:               $r \leftarrow (a, I)$ 
12:               $A \leftarrow A \cup \{r\}$ 
13:   return  $A$ 

```

Algorithm 3 GUARDSFOR

Arguments: P the program, an action invocation act **Returns:** G a set of clauses

```

1: procedure GUARDSFOR( $P, act$ )
2:    $G = \emptyset$ 
3:   for  $g = (x, l, o) \in a.guards$  do
4:     Create a list  $L$  of length  $g.l$ , initialized to  $*$ 
5:     for  $i \in 1, len(g.l)$  if  $g.l[i] \neq *$  do
6:        $L[i] \leftarrow act.I[g.l[i]]$ 
7:      $G \leftarrow G \cup \{(g.x, L, g.o)\}$ 
8:   return  $G$ 

```

Algorithm 4 STATICANALYSIS

Arguments: P the program, S_{init} a set of clauses**Returns:** S a set of clauses, A a set of action invocations

```

1: procedure STATICANALYSIS( $P, S_{init}$ )
2:    $S \leftarrow S_{init}$ 
3:    $S_{prev} \leftarrow \emptyset$ 
4:    $A \leftarrow \emptyset$ 
5:   while  $S \neq S_{prev}$  do
6:      $S_d \leftarrow S \setminus S_{prev}$ 
7:      $S_{prev} \leftarrow S$ 
8:     for each clause  $c$  in  $S_d$  do  $\triangleright$  For each new clause
9:        $A_t \leftarrow ACTIONSTHATSET(P, c)$ 
10:      for each action invocation  $act$  in  $A_t$  do
11:         $S \leftarrow S \cup GUARDSFOR(P, act)$ 
12:       $A \leftarrow A \cup A_t$ 
13:   return  $S, A$ 

```

from $\mathcal{D}_L \setminus \mathcal{D}_L^v$ as they will never appear in the simulation relation or lockstep.

- If S or A obtained from the static analysis has components containing $*$ in any field of the node type, we map all the nodes from $\mathcal{D}_L \setminus \mathcal{D}_L^v$ to one of the nodes in \mathcal{D}_C .

Intuitively, the simulation relation states that for all the clauses that are relevant to the violation (as obtained by the static

analysis procedure) in the larger system L , the same state components are maintained in the cutoff system but in the state component of the simulating nodes (as per the sim mapping). Similarly, the lockstep states that the relevant actions are performed in the cutoff system, but by the simulating nodes.

Given S, A and sim , we obtain the simulation relation and lockstep using the procedure SIMANDLOCKSTEP(S, A, sim) in Algorithm 5. The procedure returns the simulation relation γ as a FOL formula and the lockstep τ as an abstract map from action invocations of the larger system to action invocations of the cutoff system. The main idea is to simply perform the relevant actions of A in the cutoff system, whenever they are performed in the larger system, synthesizing the appropriate mapping of the action arguments, and thus maintaining a simulation relation for the relevant state components in S .

Cutoff Verification. To prove that the synthesized cutoff instance is actually a cutoff for the protocol, we generate FOL formulae for each of the 3 properties $\varphi_{init}(\gamma_L), \varphi_{step}(\gamma_L)$ and $\varphi_{safety}(\gamma_L)$ mentioned in §3, using the simulation relation γ synthesized by Algorithm 5. Furthermore, for $\varphi_{step}(\gamma_L)$, we remove the existential quantifier over the state σ_C after the transition by providing a candidate transition in the system C as per the lockstep τ .

D. Synthesis for Consensus Protocols

We now describe how the above technique can be adapted to work for quorum-based consensus protocols. Such protocols are used to achieve consensus amongst the nodes on some decision such as proposing a value or choosing a leader, with the safety property being the uniqueness of the decision taken i.e. no two nodes learn of two different decisions.

Quorum-based consensus protocols define a notion of a *quorum* which refers to a set of nodes and a *quorum-set* which is a set of such quorums. Additionally, the quorum-set satisfies the *quorum-intersection* property i.e. any two quorums belonging to a quorum-set intersect. These protocols also involve a voting phase where nodes cast their unique votes for values, and values

Algorithm 5 Function to obtain simulation relation and lockstep

Arguments: Set of clauses S , action invocations A and mapping $sim : \mathcal{D}_L \rightarrow \mathcal{D}_C$

Returns: FOL formula γ representing the simulation relation and lockstep τ as a map from actions of the larger system to actions of the cutoff system

```
1: procedure SIMANDLOCKSTEP( $S, A, sim$ )
2:    $\gamma \leftarrow true$ 
3:   for each clause  $c = (x, L, o) \in S$  do
4:     For each  $*$  entry in  $L$ , replace it with a unique variable name from  $\bar{v}$ , and add those variables to  $L$  to get  $\mathcal{L}_{args}$ ;
5:     Replace each node variable  $n$  in  $\mathcal{L}_{args}$  with  $sim(n)$  to get  $\mathcal{C}_{args}$ ;
6:     if  $o == *$  then
7:        $\triangleright$  In this case, we assert that the function/relation entries are equal in the larger system and cutoff system
8:        $\gamma \leftarrow \gamma \wedge (\forall \bar{v}. x(\mathcal{L}_{args}) = x(\mathcal{C}_{args}))$ ;
9:     else
10:       $\triangleright$  In this case, we assert that if the relation/function entry takes the value  $o$  in  $L$ , it also does so in  $C$ 
11:      if  $x.out$  is of node type then
12:         $\gamma \leftarrow \gamma \wedge (\forall \bar{v}. (x(\mathcal{L}_{args}) = o) \implies (x(\mathcal{C}_{args}) = sim(o)))$ ;
13:      else
14:         $\gamma \leftarrow \gamma \wedge (\forall \bar{v}. (x(\mathcal{L}_{args}) = o) \implies (x(\mathcal{C}_{args}) = o))$ ;
15:   Initialize an empty map  $\tau$ 
16:   for each action invocation  $act \in A$  do
17:     For each  $*$  value in  $act.I$ , replace it with a unique variable name from  $\bar{v}$ , to get  $act_L.I$ ;
18:     Replace each node value  $n$  in  $act_L.I$  with  $sim(n)$  to get  $act_C.I$ ;
19:     Define  $act_L = (act.a, act_L.I)$  and  $act_C = (act.a, act_C.I)$ ;
20:      $\forall \bar{v}. \tau(act_L) \leftarrow act_C$ ;
21:   return  $\gamma, \tau$ 
```

which receive a quorum of votes are considered as decided. The core safety argument for such protocols typically relies on the quorum-intersection property and the uniqueness of votes i.e. if two values were decided, they both must have received a quorum of votes but since any two quorum-sets intersect, there must be a node that has voted twice which is disallowed by the protocol. Most protocols for achieving consensus such as Raft [12], Paxos [13] and Two-phase commit are designed around these core principles. However, obtaining an inductive invariant for formal verification of these protocols is still a challenging task.

For such protocols, we assume a sort *quorum* for quorums and a fixed relation *member* : *node*, *quorum* which governs the membership of nodes to quorums. In our pre-processing, for guard atoms in quorum-based consensus protocols, we also track state components on which a quorum-agreement is required.

At a high-level, similar to the non-consensus case, we collect the actions and the state components responsible for a violation through a similar static analysis procedure. However, simulating the violation in the cutoff system by maintaining these states now requires a more complicated lockstep. In particular, the cutoff system tries to maintain the quorum agreement on state components required to reach the violation through staggered actions i.e. the cutoff system waits for a quorum agreement on some necessary state component and then performs the set of actions required to reach quorum agreement in the cutoff system all at once thereby ensuring that a quorum agreement on a state component in the larger

system is maintained in the cutoff system.²

VI. EXPERIMENTAL RESULTS

We have applied the proposed strategy on a variety of different distributed protocols^{3,4} given in Table II. Our technique works in two parts, where we first attempt to automatically synthesize the cutoff instance, and then attempt to prove its correctness. For proving correctness of a cutoff instance, we generate a FOL encoding of the 3 conditions $\varphi_{init}(\gamma_{\mathcal{L}})$, $\varphi_{step}(\gamma_{\mathcal{L}}, \tau_{\mathcal{L}})$ and $\varphi_{safety}(\gamma_{\mathcal{L}})$. We reduce the problem of checking correctness to satisfiability of the generated FOL formulae. For example, for checking the $\varphi_{step}(\gamma_{\mathcal{L}}, \tau_{\mathcal{L}})$ condition which is a condition of the type $p \implies q$ to be correct, we check whether $p \wedge \neg q$ is unsatisfiable. We use Z3 [14] as our backend SMT solver. The experiments were run on a system with a 12-core Apple M2 Pro processor and 16GB RAM. Table II summarizes our experimental results. Notice that the time taken for each protocol is in the order of few milliseconds except for the Consensus protocol which takes significantly longer due to the larger number of quantifiers used in the encoding.

VII. RELATED WORK AND CONCLUSION

In the recent past, there has been a lot of interest in automated and mechanised verification of distributed protocols

²We provide an example of our technique for consensus protocols on the Toy Consensus protocol in the full version of our paper at <https://github.com/shreesha00/FMCAD.git>

³The RML descriptions and the SMT encoding of the simulation relation and cutoff protocol for each protocol can be found at the following link: <https://github.com/shreesha00/FMCAD.git>

⁴We provide detailed descriptions of each protocol and its cutoff instance in the full version of our paper at <https://github.com/shreesha00/FMCAD.git>

Protocol	Cutoff	Time Taken(s)	$ \gamma $
Sharded Key-Value Store[15]	2	0.02	5
Leader Election in a Ring[16]	2	0.03	4
Centralized Lock Server[17]	2	0.02	5
Lock Server Sync[18]	2	0.01	2
Ricart Agrawala[19]	2	0.01	6
Two Phase Commit[20]	2	0.02	9
Toy Consensus ForAll[18]	1	0.07	5
Consensus[18]	2	29.7	11

TABLE II: γ is a FOL formula of the type $\bigwedge_{i=1}^{|\gamma|} (p \implies q)$ therefore $|\gamma|$ represents the number of clauses of the type $p \implies q$ in the simulation relation. Time taken refers to the total time taken by our synthesis+verification procedure.

([1]–[6]). Ironfleet [15] and Verdi [17] are some of the earliest works which are more focused towards verifying real-world implementations of distributed protocols, and typically assume that an abstract model of the protocol works correctly. Many of the recent approaches towards protocol verification rely on constructing and proving some form of inductive invariant. Padon et. al. [11] introduced the Ivy framework along with the RML language which allows a protocol developer to interactively generate an inductive invariant for verifying safety. Other approaches ([1], [2], [5]) have continued along this line of work, by attempting to automate the process of deriving the inductive invariant using techniques like IC3/PDR or data-driven approaches. While these approaches have been successful to some extent, we note that the problem of deriving inductive invariants is a fundamentally hard problem, and our work allows us to sidestep it. In fact, it could be useful to apply these techniques to the comparatively simpler problem of finding and proving a cutoff instance.

While previous works have also attempted to use cutoff-based approaches for verification ([7]–[10]), they have mostly been limited to either a restricted class of protocols or a restricted class of specifications. We note that none of these works actually mechanize and automate the proof that a protocol instance is actually a cutoff instance. To our best knowledge, ours is the first work that enables automated cutoff based verification.

In this work, we investigated the applicability of cutoff based verification for a variety of distributed protocols. We observe that cutoff based verification allows us to naturally sidestep the harder problem of finding inductive invariants. We identify sufficient conditions which can be used to verify that a protocol instance is indeed a cutoff instance and which can be encoded using SMT. We develop a simple static analysis-based approach to automatically synthesize the cutoff instance for many protocols.

We note that our approach has limitations. In particular, it can fail in one of two ways. Firstly, the cutoff value itself could be higher than the one chosen by our analysis. Secondly, it is possible that the simulation relation and the lockstep synthesized by our analysis may not work (i.e. they may not satisfy the φ_{step} or φ_{safety} constraints). In either case, our analysis will not succeed in verifying the protocol. Intuitively,

this could happen because the nodes in our synthesized cutoff instance cannot simulate a violation of the safety property, in which case, either of the φ constraints will not hold. One can construct an artificial example to demonstrate this; however, we note that we have not encountered this issue in our experiments. It is a well-established empirical result that most bugs in real-world protocol implementations and designs can be discovered within a small scope of parameter values. Our work takes a step towards generalizing and formalizing this result by providing a generic simulation-based strategy to synthesize cutoff instances and cutoff proofs.

To conclude, our cutoff-based verification approach demonstrates how a combination of static analysis, SMT-based verification, and model checking can simplify the hard problem of protocol verification. Our experimental results indicate that cutoff results are ubiquitous and applicable for different types of protocols. Our vision is that this work can pave the way for more investigations into automating cutoff results for more complex protocols.

REFERENCES

- [1] Y. M. Y. Feldman, J. R. Wilcox, S. Shoham, and M. Sagiv, “Inferring inductive invariants from phase structures,” in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 11562. Springer, 2019, pp. 405–425.
- [2] H. Ma, A. Goel, J. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, “I4: incremental inference of inductive invariants for verification of distributed protocols,” in *SOSP*. ACM, 2019, pp. 370–384.
- [3] K. L. McMillan and O. Padon, “Ivy: A multi-modal verification tool for distributed algorithms,” in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 12225. Springer, 2020, pp. 190–202.
- [4] O. Padon, G. Losa, M. Sagiv, and S. Shoham, “Paxos made EPR: decidable reasoning about distributed protocols,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 108:1–108:31, 2017.
- [5] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, “Distai: Data-driven automated invariant learning for distributed protocols,” in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 405–421.
- [6] A. Damian, C. Dragoi, A. Militaru, and J. Widder, “Communication-closed asynchronous protocols,” in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11562. Springer, 2019, pp. 344–363. [Online]. Available: https://doi.org/10.1007/978-3-030-25543-5_20
- [7] E. A. Emerson and K. S. Namjoshi, “Reasoning about rings,” in *POPL*. ACM Press, 1995, pp. 85–94.
- [8] N. Jaber, S. Jacobs, C. Wagner, M. Kulkarni, and R. Samanta, “Parameterized verification of systems with global synchronization and guards,” in *CAV (1)*, ser. Lecture Notes in Computer Science, vol. 12224. Springer, 2020, pp. 299–323.
- [9] O. Maric, C. Sprenger, and D. A. Basin, “Cutoff bounds for consensus algorithms,” in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 10427. Springer, 2017, pp. 217–237.
- [10] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder, *Decidability of Parameterized Verification*, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
- [11] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: safety verification by interactive generalization,” in *PLDI*. ACM, 2016, pp. 614–630.
- [12] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319.
- [13] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, p. 133–169, may 1998.

- [14] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.
- [15] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill, “Ironfleet: proving practical distributed systems correct,” in *SOSP*. ACM, 2015, pp. 1–17.
- [16] E. Chang and R. Roberts, “An improved algorithm for decentralized extrema-finding in circular configurations of processes,” *Commun. ACM*, vol. 22, no. 5, p. 281–283, may 1979.
- [17] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, “Verdi: a framework for implementing and formally verifying distributed systems,” in *PLDI*. ACM, 2015, pp. 357–368.
- [18] J. Yao, R. Tao, R. Gu, and J. Nieh, “DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA, Jul. 2022, pp. 485–501.
- [19] G. Ricart and A. K. Agrawala, “An optimal algorithm for mutual exclusion in computer networks,” *Commun. ACM*, vol. 24, no. 1, p. 9–17, jan 1981.
- [20] J. N. Gray, *Notes on data base operating systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 393–481.
- [21] K. S. Namjoshi, “Symmetry and completeness in the analysis of parameterized systems,” in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 4349. Springer, 2007, pp. 299–313.
- [22] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos, “Modularity for decidability of deductive verification with applications to distributed systems,” in *PLDI*. ACM, 2018, pp. 662–677.
- [23] S. Chand, Y. A. Liu, and S. D. Stoller, “Formal verification of multipaxos for distributed consensus,” in *International Symposium on Formal Methods*. Springer, 2016, pp. 119–136.
- [24] V. Rahli, D. Guaspari, M. Bickford, and R. L. Constable, “Formal specification, verification, and implementation of fault-tolerant systems using eventml,” *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 72, 2015. [Online]. Available: <https://api.semanticscholar.org/CorpusID:46662559>
- [25] S. Paul, G. A. Agha, S. Patterson, and C. A. Varela, “Verification of eventual consensus in synod using a failure-aware actor model,” in *NASA Formal Methods Symposium*. Springer, 2021, pp. 249–267.
- [26] P. Küfner, U. Nestmann, and C. Rickmann, “Formal verification of distributed algorithms,” in *Theoretical Computer Science*, J. C. M. Baeten, T. Ball, and F. S. de Boer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 209–224.
- [27] B. Charron-Bost and A. Schiper, “Schiper, a.: The heard-of model: computing in distributed systems with benign faults. distributed computing 22(1), 49-71,” *Distributed Computing*, vol. 22, 04 2009.
- [28] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, “Verifying safety properties with the tla+ proof system,” in *Automated Reasoning*, J. Giesl and R. Hähnle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 142–148.

Optimal Bounded Partial Order Reduction

Iason Marmanis 

Max Planck Institute for Software Systems
Kaiserslautern, Germany
imarmanis@mpi-sws.org

Viktor Vafeiadis 

Max Planck Institute for Software Systems
Kaiserslautern, Germany
viktor@mpi-sws.org

Abstract—Preemption bounding (PB) and dynamic partial order reduction (DPOR) are two key techniques for scaling up the model checking of concurrent software. Attempts to combine them have so far been suboptimal: they either explore redundant executions (that DPOR alone would eliminate) or executions exceeding the desired bound (that PB alone would not consider).

By bounding the number of rounds of a round-robin scheduler instead of the number of preemptions, we obtain the first optimal bounded partial order reduction algorithm. Our approach has two additional benefits: (1) it makes checking boundedness of a Mazurkiewicz trace linear-time (instead of NP-hard) and (2) it extends smoothly to weak memory models.

I. INTRODUCTION

Even under *sequential consistency* (SC) [18], to make automated verification of concurrent programs feasible, one typically has to restrict their state space in several unsound ways, such as considering only executions with up to K recursive calls, L loop iterations, N concurrent threads, and even M preemptive context switches between them. In this paper, we will focus on the latter restriction, which is known as *preemption bounding* (PB) or *context bounding* [23].

Bounding such quantities is generally sufficient for finding safety errors in programs and can provide reasonable confidence in the correctness of programs whose full verification is intractable. PB is especially good in that regard: it achieves great state-space reduction (since the number of executions of a concurrent program is exponential in the number of preemptions) and bug coverage (because bugs in practice can be exposed with a very small number of preemptions [22]).

Bounding, however, often destroys symmetries in a program, which lessens the effect of sound state-space reduction techniques. In particular, PB does not work well with *dynamic partial order reduction* (DPOR) [9], which calls two executions of a concurrent program equivalent if they differ only in the order of commuting operations (e.g., two accesses to different shared memory locations) and strives to explore only one execution per equivalence class.

Combining DPOR and bounding optimally is non-trivial. Coons et al. [8] weaken the benefit of DPOR leading to the (redundant) exploration of multiple equivalent interleavings; whereas Marmanis et al. [19] weaken the benefit of PB and often require the exploration of executions with more preemptions than the desired bound. Moreover, both approaches suffer from the NP-hardness of checking whether a given execution is equivalent to some execution with at most M preemptions.

In this paper, we provide an optimal combination of these two techniques by changing the bounded quantity. Rather than assuming a completely non-deterministic scheduler and bounding the number of preemptive context switches, we assume the presence of a round-robin scheduler under a fixed ordering of the threads (e.g., in increasing thread-identifier order) and bound the number of rounds such a scheduler can take. This change has two immediate consequences:

- 1) Checking whether an execution is below the desired bound can be decided in linear time (see §II).
- 2) The optimal DPOR exploration procedure of Kokologiannakis et al. [12] is monotone in the number of scheduling rounds (see §III).

Therefore, by stopping the exploration of any execution prefix that exceeds the desired bound, we immediately obtain a sound, complete, and optimal bounded partial order reduction algorithm called **ROUNDER** (see §IV), which enables the bounded verification of programs whose unbounded verification is intractable (see §VI). Our optimal bounding approach extends seamlessly to weak memory models for bounding metrics that are similar to the number of scheduling rounds or that constrain only the non-SC part of executions (see §V).

II. BOUNDING THE NUMBER OF SCHEDULING ROUNDS

A. Program Traces and Execution Graphs

A program trace τ is a sequence of *events*, each corresponding to the execution of a single thread instruction, such as a *read* (R) or a *write* (W) of a certain location and value. In a sequentially consistent trace, every read event r in the trace reads the value written by the last write event in the same location that appears before r in the trace. Two traces are (*Mazurkiewicz*-)equivalent if they only differ in the order of commuting instructions (of different threads) [20].

Instead of using traces, several recent DPOR algorithms [11, 12, 14] directly explore their equivalence classes, succinctly represented as *execution graphs*. An execution (graph) G consists of a set of events $G.E$ including one initialization write event for each memory location and the following set of directed edges that reflect the ordering between the events:

- the *program order* $G.po$, which orders events of the same thread in their control-flow order and initialization write events before all non-initialization events,
- the *coherence order* $G.co$, which (totally) orders same-location write events, and

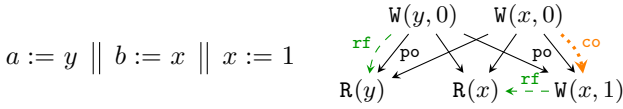


Fig. 1. A program (left) and one of its execution graphs (right).

- the *reads-from* $G.\text{rf}$, which orders every read event after the write event that it reads from.

We write $G|_S$ for the restriction of execution G to the set of events S ; and we say that G is a prefix of G' if $G = G'|_{G.E}$.

In Fig. 1, we show a program with three threads and one of its execution graphs. The graph contains two initialization writes (one for x and one for y along with three events corresponding to the memory accesses of the program.

Given two relations X and Y , we define their composition $X;Y \triangleq \{\langle x,z \rangle \mid \exists y. \langle x,y \rangle \in X \wedge \langle y,z \rangle \in Y\}$ and the inverse $X^{-1} \triangleq \{\langle x,y \rangle \mid \langle y,x \rangle \in X\}$ of X . Finally, we write X^+ for the transitive closure of X .

We now define the following four derived relations:

- The *causality order*, $G.\text{porf} \triangleq (G.\text{po} \cup G.\text{rf})^+$, captures dependencies between events due to the program order and the reads-from relation: an event b causally ordered after an event a cannot be executed before a because it depends on a 's execution.
- The *from-reads* relation, $G.\text{fr} \triangleq G.\text{rf}^{-1}$; $G.\text{co}$, orders every read event before the same-location writes that are co -after than the write that the read it is reading from.
- The *extended coherence order*, $G.\text{eco} \triangleq (G.\text{rf} \cup G.\text{co} \cup G.\text{fr})^+$, orders all same-location access events apart from two reads that read from the same write—their order is immaterial.
- The *SC order*, $G.\text{sc} \triangleq (G.\text{po} \cup G.\text{eco})^+$, puts together orderings due to the program and due to coherence.

An execution G is (*sequentially*) *consistent* if $G.\text{sc}$ is irreflexive. A consistent execution G represents the equivalence class of the set of all linearizations of $G.\text{sc}$.

Other memory models require only certain subsets of the $G.\text{sc}$ relation to be irreflexive. For example, coherence requires $G.\text{po}; G.\text{eco}$ to be irreflexive, while release-acquire consistency requires $G.\text{porf}; G.\text{eco}$ to be irreflexive.

B. Round-Robin Rounds

We define the (round-robin) *rounds* of a program trace $\tau = e_1, e_2, \dots, e_k$ to be the number of times a round-robin scheduler needs to start again from the first thread to generate the trace τ , i.e., $\text{rounds}(\tau) \triangleq |\{e_i \mid \text{tid}(e_i) > \text{tid}(e_{i+1})\}|$, where $\text{tid}(e)$ returns the thread identifier of event e .

The notion of rounds can be naturally lifted to execution graphs: the rounds of an execution graph is the least among the rounds of the traces it represents, i.e., $\text{rounds}(G) \triangleq \min \{\text{rounds}(\tau) \mid \tau \text{ linearizes } G.\text{sc}\}$.

The execution graph in Fig. 1 represents the following three traces: (a) $a := y; x := 1; b := x$, (b) $x := 1; a := y; b := x$, and (c) $x := 1; b := x; a := y$. It has one round, since traces (a) and (b) have one round, whereas trace (c) has two rounds.

Algorithm 1 Greedy algorithm for rounds(G)

```

1: procedure rounds( $G$ )
2:    $S \leftarrow G.E$ 
3:    $\text{rounds} \leftarrow 0$ 
4:   while  $S \neq \emptyset$  do
5:      $\text{rounds} \leftarrow \text{rounds} + 1$ 
6:     for  $i \leftarrow 1$  to  $N$  do
7:       while  $\left( \begin{array}{l} \exists e \in S. \text{tid}(e) = i \\ \wedge \nexists e' \in S. \langle e', e \rangle \in G.\text{sc} \end{array} \right)$  do
8:          $S \leftarrow S \setminus \{e\}$ 
9:   return  $\text{rounds} - 1$ 

```

Clearly, to compute the rounds of an execution G we do not have to enumerate the traces of G . Assuming G has N threads, Algorithm 1 computes $\text{rounds}(G)$ with a greedy approach: it follows the scheduling of the round-robin scheduler adding as many events from the current thread as possible. Any trace τ' has at least as many rounds as the trace τ that $\text{rounds}(\cdot)$ (implicitly) constructs. To see this, observe that at the first point where τ and τ' differ, τ' could be extended with the event e of thread t , but instead moved to the next thread and possibly incurred an additional round.

We note that $\text{rounds}(\cdot)$ is *monotone* w.r.t. the prefix relation.

Proposition 1. *Given two consistent executions G and G' , if G is a prefix of G' , then $\text{rounds}(G) \leq \text{rounds}(G')$.*

Proof. Consider a trace τ' of G' with $\text{rounds}(G')$ rounds. Restricting it to the events of G yields a trace τ of $G.E$ with at most $\text{rounds}(G')$ rounds. \square

C. Rounds versus Context Switches

We say that a program trace incurs a *context switch* whenever adjacent elements of the trace belong to different threads. A *preemptive* context switch between two events is one where the thread of the first event could have continued execution: it is neither blocked nor finished. As with scheduling rounds, we straightforwardly count the number of (preemptive) context switches in a trace and lift that definition to execution graphs.

Although the formal definitions of scheduling rounds and context switches are very similar, the number of scheduling rounds and the number of context switches of a particular trace can differ widely. The reason is that one scheduling round in a program with N threads can contain events from at least one and at most N threads. Consequently, a round-robin execution of a program with N threads and K rounds can have at most $K \times N$ context switches.

Conversely, an arbitrary execution with N threads and C context-switches can be generated by a round-robin scheduler with at most $C - \lfloor C/N \rfloor$ rounds. To see this, consider the worst-case scenario where as many of the context-switches as possible incur a new round: at least $\lfloor C/N \rfloor$ of them originate from the same thread and therefore at least $\lfloor C/N \rfloor$ increase the thread identifier of the current thread, which does not incur a new round.

III. OPTIMAL UNBOUNDED DPOR

In this section, we recall the TruSt algorithm [12] in Algorithm 2. With every execution graph G , TruSt keeps track of a total order $<_G$ on $G.E$, which corresponds to the order they were added to the graph.

Algorithm 2 TruSt's exploration algorithm

```

1: procedure VERIFY( $P$ )
2:   VISIT $_P(G_\emptyset)$ 

3: procedure VISIT $_P(G)$ 
4:   if  $\neg$ consistent( $G$ ) then return
5:   switch  $a \leftarrow \text{next}_P(G)$  do
6:     case  $a = \perp$ 
7:       return "Visited full execution graph  $G$ "
8:     case  $a \in \text{error}$ 
9:       return "Visited erroneous execution  $G$ "
10:    case  $a \in R$ 
11:      for  $w \in G.W_{\text{loc}(a)}$  do VISIT $_P(\text{SetRF}(G, a, w))$ 
12:    case  $a \in W$ 
13:      VISITCO $_P(G, a)$ 
14:      for  $r \in G.R_{\text{loc}(a)}$  s.t.  $\langle r, a \rangle \notin G.\text{porf}$  do
15:         $D \leftarrow \{e \in G.E \mid r <_G e \wedge \langle e, a \rangle \notin G.\text{porf}\}$ 
16:        if ISMAXIMAL( $G, \{r\} \cup D, a$ ) then
17:          VISITCO $_P(\text{SetRF}(G|_{G.E \setminus D}, r, a), a)$ 
18:    case  $-$ 
19:      VISIT $_P(G)$ 

20: procedure VISITCO $_P(G, a)$ 
21:   for  $w_p \in G.W_{\text{loc}(a)}$  do VISIT $_P(\text{SetCO}(G, w_p, a))$ 

```

TruSt's exploration of the execution of program P starts by invoking VISIT $_P$ on the empty execution graph. At each step, TruSt checks that the current execution graph is inconsistent and drops it if so (line 4). Otherwise, TruSt augments the current execution with the next event a picked by the scheduler (line 5), and proceeds differently depending on the type of a . The interesting cases are when a is a read or a write.

In the case of a read event, TruSt considers all the executions where a reads from some write event w in the same location as a . (SetRF(G, a, w) modifies G so that a reads from w .)

In the case of a write event a , TruSt first considers every possible oo -placement for a (placing it directly after each write w_p to the same location as a via SetCO(G, w_p, a), line 21). Second, it considers *revisiting* each previously added read r of the same location as a that does not causally precede a (line 14). The revisit operation removes from the execution all events added after r that do not causally precede a (line 17).

To perform the revisit, TruSt checks a *maximality* condition for the set of events that would be removed from the execution graph (line 16). Intuitively, ISMAXIMAL(G, S, a) checks if it is possible to reconstruct G from the restriction of G to the events of $G.E \setminus S$, by adding the events of S one by one in the order prescribed by $<_G$ in a *coherence-maximal* way: each read event must read from the oo -maximal same-location

write, and each write must be added at the end of oo . This condition is necessary to guarantee *optimality*, i.e., no execution graph is explored twice. We omit the concrete definition of ISMAXIMAL and refer the reader to Kokologiannakis et al. [12] for details about this condition.

Assuming that next $_P(\cdot)$ always picks an event from the leftmost available thread, we can prove that the steps of TruSt are monotone w.r.t. to the rounds function, i.e., if VISIT $_P(G)$ invokes VISIT $_P(G')$, then rounds(G) \leq rounds(G').

To prove this monotonicity, we need the following corollary that follows directly from TruSt's proof of correctness [13, Prop A.22 (P4)]:

Corollary 1. *Let G be a consistent execution visited by Algorithm 2 and e be either the revisited read, if the last step was a revisit, or the last event added, otherwise. Then, there is no $G.\text{porf}$ -maximal event e' such that $\text{tid}(e') > \text{tid}(e)$.*

Proposition 2. *Assuming that next $_P(\cdot)$ always picks an event from the leftmost available thread, if a call to VISIT $_P(G)$ directly calls VISIT $_P(G')$, then rounds(G) \leq rounds(G').*

Proof. For calls from lines 11 and 13, we immediately have rounds(G) \leq rounds(G') from Prop. 1.

The remaining case is when the call to VISIT(P, G') results from a revisit at line 17. Let \hat{G} be the execution that results from G' by removing the added write a and the revisited read r , and S be the linearization of $G.\text{po}$ on the events in $G.E \setminus \hat{G}.E$ in non-decreasing thread identifier order. Note that r is the first event in S . From TruSt's proof of correctness [13, Prop A.22 (P3)], G can be obtained from \hat{G} by adding the missing events in the order they appear in S in a coherence-maximal way. We now consider two cases, depending on whether there exists a trace τ of \hat{G} with rounds(\hat{G}) rounds such that $\text{tid}(\text{last}(\tau)) \leq \text{tid}(r)$, where last(\cdot) returns the last event of a trace.

If there is such a trace τ , then it is easy to see that $\tau ++ S$ is a trace of G and rounds($\tau ++ S$) = rounds(τ). Thus we have rounds(G) \leq rounds($\tau ++ S$) = rounds(τ) = rounds(\hat{G}). From monotonicity, we have rounds(\hat{G}) \leq rounds(G'), which gives us the desired rounds(G) \leq rounds(G').

Otherwise, let $\hat{\tau}$ be a trace of \hat{G} with rounds(\hat{G}) rounds. Again, $\hat{\tau} ++ S$ is a trace of G , but rounds($\hat{\tau} ++ S$) = rounds($\hat{\tau}$) + 1, because $\text{tid}(\text{last}(\hat{\tau})) \leq \text{tid}(r)$. Since rounds(G) \leq rounds($\hat{\tau} ++ S$) and rounds($\hat{\tau}$) = rounds(\hat{G}), showing that rounds(\hat{G}) \leq rounds(G') - 1 suffices to prove that rounds(G) \leq rounds(G').

Assume the opposite, i.e., rounds(\hat{G}) \geq rounds(G') and let $K = \text{rounds}(G')$. From monotonicity, rounds(\hat{G}) $\leq K$, and thus rounds(\hat{G}) = K . Let G'' be the execution that results from removing r from G' . Since $\hat{G} \sqsubseteq G'' \sqsubseteq G'$, from monotonicity, it is also rounds(G'') = K . From TruSt's proof of correctness [13, Prop A.22 (P9)], because a revisited r in G' , $\text{tid}(a) > \text{tid}(r)$. From Corollary 1 for G' , any event e' with $\text{tid}(e') > \text{tid}(r)$ is not $G'.\text{porf}$ -maximal, and therefore the only event e' in G'' with $\text{tid}(e') > \text{tid}(r)$ that is $G''.\text{porf}$ -maximal is the write a . Any trace τ'' with K rounds must end with a , otherwise we can remove a from τ'' and obtain a trace

$\hat{\tau}'$ of \hat{G} with at most K (and therefore exactly K) rounds such that $\text{tid}(\text{last}(\hat{\tau}')) \leq \text{tid}(r)$, which contradicts the hypothesis. Let τ' be a trace of G' with K rounds. Removing r from τ' results in a trace τ'' of G'' with at most K (and therefore exactly K) rounds. Since τ'' ends with a , and r must be after a in τ' , it is $\tau' = \tau ++ [r, a]$, for a trace τ of \hat{G} . Therefore $\text{rounds}(\tau') = \text{rounds}(\tau) + 1$ ($\text{tid}(a) > \text{tid}(r)$). This leads to a contradiction: $\text{rounds}(\hat{G}) = \text{rounds}(\tau) = \text{rounds}(\tau') - 1 = \text{rounds}(G') - 1 = \text{rounds}(\hat{G}) - 1$. \square

IV. OPTIMAL BOUNDED DPOR

Given Prop. 2, we can trivially obtain an algorithm that explores all executions of a program P with up to k rounds. Let **ROUNDER** be Algorithm 2 that, apart from consistency, also checks whether $\text{rounds}(G) \leq k$ at line 4.

ROUNDER is sound, complete, and optimal. Soundness is trivial because any execution that is not consistent or exceeds the bound k is dropped. Completeness of **ROUNDER**, i.e., **ROUNDER** explores every consistent execution of P with up to k rounds, follows from the completeness of **TruSt** and Prop. 2. Optimality, i.e., no execution graph is explored twice, is inherited from the **TruSt** algorithm because **ROUNDER** explores a subset of the executions that **TruSt** does.

Theorem 1. ***ROUNDER** is sound, complete, and optimal.*

V. EXTENSIONS FOR WEAK MEMORY MODELS

While in the previous sections we have focused on sequentially consistent executions, the framework can also be used for weaker memory models, such as x86-TSO, PSO, and RC11. In fact, **TruSt** is parametric in the choice of the memory model, provided it respects some common assumptions, such as ruling out **porf** cycles.

Our optimal bounding approach can be similarly generalized to such memory models by choosing a bounding function that validates Prop. 2. A sufficient condition is for the function (a) to be monotone w.r.t. the prefix relation and (b) to not be affected by the coherence maximal addition of an event. To see this, note that any execution graph that **TruSt** visits in order to reach a final execution graph G_f is a prefix of G_f that is (possibly) extended with coherence-maximally added events [19].

One suitable such function is the modification of Algorithm 1 by changing $G.\text{sc}$ to just $G.\text{porf}$. Another is to count the number of simple **sc** cycles in an execution graph, or the number of events participating in such cycles. It is also possible to combine the results of multiple such functions by any monotone operation (e.g., addition or to return a tuple).

VI. EVALUATION

We implemented **ROUNDER** on top of the **GENMC** tool [15], which implements the **TruSt** algorithm. To evaluate **ROUNDER**, we investigate (a) how many rounds are usually enough to discover concurrency bugs (§ VI-A), and (b) how efficient is **ROUNDER** for that number of rounds (§ VI-B).

Our evaluation shows that 2 rounds suffice to uncover almost all concurrency bugs, and for a bound of 2, bounded search is generally much faster than a plain DPOR algorithm.

TABLE I
ROUNDER’S SPEEDUP COMPARED TO GENMC

Benchmark	GENMC Time (s)	$k = 2$ Speedup	$k = 3$ Speedup
bstack(5)	90.37	37.50	5.11
bstack(6)	859.83	104.35	9.89
bstack2(8)	174.72	95.48	9.84
bstack2(9)	730.50	243.50	19.37
dglm-queue(6)	105.58	9.36	2.17
dglm-queue(7)	589.54	22.49	3.88
dglm-oe(7)	22.23	3.01	1.02
dglm-oe(8)	33.33	3.60	1.11
dglm-fifo(7)	24.40	1.81	1.10
dglm-fifo(8)	40.48	1.88	0.83
ms-queue(6)	296.69	42.44	4.96
ms-queue(7)	148.64	34.09	4.61
ms-queue(8)	660.85	81.39	8.50
ms-oe(6)	242.88	23.13	4.26
ms-oe(7)	490.78	34.22	5.62
ttas-lock2(7)	28.13	7.99	2.12
ttas-lock2(8)	149.35	19.20	3.93
ttas-lock3	424.31	22.30	3.80

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

A. Rounds and Bug Discovery

To evaluate how many rounds are sufficient to discover concurrency bugs in practice, we run **ROUNDER** on the two sets of benchmarks used in the evaluation of **BUSTER** [19].

For the first set of benchmarks consisting of programs from **SV-COMP** [24] and **SCTBench** [25], **ROUNDER** discovered all bugs with only two rounds, apart from one benchmark with 100 threads where it times out before discovering the bug. **ROUNDER** managed to find one more bug than **BUSTER** before timing out because of the significantly reduced overhead of the bound calculation.

For the second set of benchmarks, consisting of concurrent data structures with induced bugs, **ROUNDER** discovered again all bugs with two rounds, with the exception of two benchmarks where it timed out, while **BUSTER** does not. The reason for this is that the number of executions grows faster as the round-robin bound increases compared to when the preemption-bound increases.

B. Bounding Efficiency

To evaluate how efficient **ROUNDER** is, we compared its execution time for bounds of two and three against the unbounded DPOR algorithm implemented in **GENMC**. We again used a set of correct concurrent data structures as benchmarks.

Our results are summarized in Table I. We report the execution time of **GENMC** and the speedup when run with **ROUNDER** for bounds of two and three. In most benchmarks, **ROUNDER** is significantly faster under both bound values. For the “dglm-oe” and “dglm-fifo” benchmarks, little to no

speedup is observed because **ROUNDER** explores most to all program executions for these small bounds.

In comparison to **BUSTER**, we note that the execution time of **ROUNDER** grows faster as the bound increases. This happens because scheduling rounds are a coarser-grained bounding metric than preemptions: one additional round typically allows many more executions than one additional preemption.

VII. RELATED WORK AND CONCLUSIONS

In this paper, we have presented the first optimal bounded DPOR algorithm. While bounding the number of round-robin scheduling rounds in the context of DPOR is a novel contribution of this paper, the bound itself is not new. It was first used by Lal et al. [17] as a technical device to show that preemption-bounded verification of concurrent pushdown automata is decidable, and has since been used for related decidability results.

Two other works have tried to integrate notions of concurrency bounding into DPOR algorithms, albeit nonoptimally. Specifically, Musuvathi et al. [21] developed the BPOR algorithm, which combines DPOR with a preemption-bound search by weakening the reduction obtained by DPOR to avoid exploring any executions with a larger number of preemptions than the desired bound. As a result, their algorithm explores redundant equivalent executions leading to poor performance, which is often worse than the state of the art in unbounded DPOR. More recently, Marmanis et al. [19] developed a different sound approach for combining DPOR and PB by allowing the exploration of executions that exceed by the desired bound by a certain margin equal to the number of threads in the program minus two. Their tool, **BUSTER**, avoids any redundant exploration, and so is generally faster than unbounded DPOR; it does, however, typically explore a large number executions exceeding the bound, negatively impacting its performance. Both approaches are further limited by the need to determine whether a given execution graph (Mazurkiewicz trace) exceeds the desired preemption bound, which is an NP-complete problem [21].

A large body of work on DPOR devoted on coarser equivalence relations [4, 6, 7, 10, 14] and on supporting weak memory models [1, 3, 11, 14, 16]. These works are mostly orthogonal to our extension over **TruSt** [12], and can likely be integrated into **ROUNDER**.

Finally, Abdulla et al. [2] and Atig et al. [5] have proposed bounds for the TSO and Power memory models, but being based on preemption bounding, they are not very suitable for integration with DPOR. In contrast, the proposed bounds of §V, while coarser, can be integrated smoothly into **ROUNDER**.

ACKNOWLEDGMENTS

We would like to thank the anonymous FMCAD reviewers for their feedback. This work has received funding from Amazon and from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101003349).

REFERENCES

- [1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. “Stateless model checking for TSO and PSO”. In: *TACAS 2015*. Vol. 9035. LNCS. Berlin, Heidelberg: Springer, 2015, pp. 353–367. DOI: 10.1007/978-3-662-46681-0_28. URL: http://dx.doi.org/10.1007/978-3-662-46681-0_28.
- [2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. “Context-Bounded Analysis for POWER”. In: *TACAS 2017*. Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 56–74. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_4.
- [3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. “Optimal stateless model checking under the release-acquire semantics”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 135:1–135:29. ISSN: 2475-1421. DOI: 10.1145/3276505. URL: <http://doi.acm.org/10.1145/3276505>.
- [4] Pratyush Agarwal, Krishnendu Chatterjee, Shreyas Pathak, Andreas Pavlogiannis, and Viktor Toman. “Stateless Model Checking Under a Reads-Value-From Equivalence”. In: *CAV 2021*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, July 2021, pp. 341–366. ISBN: 978-3-030-81685-8. DOI: 10.1007/978-3-030-81685-8_16.
- [5] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. “Context-Bounded Analysis of TSO Systems”. In: *FPS 2014*. Ed. by Saddek Bensalem, Yassine Lakhnech, and Axel Legay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 21–38. ISBN: 978-3-642-54848-2. DOI: 10.1007/978-3-642-54848-2_2.
- [6] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. “Data-centric dynamic partial order reduction”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 31:1–31:30. ISSN: 2475-1421. DOI: 10.1145/3158119. URL: <http://doi.acm.org/10.1145/3158119>.
- [7] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. “Value-Centric Dynamic Partial Order Reduction”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360550. URL: <https://doi.org/10.1145/3360550>.
- [8] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. “Bounded Partial-Order Reduction”. In: *OOPSLA 2013*. Indianapolis, Indiana, USA: ACM, 2013, pp. 833–848. ISBN: 9781450323741. DOI: 10.1145/2509136.2509556. URL: <https://doi.org/10.1145/2509136.2509556>.
- [9] Cormac Flanagan and Patrice Godefroid. “Dynamic partial-order reduction for model checking software”. In: *POPL 2005*. New York, NY, USA: ACM, 2005,

- pp. 110–121. DOI: 10.1145/1040305.1040315. URL: <http://doi.acm.org/10.1145/1040305.1040315>.
- [10] Jeff Huang. “Stateless model checking concurrent programs with maximal causality reduction”. In: *PLDI 2015*. New York, NY, USA: ACM, 2015, pp. 165–174. DOI: 10.1145/2737924.2737975. URL: <http://doi.acm.org/10.1145/2737924.2737975>.
 - [11] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. “Effective stateless model checking for C/C++ concurrency”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 17:1–17:32. ISSN: 2475-1421. DOI: 10.1145/3158105. URL: <http://doi.acm.org/10.1145/3158105>.
 - [12] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. “Truly stateless, optimal dynamic partial order reduction”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498711. URL: <https://doi.org/10.1145/3498711>.
 - [13] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. “Truly Stateless, Optimal Dynamic Partial Order Reduction (supplementary material)”. In: (Jan. 2022). URL: <https://plv.mpi-sws.org/genmc>.
 - [14] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. “Model checking for weakly consistent libraries”. In: *PLDI 2019*. New York, NY, USA: ACM, 2019. DOI: 10.1145/3314221.3314609.
 - [15] Michalis Kokologiannakis and Viktor Vafeiadis. “GenMC: A model checker for weak memory models”. In: *CAV 2021*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. LNCS. Springer, 2021, pp. 427–440. DOI: 10.1007/978-3-030-81685-8_20.
 - [16] Michalis Kokologiannakis and Viktor Vafeiadis. “HMC: Model checking for hardware memory models”. In: *ASPLOS 2020*. ASPLOS ’20. Lausanne, Switzerland: ACM, 2020, pp. 1157–1171. ISBN: 9781450371025. DOI: 10.1145/3373376.3378480. URL: <https://doi.org/10.1145/3373376.3378480>.
 - [17] Akash Lal and Thomas Reps. “Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis”. In: *CAV 2008*. Ed. by Aarti Gupta and Sharad Malik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 37–51. ISBN: 978-3-540-70545-1.
 - [18] Leslie Lamport. “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”. In: *IEEE Trans. Computers* 28.9 (Sept. 1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439. URL: <http://dx.doi.org/10.1109/TC.1979.1675439>.
 - [19] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. “Reconciling Preemption Bounding with DPOR”. In: *TACAS 2023*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 85–104. ISBN: 978-3-031-30823-9.
 - [20] Antoni Mazurkiewicz. “Trace Theory”. In: *PNAROMC 1987*. Vol. 255. LNCS. Berlin, Heidelberg: Springer, 1987, pp. 279–324. DOI: 10.1007/3-540-17906-2_30. URL: http://dx.doi.org/10.1007/3-540-17906-2_30.
 - [21] Madalan Musuvathi and Shaz Qadeer. *Partial-Order Reduction for Context-Bounded State Exploration*. Tech. rep. MSR-TR-2007-12. Microsoft Research, 2007. URL: <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2007-12.pdf>.
 - [22] Madanlal Musuvathi and Shaz Qadeer. “Iterative Context Bounding for Systematic Testing of Multithreaded Programs”. In: *PLDI 2007*. San Diego, California, USA: ACM, 2007, pp. 446–455. ISBN: 9781595936332. DOI: 10.1145/1250734.1250785. URL: <https://doi.org/10.1145/1250734.1250785>.
 - [23] Shaz Qadeer and Jakob Rehof. “Context-Bounded Model Checking of Concurrent Software”. In: *TACAS 2005*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. LNCS. Springer, 2005, pp. 93–107. DOI: 10.1007/978-3-540-31980-1_7. URL: https://doi.org/10.1007/978-3-540-31980-1_7.
 - [24] SV-COMP. *Competition on Software Verification (SV-COMP)*. 2019. URL: <https://sv-comp.sosy-lab.org/2019/> (visited on 03/27/2019).
 - [25] Paul Thomson, Alastair F. Donaldson, and Adam Betts. “Concurrency testing using schedule bounding: an empirical study”. In: *PPoPP 2014*. ACM, 2014, pp. 15–28. DOI: 10.1145/2555243.2555260. URL: <https://doi.org/10.1145/2555243.2555260>.

Datapath Verification via Word-Level E-Graph Rewriting

Samuel Coward^{1,2}, Emiliano Morini¹, Bryan Tan¹, Theo Drane¹ and George A. Constantinides²

¹ Intel Corporation, ² Imperial College London,

Email: {samuel.coward, emiliano.morini, bryan.tan, theo.drane}@intel.com, g.constantinides@imperial.ac.uk

Abstract—Formal verification of datapath circuits is challenging as they are subject to intense optimization effort in the design phase. Industrial vendors and design companies deploy equivalence checking against a golden or existing reference design to satisfy correctness concerns. State-of-the-art datapath equivalence checking tools deploy a suite of techniques, including rewriting. We propose a rewriting framework deploying bitwidth-dependent rewrites based on the e-graph data structure, providing a powerful assistant to existing tools. The e-graph allows generation of a path of rewrites between the reference and implementation designs, which can then be checked by a trusted industry tool. We demonstrate that the intermediate proofs generated by the assistant enable convergence in a state-of-the-art tool, without which the industrial tool runs for 24 hours without making progress. The intermediate proofs automatically introduced by the assistant also reduce the total proof runtime by up to 6×.

Index Terms—Formal Verification, Datapath, E-Graph, Equivalence Checking.

I. INTRODUCTION

Arithmetic datapath circuits like adders and multipliers are included in almost every electronic device. Designers of these circuits implement low-level optimizations targeting the best power, performance and area. As a result, the verification of datapath circuits is challenging since the code can be difficult to review, making it hard to identify a sufficient test suite. Typically, exhaustive simulation is infeasible due to the size of the input space. Undetected bugs lead to system failures and reputational damage [1]. Formal Verification (FV) is the only scalable option to prove the absence of bugs in hardware [2].

One of the most successful FV approaches to verify datapath circuit designs is based on Equivalence Checking (EC), where the design under test, usually called the *implementation*, is proven to be equivalent to a golden reference design, often called the *specification*. Electronic Design Automation (EDA) vendors have developed commercial tools drastically lowering the entry barrier [3], allowing semiconductor companies to fully verify many different designs [4], [5].

Commercial tools orchestrate a suite of solver technologies [3], including SAT, SMT and BDD based solvers. Yet still some simple designs can not be proven equivalent. For example, an industrial state-of-the-art tool is unable to prove the equivalence of the two designs shown in Figure 1 without requiring manual effort to apply advanced formal techniques. We enhance the capabilities of such tools by deploying word-level rewriting in combination with a data structure, known as an e(quality)-graph. E-graphs are found at the heart

<pre> module spec(A,B,M,N,O); input [15:0] A, B; input [3:0] M, N; output [62:0] O; wire [30:0] D; wire [30:0] E; assign D = A << M; assign E = B << N; assign O = D * E; endmodule </pre>	<pre> module impl(A,B,M,N,O); input [15:0] A, B; input [3:0] M, N; output [62:0] O; wire [31:0] C; wire [4:0] P; assign C = A * B; assign P = M + N; assign O = C << P; endmodule </pre>
--	---

(a) Specification design.

(b) Implementation design.

Fig. 1: A motivational example, where existing EC tools fail to prove the equivalence of these two designs.

of modern SMT solvers [6], but by applying them at the abstraction level used by humans in RTL design we can tailor the rewrites to datapath verification.

In this work we modify an existing e-graph-based RTL optimization tool [7] to produce a powerful formal verification assistant. The proposed verification assistant is able to exceed the capabilities of the industrial state of the art, reduce verification runtimes and decrease the complexity of the EC problem. The approach taken here is similar to that of Stepp, Tate and Lerner, who initially developed an e-graph based LLVM optimizer [8], and later modified it to perform translation validation [9]. We differ from this previous work in that we validate numerically intense optimizations at a lower abstraction level often performed by a human rather than a compiler. We also deploy modern e-graph developments allowing us to incorporate value range analysis techniques and can generate a simplified EC problem for FV engineers. The approach presented is sound, as we check each intermediate step using a trusted EC tool. The paper contains the following novel contributions:

- a word-level e-graph framework that composes a set of sub-problems from local rewrites to assist FV tools,
- a specialized and extensible bitwidth dependent rewrite set for datapath verification,
- an e-graph extraction method minimizing the ‘distance’ between two designs,
- test cases showing an enhancement in capabilities over industrial tools, reducing the need for manual FV effort.

First, we provide the necessary background on verification and e-graphs. In Section III we describe how word-level e-

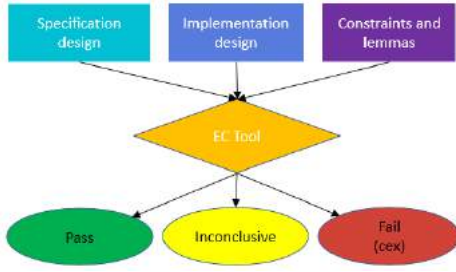


Fig. 2: The inputs of an EC tool are two designs, specification and implementation, a set of constraints to drive the possible values to tests and a set of lemmas to prove. Each lemma can pass, fail or be inconclusive. A counterexample (cex) is provided for each failing lemma.

graphs can be applied to produce a verification assistant. In Section IV we describe a case study where we outperform the industrial state of the art. Finally, in Section V we present results demonstrating overall verification runtime improvements.

II. BACKGROUND

A. Datapath Verification

Classic formal property verification methods successfully used to verify state machines and communication protocols are not able to verify datapath dominated circuits. Theorem Proving [10]–[14] and Symbolic Trajectory Evaluation [15] are valuable approaches, but their common downsides are a high barrier to entry and maintenance of complex code bases.

An alternative and successful approach is to rely on EC, defining two circuit representations to be equivalent if for all valid inputs they generate identical outputs. EC has been used in several contexts in the semiconductor industry [4], [16]. The most popular types of EC are Boolean, Sequential and Transactional, and in this paper we focus on Transactional EC of combinational circuits, where the result of a given computation in the *implementation* is compared against the result of the same computation in the trusted *specification*. The output of the comparison can be *pass*, when a property is proven, *fail*, when the property is not true (a counterexample is generated), or *inconclusive*, when the tool does not manage to either prove or disprove a property. See Figure 2.

A trusted reference is fundamental for EC. One standard verification flow used in the semiconductor industry is the following: starting from a component specification, a developer writes a high-level reference C++ design without any interaction with the designer who writes the RTL implementation, providing *diversity* and *independence* between the two, which are then formally tested for equivalence. Many more tests can be run on the C++ code, due to the great difference in simulation speed between C++ and RTL. This is usually described as *C2RTL* EC. Another common option is what is called *RTL2RTL* EC, where the reference is a trusted version of the same design in RTL, usually a version from previous projects or based on a third party library like Synopsys’ DesignWare [17].

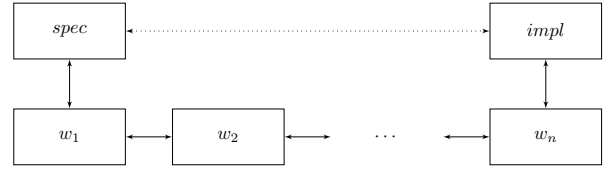


Fig. 3: Overview of the waterfall approach used by FV engineers. The dashed line between *spec* and *impl* represents an inconclusive verification. Full equivalence is achieved introducing n intermediate designs w_i and proving the equivalences of all the pairs $(spec, w_1)$, (w_1, w_2) , \dots , $(w_n, impl)$.

Inconclusive results are commonplace in real-life EC and require advanced techniques to achieve full convergence, occupying most of the FV engineer’s time. A common approach is to generate a “waterfall”, where the verification between implementation and specification is achieved by introducing intermediate designs, as shown in Figure 3. If all the intermediate equivalence steps are proven, the equivalence between specification and implementation holds.

One of the key motivations for this work derives from an overview of the technology behind Synopsys’ industry leading Datapath Validation (DPV) tool [3]. The tool orchestrates a suite of techniques and solvers to prove the equivalence of input designs. One of these techniques is a set of rewrite engines. In [3], the authors state that certain rewrite sets “are only applied selectively” or their application “can be counterproductive”. As a result these rewrite engines are heuristic and may not explore the required space. The techniques presented in Section III describe a rewrite orchestration approach that does not suffer from these limitations.

One relevant work combined rewriting and theorem proving to verify the correctness of gate-level multiplier designs in RTL [18], [19]. In this work, the authors deploy ACL2 verified [20] rewrites to transform optimized implementations into normalized implementations. Whilst our work targets a higher abstraction level, techniques and principles applied in the multiplier verification work will be relevant here.

B. E-Graphs

E-graphs cluster equivalent expressions into e(quivalence)-classes, enabling a compact representation of alternative but functionally identical implementations. In the e-graph, nodes represent variables, constants or operators that point to children e-classes. This captures the intuition that we may choose how to implement a given sub-expression at any point in the design. Due to these nested choices, an e-graph can represent exponentially many implementations in the number of nodes.

An e-graph is grown via constructive application of local equivalence preserving rewrites, $l \rightarrow r$, where the right-hand side of the rewrite is added to the e-class containing l , without removing l as would be done in a traditional rewrite engine. As a result, the e-graph avoids the phase-ordering problem, where the order of application impacts the results. This approach to growing an e-graph is known as equality saturation [8], [21],

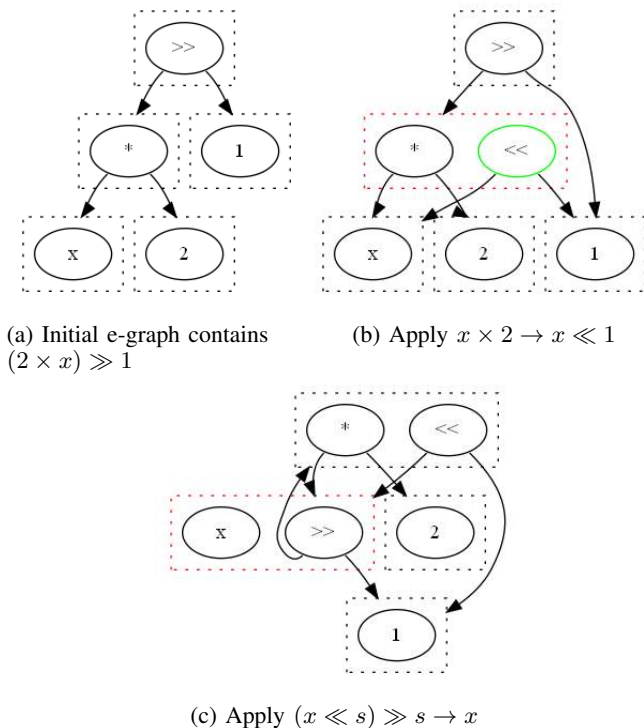


Fig. 4: Simple e-graph rewriting over the integers. The dashed boxes represent e-classes of expressions, where we have highlighted the modified e-class in red at each stage. Green nodes represent newly added nodes.

[22]. A simple e-graph rewriting example is shown in Figure 4, where the dashed boxes represent e-class boundaries and the arrows connect nodes to their child e-classes.

The e-graph data structure has been used in the formal methods community for many years [23] and can be found in modern SMT solvers such as Z3 [6]. One particularly relevant work used e-graphs to perform translation validation of an LLVM compiler [9]. The recently released *egg* library [21] has enabled researchers to quickly develop a wide range of e-graph applications, ranging from hardware design [7] to rewrite rule synthesis [24]. By using e-graphs to represent datapath circuit designs in RTL we can take advantage of the e-graphs’ ability to explore equivalent designs efficiently.

III. PROVING EQUIVALENCE VIA E-GRAPH REWRITING

The problem we tackle is, given specification and implementation RTL designs, prove them equivalent or reduce the original EC problem to a simpler one to solve. Given this objective, we will now describe how e-graph rewriting can provide an efficient solution. Figure 5 illustrates the overall flow of the assistant. In this work we used a particular commercial tool throughout, but any RTL2RTL EC tool could be substituted in its place.

A. E-Graph Initialization

Using the framework developed in [7], we produce an e-graph representation of RTL, encoding all signal bitwidth and

signage definitions. We use an intermediate language made up of nested S-expressions, as in Common Lisp [25]:

```
term ::= (operator [term] [term] ... [term])
```

For example, the following System Verilog:

```
logic [7:0] a, b;
logic [8:0] c;
always_comb c = a[7:0] + b[7:0];
```

corresponding to an unsigned addition of two primary 8-bit inputs *a* and *b*, stored in a 9-bit result is expressed as:

(+ 9 unsigned 8 unsigned *a* 8 unsigned *b*).

The + operator takes eight arguments, describing the output and operand signals.

This intermediate language is sufficient to correctly represent the functional behaviour of combinational RTL. Verilog operator definitions are context dependent meaning knowledge of all bitwidth and signage definitions is essential [26]. In this work we target word-level RTL written in System Verilog. Using the open-source Slang parser [27], we implemented an automated flow converting System Verilog into this intermediate language. We parse both RTL designs and generate expressions, *S* and *I*, in the intermediate language, for the specification and implementation respectively.

In most e-graph applications built using *egg*, the e-graph is initialized with a single expression representing the design to be optimized. However, in our work we initialize the e-graph with both *S* and *I*, such that the e-graph has two roots. The nodes common to both designs are automatically shared by *egg*. In this paper we describe RTL generating a single output, but using constructs from [7] it is trivial to generalize to multiple outputs from each design.

In Figure 6, we represent the two designs shown in Figure 1 in a single e-graph. Colors indicate the design in which each node is used. Note that the designs initially only share the input variables and no intermediate signals. In the following sections we will discuss how as the e-graph is grown, common intermediate signals can be discovered. Initialising the e-graph with both designs means that we can simultaneously rewrite both designs in order to find a common equivalent.

B. Bitwidth Dependent Rewriting

The rewrites define the space of equivalent designs that can be reached as the e-graph grows. We build upon a subset of the bitwidth dependent rewrites described in [7], which was originally designed for optimization and was learnt from industrial RTL engineers. The optimization rewrite set deployed specific rewrites to improve correlation with the downstream logic synthesis tool. These rewrites are not deployed in the verification rewrite set. It is natural that the verification rewrite set should include many of the optimization capabilities but also incorporate additional verification specific rewrites that ‘undo’ optimizations. For example, it may be productive to include transformations that introduce redundant logic that enables further sharing.

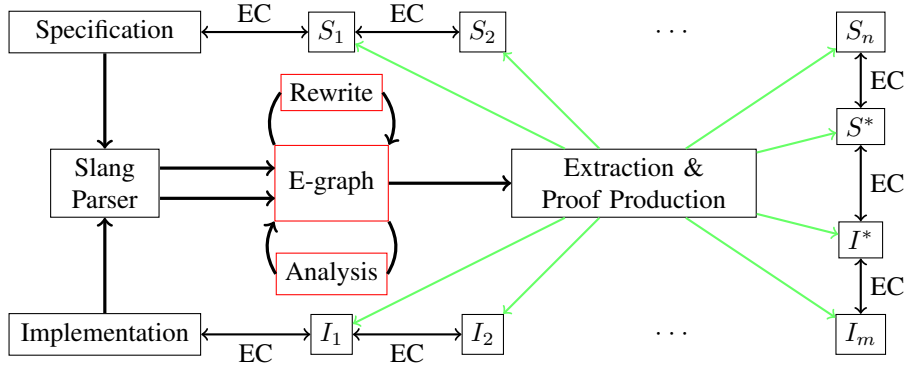


Fig. 5: Flow diagram for the verification assistant, taking a specification and implementation circuit design in System Verilog. The designs are parsed and an e-graph is constructed. From the rewritten e-graph, extract two designs S^* and I^* along with intermediate designs forming a verification waterfall.

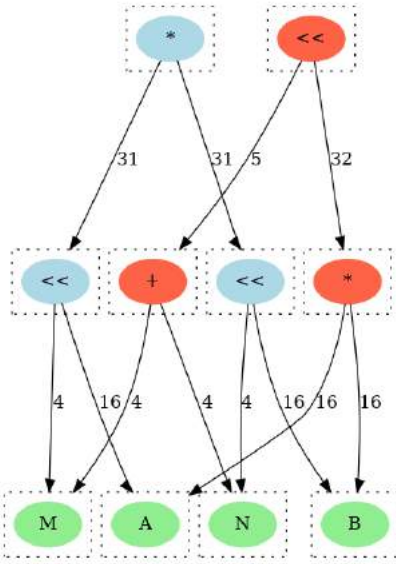


Fig. 6: Initial e-graph representing two designs shown in Figure 1, a specification (blue) and implementation (red). Shared nodes are colored green. Edge labels denote bitwidths. All e-classes (dashed boxes) initially contain a single node.

Table I describes the small set of additional verification specific rewrites learnt from experience using commercial EC tools. Several of these rewrites are the reverse of rewrites targeting optimization. The space of rewrites that ‘undo’ optimizations is less intuitive, so selecting valuable rewrites to include is challenging. We selected rewrites that were relevant for the test cases presented here. The assistant is designed such that it is simple for users to extend the rewrite set with their own transformations that are applicable to their designs.

One important consideration is to ensure that few rewriting opportunities are missed. To achieve this we parameterize the pattern matching left-hand side and apply the rewrites conditionally. We construct necessary and sufficient conditions that are functions of the rewrite parameters. Letting $l(\cdot)$

TABLE I: An example set of bitwidth dependent datapath verification rewrites. All rewrites are conditionally applied to ensure correctness. Bitwidth and signage information of operators and operands is omitted here for concision.

Name	Left-hand Side	Right-hand Side
Unmerge Shift	$a \ll (b + c)$	$(a \ll b) \ll c$
Mult Left Shift	$a \times (b \ll c)$	$(a \times b) \ll c$
Shift to Mult	$a \ll \text{const}$	$a \times 2^{\text{const}}$
Mult to Add	$a \times 2$	$a + a$

and $r(\cdot)$ denote functions mapping a vector of parameters \vec{p} , encoding operand bitwidth and signage, to expressions in the intermediate language. Given a parameterized rewrite, $l(\vec{p}) \rightarrow r(\vec{p})$, we construct a condition, ϕ , such that $l(\vec{p}) \cong r(\vec{p}) \iff \phi(\vec{p})$. The sufficiency ensures that only valid, equivalence preserving, rewrites are applied. The necessity guarantees that no rewriting opportunities are missed for this rewrite. Missed opportunities can be the difference between a proven equivalence check and an inconclusive result. We will see this in Section IV.

A challenge for RTL level verification is that functional behaviour is bitwidth dependent, for example the addition of two 8-bit values stored in an 8-bit and a 9-bit result differ in general but may be equivalent under certain design constraints. We use the interval analysis and bitwidth reduction rewrites described in [28], deploying `egg`’s built-in e-class analysis feature. These rewrites detect and reduce operators to the minimum bitwidth required to store the result, hence normalizing the operations. Such techniques are also deployed in commercial tools [3], but program analysis on e-graphs is able to provide more precise abstractions [29].

Having defined a set of rewrites, we use equality saturation to apply them to the e-graph initialized as described in Section III-A. Rewrites are applied to both the specification and implementation designs simultaneously with the objective being to discover equivalent sub-expressions across the two designs. As rewrites are applied, new nodes are added to the e-graph and the e-classes grow, as we see in Figure 8. We vary the number of e-graph rewriting iterations to control the e-graph growth

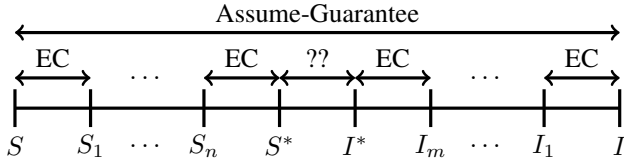


Fig. 7: E-graph extracted waterfall, generated automatically by the assistant. We use EC tools to prove the equivalence of the intermediate steps. The central equivalence check between S^* and I^* , which may not be true, may not be provable using the EC tool, but represents a simplified problem.

throughout this work. Constructive rewrite application adds the overhead of maintaining many equivalent representations of the two designs in the e-graph, but greatly simplifies the problem of determining a correct rewrite application order.

C. Extraction

Once the e-graph has saturated or reached a timeout, the e-graph represents two sets of equivalent designs, one for the specification and one for the implementation. From the e-graph we now seek to extract two designs, $S^* \cong S$ and $I^* \cong I$ that share the maximum number of common nodes. If S and I are found in the same e-class, namely the tool found a path of rewrites between the two designs, then S^* and I^* are identical. If they are found in different e-classes, we extract distinct S^* and I^* sharing as many of the common sub-expressions as is feasible from the e-graph. Figure 5 shows the flow.

To extract S^* and I^* we first identify which e-classes in the e-graph are associated with each design. Let C denote the set of all e-classes. Given a root e-class, r , we recursively construct an associated $C_r \subseteq C$. Starting from $C_r = \emptyset$, we iterate through each node in r , adding its children e-classes to C_r . We continue, recursively visiting each of the child e-classes and iterating through the contained nodes until C_r stops growing. This construction is guaranteed to terminate.

Letting $\text{root}(e)$, be a function returning the root e-class for a given expression e in the intermediate language, we use the algorithm described above to construct $C_{\text{spec}} \subseteq C$ starting from $\text{root}(S)$ and $C_{\text{impl}} \subseteq C$ starting from $\text{root}(I)$. We construct the shared e-class set, $C_{\text{shared}} = C_{\text{spec}} \cap C_{\text{impl}}$, which is used to identify the S^* and I^* that share the most common nodes. We update the spec and impl sets, $C'_{\text{spec}} = C_{\text{spec}} \setminus C_{\text{shared}}$ and $C'_{\text{impl}} = C_{\text{impl}} \setminus C_{\text{shared}}$. In Figures 6 and 8, we highlight C'_{spec} in blue, C'_{impl} in red and C_{shared} in green.

In previous work we have deployed hardware specific cost functions, for example circuit area or delay, that we seek to minimize in the extraction phase [7], [28]. In this instance, we use a simpler objective function of e-graph nodes n :

$$\text{obj}(n) = \begin{cases} K, & \text{if } \text{class}(n) \in C_{\text{shared}}, \\ -1, & \text{otherwise,} \end{cases} \quad (1)$$

where $\text{class}(n)$ returns the e-class containing the node n and K is the total number of e-classes in the e-graph. We maximize this objective function to ensure that we share the maximum

number of nodes possible, where the negative scoring of unshared nodes ensures that amongst designs sharing the same number of nodes, we extract the simplest one. We formulate the problem as an integer linear programming problem (ILP) [30]. We define N to be the set of nodes and $E \subseteq N \times C$ the set of edges. We also introduce N_c to denote the set of nodes in a given e-class c and P_c to denote the set of parent nodes of c . For each node $n \in N$ we associate an objective, $\text{obj}(n)$, and a binary variable $x_n \in \{0, 1\}$, which indicates whether n is implemented in either of the extracted RTL designs. Lastly we introduce $R = \text{root}(I) \cup \text{root}(S)$. With these definitions the problem formulation is the following:

$$\text{maximize} \quad \sum_{n \in N} \text{obj}(n) \cdot x_n \quad (2)$$

$$\text{subject to} \quad \forall (n, c) \in E \quad : \quad x_n \leq \sum_{\hat{n} \in N_c} x_{\hat{n}} \quad (3)$$

$$\forall c \in R \quad : \quad \sum_{n \in N_c} x_n = 1 \quad (4)$$

$$\forall c \in C \quad : \quad \sum_{n \in N_c} x_n \leq 1 \quad (5)$$

$$\forall c \in C \text{ s.t. } P_c \neq \emptyset : \quad \sum_{n \in N_c} x_n \leq \sum_{\hat{n} \in P_c} x_{\hat{n}}. \quad (6)$$

In the ILP problem, (3) guarantees that for every node n , we implement a node from each of its child e-classes, extracting only valid designs. (4) then ensures that the outputs from both specification and implementation designs are produced by the extracted design. Lastly, (5) allows at most one node in each e-class to be implemented and (6) ensures that only e-classes with implemented parents are selected, namely there are no unused signals in the generated RTL. We deploy topological sorting variables to handle cycles in the e-graph [7], [30]. We use the Coin-Or CBC solver to solve the ILP problem.

For improved performance, we also use a comparable objective function that computes a greedy extraction based on *egg*'s built-in method. Such an approach is faster but fails to correctly account for common sub-expressions so may generate designs that are not as 'close' as the ILP approach. We would recommend the ILP approach for solving EC problems that will require manual intervention.

The extracted solution corresponds to two expression in the intermediate representation, S^* , equivalent to the specification and I^* , equivalent to the implementation, from which the tool automatically generates RTL. Using the recently added proof production feature in *egg* [31], two sequences of intermediate designs separated by a single rewrite are produced such that

$$S \cong S_1 \cong \dots \cong S_n \cong S^* \text{ and } I \cong I_1 \cong \dots \cong I_m \cong I^*.$$

To remove the need to trust the correctness of the rewrites, the assistant generates System Verilog implementations for each of the intermediate designs and deploys the EC tool to formally verify the equivalence at each step as shown in Figure 7. If the EC tool can prove each step including $S^* \cong I^*$, we have proven the equivalence of S and I . Each intermediate proof is independent and can thus be proven in

parallel. We can also specialize the solver configuration for each intermediate proof, since we are able to map rewrites to an optimal solver setup. For example, the commercial tool provides a set of solve scripts that handle proof orchestration with different capabilities. These scripts can be enabled by a user. We encoded a mapping from rewrites to the most efficient solve script in the assistant. With limited effort the assistant can be extended to target additional solvers.

To ensure soundness of the generated waterfall, a final “Assume-Guarantee” lemma proving $S \cong I$ is included, which uses all of the intermediate proofs (assuming they passed). This provides confidence that no gaps were left in the reasoning. If the tool is unable to prove $S^* \cong I^*$ then human intervention is required. However the EC problem is simplified, as these designs share more common signals than the original S and I .

IV. CASE STUDY

We present a case study of a real world problem where this technique proves beneficial. In all the following results we use an up-to-date version of the commercial EC tool running on SLES 12 on Intel Xeon W-2155 CPUs.

The designs shown in Figure 1 are alternative ways to implement floating point multiplication of denormal numbers. More precisely, given two denormals $2^{1-bias} \times 0.mant_a$ and $2^{1-bias} \times 0.mant_b$, the product of their mantissas is usually reduced to a standard non-denormal multiplication by shifting the values, expressing it as either $(mant_a \ll m) \times (mant_b \ll n)$ or equivalently as $(mant_a \times mant_b) \ll (m + n)$, where $m = lzc(mant_a) + 1$, $n = lzc(mant_b) + 1$ and $lzc(\cdot)$ is the leading zero counter function.

In three iterations of rewriting the e-graph applies a sequence of rewrites such that the specification and implementation are found within the same e-class. The progress of the e-graph can be seen in Figures 8. After two iterations of rewriting the first shared signal is detected, see the green left-shift in Figure 8a, where we have highlighted the initial specification and designs sharing the green node with brighter arrows. The e-graph shown in Figure 8b, after three iterations of rewriting, contains only green nodes, since the tool was able to apply a sequence of rewrites such that the original root nodes of S and I were merged into the same equivalence class. As a result, all e-classes are shared, meaning $C_{shared} = C$.

From the final e-graph, Figure 8b, the tool then extracts identical S^* and I^* along with the sequence of rewrites that were applied to reach it. We summarise the rewrites applied below, omitting bitwidth alteration and commutativity steps.

$$(A \times B) \ll (M + N) \rightarrow \quad (7)$$

$$\text{Unmerge Left-Shift } ((A \times B) \ll N) \ll M \rightarrow \quad (8)$$

$$\text{Left-Shift Mult } (A \times (B \ll N)) \ll M \rightarrow \quad (9)$$

$$\text{Left-Shift Mult } (A \ll M) \times (B \ll N) \quad (10)$$

The e-graph assistant runs in 0.14 seconds, growing an e-graph comprised of 77 nodes. The EC tool is unable to prove the “Left-Shift Mult” and “Mult Left-Shift” transformations

when non-uniform bitwidths are used. We resolve this by automatically inserting an additional intermediate step with standardized bitwidths. We hypothesize that this is due to a rewrite rule only being applied under certain parameterizations in the EC engine.

Including all commutativity and bitwidth alteration rewrites, the assistant generated a total of 20 intermediate equivalence checks (including the “Assume-Guarantee” lemma) for the EC tool to prove. All intermediate proofs and the final completeness lemma are proven in 0.1 seconds by the EC tool. In contrast, when passed the original EC problem, $S \cong I$ with no assistance, the tool did not return a result within 24 hours.

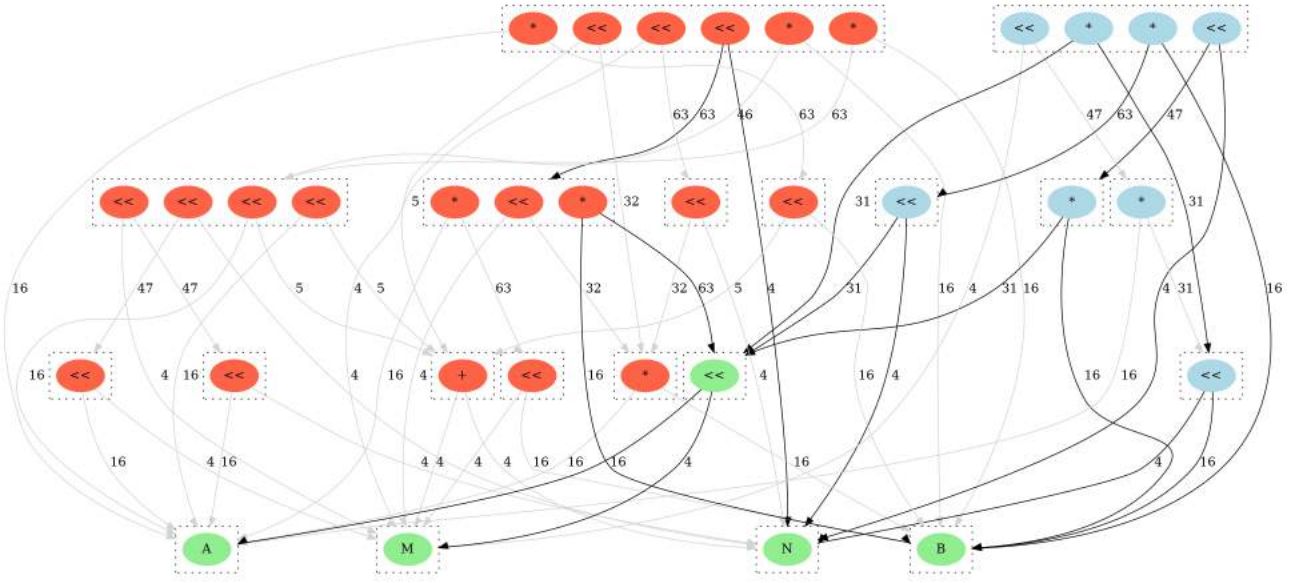
V. RESULTS

Having demonstrated how the verification assistant can provide the intermediate steps transforming a previously inconclusive proof into one solved in under one second, we will now demonstrate how the assistant can improve overall verification runtimes across a datapath optimization benchmark set. We take benchmarks from [32] and implement original and optimized RTL for those designs that are fully described in this paper. The ADPCM decoder is an approximate multiplication implementation. We include two instances of a kernel from the H.264 VBSME (variable block size motion estimator), which correspond to absolute difference summation trees of size four and eight, $\sum_i |a_i - b_i|$. The FIR Filter is a typical finite impulse response filter of depth eight. The case study and box filter are Intel provided benchmarks. The box filter is a reconfigurable square filter, sampling four pixels at a time. The dataflow graph for this design is shown in Figure 9. The optimized design deploys constant factorization and mux rewriting which is relatively challenging for the EC tool to prove.

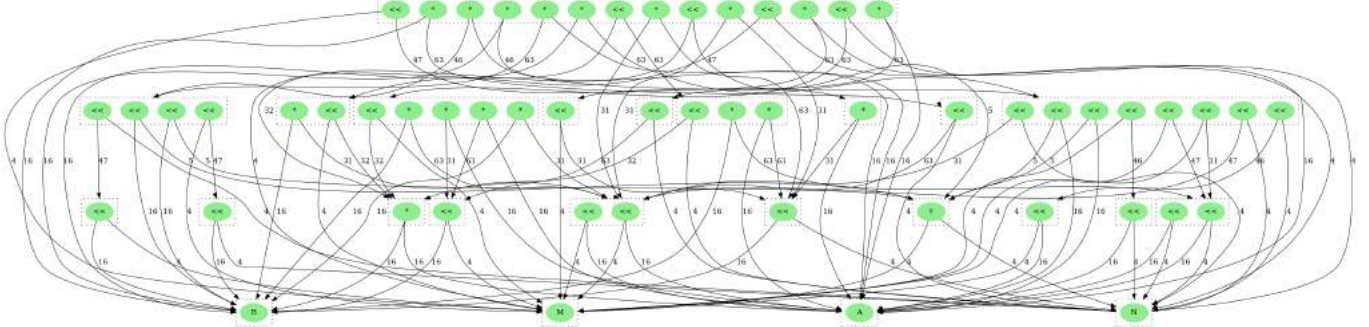
The benchmarks include a range of arithmetic and logical operators, representative of typical RTL optimizations that may be performed by hand or by a specialized datapath optimization tool. For each benchmark, we run the assistant until either, it discovers a complete path between specification and implementation or it deploys five iterations of rewriting, whichever comes first. The e-graph applies all rewrites in parallel at each iteration, meaning that many parts of the designs can be simultaneously transformed in each iteration.

In these results, the EC tool does not report any increase in the initial compilation time, which is less than a second for all cases presented here. We report the runtime from when the solvers start running. For the baseline, we deploy all the EC tool’s solvers in parallel and take the fastest proof. When the verification assistant has generated a sequence of intermediate proofs, we report the maximum time taken to solve a single sub-problem, since each proof can be run in parallel. In practice, the industrial tool’s multi-processor environment introduced runtime overhead that was not related to the proof. Namely, running a proof on a server grid produced unpredictable runtime results due to the licence checks and interactions with the workload management software.

Table II presents the performance impact of the assistant on the total verification time. In the first example, ADPCM



(a) E-graph after two iterations of rewriting. Designs sharing an intermediate signal are highlighted with black arrows.



(b) E-graph after three iterations of rewriting (77 nodes), where S and I have been merged into the same e-class.

Fig. 8: Stages of e-graph growth starting from the initial e-graph in Figure 6.

Decoder, the EC tool efficiently proves the correctness of the two designs, meaning that the overhead of the assistant is detrimental, increasing total runtime. It is worth noting that the intermediate proofs do help reduce the solve time.

In the remaining benchmarks, the baseline EC tool takes longer to prove equivalence. The introduction of intermediate proofs reduces the EC solve time by up to 465x, when we just compare the EC tool runtimes and discount the assistant's runtime. Including the runtime to generate the intermediate proofs, the total verification time is reduced by up to 6x. In most cases, the EC tool solves each of the intermediate proofs in less than 0.5 seconds as each step represents a single local modification to the design. The assistant can effectively select the most optimal solver orchestration script per intermediate proof, which greatly helps performance. This is possible because the assistant understands what transformation has been applied at each stage. Such an approach avoids wasted compute resources, since there is no need to run different solvers in parallel for each of the intermediate problems.

The box filter is an Intel provided benchmark and is the only

example where the assistant is unable to find a complete path. This verification problem may require additional rewriting iterations or entirely new rewrites to reach the implementation design. To minimize runtime, we deploy the faster greedy extraction method. To solve the EC problem, $S^* \cong I^*$, we default to one of the slower but more powerful solver orchestration scripts. In this case, the $S^* \cong I^*$ EC problem takes significantly longer to prove than the other sub-problems. In general, as the assistant is able to deploy longer sequences of dependent rewrites, corresponding to more iterations of rewriting, we expect to find S^* and I^* that are increasingly close. In Table II, we reported box filter results based on five iterations of rewriting. If we instead limit the e-graph to three iterations of rewriting, the assistant's runtime is reduced from 16 seconds to 2 seconds. The intermediate proofs generated by this smaller e-graph can be proven in 1.12 seconds, reducing the total verification time to approximately 3 seconds, corresponding to a 24x speedup over the baseline.

The box filter results highlight a tradeoff between resource investment into generating intermediate proofs and into solv-

TABLE II: Industrial EC tool performance with and without intermediate proofs generated by the assistant. We report the baseline EC tool performance when solving the original EC problem. We also report the runtime of the e-graph assistant and the runtime of the EC tool when solving the problem with the intermediate proofs. The sum provides a total verification time for the assisted proof. The last column shows the speedup ratio achieved using the assistant. Runtimes are in seconds.

Benchmark	EC without assistance	Assistant	EC with assistance	Assisted Total	Speedup (without/with)
ADPCM Decoder	0.68	0.38	0.49	0.87	0.78
H-264 VBSME-4	7.93	7.04	0.71	7.75	1.02
H-264 VBSME-8	93.13	14.3	0.20	14.50	6.42
FIR Filter	5.50	3.49	0.79	4.28	1.29
Box Filter	79.56	16.10	1.61	17.71	4.49
Case Study	-	0.14	0.10	0.24	-

TABLE III: Summary of e-graph assistant properties across the benchmarks. We report the number of rewriting iterations, the e-graph size in terms of node count, the number of intermediate proofs generated, and whether the e-graph found a complete path of rewrites between S and I .

Benchmark	Num. Iter.	E-graph Nodes	Num. Proofs	Full Path
ADPCM	3	469	20	Y
VBSME-4	5	5640	26	Y
VBSME-8	5	5800	46	Y
FIR Filter	5	4700	23	Y
Box Filter	5	21400	115	N
Case Study	3	149	20	Y

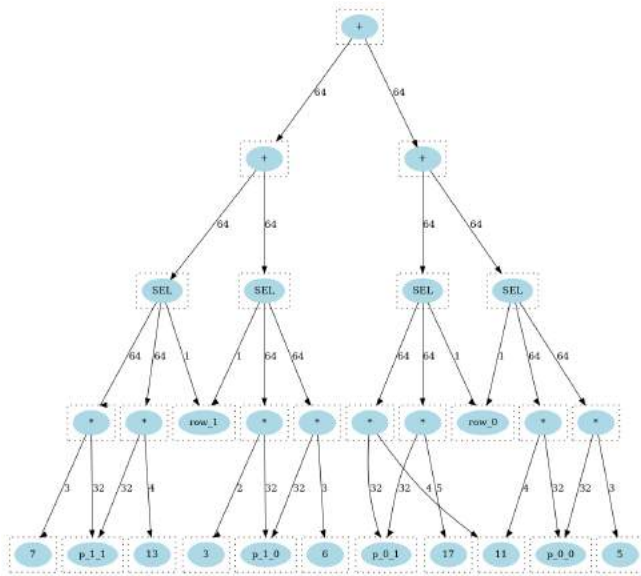


Fig. 9: Dataflow graph of the initial box filter design. The SEL nodes represent muxes.

ing these proofs. Understanding the turning point would allow the assistant to automatically identify which intermediates will be most beneficial, a task beyond the current tool.

In addition to the solvers described so far, we also investigated the open-source SymbiYosys equivalence checker [33]. However, for the instances we tried, we were not able to solve any equivalence problems since the tool is SAT/SMT based and does not handle datapath problems efficiently. A key advantage of a rewrite based approach is that performance is not affected by bitwidths, whilst SAT-based solvers will suffer

from exponential slowdowns as the bitwidths are increased. This approach is promising because we do not typically need the full power of a SAT or SMT solver on the entire design, meaning that a specialized tool that does not target notions of completeness can prove valuable.

VI. CONCLUSION

This paper applies recent advances in e-graph rewriting to datapath equivalence checking to develop an automated formal verification assistant that enhances the capabilities of industrial tools. By incorporating both the specification and implementation into a single data structure, the assistant simultaneously rewrites both designs to efficiently identify common equivalent sub-expressions. From the e-graph, the tool extracts a sequence of intermediate designs, breaking the complete equivalence check into a sequence of smaller sub-proofs which can be proven by *trusted* tools. In cases where the assistant is unable to identify a complete path between the specification and the implementation designs, the e-graph rewriting may still reduce the equivalence checking to a simpler sub-problem. This enables FV engineers to focus on the challenging core of the verification task and helps the EC tool to identify additional internal equivalence pairs automatically, reducing the complexity of the overall equivalence check.






The assistant developed through this work is able to find a complete sequence of intermediate designs, enabling a commercial EC tool to prove equivalence in under a second on a problem that was previously beyond its capabilities. We also demonstrated test cases where the verification assistant was able to reduce verification runtimes by up to 6x.

Future work will primarily investigate integration of the techniques presented in this paper into complete solvers to improve the rewrite engines in such tools. We will also explore different front-ends to enable C to RTL equivalence checking and will incorporate registers to facilitate equivalence checking across multiple cycles. Exploring alternative applications, such as the gate-level multiplier verification challenge discussed in the background, would highlight the generality of the approach. Lastly, there are many performance optimizations that we will make to the assistant. For example, having discovered shared classes in the e-graph, we could freeze these sub-graphs to limit e-graph growth. Such optimizations and better orchestration would allow us to extend the evaluation to larger inconclusive problems requiring deeper e-graph exploration.

REFERENCES

- [1] V. Pratt, “Anatomy of the Pentium bug,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 915, 1995.
- [2] A. Darbari, “Smart Formal for Scalable Verification,” in *Design and Verification Conference and Exhibition (DVCon) United States*, 2019.
- [3] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, “Solver technology for system-level to RTL equivalence checking,” in *Proceedings -Design, Automation and Test in Europe, DATE*, 2009.
- [4] B. Xue, P. Chatterjee, and S. K. Shukla, “Simplification of C-RTL equivalent checking for fused multiply add unit using intermediate models,” in *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, 2013.
- [5] E. Morini and S. Elliott, “Formal verification of integrated circuit hardware designs to implement integer division,” 2019.
- [6] L. De Moura and N. Björner, “Z3: An efficient SMT Solver,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4963 LNCS. Springer, 2008.
- [7] S. Coward, G. A. Constantinides, and T. Drane, “Automatic Datapath Optimization using E-Graphs,” in *IEEE 29th Symposium on Computer Arithmetic (ARITH)*. IEEE, 9 2022, pp. 43–50.
- [8] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: A new approach to optimization,” in *ACM SIGPLAN Notices*, vol. 44, no. 1. Association for Computing Machinery, 2009.
- [9] M. Stepp, R. Tate, and S. Lerner, “Equality-based translation validator for LLVM,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6806 LNCS, 2011.
- [10] W. A. Hunt, M. Kaufmann, J. S. Moore, and A. Slobodova, “Industrial hardware and software verification with ACL2,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, 2017.
- [11] J. S. Moore, T. W. Lynch, and M. Kaufmann, “A mechanically checked proof of the AMD5K86™ floating-point division program,” *IEEE Transactions on Computers*, vol. 47, no. 9, 1998.
- [12] D. M. Russinoff, “A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division and Square Root Algorithms of the AMD-K7™ Processor,” *LMS Journal of Computation and Mathematics*, vol. 1, 1998.
- [13] J. Harrison, “A machine-checked theory of floating point arithmetic,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1690, 1999.
- [14] —, “Handbook of practical logic and automated reasoning,” *Choice Reviews Online*, vol. 47, no. 06, 2010.
- [15] C. J. H. Seger and R. E. Bryant, “Formal verification by symbolic evaluation of partially-ordered trajectories,” *Formal Methods in System Design*, vol. 6, no. 2, 1995.
- [16] T. Drane and H. Jain, “Formal Verification and Validation of High-Level Optimizations of Arithmetic Datapath Blocks,” in *SNUG*, 2011.
- [17] Synopsys, “Design Compiler User Guide S-2021.06-SP2,” Synopsys, Mountain View, Tech. Rep., 6 2021.
- [18] M. Temel and W. A. Hunt, “Sound and Automated Verification of Real-World RTL Multipliers,” in *Proceedings of the 21st Formal Methods in Computer-Aided Design, FMCAD 2021*, 2021.
- [19] M. Temel, A. Slobodova, and W. A. Hunt, “Automated and Scalable Verification of Integer Multipliers,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12224 LNCS, 2020.
- [20] M. Kaufmann and J. S. Moore, “ACL2: An industrial strength version of Nqthm,” in *COMPASS - Proceedings of the Annual Conference on Computer Assurance*, 1996.
- [21] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panckekha, “Egg: Fast and extensible equality saturation,” in *Proceedings of the ACM on Principles of Programming Languages*, vol. 5, no. POPL, 2021.
- [22] R. Joshi, G. Nelson, and K. Randall, “Denali: A goal-directed super-optimizer,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2002.
- [23] C. G. Nelson, “Techniques for program verification,” Ph.D. dissertation, Stanford University, 1980.
- [24] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, “Rewrite rule inference using equality saturation,” in *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, 2021.
- [25] G. Steele, *Common LISP: the language*. Elsevier, 1990.
- [26] D. Thomas and P. Moorby, *The Verilog® hardware description language*. Springer Science & Business Media, 2008.
- [27] M. Popoloski, “Slang,” 2023. [Online]. Available: <https://github.com/MikePopoloski/slang>
- [28] S. Coward, G. A. Constantinides, and T. Drane, “Automating Constraint-Aware Datapath Optimization using E-Graphs,” 2023. [Online]. Available: <https://arxiv.org/abs/2303.01839>
- [29] —, “Abstract Interpretation on E-Graphs,” 3 2022. [Online]. Available: <https://arxiv.org/abs/2203.09191>
- [30] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu, “SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra,” *Proceedings of the VLDB Endowment*, vol. 13, no. 11, 2020.
- [31] O. Flatt, S. Coward, M. Willsey, Z. Tatlock, and P. Panckekha, “Small Proofs from Congruence Closure,” in *Formal Methods in Computer-Aided Design*, 9 2022.
- [32] A. K. Verma, P. Brisk, and P. Ienne, “Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1761–1774, 2008.
- [33] YosysHQ GmbH, “SymbiYosys.” [Online]. Available: <https://symbi Yosys.readthedocs.io/en/latest/index.html>

μARCHIFI: Formal Modeling and Verification Strategies for Microarchitectural Fault Injections

Simon Tollec^{*}, Mihail Asavae^{*}, Damien Couroussé[§], Karine Heydemann^{¶‡} and Mathieu Jan^{*}

^{*}Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France – firstname.lastname@cea.fr

[§]Univ. Grenoble Alpes, CEA, List, F-38000, Grenoble, France – firstname.lastname@cea.fr

[¶]Thales DIS, France – firstname.lastname@thalesgroup.com

[‡]Sorbonne Univ., CNRS, LIP6, F-75005, Paris, France

Abstract—This paper introduces μARCHIFI, an open-source tool dedicated to the formal modeling and verification of microarchitecture-level fault injections and their effects on complex hardware/software systems. First, we address the problem of the system modeling, and our implementation is integrated into the Yosys toolchain. Second, we introduce verification strategies to evaluate the fault effects for software-level security. We demonstrated the practical use of μARCHIFI on RISC-V use cases using state-of-the-art model-checking tools for hardware verification.

Index Terms—Faulty transition system ; Bounded model checking ; SW/HW co-verification ; Security

I. INTRODUCTION

Context. Fault Injection (FI) attacks aim at applying abnormal execution conditions to an embedded system, such as high temperature or electromagnetic radiation. These disturbances induce computational errors in the system, leading to undesired behaviors. From a security point of view, these FI attacks can create vulnerabilities, such as the ability of an attacker to retrieve sensitive data or to acquire execution privileges on the platform. Consequently, there is a growing desire to study fault injections and better understand their effects to analyze the system's security or develop countermeasures.

Different abstraction layers are involved in describing faults in a system [1]. Faults initially appear in the circuit, and representing faults at this level permits to describe their initial effect, e.g., bit-flip, bit-reset. The consequences of the fault then propagate in the microarchitecture, can be captured by sequential logic, and induce a different behavior at the software level. An analysis at the hardware level, e.g., [2]–[4], can show that a module is functionally incorrect due to the perturbation induced by fault injections. Such approaches are sufficient for the robustness analysis of standalone components such as cryptographic IPs, but usually, the exploitation of a fault injection, in an attack, involves software. On the other side, a pure software analysis, e.g., [5]–[7], struggles to model many subtle behavioral effects induced by fault injection [8].

Recent research motivates the need to consider both the hardware and the software in the same analysis [8]–[10]. Laurent et al. show how faults on the forwarding mechanism permit to retrieve a previously computed value and reintroduces it in one of the processor pipeline stages [11]. Tollec et al. show how faults on the prefetch buffer can result in various

software-level consequences [10], such as: immediate replay of instructions that are alive in the prefetch buffer; execution of instructions in incorrect order; and corruption of the next branch target. However, such effects, induced by faults in the processor microarchitecture, can only be leveraged in an attack by specific software conditions, in particular the sequence of program instructions executed, such that the fault effects propagate until the attack target is reached.

Problem Statement. There is a need for modeling and analysis methods to better understand fault effects while considering the software and the hardware together. Such techniques can help to highlight microarchitectural implementation details impacting the system's security. We need to automatically build a model that encompasses both hardware and software implementation details, and fault effects. Such a model needs to be tractable by verification tools in order to leverage automated verification techniques.

Proposal and contributions. We propose a complete workflow for the formal analysis of a full system composed of hardware and software components under fault injection. Faults are modeled at the microarchitectural level to accurately analyze the impact of a fault injection at the hardware level and their effects at the software level. We leverage bounded model checking in order to reason about the impact of fault injections on the system and their possible exploitation by an attacker. This work is a follow-up of [10], and we bring improvements in two directions. First, we formalize the model-checking problem with a transition system including the attacker model. We describe its implementation in an end-to-end formal analysis workflow, named μARCHIFI, based on Yosys [12] and a third-party model checker. μARCHIFI generates the system's formal model from the RTL implementation of the hardware, an input binary program, and the attacker model to analyze the system's robustness under fault injection. The attacker model supports various fault-injection models. Second, we discuss practical strategies to improve the efficiency of the workflow, leveraging well-known optimization techniques from formal methods. We illustrate the use of μARCHIFI on several case studies and evaluate the impact of the proposed strategies. μARCHIFI is open-source and will be publicly available on the GitHub of Yosys¹.

¹Currently available on <https://zenodo.org/record/7958412>

Paper outline. Section II introduces microarchitectural fault injection models and hardware transition systems. Section III describes the faulty transition system we introduce to analyze microarchitectural fault consequences at the software level. Section IV introduces our verification problem. The formal model is then automatically generated by the tool detailed in Section V. Section VI evaluates our approach in three case studies. Section VII discusses our contributions wrt. related work, and Section VIII concludes.

II. BACKGROUND

This section introduces microarchitectural fault models, provides definitions on hardware transition systems modeling, and describes the Yosys framework that can translate a hardware description to formal models.

A. Microarchitectural Faults

Fault Injection (FI) attacks are a powerful threat against embedded systems that cover various physical injection means like clock glitches, electromagnetic pulses, or laser fault injections. In [13], Brockmann et al. propose a unified fault injection model that describes fault effects in the microarchitecture. The authors represent the synchronous digital circuit as a directed graph (V, E) composed of vertices V and edges E . Vertices V represent logic gates, state-holding elements, inputs, and outputs in the circuit, while edges E represent circuit wires connecting two vertices and carrying a digital value. Each vertex representing a logic gate in the graph is associated with a Boolean function describing the gate behavior. By definition, a *fault* occurs in the circuit when a given logic gate is not evaluated with its correct Boolean function.

Microarchitectural faults in the graph are parametrized with the three following attributes: *location*, *effect*, and *number*. The *location* lists the vertices $v \in V$ targeted by the fault injection. The *effect* specifies the fault effect by associating a faulty Boolean function with the targeted vertices. Finally, the *number* describes the maximum number of vertices simultaneously affected by a fault.

B. Transition System for Sequential Hardware Circuits

We model sequential hardware circuit [14, §2.1.2] as a transition system $\mathcal{M} = (S, S_0, X, T)$ where:

- S is the set of circuit states,
- $S_0 \subseteq S$ is the set of initial states,
- X is the set of circuit inputs,
- $T: S \times X \rightarrow S$ is the transition function of the circuit.

A *system state* $s \in S$ corresponds to a valuation of state-holding elements (e.g., microarchitectural registers, memories) in the hardware design. Assuming there are n state-holding elements, denoted as *registers* in the following, the state s can be seen as a vector of the n register values $s := \langle r_1, \dots, r_n \rangle$.

An *initial state* $s_0 \in S_0$ is a system state where each register $r_{j, 1 \leq j \leq n}$ is evaluated with its initial values. Initial states can be determined according to register reset values, for

instance. Uninitialized registers imply multiple initial states in the system.

An *input* $x \in X$ is a vector $x := \langle i_1, \dots, i_m \rangle$ composed with a valuation of the m input variables given to the system. Circuit outputs are not considered in the formalization as they are not useful for the rest of this work.

The *transition function* T describes the valid transitions from a state s_i and an input vector x_i to the state s_{i+1} at the next circuit clock cycle. The function T is determined with the combinational logic of the hardware design and can be decomposed in n *register transition functions* δ_{r_j} where each register next state value is computed by applying δ_{r_j} to the current state and the input vector.

$$s_{i+1} = T(s_i, x_i) = \langle \delta_{r_1}(s_i, x_i), \dots, \delta_{r_n}(s_i, x_i) \rangle$$

In hardware circuits, intermediate combinational results are often factorized to minimize the number of operations. This optimization avoids duplicating identical operations to reduce hardware costs. We denote *combinational functions* these intermediate results. Consequently, the register transition functions δ_{r_j} can be expressed as a composition of these intermediate combinational functions.

C. Yosys Framework

Yosys [12] is an open-source synthesis tool with a compiler-like infrastructure. Its frontend takes as input a design description using a hardware description language like Verilog. The Yosys intermediate language, called RTLIL, is a netlist composed of gates and wires. The backend converts the RTLIL design into various outputs ranging from technological targets like FPGA or ASIC to formal languages.

In particular, Yosys can transform a hardware design description into a hardware transition system. Supported formal languages are AIGER, SMV, BTOR2 and SMT-LIB. AIGER [15] describes hardware systems at the bit level using an and-inverter graph. The SMV language [16] is provided by the symbolic model checker NUXMV [17] and describes finite and infinite transition systems. NUXMV lifts the verification from the bit level to the word level with data types like bit vectors or memories. BTOR2 [18] is a word-level generalization of AIGER and provides word-level data types, registers, and memories. Finally, the SMT-LIB language [19] is the standard specification to describe SMT problems and is broadly supported by SMT solvers. SMT-LIB can also specify hardware transition systems using the quantifier-free bit-vector theory. Yosys can produce VCD waveform traces of the successive hardware states from model-checker outputs.

Finally, Yosys has built-in options to simulate the design and set the register's initial values. These functionalities allow a user to configure the initial state of the circuit before converting it into a hardware transition system \mathcal{M} .

III. FAULTY SYSTEM MODELING

Hardware analysis often relies on equivalence-checking techniques [4], [20] to capture faults that induce a different circuit behavior compared to a reference model. But these

methods can only classify fault effects at the circuit level and cannot determine whether the faults have consequences on the running software. To analyze the consequences of a fault described at the microarchitectural level on the software, we need to observe its propagation in the system. For this purpose, in the following, we perform model checking to capture the successive system states between the fault injection and the fault manifestation. This section defines a faulty transition system comprising the program, the hardware, and the attacker model.

A. Bringing the Software and the Hardware Together

The hardware processor design is modeled as a transition system $\mathcal{M} = (S, S_0, X, T)$, as introduced in Section II. The software program is the sequence of instructions to be executed on the processor. The program is encoded in the initial state of a memory modeled simultaneously with the processor. Accordingly, the initial state S_0 of the system restricts the possible processor executions to the software program under study. The input set X of the system does not represent the program as it is already encoded in the transition system. Instead, system inputs are used to model the fault injections applied during the processor operation. The attacker model and the faulty transition system are introduced in the following of this section.

B. Attacker Model

We define an *attacker model* which specifies how the attacker can perturb the system operation. This model relies on the definition of microarchitectural faults introduced in Section II-A and extends this definition to describe the attacker's capabilities on a hardware transition system. The attacker model comprises *i*) the attacker's goal expressed as a reachability property φ , *ii*) the number of faults N that the attacker can inject into the system, and *iii*) the fault model. The *fault model* is parametrized by the triplet $(\mathcal{L}, \mathcal{T}, \mathcal{E})$ and describes the possible modifications that a fault may induce on the system.

- \mathcal{L} is the set of possible locations of the fault,
- \mathcal{T} is the timing range of the fault injection,
- \mathcal{E} is the set of possible effects of the fault.

The *fault location* \mathcal{L} is a set that denotes the registers targeted by the fault injection, i.e., $L \subseteq \{r_1, \dots, r_n\}$.

The *timing range* $\mathcal{T} \subset \mathbb{N}$ of the fault is a set of non-negative integers that specifies when the fault injection can occur in the system. For example, a fault can be injected in the transition system between states s_i and s_{i+1} if $i \in \mathcal{T}$.

The *fault effect* $\mathcal{E} \subset \{set, reset, flips, \dots\}$ is a set of functions that modifies how a register is updated in the next state. For instance, a fault $e \in \mathcal{E}$ injected in register r_j consists in replacing the transition function δ_{r_j} with the faulty transition $\delta_{r_j}^e$ within the same domain, i.e., it produces the same output data type. A non-exhaustive list of possible effects is given below for an 8-bit register:

$$\begin{array}{l} e \in \mathcal{E} : \delta_{r_j}(s) \mapsto \delta_{r_j}^e(s) \\ \hline reset : \delta_{r_j}(s) \mapsto 0x00 \\ set : \delta_{r_j}(s) \mapsto 0xff \\ flips : \delta_{r_j}(s) \mapsto \neg \delta_{r_j}(s) \\ flip_{lsb} : \delta_{r_j}(s) \mapsto \delta_{r_j}(s) \oplus 0x01 \end{array}$$

The *number of fault injections* N restricts the possibilities offered by the fault model $(\mathcal{L}, \mathcal{T}, \mathcal{E})$. An attacker can use at most N faulty transitions δ_r^e to compute the next system states.

The *attacker's goal* φ is a reachability property defined on the transition system \mathcal{M} . It represents a vulnerability that the attacker wants to reach in order to create an exploit on the system by injecting faults. In the system's normal operation, such an exploit should not exist. In other words, $\neg \varphi$ is a system's invariant that the attacker wants to break.

Let us illustrate some practical instantiations of the attacker model we defined. Laser fault injections are accurate in space and time and can be modeled with only one or two bit-flip. On the other hand, voltage or clock glitches are less accurate and can affect the whole design. We may model them with multiple bit-set and bit-reset.

C. Faulty Hardware Transition System

The *faulty transition system* $\mathcal{M}^F = (S, S_0, X, T)$ results from the modification of the hardware and software transition system \mathcal{M} and the attacker model $((\mathcal{L}, \mathcal{T}, \mathcal{E}), N, \varphi)$. First, a new variable $cnt \in \llbracket 0, N \rrbracket$ is added in the system model to encode the maximum number of fault injections N . The cnt is incremented each time a faulty transition is applied and cannot be targeted by the fault model, i.e., $cnt \notin \mathcal{L}$. Then, for each targeted register r_l in \mathcal{L} , we add a new input x_l to the system to control the fault injection. The input x_l determines whether a fault is injected in register r_l , and hence, if the value of register r_l in the next state should be computed using the normal transition function δ_{r_l} or the faulty transition $\delta_{r_l}^e$.

$$r'_l = \begin{cases} \delta_{r_l}(s) & \text{if } x_l = \text{False} \\ \delta_{r_l}^e(s) & \text{if } x_l = \text{True} \end{cases}$$

A possible extension of this faulty transition system is to expose intermediate *combinational functions* often used in hardware circuits, as introduced in Section II. We can then extend our fault model and the resulting faulty transition system to target these combinational functions with fault injection. This extension is not formalized here but is implemented in the $\mu\text{ARCHIFI}$ tool.

IV. TRANSITION SYSTEM VERIFICATION

This section introduces verification techniques on a faulty transition system. In addition, we describe how the knowledge of the running software can be leveraged to refine the transition system verification.

A. Verification Problem Statement

In Section III, we model the system under attack as a transition system $\mathcal{M}^F = (S, S_0, X, T)$. The set of initial states S_0 describes the possible software execution path to analyze, and the inputs X control the possibilities of the attacker to inject faults in the system. The verification problem is then a reachability property verification where an attacker wants to find a sequence of states $(s_0, s_1, \dots, s_k) \in S^{k+1}$ such that:

- s_0 is an initial state, i.e., $s_0 \in S_0$,
- transition between states s_i and s_{i+1} is valid, i.e., it exists an input $x_i \in X$ such that $s_{i+1} = T(s_i, x_i)$,
- the number of faults injected in the system does not exceed the attacker capacity, i.e., $\text{cnt} \leq N$ and,
- $\varphi(s_k)$ is true.

Such a path in the transition system allows an attacker to identify an instance of the fault model and a software execution trace that verifies the property φ .

Different strategies exist to iterate over the transition model to verify the property. *Unbounded verification techniques* [21]–[24] prove the property in the general case, but the data dependency and the transient nature of faults make these techniques ill-suited [25] on the fault injection problem. *Bounded verification techniques* like Bounded Model Checking (BMC) [26], [27] prove the property from an initial state for a limited number of transitions, fixed a priori with a bound. This *bound* is typically set according to the length of execution trace of the analyzed software. In this work, we rely on bounded verification techniques to address the fault verification problem.

The remainder of this section introduces software-related considerations for applying well-known optimization techniques to speed up our BMC verification.

B. Sandboxing Execution Paths

Sandboxing is a general technique that adds a global constraint on the model to reduce the problem's state space. We apply sandboxing to restrict the Program Counter (PC) to a range of values that a simple static analysis can retrieve from the addresses in the binary, e.g., using objdump-like tool. The verification framework then stops exploring software execution paths that do not satisfy this sandboxing constraint. Consequently, the BMC procedure can also terminate faster when the entire state space has been explored. However, adding such a global constraint on the model may lose the k -completeness of the bounded verification procedure. This technique must therefore be used to explore possible vulnerabilities rather than prove the system's robustness.

While only one PC exists at the software level, several microarchitectural registers store its value in the processor design. The fetch stage PC speculates on the next addresses to be read from memory. Applying the sandboxing technique on this register would thus require relaxing the sandboxing constraint. The execute stage PC misses unconditional branches that are resolved directly in the decode stage. We therefore implement sandboxing by constraining the PC of the decode stage of in-order processors, as presented later in Section VI.

Algorithm 1: Bounded model checking (BMC) with concretization

Input: transition system $\mathcal{M} = (S, S_0, X, T)$, reachability property φ , BMC bound k , concretization depth m , number of concretizations L

Output: a path π if a φ is reachable, *None* otherwise

```

1 Function BMC_Concretizing( $\mathcal{M}, \varphi, k, m, L$ ) is
   // BMC() checks the reachability of  $\varphi$  up to
   // the bound.  $\mathcal{M}, \varphi, \phi, i$  are global variables.
2   Function BMC( $bound$ ) is
3     while  $i < bound$  do
4       if  $\phi \wedge \varphi(s_i)$  is SAT then
5         exit( $\pi := (s_0, \dots, s_i)$ )
6        $i \leftarrow i + 1$ 
7        $\phi \leftarrow \phi \wedge (s_i = T(s_{i-1}, x_{i-1}))$  // Unrolling
8     end
9   end

   // Initial BMC verification up to step  $m$ 
10   $\phi \leftarrow S_0(s_0)$ ;  $i \leftarrow 0$ 
11  BMC( $m$ )
   // Concretization loop
12   $\psi \leftarrow \phi$ ;  $\text{set}_{PC} \leftarrow \emptyset$ ;  $\text{iter} \leftarrow 1$ 
13   $\text{incomplete\_enum} \leftarrow \text{False}$ 
14  while  $\psi$  is SAT do
15     $\text{address} \leftarrow \text{get\_model}(\psi)(PC)$ 
16     $\text{set}_{PC} \leftarrow \text{set}_{PC} \cup \text{address}$ 
17     $\psi \leftarrow \psi \wedge (PC \neq \text{address})$ 
18    if  $\text{iter} = L$  then
19       $\text{incomplete\_enum} \leftarrow \text{True}$ ; Break
20     $\text{iter} \leftarrow \text{iter} + 1$ 
21  end
   // Parallel BMC verifications for each
   // concretized path up to bound  $k$ 
22  for  $\text{address} \in \text{set}_{PC}$  do
23     $\phi \leftarrow \phi \wedge (PC = \text{address})$ 
24    BMC( $k$ ) // run BMC on  $\phi$  (concretized paths)
25  end
26  if  $\text{incomplete\_enum}$  then
27     $\phi \leftarrow \phi \wedge (PC \notin \text{set}_{PC})$ 
28    BMC( $k$ ) // run BMC on  $\phi$  (remaining paths)
29 end

```

C. Concretizing Execution Paths

BMC algorithms [27] typically unroll the transition system in a formula ϕ to check the reachability of a property φ . As a result, solving the formula ϕ suffers from the increasing number of variables and clauses.

We apply the general *concretizing* technique to split ϕ into sub-formulas encoding the different software execution paths. Like *sandboxing*, we rely on the PC to distinguish between these different execution paths. However, a given PC value can refer to several microarchitectural contexts if more than one execution path can reach this address at the same time. This aspect is discussed at the end of this section.

The concretization procedure is detailed in Algorithm 1. After initializing the formula ϕ with an initial state s_0 and performing BMC up to bound m (lines 10-11), a concretization loop enumerates the possible values for the PC (lines 12-21). This loop successively asks an SMT solver to give models

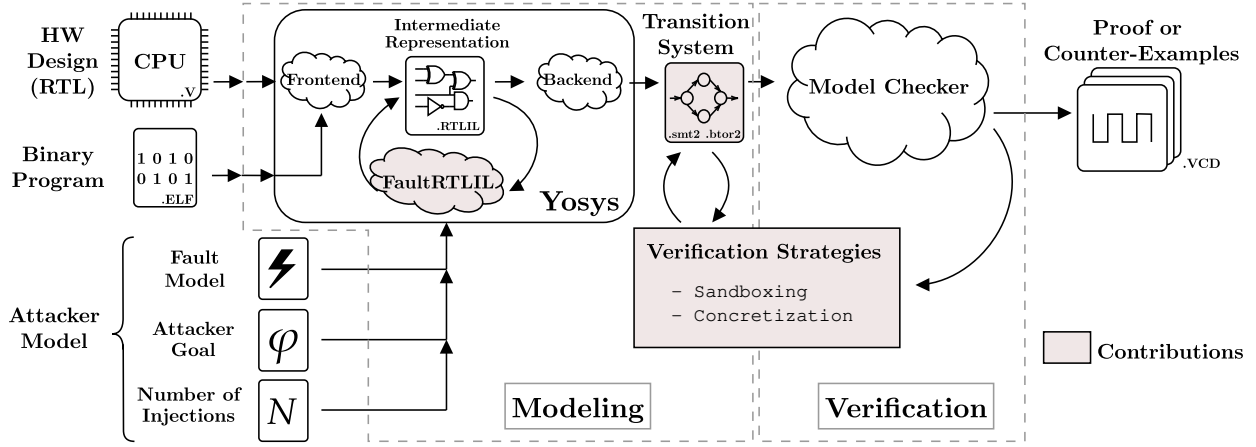


Fig. 1: μ ARCHIFI architecture and verification tool-chain.

of the system with different PC values. It stops when no more system model exists, i.e., ψ becomes unsatisfiable, or after a given number of concretizations L (lines 18-19). A new BMC procedure is performed for each enumerated PC value until bound k (lines 22-25). When the PC enumeration is incomplete, the remaining paths are encoded within a single formula and checked together (lines 26-28). A program analysis can identify branches' locations in the program and determine the optimal depth m the user should perform the concretization.

D. Discussion

Both the *sandboxing* or the *concretization* techniques reduce the state space to explore by adding terms and clauses to the formula encoding the problem. However, this general approach has some limitations. To speed up verification, a trade-off must be found between eliminating execution paths and the number of additional state variables that increase the complexity of the formula to check. For example, we might want to add constraints ensuring that injected faults lead to a different system behavior than the fault-free reference model. This technique allows to focus on analyzing effective faults while ineffective ones are ignored. However, it requires capturing the complete microarchitectural state twice, leading to an excessively complex system encoding formula. The verification times thus increase.

V. TOOL IMPLEMENTATION

This section introduces the μ ARCHIFI tool we developed to generate a formal transition system from a hardware description, a program, and an attacker model. First, we detail how a user can use the tool, then we give its implementation details.

A. μ ARCHIFI Usage

μ ARCHIFI, illustrated in Fig. 1, takes as input a processor hardware description in Verilog, a binary software program, and an attacker model comprising the fault model. First, the user can simulate the execution of the target program, compiled for the corresponding ISA, on the hardware design to

set the initial state of the hardware right before the instruction sequence to analyze formally. Then, the user needs to specify the attacker model comprising the goal φ , the maximal number of faults N , and the fault model (location, timing, and fault effects). This model is automatically integrated into the system through the `FaultRTLIL` pass. The attacker's goal can also be specified into the hardware design using the SystemVerilog Assertion subset supported by Yosys. Finally, the μ ARCHIFI tool produces a transition system, as introduced in Section III-B, in SMT-LIB or BTOR2 format. The faulty transition system can be verified using external model-checking tools compatible with these input formats, like AVR [25], PONO [28] or BTORMC [18].

When an external model checker finds a counterexample, as illustrated in the *verification* box in Fig. 1, a VCD file reports precisely where the fault is injected and when the attacker's goal is reached. However, understanding the propagation of faults and their consequences requires human expertise, but this task can be facilitated by external tools that perform differential traces comparison against a reference model.

Additional global constraints for the *sandboxing* technique can be specified to the model checker or can be directly included in the input Verilog design parsed by Yosys. The *concretization* technique requires an external model checker to enumerate possible execution paths and is thus not integrated into the μ ARCHIFI tool.

B. μ ARCHIFI Architecture

Fig. 1, in the *modeling* box, illustrates the integration of the μ ARCHIFI tool with the Yosys framework. Yosys can parse and translate a hardware specification into formal languages, allowing us to focus on the automated integration of an attacker model into the system. We work on the RTLIL intermediate representation of Yosys to get the best expressivity and exhaustiveness to specify the fault model. Besides, as the Yosys RTLIL translation preserves all signal and register names from the Verilog processor design, the user can accurately select fault locations based on name pattern matching, cell type, or cell-width filtering. In addition, the

TABLE I: Use Cases description and expected verification result.

	Hardware Design			Software Program		Attacker Model				BMC Results		
	Name	Logic Gates	Flip-Flops	Name	gcc Flag	Attacker goal φ	Location	Timing	Effect	N	k	Reachability
Use Case I	CV32E40P	2842	179	VerifyPIN_V7	Og	Bypass authentication	Flip-Flops in Control path	60:75	Symbolic	1	75	φ reachable
Use Case II	Secure Ibex	4422	211	VerifyPIN_V1	Os	Bypass authentication	Flip-Flops in Lockstep	*	Symbolic	5	46	φ unreachable
Use Case III	Ibex	1983	114	KeySchedule (AES)	Os	Set expanded key to 0	Combinational in EX Stage	*	Reset	2	38	φ unreachable

preservation of names facilitates the generation of comprehensive counterexamples.

Our work extends the Yosys tool by proposing a `FaultRTLIL` translation pass, illustrated in Fig. 1, that takes the attacker’s model $(\mathcal{L}, \mathcal{T}, \mathcal{E}), N, \varphi$ in input and integrates it into the system. This integration is achieved in several steps. First, a new RTLIL register is created to encode the maximum number of fault injections N the attacker can inject into the system. Then, a clock is added to control the timing range of the fault injection \mathcal{T} . Finally, additional logic functions are inserted into the intermediate representation for each location $l \in \mathcal{L}$ potentially targeted by a fault injection, modeling the possible fault effects \mathcal{E} . The targeted elements are then replaced by an *if-then-else* structure controlled by a *fault selector*. Fault selectors are exposed as system inputs and indicate whether the fault should be injected or not. The maximum counter value N and the clock for fault injection timing \mathcal{T} , introduced previously, are used to constrain the fault selectors.

VI. EVALUATION

This section illustrates the use of μ ARCHIFI in three case studies, applies the verification strategies introduced in Section IV, and discusses the tool’s limitations. The μ ARCHIFI implementation, the case studies, and the experimental results are publicly available².

All verifications have been executed on an 11th Gen Intel(R) Core(TM) i7-1185G7 CPU platform. Every program presented in this section is compiled with the RISC-V toolchain for the RV32IMC architecture (gcc version 10.2.0). For each verification, the BMC bound k is fixed according to the longest program execution trace plus a 10-percent increment to capture possible modifications in the control flow.

A. Use Case I: Robust Software

Use Case I illustrates the possibility for a user to analyze the robustness of a secure program running on a processor.

Software. We consider a `memcmp`-like authentication mechanism from the FISSC benchmark suite [29]. This collection provides eight versions of the VerifyPIN program embedding software countermeasures against fault injections. The VerifyPIN program compares two 3-digit³ PIN codes stored in memory: a user and a secret PIN. The user can authenticate when the two codes are identical. In the following, PIN values are symbolic, but we assume that the user PIN and the secret PIN are different in each of their digits. In

Use Case I, we target VerifyPIN_v7 with the most software countermeasures. It implements hardened booleans, constant iteration, loop counter check, inline PIN comparison, and duplication of critical tests. VerifyPIN_V7 is compiled with the optimization flag `Og` to prevent the compiler from removing the countermeasure. The program runs in constant time, in 69 clock cycles.

Hardware. We execute the program on the 32-bit, in-order, 4-stage pipeline CV32E40P RISC-V core from the OpenHW group [30]. The version under study does not provide any security countermeasures.

Attacker Model. In this system, the attacker aims to bypass the secure authentication mechanism without triggering the software countermeasures.

$$\varphi_1 := (\text{authenticated} \wedge \neg \text{software_alert})$$

The VerifyPIN_V7 program implements the authentication process in two steps. First, a constant-time loop sets a Boolean to True if a difference is detected between the two PINs. Second, a comparison is performed to test the Boolean value and allow the authentication. We evaluate the robustness of the second comparison block against a single fault injected on the sequential logic of the processor control path. The considered fault model targets 102 registers among 179 in the processor. Use Case I is summarized in Table I.

Verification Results. Table II compares verification performance between three model checkers with and without faults. Performing the verification without fault ensures that the attacker goal φ does not hold outside of an attack. The analysis results in Table I highlight that the attacker can bypass the authentication by injecting a single fault. Counterexamples provided by the model checkers permit the user to find the exact location of the fault that leads to the vulnerability φ_1 . All solvers found the same fault model on this use case, but we can observe that PONO is faster to solve the model-checking problem.

B. Use Case II: Robust Hardware

Use Case II details how a user can determine whether a fault injected into a secure processor can induce a vulnerable behavior on the software without being detected by the hardware countermeasure.

Software. We consider VerifyPIN_V1, the baseline version of the VerifyPIN collection, without any countermeasure. As in Use Case I, the same constraint is applied to user- and secret-PIN, which are still symbolic. VerifyPIN_V1 is compiled with the optimization flag `Os`.

Hardware. The Ibex [31] is a parametrizable open-source 32-bit, in-order processor. We analyze the *small* version of the

²<https://doi.org/10.5281/zenodo.7958412>

³VerifyPIN uses 4-digit PINs in its original version.

core [32] in its *secure* configuration. The Secure Ibex implements protections against physical attacks like the redundancy-based Lockstep mechanism that instantiates the core twice and compares the outputs. The duplicated core is called the *Shadow Core* and an alert signal is triggered if an attack has been detected during the operation of the processor.

Attacker Model. In this second use case, the attacker still aims to bypass the secure authentication mechanism without triggering the hardware countermeasures.

$$\varphi_{II} := (\text{authenticated} \wedge \neg \text{hardware_alert})$$

The considered attacker model cannot inject more than five faults into the system. Fault Locations are limited to the sequential logic in the Shadow core since we do not want to inject the same fault in both cores.

Verification Results. Table I reports that an attacker cannot bypass the secure authentication with the considered fault model. This use case leverages the fact that the Secure Ibex implements hardware countermeasures. On the one hand, assuming that the *hardware_alert* cannot be triggered makes sense as the attacker wants to bypass the authentication without being detected. On the other hand, it helps the solver simplify the formula during the verification. Table II reports verification performance. BTORMC fails to solve the problem, and we stop the verification after 2 hours.

C. Use Case III: Cryptographic Software

Use Case III details how a user can apply the tool to software implementations of cryptographic algorithms.

Software. Tiny AES [33] is a small software implementation of the encryption algorithm. The key schedule function of the AES program expands the key into several separate keys for each round of AES. We focus here on a round of the key schedule function from the 128-bit AES. The program is compiled with the optimization flag `Os`.

Hardware. We run the key schedule function on the baseline version of the *small* Ibex core without any countermeasure.

Attacker Model. The attacker wants to set to zero a byte in the penultimate round key. An attacker can then use the observation of such an effect to perform differential fault analysis [34], [35]. Fault consequences are observed at the end of the key schedule function to limit the analysis to a small sequence of instruction.

$$\varphi_{III} := (9^{th} \text{ Round_key}_{\text{byte}} = 0)$$

To attempt to reach the property φ_{III} , we allow an attacker to inject up to two word-reset faults anywhere in the execute stage of the Ibex.

Verification Results. As reported in Table I, an attacker cannot reach his goal with the considered fault model. Additional verification not described here shows that a more powerful attacker reaches his goal with four fault injections instead of two. We can also note that the verification of φ_{III} on the AES program without fault is faster than both Use Case I and II because the AES key is fixed for while the two 3-digit PINs are symbolic for the VerifyPIN program.

TABLE II: Use-cases verification time with three model checkers.

	Without Fault			With Faults		
	PONO	YOSYS-BMC	BTORMC	PONO	YOSYS-BMC	BTORMC
Use Case I	12.6s	11.1s	1.5s	107s	249s	273s
Use Case II	20.7s	10.6s	3.5s	250s	373s	timeout
Use Case III	0.3s	2.4s	0.1s	313s	1945s	3427s

TABLE III: Verification time improvement with the sandboxing technique wrt. the baseline verification time with faults in Table II.

	PC Sandboxing	PONO	YOSYS-BMC	BTORMC
Use Case I	$0x1c4 \leq PC \leq 0x234$	110s (+2.8%)	242s (-2.8%)	205s (-24.9%)
Use Case II	$0x84 \leq PC \leq 0x114$	206s (-17.6%)	297s (-20.4%)	timeout
Use Case III	$0x40 \leq PC \leq 0xc0$	107s (-65.8%)	1454s (-25.2%)	1659s (-52.0%)

TABLE IV: Verification time improvement with the concretization technique wrt. the baseline verification time with faults from Table II.

	Concretized step	Baseline	Concretization	
			Parallelized	Accumulated
Use Case I	62 (Status comparison)	249s	189s (-24.1%)	509s (+104%)
Use Case II	31 (PIN comparison)	373s	304s (-18.5%)	891s (+139%)
Use Case III	23 (No branch instruction)	1945s	1504s (-22.7%)	2955s (+51%)

D. Influences of Verification Strategies

Sandboxing Execution Paths. For each use case introduced before, we determine the range of possible values for the program counter (PC) by dumping addresses from the binary file. Here, the possible addresses are contiguous, and we add a global constraint on the system to force the PC to stay in this set of values. Table III illustrates that the sandboxing strategy results in an improvement of the performances up to 65%, and these additional constraints do not prevent model checkers from retrieving the vulnerability highlighted in Use Case I

Such improvements are due to two factors. First, some fault effects are not analyzed if they lead to PC values out of the memory range considered. Secondly, the verification may end before the bound k if all execution paths in the system exit from the considered address range. We also observe that improvements vary between the different solvers even if PONO remains more efficient on the use cases analyzed.

Concretizing Execution Paths. We apply the concretization strategy for each use case with an enumeration bound $L = 3$ to split the bounded verification procedure into $L + 1$ sub-verifications (c.f., Algorithm 1). We arbitrarily set $L = 3$ as it provides the best performance in these practical use cases. A higher value of L increases the *accumulated* verification time without improving the *parallelized* time.

Table IV reports the concretization steps, the baseline verification time from Table II, and the concretization performance. We show each experiment's wall-clock time and accumulated verification time since we can parallelize the executions. Performance is given for the YOSYS-BMC since other evaluated model checkers do not permit to retrieve the SMT formula encoding the unrolled system.

On Use Case I, we concretize the execution at the first branching instruction targeted by fault injection. It corresponds to the PIN-status comparison to allow authentication (step 62).

This results in an improvement of the verification time by 24%. On Use Case II, we apply concretization during a PIN-digit comparison and enumerate PC values associated to different execution paths. However, few performance improvements are observed, especially regarding the accumulated verification time. We believe this is due to the hardware countermeasure that already prevents executing different paths due to the faults. No branching instruction exists on Use Case III. However, many execution paths are possible due to fault injections. Concretization is applied at step 23, at half of the verification time. This results in a 22.7% verification time improvement.

In conclusion, concretization often improves the verification time thanks to the parallelization of the executions. However, these verification times remain higher than the one obtained when using the PONO model checker (Table II).

VII. RELATED WORK

Similar works propose modeling and verification methodologies to study fault injection effects. Classical verification methods like simulation are used [3], [6], [36], [37], but they are often not exhaustive, and it is often difficult to highlight corner cases, like the Prefetch Buffer introduced in Section I. For instance, VERFI [3] needs to set a fixed input test vector to evaluate cryptographic implementation robustness to faults. In the following, we will discuss papers that propose a formal framework to analyze fault effects on the system.

First, some works analyze fault effects on hardware implementation [2], [4], [20]. Formal techniques were first dedicated to analyzing cryptographic circuits with equivalence checking. AutoFAULT tool [2] can parse and transform a small block cipher written in VHDL into a SAT formula to determine if a fault can induce a wrong ciphertext. The FIVER tool [4] translates Verilog netlists to Binary Decision Diagram to compare a fault-free circuit with a faulty copy to determine the fault effects. FIVER symbolically checks every possible input and classifies fault effects according to the expected reference behavior. Faults are classified as *effective*, *ineffective*, or *detected*, depending on whether they induce a different behavior and if the countermeasure (if any) detects them. SYNFI [20] can parse technological netlists to prove the equivalence between golden and faulty circuits to detect if the synthesis step removes countermeasures. However, SYNFI does not handle sequential verification since the design to analyze is unrolled to perform equivalence checking, and thus, the tool cannot analyze software. In comparison, μ ARCHIFI does not support advanced technological netlists, but we still support any Verilog or SystemVerilog design by plugging our translation pass into the Yosys tool. In addition, we take advantage of a simplified word-level netlist to bridge the gap with the software and facilitate the analysis of the transitional system. We also keep the sequential logic instead of unrolling and flattening the whole design to use model-checking verification techniques.

On the other hand, some additional works model and study faults at the software level and analyze fault effects on the control flow [5], [7], [38]–[40]. These approaches

address the binary or the Instruction Set Architecture level and propose methodologies to analyze the robustness of the software programs. SAMVA [40] assesses a binary program against multiple instruction-skip attacks with static analysis. The proposed method by Ducouso et al. [7] permits scaling on large programs like bootloader with up to 10 fault injections. However, these works do not consider the execution platform, and the generic fault models used are sometimes inadequate to model microarchitectural implementation details.

Furthermore, commercial tools offer all the building blocks required for such a fault injection analysis, but their closed nature prevents users from integrating them into the same verification framework. SystemVerilog Assertion (SVA), supported by tools such as Synopsys VC Formal or Siemens QuestaVerify, could define the attacker’s goal, but is not suitable for fault modeling. On the other hand, tools such as Cadence JasperGold offer support for fault injection but do not consider software. In short, none of these tools address the verification of software and hardware against fault injection.

Finally, apart from fault injection, some works [41], [42] tackle the problem of hardware-software co-verification using BMC verification. Schmidt et al. [42] propose to separate the control path and the computation in the modeling to cope with system complexity. However, this compositional approach is undermined when the underlying hardware is corrupted by fault injection since data and control are then both impacted.

VIII. CONCLUSION


In this paper, we propose a faulty transition system to model the hardware implementation of a processor and the software program conjointly. This modeling allows to formally analyze and study the propagation of faults in the microarchitecture and their consequences on the system behavior. The μ ARCHIFI tool automatically implements this model, from the hardware design description at the RTL level, in Verilog, a binary program, and a specification of the attacker model. μ ARCHIFI allows to specify a large variety of microarchitectural fault models with high expressiveness. We illustrate the use of μ ARCHIFI on three use cases encompassing complete microarchitectural designs of RISC-V processors representative of the embedded market and binary programs of hundreds of machine instructions. We discuss possible strategies to improve the verification performance. At this stage, the user of μ ARCHIFI must find a sweet spot between the size of the hardware design, the size of the analyzed program, and the complexity of the fault model. Future work will focus on combining several verification strategies leveraging software, such as sandboxing and concretization techniques, but also robust hardware embedding countermeasures to analyze fault injections on a larger scale.

REFERENCES


- [1] B. Yuce, P. Schaumont, and M. Witteman, "Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation," *Journal of Hardware and Systems Security*, Jun. 2018.
- [2] J. Burchard, M. Gay, A.-S. M. Ekosono, J. Horáček, B. Becker, T. Schubert, M. Kreuzer, and I. Polian, "AutoFault: Towards Automatic Construction of Algebraic Fault Attacks," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017.
- [3] V. Arribas, F. Wegener, A. Moradi, and S. Nikova, "Cryptographic Fault Diagnosis using VerFI," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Dec. 2020.
- [4] J. Richter-Brockmann, A. Rezaei Shahmirzadi, P. Sasdrich, A. Moradi, and T. Güneysu, "FIVER – Robust Verification of Countermeasures against Fault Injections," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Aug. 2021.
- [5] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections," in *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, Mar. 2014.
- [6] M. Hoffmann, F. Schellenberg, and C. Paar, "ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries," *IEEE Transactions on Information Forensics and Security*, vol. 16, 2021.
- [7] S. Ducousso, S. Bardin, and M.-L. Potet, "Adversarial Reachability for Program-level Security Analysis," in *32nd European Symposium on Programming (ESOP)*, 2023, pp. 59–89.
- [8] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle, "Bridging the Gap between RTL and Software Fault Injection," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 17, no. 3, May 2021.
- [9] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software Fault Resistance is Futile: Effective Single-Glitch Attacks," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Aug. 2016.
- [10] S. Tollec, M. Asavaoae, D. Couroussé, K. Heydemann, and M. Jan, "Exploration of Fault Effects on Formal RISC-V Microarchitecture Models," in *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, Sep. 2022.
- [11] J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula, "Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.
- [12] C. X. Wolf, "Yosys open synthesis suite," <https://github.com/YosysHQ/yosys>, 2016.
- [13] J. Richter-Brockmann, P. Sasdrich, and T. Güneysu, "Revisiting Fault Adversary Models – Hardware Faults in Theory and Practice," *IEEE Transactions on Computers*, 2022.
- [14] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, Apr. 2008.
- [15] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," 2011.
- [16] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2SMV: A Tool for Word-level Verification," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.
- [17] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv Symbolic Model Checker," in *Computer Aided Verification*, 2014.
- [18] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , BtorMC and Boolector 3.0," in *Computer Aided Verification*, 2018.
- [19] C. Barrett, P. Fontaine, and A. Stump, "The SMT-LIB Standard," 2010.
- [20] P. Nasahl, M. Osorio, P. Vogel, M. Schaffner, T. Trippel, D. Rizzo, and S. Mangard, "SYNFI: Pre-Silicon Fault Analysis of an Open-Source Secure Element," May 2022.
- [21] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Formal Methods in Computer-Aided Design*, 2000, pp. 127–144.
- [22] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Computer Aided Verification*, vol. 2725. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.
- [23] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *Verification, Model Checking, and Abstract Interpretation*, 2011.
- [24] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2011.
- [25] A. Goel and K. Sakallah, "AVR: Abstractly Verifying Reachability," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds., 2020.
- [26] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design*, 2001.
- [27] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer International Publishing, 2018.
- [28] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett, "Pono: A Flexible and Extensible SMT-Based Model Checker," in *Computer Aided Verification*, 2021, pp. 461–474.
- [29] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: A Fault Injection and Simulation Secure Collection," in *Computer Safety, Reliability, and Security*, 2016.
- [30] OpenHW group, "OpenHW Group CV32E40P User Manual," <https://cv32e40p.readthedocs.io/en/latest/>.
- [31] LowRISC, "Ibex: An embedded 32 bit RISC-V CPU core," <https://ibex-core.readthedocs.io/en/latest/>.
- [32] "Ibex RISC-V Core github repository," <https://github.com/lowRISC/ibex#configuration>.
- [33] kokke, "Tiny AES," <https://github.com/kokke/tiny-AES-c>, 2019.
- [34] J. Takahashi, T. Fukunaga, and K. Yamakoshi, "DFA Mechanism on the AES Key Schedule," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. Vienna, Austria: IEEE, Sep. 2007.
- [35] S. S. Ali and D. Mukhopadhyay, "A Differential Fault Analysis on AES Key Schedule Using Single Fault," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, Sep. 2011.
- [36] J. Grycel and P. Schaumont, "SimpliFI: Hardware Simulation of Embedded Software Fault Attacks," *Cryptography*, vol. 5, no. 2, Jun. 2021.
- [37] T. Given-Wilson, N. Jafri, and A. Legay, "Combined software and hardware fault injection vulnerability detection," *Innovations in Systems and Software Engineering*, vol. 16, no. 2, Jun. 2020.
- [38] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLIFIED: Symbolic program-level fault injection and error detection framework," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, Jun. 2008.
- [39] J.-B. Bréjon, K. Heydemann, E. Encrenaz, Q. Meunier, and S.-T. Vu, "Fault attack vulnerability assessment of binary code," in *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*. Valencia Spain: ACM, Jan. 2019.
- [40] A. Gicquel, D. Hardy, K. Heydemann, and E. Rohou, "SAMVA: Static Analysis for Multi-fault Attack Paths Determination," in *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2023, pp. 3–22.
- [41] D. Große, U. Kühne, and R. Drechsler, "HW/SW co-verification of embedded systems using bounded model checking," in *Proceedings of the 16th ACM Great Lakes Symposium on VLSI - GLSVLSI '06*. Philadelphia, PA, USA: ACM Press, 2006.
- [42] B. Schmidt, C. Villarraga, T. Fehmel, J. Bormann, M. Wedler, M. Nguyen, D. Stoffel, and W. Kunz, "A New Formal Verification Approach for Hardware-dependent Embedded System Software," *IPSI Transactions on System LSI Design Methodology*, pp. 135–145, 2013.

Sylvia: Countering the Path Explosion Problem in the Symbolic Execution of Hardware Designs

Kaki Ryan

 University of North Carolina
Chapel Hill, NC, USA
kakiryan@cs.unc.edu

Cynthia Sturton

 University of North Carolina
Chapel Hill, NC, USA
csturton@cs.unc.edu

Abstract—Symbolic execution is a powerful verification tool for hardware designs, in particular for security validation. However, symbolic execution suffers from the path explosion problem in which the number of paths to explore grows exponentially with the number of branches in the design. We introduce a new approach, *piecewise composition*, which leverages the modular structure of hardware to transfer the work of path exploration to SMT solvers. Piecewise composition works by recognizing that independent parts of a design can each be explored once, and the exploration reused. A hardware design with N independent `always` blocks and at most b branch points per block will require exploration of $O(2^{bN})$ paths in a single clock cycle with our approach compared to $O(2^{bN})$ paths using traditional symbolic execution.

We present *Sylvia*, a symbolic execution engine implementing piecewise composition. The engine operates directly over RTL without requiring translation to a netlist or software simulation. We evaluate our tool on multiple open-source SoC and CPU designs, including the OR1200 and PULPissimo RISC-V SoC. The piecewise composition technique reduces the number of paths explored by an order of magnitude and reduces the runtime by 97% compared to our baseline. Using 84 properties from the security literature we find assertion violations in open-source designs that traditional model checking and formal verification tools do not find.

Index Terms—symbolic execution, verilog, register transfer level, verification, formal methods, hardware security

I. INTRODUCTION

The verification of hardware designs is a key activity for ensuring the correctness and security of a design early in the hardware lifecycle. Current best practice includes assertion-based verification (ABV) [17], which has simulation-based testing as the underlying means of verification, and formal verification techniques, an umbrella term encompassing many techniques with the goal of proving a given property of a design. One technique that has gained recent attention, especially in security verification applications, is symbolic execution [8], [40], [44], [48]. In addition to finding property violations, symbolic execution has been used to verify information-flow properties [25] or to find hardware trojans [44].

Symbolic execution generalizes testing by replacing input values with symbols, where each symbol represents the set of possible values of the input parameter. A symbolic execution engine drives symbolic execution using the semantics of the program's language, but updated to include symbols. As execution proceeds the symbols are used in place of literal values. When a branch point, or control flow statement, is reached

(e.g., an `if` statement), both possible branches are explored separately. The result of symbolically executing a design for one clock cycle is a tree of paths, each one associated with a unique *path condition* that describes the conditions satisfied by branches taken along the path. Symbolic execution is often used to find assertion violations. If any path is found to violate a given assertion, then the associated path condition acts as a precise description of the inputs that will drive (concrete) execution along the same path. Concrete values that satisfy the path condition are a counter-example to the assertion.

Unfortunately, symbolic execution suffers from the path explosion problem: the number of paths grows exponentially with the number of branch points in the design. Prior work has sought to avoid the path explosion problem by combining symbolic execution with model checking [6], concrete execution traces [40], or by limiting the use to small designs [42].

We introduce *piecewise composition*, a technique that leverages the structure of hardware designs and the power of satisfiability modulo theories (SMT) solving to reduce the amount of repeated work. A single clock cycle of symbolic execution produces a full tree of paths, where the root of the tree is the initialized reset state and the leaves are realizable design states in the next clock cycle. The inspiration behind piecewise composition is the recognition that independent parts of the design are being re-explored unnecessarily in each root-to-leaf path. Instead, each independent block of logic can be explored once, without consideration of the other blocks. To reconstruct full root-to-leaf paths through the design, whether for finding assertion failures, describing how information flows through a design, or to generate testcases, the algorithm uses SMT queries to combine the independently explored path fragments.

Perhaps surprisingly, we show that with piecewise composition, a design with N `always` blocks, each with at most b binary branch points, symbolic execution for a single clock cycle requires exploring $O(2^{bN})$ paths, instead of the $O(2^{bN})$ paths typical of standard symbolic execution. The number of paths to explore grows exponentially with only the number of branch points in any one independent block, and linearly with the number of blocks.

Symbolic execution is closely related to symbolic simulation [2] [3] [6]. In both, concrete input values are replaced with symbolic values, representing any possible value, and the symbolic values are allowed to propagate through the design. How-

ever, in symbolic simulation, the analysis is centered around dataflow. At the end of a simulation run, each signal may hold the value true, false, or a boolean expression characterizing the entire circuit that drives that particular signal. Where there are control points in the circuit, they are expressed as *if-then-else* (ITE) statements in the boolean expression. In symbolic execution, the analysis is centered around control flow. At the end of one iteration of symbolic execution, each signal holds a symbolic expression in a subset of first-order logic that characterizes the particular path taken through the register-transfer level (RTL) code. In addition, there is a path condition that represents the conditions under which execution would follow the particular path through the design.

In comparing symbolic simulation and symbolic execution, there is a trade-off being made between the complexity of queries sent to the SMT solver (symbolic simulation) and the number of paths to explore (symbolic execution). With piecewise composition, we examine a new point in the design space, reducing the number of paths to explore to a tractable amount, while still keeping SMT queries simple enough for modern solvers. The result is a symbolic execution engine that can handle large designs and operate directly over the register-transfer level design. *Sylvia* targets the Verilog hardware description language (HDL), however the approaches and principles presented in this work are applicable to other HDLs.

This paper presents the following contributions:

- 1) Introduction and definition of *piecewise composition*, a technique that leverages the modular nature of hardware designs to counter the path explosion problem in symbolic execution;
- 2) Design and implementation of *Sylvia*, a symbolic execution engine for Verilog RTL using piecewise composition;
- 3) Evaluation of piecewise composition and our implementation on five open-source designs, including an SoC and two CPUs.

II. PRELIMINARIES

A. Example Verilog RTL Fragment

In the following discussion we will use the fragment of Verilog shown in Figure 1. In this example, *inpA* and *inpB* are input signals (along with *clk*), while all other named variables are state-holding *regs*. We use the set V to denote all design variables (*regs*, *wires*, etc.) other than *clk*. In Figure 1, $V = \{\text{inpA}, \text{inpB}, g0, g1, y, z\}$.

B. Symbolic Execution

In symbolic execution [33], concrete values are replaced with symbolic values. Each symbol represents an arbitrary, but fixed, value of appropriate type. As execution proceeds, the symbols are used in place of concrete values wherever they occur. Variables may take on concrete values ($\{0, 1\}^*$), symbolic values ($\alpha, \beta, \gamma, \dots$), or a symbolic expression (e), which is an expression in a quantifier-free subset of first-order logic that supports bitvector arithmetic, the standard Verilog operators and the theory of equality. The symbolic execution

```

1  always @ (posedge clk) begin
2      if (g0)
3          y <= inpA; //inpA is an input signal
4      else
5          y <= 0;
6      end
7
8  always @ (posedge clk) begin
9      if (g1)
10         z <= inpB; //inpB is an input signal
11     else
12         z <= 0;
13     end

```

Fig. 1: Verilog RTL fragment with two branches.

engine (or just *engine* from now on) implements the semantics of the RTL Verilog; there is no compilation down to a netlist.

We model the symbolic execution of a design as a transducer $SE = (RTL, \Sigma, \Pi, \sigma_0, \pi_0, \mathcal{E})$:

- RTL is the design, modeled as a partially ordered sequence of Verilog statements.
- $\Sigma \subset 2^{V \times E}$ is the set of symbolic stores. Each symbolic store $\sigma \in \Sigma$ is a function mapping program variables in V to symbolic expressions in E : $\sigma : v \rightarrow e$.
- Π is the set of path conditions. A path condition $\pi \in \Pi$ is a boolean formula in the same subset of first of first-order logic mentioned in the preceding paragraph. The path condition is composed of symbolic and concrete literals, which describe the conditions satisfied by branches taken along the current path.
- σ_0 is the initial symbolic store. All input variables other than the clock are initialized with fresh symbols.
- π_0 is the initial path condition, which is always initialized to $\pi_0 = \text{True}$.
- $\mathcal{E} \subseteq RTL \times \Sigma \times \Pi \times RTL \times \Sigma \times \Pi$ is the transition relation of the engine. Given the current RTL statement, symbolic store, and path condition the engine updates the symbolic store and path condition and moves to a next statement to execute (at branch points there is more than one to choose from).

To continue with our example, in the RTL fragment shown in Figure 1, the symbolic store σ would maintain the values of variables *inpA*, *inpB*, *g0*, *g1*, *y*, *z*. (In the following discussion, we write out only the part of the symbolic store that is relevant to the discussion.) Let the (partial) initial symbolic store and path condition be:

$$\sigma_0 = \{\text{inpA} = \alpha, g0 = \gamma\}, \quad \pi_0 = \text{True}$$

When a branching statement is reached (e.g., line 2 in Figure 1), the engine uses the current path condition to decide which path of execution to follow. If the engine has current path condition π and is at a branch statement with the boolean condition b , and if $\pi \rightarrow b$, the *then* branch is taken and the

path condition is updated: $\pi = \pi \wedge b$. Otherwise, if $\pi \rightarrow \neg b$, the `else` branch, if present, is taken and the path condition is updated: $\pi = \pi \wedge \neg b$. If neither implication holds, then both paths must be explored in turn. In our example, the engine will explore the two paths from line 1 to line 6, resulting in the following two symbolic stores and path conditions:

- 1) $\sigma_6 = \{\text{inpA} = \alpha, g0 = \gamma, y = \alpha\}$, $\pi_6 = \text{True} \wedge \gamma == 1$
- 2) $\sigma_6 = \{\text{inpA} = \alpha, g0 = \gamma, y = 0\}$, $\pi_6 = \text{True} \wedge \gamma == 0$

This example is simple, but in practice, path conditions quickly become complex, involving hundreds of terms and complex constraints in the theories necessary to express all Verilog operators.

At each branch point, the number of paths to explore doubles. This is the path explosion problem, and the result is that not all paths can feasibly be explored. Typically heuristics are used to guide the exploration toward paths that will maximize coverage or depth, or path-merging strategies are used to reduce the number of paths at the expense of less precise analysis [6], [18], [35].

C. Symbolic Execution Trees

A trace, τ , of symbolic execution is a sequence of symbolic store and path condition pairs, $\tau = \langle (\sigma_0, \pi_0), (\sigma_i, \pi_i), (\sigma_j, \pi_j), \dots, (\sigma_n, \pi_n) \rangle$, where the subscripts indicate the line of code associated with the symbolic state and path condition, and $0 < i < j < n$.¹ The complete symbolic execution of the RTL produces a tree of traces, T , as seen, for example, in Figure 2b. A path through the tree from the root node to a leaf node is a symbolic execution trace τ .

Each node (σ_i, π_i) in the tree is associated with a line of code in the RTL. More than one node in the tree will be associated with the same line of code. For example, in Figure 2b there are two distinct nodes associated with line 9, representing the two paths that can be taken to arrive at that line. These two nodes will necessarily have unique path conditions, the conjunction of which will be unsatisfiable.

D. Multiple Clock Cycles

The symbolic execution of a hardware design corresponds to a single clock cycle. Every path through the tree from the root node $n_r = (\sigma_0, \pi_0)$ to a leaf node $n_l = (\sigma_n, \pi_n)$ corresponds to a realizable step of the design from one state to the next. If n_r corresponds to a reachable state of the design (e.g., the reset state) that can be reached in k clock cycles, then n_l corresponds to a reachable state of the design that can be reached in $k+1$ clock cycles. The path condition associated with a leaf node n_l can be viewed as a predicate describing the (concrete) input values that would drive execution down the current path.

¹The engine considers the subset of Verilog that uses only statically bounded loops and unrolls loops before execution, so that the engine never executes line i after line j , $i < j$.

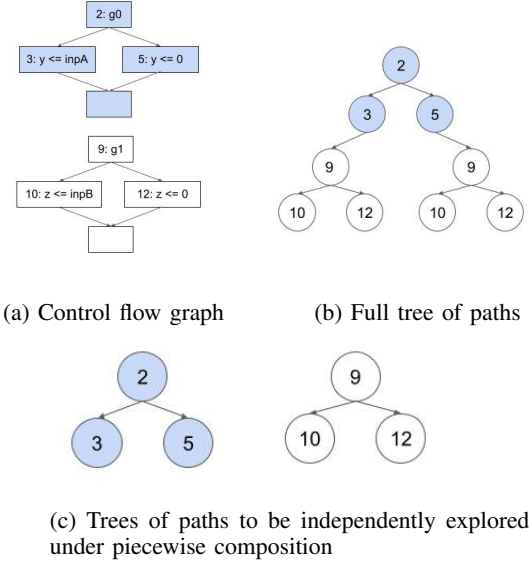


Fig. 2: Piecewise Composition

E. Comparison with Symbolic Simulation

Symbolic simulation is a well-established technique in hardware testing and verification [9], [10], [12], [13], [34], [37]. Conceptually, symbolic simulation and symbolic execution are closely related. In both, symbols are used in place of concrete values for input or state variables. The symbolic values propagate as execution proceeds, and variables in the design take on symbolic expressions as values. Both techniques generalize testing, symbolic execution in the software domain [33], and symbolic simulation in the hardware domain [43].

A key difference in the two techniques, however, is in how branch points are handled. In symbolic execution, execution proceeds separately down each path following the branch point, with the path condition π keeping track of the conditions associated with the current path of execution. In symbolic simulation, however, there is no notion of separate paths and there is no path condition. Instead, branch points are captured as conditional assignments to variables. For example, for the code in Figure 1, as before, the value of y at the end of symbolically simulating the code fragment would be²:

$$\begin{aligned} y &:= \text{ite}(\gamma, \alpha, 0) \\ &:= (\gamma \wedge \alpha) \vee (\neg \gamma \wedge 0) \\ &:= \gamma \wedge \alpha \end{aligned}$$

The value of each output variable is a symbolic expression capturing the complete dataflow path from the inputs. However, without the separation of paths, the symbolic expression for variables becomes complex; this complexity is the limiting factor for symbolic simulation and is managed by initializing control variables to concrete values, a reasonable constraint for many functional verification tasks (e.g., [31]). For example, $g0$ would be given a concrete value, rather than the symbolic γ .

²`ite` is the if-then-else operator.

In contrast, symbolic execution separates paths and uses the path condition to store the constraints along the current path. As a result, control signals can be made symbolic without adding complexity to the symbolic expressions for each variable. Symbolic control signals allow for verifying a series of control-flow dependent properties without modifying the verification environment. For example, in our evaluation, the same environment set-up was used to verify all properties of a given design: the design was started in its initialized state, all input variables were made symbolic, and the desired property was checked.

While the limiting metric for symbolic simulation is the complexity of each variable's symbolic expression, the limiting metric for symbolic execution is the number of paths to explore. In this paper we present a technique to reduce the number of explorations needed.

F. Comparison with Bounded Model Checking

In bounded model checking the initial state of the system and the transition relation of the system are formally defined in a logic system, typically a subset of first-order logic [14], [15]. The reachable states of the system are computed up to a bound and checked against a desired property. Techniques such as IC3 [26] can allow for unbounded proofs of a property.

Prior work has reported that symbolic execution is at times able to find security property violations that model checking does not (see Section VI). In addition, in recent years, the hardware security community has turned its attention to analyzing how information flows through a design [4], [27], [29]. Doing so requires reasoning about hyperproperties [16], which requires self-composition in model checking [24], adding to the complexity of the verification task.

Symbolic execution, on the other hand, is suited to information-flow analysis, as the symbolic state σ and path condition π provide precise tracking of information flow from reset to the current execution point. A number of recent papers have explored the use of symbolic execution to analyze information flow through a hardware design [6], [25], [41].

G. Symbolic Simulation, Model Checking, Symbolic Execution

We do not advocate replacing either symbolic simulation or model checking with symbolic execution. Rather, it has become clear in recent years that symbolic execution is a valuable tool to add to the formal verification toolbox, especially when it comes to security verification tasks [6], [21], [25], [40]. We present an algorithmic technique to improve the performance of symbolic execution for hardware designs.

III. PIECEWISE COMPOSITION

In conventional symbolic execution, each line of code is potentially visited multiple times, once for each path explored. Our approach is to aggressively decompose the design into independent blocks, symbolically explore each block once, then use an SMT solver to compose path conditions and symbolic stores from each block. This strategy is made possible by the inherent modular nature of hardware designs, and lets us

leverage the relative speed of modern SMT solvers compared to the cost of symbolically executing lines of code.

While the number of paths in the full symbolic execution tree is exponential in the number of branches in the design, the engine explores a number of paths exponential in only the number of branches in any single independent block and polynomial in the number of blocks.

A. Motivating Example

Figure 1 shows a snippet with two `always` blocks and branch points at lines 2 and 9. The corresponding control flow graph with an arbitrary ordering of the `always` blocks is given in Figure 2a, and the tree of paths through the design is given in Figure 2b. With conventional symbolic execution, each of the four root-to-leaf paths in Figure 2b is symbolically executed. This is the strategy taken by current approaches (e.g., [8], [25], [48]) that translate a hardware design into a C++ representation and then use the KLEE symbolic execution engine [11]. The two subtrees rooted at a node labeled 9 represent repeated work. For each subtree, the symbolic execution engine is exploring the same paths through the block starting at line 9.

The branching condition and assignments in lines 2–5 are independent of the branching condition and assignments in lines 9–12. Regardless of which path is taken at the first branch (line 2), the symbolic execution starting at the second branch point (line 9) will produce the same sub-tree. The feasibility of the second condition will be the same, and updates to the symbolic state will be the same. For example, let the initial symbolic store and path condition be:

$$\begin{aligned}\sigma_0 &= \{\text{inpA} = \alpha, g0 = \gamma_0, \text{inpB} = \beta, g1 = \gamma_1\} \\ \pi_0 &= \text{True}.\end{aligned}$$

After symbolically executing the path in which both branches are taken (nodes $\langle 2, 3, 9, 10 \rangle$ in the symbolic execution tree), the symbolic store and path condition would be:

$$\begin{aligned}\sigma_{2,3,9,10} &= \{\text{inpA} = \alpha, g0 = \gamma_0, \text{inpB} = \beta, g1 = \gamma_1, y := \alpha, z := \beta\} \\ \pi_{2,3,9,10} &= \gamma_0 \wedge \gamma_1.\end{aligned}$$

Whereas, for the path in which the first branch is not taken, but the second one is ($\langle 2, 5, 9, 10 \rangle$), the symbolic store and path condition would be:

$$\begin{aligned}\sigma_{2,5,9,10} &= \{\text{inpA} = \alpha, g0 = \gamma_0, \text{inpB} = \beta, g1 = \gamma_1, y := 0, z := \beta\} \\ \pi_{2,5,9,10} &= \neg \gamma_0 \wedge \gamma_1.\end{aligned}$$

In both paths, the updates to z are the same, despite the different updates to y .

B. Piecewise Composition

With piecewise composition, the engine explores independent blocks of the RTL separately, producing independent trees of path fragments. In the above example, piecewise composition results in the two trees shown in Figure 2c.

The engine now explores the second `if-else` block only once. Continuing with our example, piecewise composition will separately explore the two `always` blocks, producing the following four path fragments with associated path conditions and (partial) symbolic stores:

$$\begin{aligned} \langle 2, 3 \rangle : \quad & \sigma_{2,3} = \{y := \alpha\}, \quad \pi_{2,3} = \gamma_0 \\ \langle 2, 5 \rangle : \quad & \sigma_{2,5} = \{y := 0\}, \quad \pi_{2,5} = \neg\gamma_0 \\ \langle 9, 10 \rangle : \quad & \sigma_{9,10} = \{z := \beta\}, \quad \pi_{9,10} = \gamma_1 \\ \langle 9, 12 \rangle : \quad & \sigma_{9,12} = \{z := 0\}, \quad \pi_{9,12} = \neg\gamma_1 \end{aligned}$$

To find full paths through the design and to successfully find assertion violations, all realizable combinations of path fragments are composed with the help of an SMT solver. For example, to realize path $\langle 2, 5, 9, 10 \rangle$, the engine queries the SMT solver to find whether the two path fragments, $\langle 2, 5 \rangle$ and $\langle 9, 10 \rangle$ can be joined: $\text{isSAT}(y = 0 \wedge \neg\gamma_0 \wedge y = \beta \wedge \gamma_1)$. In this simple example, all four combinations of path fragments are possible, but in general that will not always be the case.

Piecewise composition will ultimately be constructing the same path conditions as conventional symbolic execution. The difference is in repeated work during the path exploration. Looking again at Figure 2c, The conventional approach will execute the following paths and their corresponding lines of code: $\langle 2, 3, 9, 10 \rangle$, $\langle 2, 3, 9, 12 \rangle$, $\langle 2, 5, 9, 10 \rangle$, $\langle 2, 5, 9, 12 \rangle$. Lines 2-3, 2-5, 9-10 and 9-12 are all explored twice. Piecewise composition will explore the following path fragments and corresponding lines of code, each only once: $\langle 2, 3 \rangle$, $\langle 2, 5 \rangle$, $\langle 9, 10 \rangle$, $\langle 9, 12 \rangle$. With this small example, piecewise composition is able to cut the path exploration workload in half. As the size of the design grows, the number of paths to explore with piecewise composition will be exponential only in terms of the number of branch points in a given `always` block and linear in the number of `always` blocks. We examine this more closely in the complexity analysis in Section III-D.

A conventional symbolic execution engine will query the SMT solver at branch 9 twice, once for path $\langle 2, 3, 9 \rangle$ and once for path $\langle 2, 5, 9 \rangle$. These queries are checking for feasibility of the branching condition at line 9 in the RTL. Piecewise composition will only query the solver for the branch on line 9 once. Piecewise composition will then require queries for all four complete paths through the design: $\langle 2, 3, 9, 10 \rangle$, $\langle 2, 3, 9, 12 \rangle$, $\langle 2, 5, 9, 10 \rangle$, $\langle 2, 5, 9, 12 \rangle$. These queries are ensuring that the accumulated path conditions are satisfiable and the execution path is realizable. In this small example, we do not reduce our SMT solving workload, but as the design becomes more complex piecewise composition yields a slight reduction in queries performed as we reduce the amount of redundant branch points explored. We evaluate the impact of piecewise composition both on lines of code explored and SMT queries in Section VI.

C. Comparison with Backtracking and Caching

Piecewise composition shares some similarities with backtracking and caching, two techniques often used in software symbolic execution engines (e.g., KLEE [11], Angr [32]). But,

there are key differences. Backtracking reduces repeated work by maintaining state at each point in a path and allowing two paths with a shared prefix to reuse the saved state. For example, If path $\langle 2, 3, 9, 10 \rangle$ has been explored, then when the engine explores path $\langle 2, 3, 9, 12 \rangle$, backtracking allows the engine to reuse the saved state at point 9 and continue exploration from there. Backtracking prevents re-exploring path $\langle 2, 3 \rangle$ for each of $\langle 9, 10 \rangle$ and $\langle 9, 12 \rangle$; piecewise composition also prevents this re-exploration. However, with backtracking, paths $\langle 9, 10 \rangle$ and $\langle 9, 12 \rangle$ will be re-explored to create the paths starting with prefix $\langle 2, 5 \rangle$; this re-exploration is prevented by piecewise composition.

Caching queries reduces the time spent in the SMT solver by reusing the results from prior queries. Caching queries is a technique orthogonal to piecewise composition. Using the two techniques together could further reduce runtime.

D. Complexity Analysis

We separately compute the lines of code visited by the engine during symbolic execution and the number of queries to the solver for both the baseline implementation and the implementation that uses piecewise composition to develop a theoretical understanding of the benefits of piecewise composition.

We perform the analysis with the following Verilog design parameters and assumptions:

- b : The maximum number of branch points in any one `always` block. All branch points are assumed to have at most two branches. Case statements can be rewritten to use only 2-branch conditionals.
- N : The number of sequential-logic `always` blocks.
- c : The maximum number of lines of code after any branch point until either the next branch point or an exit point. In other words, the maximum number of lines of code in any *basic block*.
- Assumption 1: We assume that all loops are unrolled.
- Assumption 2: We approximate combinational logic with a fixed constant for both the baseline and piecewise composition approaches. We do this because the piecewise composition technique is only applicable to sequential `always` blocks. The underlying implementation strategy used by Sylvia to ensure clock-cycle accuracy with combinational logic statements results in each block of combinational logic being executed twice per clock cycle in the worst case (see Section IV-A for more details).

When we use the baseline approach, the symbolic execution of a design is represented by a single binary tree (as in Figure 2b). To simplify the analysis, we assume a perfect binary tree in which every interior node has two children and all leaf nodes are at the same level. This assumption only holds in practice if every branch point in the code is reachable along every path, which in general is not the case. Our analysis, therefore, provides a loose upper bound on complexity; a tighter bound may be possible.

When we use piecewise composition, the symbolic execution of a design is represented by N binary trees, one for each sequential-logic `always` block (as in Figure 2c).

1) Baseline: Lines of Code Symbolically Executed

Every node in the tree is visited once per path it belongs to.

$$\underbrace{bN(2^{bN})}_{(a)} + \underbrace{cbN(2^{bN})}_{(b)} \quad (1)$$

- (a) Visit and execute each branch point (bN) once per path it is part of (2^{bN})
- (b) Visit and execute the c lines of code in the basic block of either the right or left branch for each branch point (bN), once per path (2^{bN}).

The time complexity for executing the design symbolically using the baseline implementation is $O(cbN2^{bN})$. As the design size grows, the number of lines of code explored is growing exponentially with bN , the total number of branch points in the entire design.

2) Baseline: SMT Queries

Every branching node in the tree generates one query per visit, and is visited once per path it belongs to.

$$\underbrace{bN(2^{bN})}_{(a)} \quad (2)$$

- (a) Visit each branch point (bN) once per path (2^{bN}) it is part of, and each visit generates one query.

The time complexity for querying the SMT solver under the baseline implementation is $O(bN2^{bN})$.

3) Piecewise Composition: Lines of Code Symbolically Executed

Every node in every tree is visited once per path it belongs to.

$$\underbrace{bN(2^b)}_{(a)} + \underbrace{cbN(2^b)}_{(b)} \quad (3)$$

- (a) For each tree (N), visit and execute each branch point (b) once per path it is part of (2^b)
- (b) For each tree (N), visit and execute the c lines of code in the basic block of either the right or left branch for each branch point (b), once per path (2^b).

The time complexity for executing the design symbolically using piecewise composition is $O(cbN2^b)$. Compared with the baseline implementation, piecewise composition drops the exponential factor N and explores each unique path fragment once.

4) Piecewise Composition: SMT Queries

In addition to the queries for each branching node visited, one query is generated for every combination of paths, one from each tree, in order to recreate the full root-to-leaf paths.

$$\underbrace{bN(2^b)}_{(a)} + \underbrace{(2^b)^N}_{(b)} \quad (4)$$

- (a) For each tree (N), visit each branch point (b) once per path it is part of, and each visit generates one query.
- (b) Generating each path through the full design requires one query. Each of N trees has 2^b paths, and all combinations need to be combined.

The time complexity for querying the SMT solver using piecewise composition is $O(2^{bN})$. In the limit, there is a slight advantage in the number of SMT queries compared to the baseline implementation, and in practice we do see less time spent in the solver (see Figure 3).

IV. A SYMBOLIC EXECUTION ENGINE WITH PIECEWISE COMPOSITION

We introduce Sylvia, a symbolic execution engine implementing piecewise composition. Importantly, Sylvia operates directly over the Verilog RTL without translating to C or compiling down to the netlist. This allows for greater human-readability of any found assertion violations. Sylvia is cycle accurate. We assume no combinational latches, no asynchronous resets, and `always` blocks are conditioned on input clocks. These assumptions are in keeping with prior work in this area [6].

The core data structures Sylvia builds and uses are the Verilog AST and control-flow graphs (CFG). Sylvia constructs one CFG per `always` block. The symbolic execution trees described in the preceding sections are useful as a conceptual model of symbolic execution, but in practice the engine executes over the basic blocks of statements that are collected in each CFG. A single execution path in Sylvia is encoded as a combination of individual paths through the set of CFGs.

The engine achieves piecewise composition by decomposing a design into partitions: one partition to contain all combinational logic in the design, one partition for all register declarations, and a set of N partitions, one per `always` block, to handle the sequential logic in the design. Each partition is symbolically explored once per clock cycle, with the exception of the combinational logic partition, discussed next in Section IV-A.

Each of the N sequential `always` block partitions are explored independently of the other `always` blocks, and the exploration produces a set of path fragments. The complete exploration of the full design produces N sets, one per `always` block. The set of full root-to-leaf symbolic execution paths through the design is formed by taking the cross-product of the N sets of path fragments. The SMT solver is used to ensure only those combinations that are sound – that correspond to true paths through the design – are kept.

A. Combinational Logic

The engine will check for any combinational latches, and if any appear, will exit with an error. Otherwise, Sylvia first symbolically executes each statement in the combinational

logic partition and then begins to execute the control flow paths through each `always` block. As each `always` block is executed the engine keeps track of a dirty bit for each signal, which gets set to 1 when the signal is updated within that particular clock cycle. The intuition here is that if one of the combinational logic dependencies becomes dirty during the symbolic execution of the `always` block, we need to re-evaluate the corresponding combinational `assign`. Once a path has been completed, every `assign` statement in the combinational logic partition for which the right-hand side involves a dirty signal is re-evaluated. In the worst case, this means each statement in the combinational logic partition may be symbolically executed twice. During this re-evaluation, the engine continues to track when signals become dirty and propagate updates as needed to ensure clock-cycle accuracy.

B. Sequential Logic

Each sequential `always` block is explored independently. This approach is sound if the `always` blocks are truly independent – the path condition and symbolic state of the various paths through one block are the same regardless of the paths taken through other blocks. In the following we discuss the issue of independence in more detail. Consider two sequential `always` blocks, B_0 and B_1 , both triggered on the same edge of the input clock signal.

1) Independence

In the simplest case, none of the variables that appear in B_0 appear in B_1 . The two blocks are independent and, within a single clock cycle, the execution of one block has no bearing on the execution of the second block. The two blocks can be explored separately and their paths can be composed in any order. This case is rare, however, as an input reset signal typically appears in all or most blocks.

2) Read-read dependence

In the next case, the same variable may appear in a branch condition or right-hand side of an assignment in both B_0 and B_1 . The two blocks can still be explored separately and their paths can be composed in any order. However, some combinations of paths may not be feasible, as variables that appear in branch conditions in both blocks, say b_0 and b_1 , respectively, will preclude the combination of paths from B_0 in which b_0 holds with paths from B_1 in which b_1 holds when $b_0 \wedge b_1$ is unsatisfiable.

3) Read-write dependence

In the next case, a variable may appear in a branch condition or on the right-hand side of an assignment in B_0 and on the left-hand side of an assignment in B_1 . When non-blocking assignments are used, updates to variables in B_1 take effect in the next clock-cycle, whereas reads and conditional branches in B_0 use values set in the previous clock cycle. The symbolic execution engine keeps the appropriate value and there is no conflict. The two blocks can be explored separately and their paths can be composed in any order. Sylvia does not support the use of blocking assignments within sequential `always` blocks.

4) Write-write dependence

In the final case, variables appear on the left-hand side of an assignment in both `always` blocks. This violates best practice in Verilog design. The symbolic execution engine will check for any instances of write-write dependence, and if any appear will exit with an error.

C. Further Optimizations

1) Repeat Submodules

When the modules are duplicate instantiations of the same module there is room for reduction in the total search space. The idea is similar in spirit to piecewise composition; the engine explores each submodule once for each path. Then instead of re-exploring again for each repeat instantiation, the engine merges in the symbolic store and path condition for the given root-to-leaf path using SMT queries.

2) Cone of Influence Analysis

This optimization prunes the exploration space at the block level. The symbolic execution engine will read in the expressions supplied in the assertions, perform a dependency analysis over the signals in the assertions and then complete an AST traversal to determine which blocks read from or write to the signals of interest or their dependencies. After this initial pass, the engine will only explore blocks that involve the signals of interest or their dependencies.

V. IMPLEMENTATION

Sylvia³ is built in python 3.8 and implements the Verilog semantics according to the IEEE 1364-2005 standard. We use the pyVerilog library to build the Verilog AST, networkX to manage graph search and traversal, and the Z3 python API for SMT solving. The engine reads in a design, including the assertions written according to the SystemVerilog 1800-2017 standard, and outputs replayable counterexamples.

VI. EVALUATION

We evaluate Sylvia over five open-source designs to study its viability as a platform for the verification of hardware designs. Our evaluation considers the following questions: 1) How well does piecewise composition counter the path explosion problem? 2) What effect do piecewise composition and the optimizations described in Section IV-C have on performance? 3) Does our engine produce assertion violations with replayable counter-examples for vulnerable designs?

A. Dataset and Experimental Setup

We collected five designs and 84 security critical assertions. The first three designs and associated assertions came from the Security Property/Rule Database available on TrustHub [22], [23]. These are an enhanced version of the Serial Peripheral Interface available on Motorola’s MC68HC11 family of CPUs; openMSP430, a synthesizable 16-bit microcontroller core compatible with Texas Instruments’ MSP430 microcontroller family; and a CrypTech True Random Number Generator

³Sylvia is fully open-source and can be accessed at <https://github.com/kakiryan/Sylvia>

(TRNG). For each of these designs, the database included 9, 2, and 2 security properties, respectively.

The fourth design is the buggy PULPissimo SoC used in a recent Hack@DAC competition [1]. Using the English description of the properties, as well as the walkthrough of the test-case generation in the RTL-ConTest paper [40] we developed 26 assertions for use with our tool.

The fifth design is the OR1200 processor core. We collected 30 security-critical bugs from two prior papers, SPECS [28] and SCIFinder [49] and 70 security assertions from SPECS [28], Security Checkers [7], SCIFinder [49], and Transys [50].

The experiments are performed on a machine with an Intel Xeon E5-2620 V3 12-core CPU (2.40GHz, a dual-socket server) and 62G of available RAM.

B. Mitigation of Path Explosion

For each design we compare the average number of lines of code and branch points visited to find an assertion violation both with and without piecewise composition. Table I has the results. The number of paths in a design will not change, but the amount of work to realize a path does. Piecewise composition reduces the number of lines of code visited (i.e., reduces redundant visits to the same line of code) by 92%–99% and branch points visited by 64%–99%.

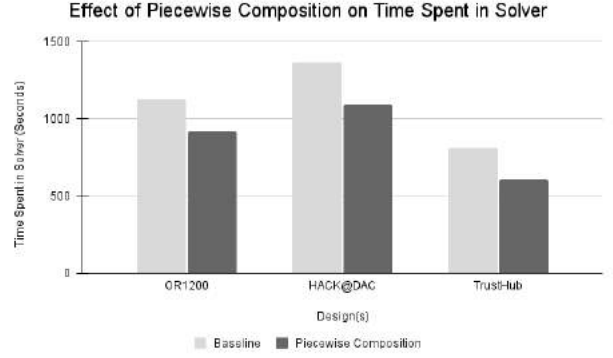
To gain a more complete picture, we symbolically explore all paths through the small MC68HC11 SPI. This design has 15 `always` blocks and 1459 possible paths. These results are reported in Table II. Without piecewise composition more than 90k lines of code and more than 7k branch points are explored. With piecewise composition, the engine needs to explore roughly only 7% of those 90k lines of code and only 4% of those 7k branch points.

The benefits of piecewise composition come from the presence of composable `always` blocks. We report on the number and dependency type (Section IV-B) of these structures in our benchmarks in Table III. As expected, all `always` blocks have at most read-write dependencies, allowing them to be composed.

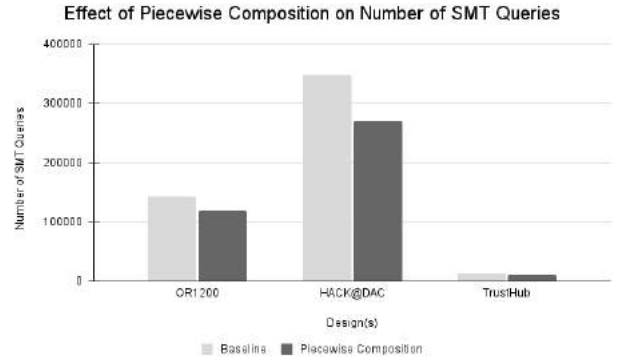
C. Effects of Optimizations

Figure 3 shows the impact of piecewise composition on the average number of SMT queries and average time spent in the SMT solver for each design. One concern might be that the win in minimizing redundant explorations comes at the expense of exploding SMT solver work. However, the opposite occurs. Reducing the number of paths explored also reduces the number of queries to the solver overall; remember that during exploration, the solver is queried at each branch point. With piecewise composition turned on there was an 18% decrease, on average, in the number of SMT queries and a 21% decrease in the amount of time spent solving.

In Table IV we measure runtime for four cases: *Baseline*, with no optimizations enabled; *Piecewise*, with piecewise composition enabled; *Repeat*, with repeated modules explored only once; and *COI*, with cone-of-influence analysis completed



(a) Solver Time



(b) SMT Queries

Fig. 3: Effects of Optimizations on Solver Time and Number of SMT Queries

before exploration. Each case is cumulative, for example, in the *Repeat* case, the *Piecewise* optimization is enabled as well. For each design we take the average when looking for each assertion. For all but the smallest design, *Baseline* could not reliably complete exploration within 30 minutes at which point we stopped searching. Table IV shows the results. Overall, the optimizations decrease the engine’s runtime by 95-99%.

D. Finding Assertion Violations

To evaluate the engine’s ability to find assertion violations, we run a set of experiments in which we have ground-truth knowledge of the (minimum) number of violations in each design. In these experiments, symbolic execution begins in the reset state with all input signals made symbolic, and execution continues until an assertion violation is found. All counterexamples generated by our engine were successfully replayed in simulation starting from the reset state using Vivado. Table V summarizes the results.

The engine finds 25 of the 31 bugs in the Hack@DAC SoC. The organizers of Hack@DAC report finding 6 and 15 bugs using the commercial tools Cadence SPV and Cadence FPV, respectively [1]. The engine finds 29 of the 30 bugs in the OR1200. The bug missed does not have a property in our dataset that covered it.

Design	Baseline		Piecewise Composition		Percent Decrease	
	LoC explored	branch points explored	LoC explored	branch points explored	LoC explored	branch points explored
OR1200	54018	7803	881	45	98%	99%
Hack@DAC	493032	15093	3525	276	99%	98%
MC68HC11 SPI	2093	158	174	57	92%	64%
openMSP430	15293	377	489	68	97%	82%
CrypTech TRNG	8930	421	336	91	96%	78%

TABLE I: Average Impact of Piecewise Composition on Path Explosion

Configuration	LoC explored	branch points explored	paths completed
Baseline	90706	7380	1459
Piecewise Composition	6783	323	1459

TABLE II: Full Exploration of MC68HC11 SPI Design

The engine is consistently able to find vulnerabilities that both commercial and open-source model checking tools are unable to find. Table VI summarizes how Cadence and SymbiYosys [2], a symbolic model checking engine built on top of Yosys, fared in finding the same known vulnerabilities.⁴

At the time of writing we don’t know why the model checking tools were unable to find all the assertion violations and further investigation is warranted. We do not believe that it is the result of a theoretical limitation of bounded model checking or the strength of the algorithms used by the underlying proof engines. More likely, our hypothesis is that there are some abstractions introduced to manage complexity that cause the property violations to be missed. The properties we are searching for are specifying system level behavior and in some cases have been automatically generated. The complexity inherent to these types of security properties compared to typical functional correctness properties may make it more difficult for a traditional formal verification approach like model checking to find the violations.

We set the bound for the model checking tools to be 5 clock cycles over the minimum bounds needed to find the violation, and let the tools run to completion. Our results align with results reported by the authors of performing comparable experiments [40] [48] and [19] and match those of the authors of the TrustHub designs and properties that we were using for evaluation.

We demonstrate how our approach allows search to scale more efficiently over multiple clock cycles compared to SymbiYosys in Figure 4. We take the MSP430 design and embed properties into the design that require an increasing number of cycles to produce a counterexample. SymbiYosys begins by outperforming our tool in terms of speed – it takes SymbiYosys under half a second to complete 4 cycles while we take around 3 seconds. As the complexity of the search space grows, piecewise composition scales more manageably.

⁴The numbers in the Cadence column are pulled from the literature [1], [23], [40], [48]. Our license does not allow for head-to-head comparisons.

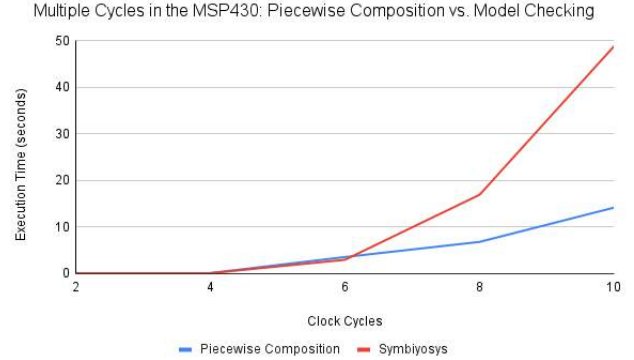


Fig. 4: Scaling Search Over Multiple Cycles

E. Comparison to Current State of the Art

Coppelia is a tool that performs symbolic execution over the C++ model of a Verilog hardware design. In comparison to Coppelia [48], we see significant performance gains. On average, Coppelia takes 4 minutes and 12 seconds to find the same known security vulnerabilities in the OR1200 that our tool is able to find in 25.22 seconds. The authors report that most (62%) of exploits in their experiments are generated within 15 minutes. However, several (7%) are found within 2–4 hours. Symbolic execution with piecewise composition, by contrast, finds the same 7% of exploits in under two minutes.

The authors of RTLConTest [40], a concolic execution engine, report that it takes around an hour and 40 minutes to complete on the PULPissimo SoC, which is a modified version of the HACK@DAC 2018 design. Once they perform the concolic execution and generate the tests, it takes 10 seconds on average to produce a counterexample. They find 14 out of the 31 bugs while our engine is able to find 25. Our tool performs the complete end-to-end symbolic execution workflow to generate counterexamples in 81.83 seconds, on average.

VII. RELATED WORK

Symbolic Simulation Of the papers presented in Section II-E, we note the early work implementing symbolic simulation at the RT level [34] that introduces a path-merging approach for handling and mitigating the complex queries characteristic of symbolic simulation. A more recent project is the Rosette/Racket solver-aided programming platform [45],

Design	LoC	Always Blocks	Branch Points	% Independent	% Read-Read	% Read-Write	% Composable
OR1200	30611	405	976	6.81%	40.98%	52.21%	100%
Hack@DAC	96444	650	4452	12.89%	42.33%	44.78%	100%
MC68HC11 SPI	527	14	43	21.45%	34.89%	44.66%	100%
openMSP430	9154	144	316	15.34%	27.27%	57.39%	100%
CrypTech TRNG	5926	54	309	8.21%	32.31%	59.48%	100%

TABLE III: Logical Structure of Benchmarks

Design	Baseline	Piecewise		Redund		COI		Overall
	runtime (sec)	runtime (sec)	% dec	runtime (sec)	% dec	runtime (sec)	% dec	% dec
OR1200	timeout (1800)	52.47	97.08%	37.56	21.31%	25.22	12.56%	98.60%
Hack@DAC	timeout (1800)	174.24	90.32%	121.94	28.34%	81.83	16.62%	95.45%
MC68HC11	962	17.53	98.18%	14.30	19.93%	0.07	99.19%	99.99%
openMSP430	timeout (1800)	37.65	97.91%	23.14	38.55%	0.73	96.83%	99.96%
CrypTech TRNG	timeout (1800)	14.92	99.17%	12.08	19.15%	0.09	99.19%	99.99%

TABLE IV: Average Effect of Optimizations on Runtime

Design	# Bugs	# Bugs Found	Avg Time (sec)	Max Clock Cycles Taken
Hack@DAC	31	25	81.83	4
OR1200	30	29	25.22	5
MC68HC11	9	9	0.07	3
openMSP430	2	2	0.73	2

TABLE V: Finding Known Bugs: Runtime Performance

Design	# Bugs	Our Engine	Cadence	SymbiYosys
OR1200	30	29	18	18
Hack@DAC	31	25	21	16
MC68HC11	9	9	8	5
openMSP430	2	2	2	1

TABLE VI: Finding Known Bugs: Comparison to Model Checking

whose use is demonstrated for verification in Notary [5]. Both works, like all symbolic simulation, merge symbolic states after each branch point, and require constraining the control-flow with concrete inputs to manage expression complexity.

Model Checking As discussed in Section II-F, model checking is a mature tool widely used in industry and research. SymbiYosys [2] is a formal verification engine for Verilog that operates at the netlist level. We compare to this tool in our evaluation (Section VI). Symbolic Quick Error Detection [20], [38] is a technique involving self-consistency checks that has been used to find bugs in open-source RISC-V processors and uses the CoSA model checker [39].

Symbolic and Concolic Execution Symbolic execution and the related technique of concolic execution are emerging from the academic research community as useful techniques for the security verification of hardware designs [8], [25], [40], [47],

[48]. However, many of these papers rely on first translating Verilog to C++ and using KLEE [11], a tool written for, and fine-tuned for, the symbolic execution of software programs.

Fuzzing Fuzzing has also been shown to be a useful technique for finding security vulnerabilities in SoCs and CPU designs. RFUZZ is a coverage-directed fuzz tester for circuits that presents a hardware-specific coverage metric called *mux control coverage* [36]. DifuzzRTL is an RTL fuzzing tool used to find unknown security bugs that measures coverage based on control registers rather than multiplexors' control signals to improve efficiency and scalability [30]. A recently developed Hardware Fuzzing Pipeline translates the RTL to a software model to improve scalability in bug finding via fuzzing [46].

VIII. CONCLUSION

We have presented piecewise composition, a technique for countering the path explosion problem in symbolic execution. We implemented Sylvia, a symbolic execution engine using the technique and evaluated the engine on five open-source designs. The engine reduces redundant work by 98%–99% compared to conventional symbolic execution, improves overall performance and successfully finds assertion violations.

IX. ACKNOWLEDGMENTS

We would like to thank Sayak Ray and the anonymous reviewers for their insightful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1816637 and Grant No. CNS-2247754, and by a Meta Security Research Award. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

REFERENCES

- [1] “Hack@DAC 2018 SoC,” <https://github.com/seth-lab-tamu/hackdac-2018-soc>, accessed: 2022-02-15.
- [2] “SymbiYosys,” <https://github.com/YosysHQ/sby>, accessed: 2022-11-21.
- [3] “Voss II,” <https://github.com/TeamVoss/VossII>, accessed: 2022-11-21.

- [4] A. Ardeshtiricham, W. Hu, J. Marxen, and R. Kastner, "Register Transfer Level information Flow Tracking for Provably Secure Hardware Design," in *DATE*, 2017, pp. 1691–1696.
- [5] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Notary: A Device for Secure Transaction Approval," in *27th Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3341301.3359661>
- [6] A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Verifying Hardware Security Modules with Information-Preserving Refinement," in *OSDI*. USENIX Association, 2022.
- [7] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security Checkers: Detecting processor malicious inclusions at runtime," in *HOST*, 2011.
- [8] N. Bruns, V. Herdt, and R. Drechsler, "Processor Verification using Symbolic Execution: A RISC-V Case-Study," 2023. [Online]. Available: https://agra.informatik.uni-bremen.de/doc/konf/2023_DATE_NB.pdf
- [9] R. E. Bryant, "Symbolic Simulation—Techniques and Applications," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, ser. DAC '90. New York, NY, USA: Association for Computing Machinery, 1991, p. 517–521. [Online]. Available: <https://doi.org/10.1145/123186.128296>
- [10] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," in *ACM/IEEE DAC*, 1991.
- [11] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *OSDI*, vol. 8, 2008, pp. 209–224.
- [12] S. Chakraborty, Z. Khasidashvili, C.-J. H. Seger, R. Gajavelly, T. Haldankar, D. Chhatani, and R. Mistry, "Symbolic Trajectory Evaluation for Word-Level Verification: Theory and Implementation," *Form. Methods Syst. Des.*, vol. 50, no. 2–3, p. 317–352, Jun. 2017.
- [13] K. Claessen and J.-W. Roorda, "An Introduction to Symbolic Trajectory Evaluation," in *Formal Methods for Hardware Verification*, M. Bernardo and A. Cimatti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 56–77.
- [14] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," vol. 19, no. 1, p. 7–34, 2001.
- [15] E. M. Clarke, "Model Checking," in *Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 54–56.
- [16] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1891823.1891830>
- [17] C. N. Coelho and H. D. Foster, *Assertion-Based Verification*. Boston, MA: Springer US, 2004, pp. 167–204.
- [18] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *USENIX Security Symposium*, 2013.
- [19] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 213–230. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky>
- [20] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, "Symbolic Quick Error Detection Using Symbolic Initial State for Pre-Silicon Verification," in *Design, Automation & Test in Europe (DATE)*, 2018, pp. 55–60.
- [21] M. R. Fadiheh, A. Wezel, J. Muller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz, "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 222–235, Jan. 2023. [Online]. Available: <https://doi.org/10.1109/2Ftc.2022.3152666>
- [22] N. Farzana, F. Farahmandi, and M. M. Tehranipoor, "SoC Security Properties and Rules," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 1014, 2021.
- [23] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "SoC Security Verification using Property Checking," in *2019 IEEE International Test Conference (ITC)*, 2019, pp. 1–10.
- [24] B. Finkbeiner, M. N. Rabe, and C. Sánchez, "Algorithms for Model Checking HyperLTL and HyperCTL," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 30–48.
- [25] F. Fowze, M. Choudhury, and D. Forte, "EISec: Exhaustive Information Flow Security of Hardware Intellectual Property Utilizing Symbolic Execution," in *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE Xplore, 2022.
- [26] A. Goel and K. Sakallah, "Model Checking of Verilog RTL Using IC3 with Syntax-Guided Abstraction," in *NASA Formal Methods*, J. M. Badger and K. Y. Rozier, Eds. Cham: Springer International Publishing, 2019, pp. 166–185.
- [27] M. Goli and R. Drechsler, "VIP-VP: Early Validation of SoCs Information Flow Policies using SystemC-based Virtual Prototypes," in *2021 Forum on specification & Design Languages (FDL)*, 2021, pp. 1–8.
- [28] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs," in *ASPLOS*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, p. 517–529.
- [29] W. Hu, A. Ardeshtiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property Specific Information Flow Analysis for Hardware Security Verification," in *ICCAD*, ser. ICCAD '18. New York, NY, USA: Association for Computing Machinery, 2018.
- [30] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs," in *42nd IEEE S&P*. IEEE, 2021, pp. 1286–1303.
- [31] R. Kaivola and N. B. Kama, "Timed Causal Fanin Analysis for Symbolic Circuit Simulation," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2022, pp. 99–107. [Online]. Available: <https://repositum.tuwien.at/handle/20.500.12708/81329>
- [32] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing Intermediate Representations for Binary Analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 353–364.
- [33] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976.
- [34] A. Kölbl, J. Kukula, and R. Damiano, "Symbolic RTL Simulation," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 47–52.
- [35] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs," in *2010 19th IEEE Asian Test Symposium*, 2010, pp. 59–64.
- [36] K. Laeuffer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs," in *ICAAD*, 2018, pp. 1–8.
- [37] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches* (Prentice Hall Modern Semiconductor Design Series). USA: Prentice Hall PTR, 2005.
- [38] F. Lonsing, K. Ganesan, M. Mann, S. S. Nuthakki, E. Singh, M. Srouji, Y. Yang, S. Mitra, and C. Barrett, "Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED: Invited Paper," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.
- [39] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "CoSA: Integrated Verification for Agile Hardware Design," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–5.
- [40] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 466–477, 2022.
- [41] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich, "rtlvc: push-button verification of software on hardware," in *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2021. [Online]. Available: <https://vm-web.pdos.csail.mit.edu/papers/rtlvc:carrv21.pdf>
- [42] R. Mukherjee, D. Kroening, and T. Melham, "Hardware Verification Using Software Analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI*, 2015, pp. 7–12.
- [43] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, no. 2, p. 147–189, Mar. 1995. [Online]. Available: <https://doi.org/10.1007/BF01383966>
- [44] L. Shen, D. Mu, G. Cao, M. Qin, J. Blackstone, and R. Kastner, "Symbolic Execution Based Test-patterns Generation Algorithm for Hardware Trojan Detection," *Comput. Secur.*, vol. 78, pp. 267–280, 2018.
- [45] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and*

- Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 530–541. [Online]. Available: <https://doi.org/10.1145/2594291.2594340>
- [46] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, “Fuzzing Hardware Like Software,” in *USENIX '22*. Boston, MA: USENIX Association, Aug. 2022, pp. 3237–3254.
 - [47] H. Witharana, Y. Lyu, and P. Mishra, “Directed Test Generation for Activation of Security Assertions in RTL Models,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 26, no. 4, jan 2021.
 - [48] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, “End-to-End Automated Exploit Generation for Validating the Security of Processor Designs,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2018.
 - [49] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, “Identifying Security Critical Properties for the Dynamic Verification of a Processor,” in *ASPLOS*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, p. 541–554.
 - [50] R. Zhang and C. Sturton, “Transys: Leveraging Common Security Properties Across Hardware Designs,” in *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2020.

Binary Decision Diagrams on Modern Hardware

Samuel Pastva and Thomas Henzinger

Institute of Science and Technology Austria

Klosterneuburg, 3400 Austria

Email: samuel.pastva@ist.ac.at, tah@ist.ac.at

Abstract—Binary decision diagrams (BDDs) are one of the fundamental data structures in formal methods and computer science in general. However, the performance of BDD-based algorithms greatly depends on memory latency due to the reliance on large hash tables and thus, by extension, on the speed of random memory access. This hinders the full utilisation of resources available on modern CPUs, since the absolute memory latency has not improved significantly for at least a decade.

In this paper, we explore several implementation techniques that improve the performance of BDD manipulation either through enhanced memory locality or by partially eliminating random memory access. On a benchmark suite of 600+ BDDs derived from real-world applications, we demonstrate runtime that is comparable or better than parallelising the same operations on eight CPU cores.

Index Terms—binary decision diagram, symbolic algorithm, hash table, cache.

I. INTRODUCTION

Binary decision diagrams (BDDs) [9] (or more specifically, reduced ordered binary decision diagrams (ROBDDs)) are one of the fundamental data structures in computer science. They are directed acyclic graphs representing Boolean functions, often exponentially more succinct compared to Boolean expressions or function tables [34].

They have a wide range of applications in formal verification [1], [10], [12], [14], [20], [43], satisfiability checking [22], [24], hardware design [26], [27], [44], test design [37], dialectical frameworks [16], and optimisation [7]. They are the building blocks for the so-called *symbolic algorithms* that are, among other applications, used for exploration of large graphs suffering from exponential state-space blow-up [3], [5], [6], [28], [45]. Many extensions of BDDs exist that attempt to improve their succinctness, typically at the cost of more complex manipulation algorithms. One example are zero-suppressed decision diagrams [32], but a more exhaustive summary of known BDD variants can be found in [2].

There are implementations of BDDs that rely on shared-memory [41] and distributed-memory [35] parallelism, external memory [38] and even GPUs [42]. Furthermore, variable ordering within the BDD has a strong impact on its succinctness and has been an intense subject of optimisation [18], [21].

In this paper, we tackle another important aspect of BDD implementation. In general, it is known that operations on BDDs are bottlenecked by memory latency due to their extensive use of large hash tables [8].

This is an unfortunate bottleneck on modern hardware, since the absolute memory latency has not improved for at least the

last 15 years [13]. Memory *capacity*, memory *bandwidth*, the number of CPU cores, as well as their *width* and *frequency* has grown significantly. However, the memory *latency* on a CPU bought today (i.e. 2023) is essentially the same as on the one bought in 2006 [13].

In this paper, we demonstrate that as a result, a *modern* CPU (2020) is in fact *worse* at BDD manipulation compared to its *legacy* (2014) counterpart once the problem size grows beyond the last-level cache (typically L3 cache). To address this problem, we propose an alternative data structure that replaces one of the underlying hash tables (node uniqueness table). We also devise additional criteria to reduce the amount of memory accesses performed during BDD manipulation. In the end, we observe that our approach to BDD manipulation indeed improves the performance on a modern CPU significantly, to an extent comparable with parallelisation on 8 CPU cores.

Finally, note that this is not the first attempt to design a more cache friendly BDD implementation. In [15], the authors propose to use a more cache-friendly hashing scheme called Hopscotch hashing. However, the paper also proposes fundamentally different BDD manipulation algorithms (based on BFS, not DFS), incorporates parallelism, a novel GC algorithm, and a number of other experimental optimizations. The work demonstrates an improvement over existing BDD packages, but does not show whether the improvement is due to improved cache friendliness or due to the fundamentally different algorithm. Meanwhile, [29] proposes to order the BDD nodes chronologically: i.e. the BDD node must appear in memory *after* both of its child nodes (this is trivially satisfied by the DFS post-order in which nodes are typically generated). This assumption then partially eliminates memory lookups that would be necessary with arbitrary node order. An improvement in runtime is demonstrated, but we are not aware of any modern work that uses this technique (the original paper is now 25 years old). Furthermore, we are not aware of any work that would directly *measure* the extent to which BDD operations are bottlenecked by memory or attempt to control for this aspect in the measurements.

A. Paper structure

First, Section II recalls the definition of ROBDDs and of the APPLY algorithm which ROBDDs use to perform logical transformations. Section III then describes our benchmark scenario involving a *modern* and a *legacy* CPU, together with the set of tested BDD operations and packages.

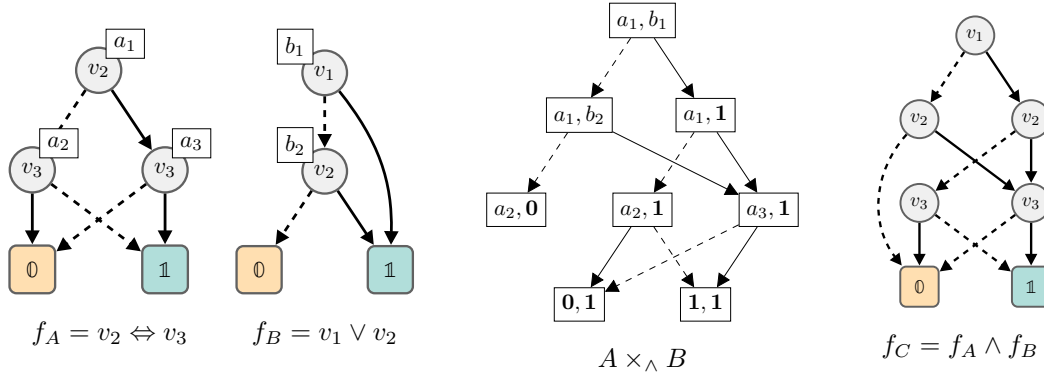


Fig. 1. Illustration of ROBDDs over the set $\mathcal{V} = \{v_1, v_2, v_3\}$. Solid edges represent the *high* successor while dashed edges represent the *low* successor. Left to right: ROBDDs of two simple functions $f_A = v_2 \Leftrightarrow v_3$ and $f_B = v_1 \vee v_2$. The product graph for the operation $\text{APPLY}_{\wedge}(a_1, b_1)$. The resulting ROBDD for the function $f_C = f_A \wedge f_B$.

Subsequently, Section IV-A demonstrates that for sufficiently large problem instances, the performance of a typical BDD package is in fact worse on the modern CPU. In Section IV-B, we then observe that BDD nodes with in-degree one are significantly over-represented in a typical BDD.

Using this observation, we define a new node table with improved memory locality (Section IV-C) and formulate a simple rule that eliminates a portion of redundant accesses to the operations cache (Section IV-D). Finally, we test the performance of this new approach in Section IV-E.

Due to the nature of this work, we also provide a reproducibility artefact¹ which contains all benchmark data and code, as well as the raw results for each experiment. To keep the main paper concise, some of the more low-level parts of the methodology and results are only available within the artefact.

II. PRELIMINARIES

Since there are already many excellent texts describing BDDs in detail, we only introduce the terminology and notation relevant for this paper. An interested reader is further referred for example to Chapter 7 of [14].

Notation: We assume \mathcal{V} is a finite set of Boolean variable symbols. We also use 1 and 0 interchangeably with *true* and *false* when appropriate.

A. Binary decision diagrams

A *binary decision diagram* (BDD) B is a directed acyclic graph with a single *root* node (denoted $\text{root}(B)$) and two *terminal* nodes 0 and 1. We write B to mean either the BDD itself or the set of its nodes when the distinction is clear from context. Each non-terminal node $x \in B$ is labelled with a variable $\text{var}(x) \in \mathcal{V}$. Furthermore, each such $x \in B$ has exactly two successor nodes denoted $\text{low}(x)$ and $\text{high}(x)$ (corresponding to the choice of $\text{var}(x) = 0$ and $\text{var}(x) = 1$ respectively). A simple example is given in Fig. 1, left.

We assign a Boolean function $f_x : \{0, 1\}^{\mathcal{V}} \rightarrow \{0, 1\}$ to each $x \in B$ s.t. $f_x = (\text{var}(x) \wedge f_{\text{high}(x)}) \vee (\neg \text{var}(x) \wedge f_{\text{low}(x)})$, with $f_0 = \text{false}$ and $f_1 = \text{true}$. In other words, every valuation of

variables from \mathcal{V} determines a path from x to either 0 or 1, corresponding to the output of the function f_x .

Now, let us assume there is *some* total ordering on the variables \mathcal{V} . We say that a BDD B is *ordered* (OBDD) when $\text{var}(\text{low}(x)) > \text{var}(x)$ and $\text{var}(\text{high}(x)) > \text{var}(x)$ for every non-terminal $x \in B$ (we also assume $\text{var}(0)$ and $\text{var}(1)$ are values greater than any $v \in \mathcal{V}$). For example, in Fig. 1, the variable ordering is $v_1 < v_2 < v_3$.

Finally, we say that B is *reduced* (ROBDD) when: (a) there is no vertex x such that $\text{low}(x) = \text{high}(x)$, and (b) there are no two vertices x and y such that $\text{var}(x) = \text{var}(y)$, $\text{low}(x) = \text{low}(y)$, and $\text{high}(x) = \text{high}(y)$. These two requirements can be also interpreted as “reduction rules” that describe how to transform an OBDD into an ROBDD. In the following, we assume all BDDs are ordered and reduced, we thus use the terms BDD and ROBDD interchangeably.

B. The APPLY algorithm

Given a fixed ordering of variables \mathcal{V} , each Boolean function $f : \{0, 1\}^{\mathcal{V}} \rightarrow \{0, 1\}$ has a unique corresponding ROBDD [9]. We also have an algorithm that, given two ROBDDs A and B , computes ROBDD C of the function $f_C = f_A \star f_B$ where \star is some binary Boolean operator. In the worst case, this APPLY algorithm operates in $\mathcal{O}(|A| \cdot |B|)$ time. However, the complexity for practical BDDs is typically much smaller than this upper bound.

In Algorithm 1, we give a recursive formulation of this APPLY procedure. In practice, one often replaces the recursion with a loop and an explicit stack to avoid overflow and to eliminate function call overhead. The APPLY algorithm also relies on two core data structures which are typically implemented using hash tables.

First is the *node table* (also called *unique table*) accessed using the $\text{ENSURE_NODE}(v, l, h)$ procedure. This function searches the node table for a node x with $\text{var}(x) = v$, $\text{low}(x) = l$, and $\text{high}(x) = h$. If such node is found, its identifier x is returned. When no such node exists, a new node is created and its identifier is returned.

Second is the *cache table* responsible for memorisation of already computed results. This table is often implemented as a

¹<https://doi.org/10.5281/zenodo.7958052>

```

1 Function APPLY $\star$ ( $x_A \in A, x_B \in B$ )
2   if  $x_A \star x_B \in \{0, 1\}$  then return  $x_A \star x_B$ ;
3   if let  $x_C \leftarrow \text{CACHE}(x_A, x_B)$  then return  $x_C$ ;
4    $v \leftarrow \min(\text{var}(x_A), \text{var}(x_B))$ ;
5    $(l_A, h_A) \leftarrow (x_A, x_A)$ ;
6   if  $v = \text{var}(x_A)$  then
7      $(l_A, h_A) \leftarrow (\text{low}(x_A), \text{high}(x_A))$ ;
8    $(l_B, h_B) \leftarrow (x_B, x_B)$ ;
9   if  $v = \text{var}(x_B)$  then
10     $(l_B, h_B) \leftarrow (\text{low}(x_B), \text{high}(x_B))$ ;
11   $l \leftarrow \text{APPLY}_\star(l_A, l_B)$ ;
12   $h \leftarrow \text{APPLY}_\star(h_A, h_B)$ ;
13   $x_C \leftarrow \text{ENSURE\_NODE}(v, l, h)$  if  $l \neq h$  else  $l$ ;
14   $\text{CACHE}(x_A, x_B) \leftarrow x_C$ ;
15  return  $x_C$ ;

```

Algorithm 1: BDD APPLY algorithm parametrised by a binary Boolean operator \star .

leaky hash table which overwrites values when hash collision occurs. Such implementation is correct (if a value is missing, it is simply recomputed), but depending on the number of collisions, it may exceed the $\mathcal{O}(|A| \cdot |B|)$ time complexity. This introduces a possible trade-off between running time and memory consumption.

Finally, let us observe that for every operation $\text{APPLY}_\star(\text{root}(A), \text{root}(B))$, there is a tighter complexity metric given by the number of unique (x_A, x_B) pairs reachable from $(\text{root}(A), \text{root}(B))$ by APPLY_\star . We call this set of tuples the *product graph* of $\text{APPLY}_\star(\text{root}(A), \text{root}(B))$ and denote it $A \times_\star B$. Note that the size of this product graph depends on \star , because some operations can short-circuit the condition on Line 2 even when one of the arguments is not a terminal node (e.g. $x_A \wedge 0 = 0$). We then observe that the complexity of the APPLY algorithm is $c \cdot |A \times_\star B|$ for some constant c , assuming the calls to CACHE and ENSURE_NODE are $\mathcal{O}(1)$ and that CACHE is not leaky. Observe that for the example in Fig. 1, we have $|A| \cdot |B| = 20$, but $|A \times_\wedge B| = 8$.

III. BENCHMARK METHODOLOGY AND HARDWARE

Due to the practical nature of this paper, we must thoroughly disclose what benchmarks are performed and how we measure the performance of BDD packages on our hardware.

A. Benchmark BDDs

Many authors test the performance of BDDs on pathological worst case scenarios like the multiplier circuit or the n -queens problem [11], [33]. While this certainly reveals some performance characteristics of the implementation, it is susceptible to over-fitting of a particular pattern of BDD operations. To mitigate this issue, we derive a large benchmark dataset based on real-world problems from model verification in systems biology, scaling from simple BDDs to millions of nodes.

Specifically, we use the tool AEON [4] which performs exhaustive formal analysis of Boolean networks, simple logical models of asynchronous biological processes. We then take

the 20 largest models from the Biodivine Boolean Models (BBM) dataset [36], ranging from 100 to 300 variables (and consequently, $2^{100-300}$ states).

We use AEON to compute the BDD representations of a set of network fixed-points and a set of reachable states based on a predefined initial state for each model. These are tasks that are also commonly performed by formal methods tools in computer science and are not specific to Boolean networks or systems biology. For each computation, we save every intermediate BDD smaller than ten million nodes into a separate file. If two different BDDs of equivalent size are encountered, we only retain the latest BDD. This generates a large dataset of realistic BDDs of increasing size.

Now, our goal is to define a set of benchmarks which cover the space of admissible BDD operations over these real-world BDDs as uniformly as possible. Each BDD B can be assigned a *bucket* $b(B) = \log_{10}(|B|)$ (the “order of magnitude” of the size of B). We then sample pairs of BDDs A, B (w.l.o.g. we assume $|A| \geq |B|$) and compute the BDD $C = A \wedge B$. Such *benchmark triple* (A, B, C) is then assigned into a bucket triple $(b(A), b(B), b(C))$. For each bucket triple, we save the first five unique benchmarks. The sampling stops once no new viable benchmark is found in the last 100 samples.

In our case, this process yields 629 benchmark instances using 963 unique BDDs. The final number of triples for each combination of buckets is summarized in Fig. 2. While the result does not cover every theoretically admissible combination of BDD sizes, it still covers a wide range of possible BDD operations. When presenting results for individual benchmarks, these are typically sorted by the size of the product graph $A \times_\wedge B$, as this gives a good approximation of the expected complexity of the BDD operation.

Finally, note that for the sake of simplicity, our tests only cover the conjunction (\wedge) operator. However, we have no reason to believe that there are significant differences in performance compared to other Boolean operators once the size of the product graph is taken into account. As such, we prioritize a wider coverage of different BDD sizes to testing more Boolean operators.

B. Hardware configuration

To compare “modern” and “legacy” CPUs, we consider the following two platforms:

- 4-core Intel i7-4790 (released in 2014) with 32GB of DDR3-1600 memory at CAS latency of 9 cycles.
- 8-core AMD Ryzen 5800X (year 2020) with 128GB of DDR4-3200 memory at CAS latency of 18 cycles.

These are both very common CPUs from their respective generations. They are paired with the maximum amount of memory available on that platform at the top speed officially supported by the manufacturer². Furthermore, notice that the

²The data rate is maximal supported on both CPUs. However, in terms of latency, the DDR3 configuration is slightly worse than the best official JEDEC configuration (9 cycles instead of 8 for DDR3-1600), while the DDR4 configuration is slightly better than the best official JEDEC configuration (18 cycles instead of 20 for DDR4-3200). Hence the modern system even has a *small* advantage compared to the officially claimed “best” configurations.

A	B	A ∧ B								
		10 ¹	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶	10 ⁷	10 ⁸	10 ⁹
10 ¹	10 ¹	5	5	—	—	—	—	—	—	—
10 ²	10 ¹	1	5	5	—	—	—	—	—	—
10 ²	10 ²	0	5	5	5	—	—	—	—	—
10 ³	10 ¹	0	0	5	5	—	—	—	—	—
10 ³	10 ²	5	0	5	5	1	—	—	—	—
10 ³	10 ³	5	0	5	5	0	0	—	—	—
10 ⁴	10 ¹	0	0	1	5	5	—	—	—	—
10 ⁴	10 ²	4	0	5	5	5	1	—	—	—
10 ⁴	10 ³	5	0	5	5	4	0	0	—	—
10 ⁴	10 ⁴	5	0	5	5	5	3	0	0	—
10 ⁵	10 ¹	0	0	0	0	5	4	—	—	—
10 ⁵	10 ²	0	0	5	5	5	5	1	—	—
10 ⁵	10 ³	5	0	5	5	5	5	0	0	—
10 ⁵	10 ⁴	5	0	5	5	5	5	4	0	0
10 ⁵	10 ⁵	5	0	2	5	5	5	5	1	0
10 ⁶	10 ¹	0	0	0	0	5	5	5	—	—
10 ⁶	10 ²	1	0	5	5	5	5	5	0	—
10 ⁶	10 ³	5	0	5	5	5	5	4	1	0
10 ⁶	10 ⁴	5	0	5	5	5	5	5	0	0
10 ⁶	10 ⁵	5	0	1	5	5	5	5	1	1
10 ⁶	10 ⁶	5	0	4	4	5	5	5	5	0
10 ⁷	10 ¹	0	0	0	0	0	1	5	2	—
10 ⁷	10 ²	5	0	5	5	5	5	5	0	0
10 ⁷	10 ³	5	0	5	5	5	5	5	1	1
10 ⁷	10 ⁴	5	0	5	5	5	5	5	1	1
10 ⁷	10 ⁵	5	0	0	5	5	5	5	5	0
10 ⁷	10 ⁶	5	0	0	0	5	5	5	5	5
10 ⁷	10 ⁷	5	0	0	1	3	5	5	5	5

Fig. 2. The distribution of the 629 benchmarks within buckets of exponentially increasing size. Dashes indicate combinations that are provably impossible.

effective latency of both memory configurations is the same: the DDR4 configuration has twice the CAS latency, but also twice the data rate of the DDR3 configuration³.

All automated overclocking features were disabled on both CPUs to improve consistency between runs and we did not observe any thermal throttling. Furthermore, we assume that no other programs were using a significant amount of resources during measurements. This is critical due to the fact that multiple CPU cores compete for the shared L3 cache.

Finally, as a sanity check, some tests were also repeated on a similar server hardware (Intel Xeon E7-8860; released in 2011, and AMD EPYC 7713; released in 2021) yielding comparable results. However, since we did not have exclusive access to these machines and thus could not prevent measurement noise caused by sharing resources with other software, we focus on the numbers obtained for the “desktop” platforms.

C. BDD packages and the benchmark harness

Our implementation is built using the Rust programming language. In several instances, we use *unsafe* operations in Rust to remove unnecessary array bounds check in the core algorithm. Aside from these instances, the memory safety of the implementation has been validated by the Rust compiler. Testing was performed using Debian 12 with `gcc 12.2.0`

³Currently, the fastest officially supported memory configuration in consumer CPUs is roughly DDR5-5600 CL32 (faster configurations do exist but require CPU overclocking and are typically not feasible with high amounts of memory). Unfortunately, we did not have access to such configuration. However, its effective latency is again very similar to our tested scenarios.

and `rustc 1.71.0`. Each measurement was performed over at least three runs. When we observed standard deviation higher than 5% of the average, we repeated the experiment up to ten times to improve reliability. However, this was only rarely necessary. Due to this low run-to-run variance, we only report the average runtime for each experiment.

Aside from our purpose-built implementation, we consider the following three BDD packages:

- `cudd 3.0` [39] as one of the best known BDD packages. While originally designed over 25 years ago, CUDD is still one of the most widely used BDD packages.
- `sylvan` [41] is one of the first BDD packages to demonstrate practical multi-core scalability. When presenting results, we suffix its name with the number of employed cores (e.g. `sylvan-4`).
- `lib-bdd` of the tool AEON [4] is an example of a “naive” BDD implementation: it uses hash tables provided by Rust’s standard library and generally does not implement any advanced features like variable reordering.

To compare performance, we focus on individual BDD operations, i.e. on the individual runs of the APPLY algorithm with the conjunction (\wedge) operator. For each package we prepared a test harness where the runtime of the operation is isolated from other overhead such as package initialization and loading of test BDDs into memory. Where applicable, we disable garbage collection or dynamic variable reordering, as it is not relevant for our testing.

Furthermore, the performance of BDD packages often strongly depends on the initial size of the node and cache table [40]. While the packages can grow these data structures dynamically, the combination of the problem size and the growth rate can influence runtime significantly [40].

To reduce the impact of this variable on the final runtime, we allow each package to pre-allocate as much memory as possible during initialization (not counted towards the total runtime)⁴. While this can impact performance on smaller BDDs, it seems necessary to allow each package to reach its full potential on larger benchmarks and is outright mandatory in some instances.⁵ Furthermore, this models the situation where our single benchmarked BDD operation is part of a larger symbolic computation which amortizes the allocation of the necessary data structures across many operations.

Finally, when reporting the *average* value for additive metrics (like runtime), this stands for the standard arithmetic mean. Meanwhile, for multiplicative metrics (like speed-up), this corresponds to the geometric mean which is more appropriate in such instances. We also use internal CPU performance counters to measure executed instructions per clock and L3

⁴Package `lib-bdd` cannot perform any pre-allocation and thus its memory allocation counts towards the total runtime. Also note that while each package is allowed to *reserve and initialize* as much memory as needed, it can still choose to initially use a smaller portion and grow the hash tables gradually. In particular, `sylvan` grows the hash table utilization gradually, while `cudd` always uses the whole table from the start.

⁵In `cudd`, the growth is extremely slow beyond the first few gigabytes of memory, making it practically unusable unless we explicitly override the table sizes to reserve as much memory as necessary beforehand.

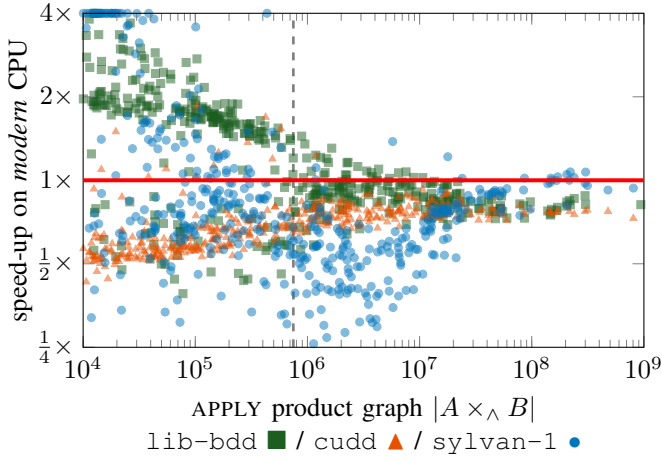


Fig. 3. The speed-up in runtime of the *modern* vs. the *legacy* system. Speed-up greater than $4\times$ is truncated to $4\times$. Points below the red line represent slow-down instead of speed-up. Points to the right of the vertical line represent 200 largest benchmark instances. Note that both axis are logarithmic.

cache miss ratio. However, their usage has negligible impact on runtime.⁶

IV. ALGORITHMS AND RESULTS

We structure this section as follows: First, we present several empirical findings about BDDs and the current APPLY implementations. Based on these findings, we then propose a variant of the node and cache table which should help improve performance on modern CPUs. Finally, we evaluate these claims empirically on our benchmark dataset.

A. Comparing modern and legacy systems

While it is generally *known* that the performance of BDD operations is bottlenecked by the memory latency, it is useful to demonstrate the extent of this problem. To do so, we benchmarked each BDD package on both the *legacy* and the *modern* system, calculating the relative *speed-up* achieved by the modern system. Note that for this test, we used the same pre-allocation settings on both systems and excluded six benchmarks that we could not reliably complete on the legacy system due to insufficient memory. The results of this analysis are presented in Fig. 3.

If we consider the full benchmark suite, the results appear to be largely positive: For *lib-bdd* and *sylvan-1*, two packages that gradually increase their table sizes with benchmark size, we see $1.52\times$ and $1.78\times$ average improvement, respectively. Only for *cudd*, which was instructed to use the maximal table size for each benchmark due to issues with table growth, we actually observe a slow-down of $0.69\times$. However, once we focus on the 200 largest benchmarks, every package is actually slower compared to the legacy system: a *lib-bdd* operation is $0.88\times$ slower on a modern system,

⁶Performance counters are thread-local and thus could not be reliably used for the multithreaded BDD package *sylvan*, even with one worker.

cudd operation is $0.77\times$ slower, and a *sylvan-1* operation is even $0.61\times$ slower.

From Fig. 3, it is clear that small benchmarks benefit the most from the modern CPU (except for *cudd*), which aligns with our assumption that additional L3 cache on the modern CPU improves BDD performance. In case of *cudd*, the reason for the poor performance on the smaller problem instances is the table growth setting: When the hash tables grow gradually, they maintain high density and thus utilise the available cache lines well. However, when the table is set to its maximal size from the start, it is very sparse for small problems. As such, each cache line will typically store only one element, making it much less effective. Intuitively, for this *cudd* configuration, every benchmark behaves like a “large” benchmark and it further amplifies the difference in practical latency of the modern and legacy system.

Also note that the relatively good results of *lib-bdd* compared to the other packages can be primarily attributed to its naive architecture: Each BDD operation in *lib-bdd* performs *more* instructions to accomplish the same task compared to the other packages (evidence for this is given in the subsequent text), which leaves the CPU more headroom for optimization and reordering while waiting for memory. In other words, *lib-bdd* achieves the best speed-up on large problems because it is the slowest, most inefficient package with the most space to improve.

To support these claims experimentally, we use CPU performance counters to measure the number of CPU cycles, executed instructions, L3 cache references and L3 cache misses for each benchmark. Interestingly, both *cudd* and *lib-bdd* seem to perform approx. 22 L3 cache references for each node of the APPLY product graph on average. This number is also independent on the size of the product graph. Similarly, the absolute number of executed instructions per product graph node is also largely constant, but here *cudd* is consistently almost $2\times$ better than *lib-bdd*. Both implementations thus perform as expected: their complexity in terms of instructions is in fact $c \cdot |A \times_A B|$, with $c_{\text{cudd}} < c_{\text{lib-bdd}}$.

However, for the 200 largest problem instances, the average IPC (instructions per cycle) of both implementations is very low: 0.35 for *cudd* and 0.85 for *lib-bdd*⁷. This is because both implementations miss almost 40% of the L3 cache requests while an L3 cache reference occurs on average every 14 (*cudd*) and 32 (*lib-bdd*) instructions.

In theory, the modern CPU can achieve up to 4 IPC, and we actually observed 3.49 IPC on some small problem instances. At the same time, the worst IPC observed on the large benchmarks was just 0.18. Intuitively, this means that for the larger problem instances, the CPU is utilizing *less than 10%* of its available compute resources. Furthermore, the results indicate that any speed-up between the modern and the legacy system occurs for problem instances that fit into (or are close to) the available L3 cache.

⁷Note that higher IPC generally does not guarantee higher performance. While *lib-bdd* achieves higher IPC, it also performs more instructions and is overall $0.84\times$ slower than *cudd* in these tests.

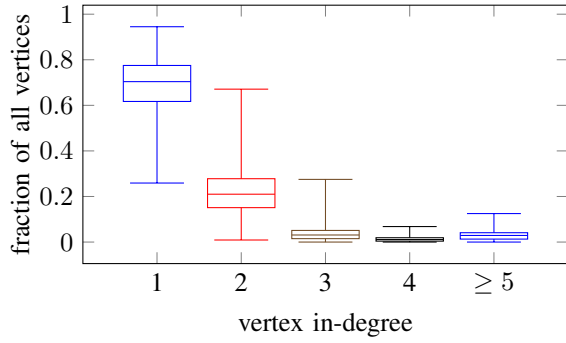


Fig. 4. The in-degree distribution within the BDDs of our benchmark dataset larger than 1000 nodes. Each box plot summarises the proportion of vertices with the corresponding in-degree across all available benchmark BDDs. Last box plot covers all in-degrees greater than or equal to five.

B. In-degree distribution in BDDs

Next we state a very simple but powerful observation: Due to each non-terminal node having exactly two outgoing edges, a BDD B has $2 \cdot (|B| - 2)$ edges. Furthermore, each node aside from root has at least one *incoming* edge. As such, for every node with in-degree $k > 2$, we expect to have $k - 2$ nodes with only one incoming edge. This leads us to an intuitive hypothesis that single-parent nodes are greatly overrepresented among BDD nodes.

We can easily verify that this property holds in our benchmark dataset. In Fig. 4, we show the portion of nodes for individual in-degrees, cutting off at ≥ 5 incoming edges. As expected, our observations strongly resemble an exponential distribution: if we draw a random BDD node, there is roughly a 70% chance the node has a single parent, 20% chance it has two parents, 3% chance of it having three parents, and so on.

C. Parent-local node table design

We can now use the previous claim to design a new node table which is simpler and more cache friendly than current implementations. For simplicity, we assume that the node table stores entries in a continuous array, with new entries simply added to the end of the node list. This mirrors the assumption form [29], but as we later show, the node ordering is not a strict requirement for our approach. To resolve an ENSURE_NODE query, each node maintains a pointer structure akin to a *prefix trie* consisting of a subset of its parent nodes.

We know that due to the in-degree distribution, the vast majority of such tries will only store a handful of elements. Furthermore, due to the recursive nature of APPLY, whenever ENSURE_NODE(v, l, h) is executed, it is likely that the entries for nodes l and h are still in the CPU cache.

The pseudocode for this approach is given in Algorithm 2. Here, CREATE allocates a new table slot for the given node and returns the slot identifier. MSB stands for *most significant bit*, and ROTATE_ONE is a simple single-bit left rotation. Expression $node(t)$ is a shorthand for $(var(t), low(t), high(t))$.

Finally, for each node x , we introduce three additional identifiers which are all initialised to *nil* and stored together

```

1 Function ENSURE_NODE( $v, l, h$ )
2    $(c_{max}, c_{min}) \leftarrow (max(l, h), min(l, h));$ 
3    $hash \leftarrow HASH(v, c_{min});$ 
4    $t \leftarrow parent(c_{max});$ 
5   if  $t$  is nil then
6      $x \leftarrow CREATE(v, l, h);$ 
7      $parent(c_{max}) \leftarrow x;$ 
8     return  $x;$ 
9   loop
10    if  $(v, l, h) = node(t)$  then return  $t;$ 
11     $msb \leftarrow MSB(hash);$  //  $msb \in \{0, 1\}$ 
12     $hash \leftarrow ROTATE\_ONE(hash);$ 
13    if  $next_{msb}(t)$  is nil then
14       $x \leftarrow CREATE(v, l, h);$ 
15       $next_{msb}(t) \leftarrow x;$ 
16      return  $x;$ 
17    else
18       $t \leftarrow next_{msb}(t);$ 

```

Algorithm 2: The ENSURE_NODE procedure that resolves node duplicates through a parent-local trie.

with the node data in the node table: $parent(x)$, $next_0(x)$, and $next_1(x)$. Here, $parent(x)$ is the reference to the root of the prefix trie. This trie then references all nodes where x is the maximal child (see Lines 2-4). Subsequently, $next_0(x)$ and $next_1(x)$ reference the “successor” nodes within the trie. Which successor is taken depends on the MSB prefix of the node hash (see Lines 11-12).

We’d like to highlight several properties of Algorithm 2:

- We choose c_{max} to store the node because it avoids terminal nodes, which typically have very high in-degrees. However, other suitable conditions could be considered, as long as they only depend on the values (v, l, h) .
- We do not include c_{max} in the node hash. Furthermore, the hash need not be truncated to a specific node table length, which further simplifies the hash function⁸.
- Consequently, our HASH function is a simple multiplicative hash $(v \mathbf{xor} c_{min}) \cdot p$ where p is a large prime number and the multiplication is natively truncated to 64 bits.
- If the table grows such that it needs to be completely re-allocated, the existing nodes can be simply copied: there is no need to relocate nodes or recompute hashes.

To test the viability of this approach in practice, we prepare a simple benchmark that recreates all of our test BDDs one-by-one: first in a standard hash map with quadratic probing and a fast industry-standard hash function, and then in our parent-local node table. The nodes are created in DFS post-order, i.e. in the same order as within the APPLY algorithm. Averaging all BDDs, the parent-local table is $2.38\times$ faster. However, on BDDs with at least one million nodes, the parent-

⁸Note that internally, a $x \bmod y$ operation (except for $x \bmod 2^k$) translates to division, which is still a relatively costly operation even on modern CPUs. A single division instruction may require as many CPU cycles as the whole ENSURE_NODE method if the required data is present in cache.

local approach is even $4.08\times$ faster than the standard hash table, with an L3 cache miss rate of just 18% vs. 27% for the standard hash table. Keep in mind that these results do not account for the rest of the APPLY algorithm. We will revisit this aspect in further experiments.

D. Excluding single-parent tasks from CACHE

Unfortunately, we cannot apply the same principle to the CACHE table, because its entries (i.e. the nodes of the product graph $A \times_\star B$) are queried in DFS pre-order, unlike the output BDD nodes which are queried in DFS post-order. This means that when we first query a particular entry (x_A, x_B) , we do not have any information about its child nodes yet.

However, the observation about the number of single parent nodes nevertheless applies to the product graph as well. That is, for the majority of (x_A, x_B) pairs explored by the APPLY algorithm, there is a single parent (y_A, y_B) through which (x_A, x_B) is discovered.

Proposition 1: Let (x_A, x_B) be a node of the product graph $A \times_\star B$ such that it has a single parent node (y_A, y_B) . Then it is unnecessary to store the result for (x_A, x_B) in CACHE as long as (y_A, y_B) is saved properly.

Intuitively, every time (x_A, x_B) is visited by the APPLY algorithm, it is through the node (y_A, y_B) . As such, once the result for (y_A, y_B) is saved in the cache, (x_A, x_B) is never visited again. Consequently, due to the distribution of node in-degrees, we know that the majority of nodes in the product graph do not actually need to be stored in CACHE.

However, this proposition does not tell us how to detect these “redundant” cache entries. Furthermore, detecting all such entries appears to be a hard problem: when a product graph node is first visited, we do not know if it *can* be visited again from some other source. If we mistakenly exclude it from CACHE, it could result in re-computation of a non-trivial portion of the product graph.

Proposition 2: Assume x_A has a single parent in BDD A and x_B has a single parent in BDD B . Then (x_A, x_B) has a single parent in any product graph $A \times_\star B$.

While this certainly does not cover *all* redundant entries, such observation gives us a way of eliminating at least *some* of the redundant work. To implement it, we only need to maintain a simple 2-bit $\{0, 1, \text{many}\}$ parent counter for each BDD node. It is then easy to verify the conditions of our proposition during each CACHE access and to skip any unnecessary queries.

In our benchmark dataset, this criterion leads to an average 22% reduction in the number of product graph nodes stored in CACHE. However, we should note that this method is quite uneven: the eliminated node ratio ranges from more than 90% to less than 1% depending on the BDD. Nevertheless, due to its low overhead, we still consider it a worthwhile improvement.

In the future, it is possible to explore additional heuristics that eliminate a higher percentage of single-parent nodes from the operations cache.

Other cache table considerations: Aside from the aforementioned improvements, we use a relatively standard cache table design: (1) we overwrite results on collision; (2) we grow

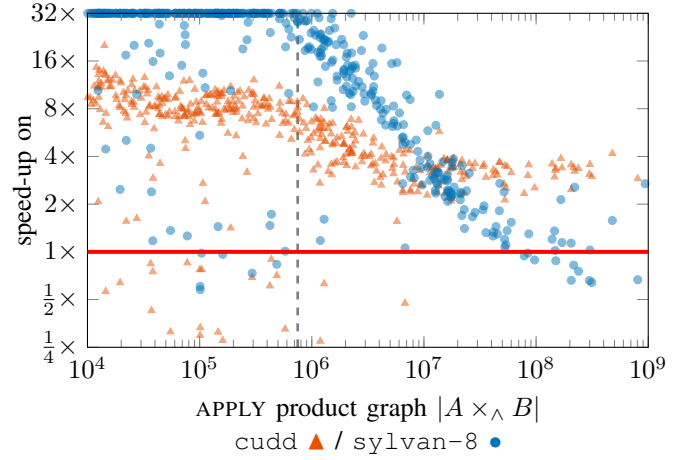


Fig. 5. The speed-up in runtime of our implementation compared to `cudd` and `sylvan-8`, truncated to $32\times$. Points below the red line represent slow-down instead of speed-up. Points to the right of the vertical line represent 200 largest benchmark instances. Note that both axis are logarithmic.

the table in exponents of two once the number of insertions exceeds the current length; (3) to determine the cache slot, we use $\log_2(\text{length})$ most significant bits of the same prime multiplicative hash as for the node table, but the input is a single integer concatenation of x_A and x_B .

Note that when this hash is truncated to the same bit-length as the input, it is in fact *perfect*: it maps each input to a unique output [31]⁹. As such, we only need to save the hash of (x_A, x_B) instead of the whole key. This does not necessarily reduce memory consumption, but there is no need to recompute the hash during table growth. Furthermore, when the table grows, this hash has a very predictable node placement since the new slot is a single-bit extension of the previous slot. Consequently, similar to the node table, growing the cache is essentially a memory copy operation.

E. Performance evaluation

1) State-of-the-art BDD packages: To evaluate the performance of this approach, we compare the implementation to the `cudd` and `sylvan-8` runtime on our modern CPU (we omit `lib-bdd` as it is largely superseded by at least one of the methods on every benchmark). The effective speed-up is shown in Fig. 5. If we focus on the 200 largest benchmarks, our implementation achieves on average a $3.70\times$ speed-up compared to `cudd` and $5.69\times$ speed-up compared to `sylvan-8`. However, we see that especially for `sylvan`, the improvements are diminishing as the benchmark size grows.

Therefore, we also investigate the 10 largest benchmarks. Here, `sylvan-8` is sometimes faster than our implementation, however our method is still on average $1.03\times$ as fast as `sylvan-8`. Note that these are tasks that all consume at least 16 GBs and sometimes more than 32 or 64 GBs

⁹This hash is *similar* and often confused with the notorious *Knuth multiplicative hash* [25] and the so-called *binary multiplicative hash* [17]. However, these do not use a prime as the multiplier and are consequently not perfect [30].

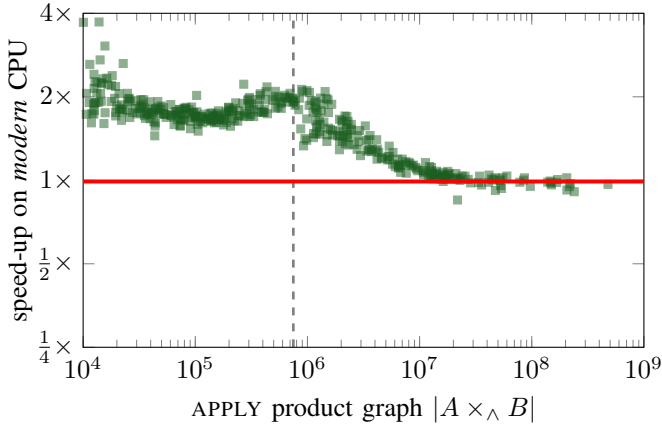


Fig. 6. The speed-up in runtime on the *modern* vs. the *legacy* system for our implementation. Points below the red line represent slow-down instead of speed-up. Points to the right of the vertical line represent 200 largest benchmark instances. Note that both axis are logarithmic.

of system memory over hundreds of millions of product graph nodes. This means that our implementation is better on small and medium tasks while achieving roughly comparable performance on very large tasks.

Furthermore, note that our implementation is always faster than *sylvan-4* (not shown in the figure), and on average $1.5\times$ faster on the 10 largest tasks. We also note that in our testing, *sylvan-4* was on average $2.27\times$ (up to $3.9\times$) faster and *sylvan-8* on average $3.22\times$ (up to $6.3\times$) faster than *sylvan-1* for the 200 largest benchmarks. While this is not perfect scaling, it appears to be roughly in line with previous reported results for *sylvan* [41].

We also investigated the performance counters to explain this improvement. For the 200 largest instances, we see average IPC of 1.26 (as opposed to *cudd*’s 0.35), while the L3 cache miss rate is only 27% compared to *cudd*’s 40%. Furthermore, our implementation needs on average only 12 L3 cache references per product graph node compared to *cudd*’s 21, and the number of instructions executed between such references increased to 42 from 14 on average. In other words, our implementation performs more work with fewer requests to the main memory, just as we were trying to achieve. For the top 10 benchmarks, these numbers are less ideal, with just 0.73 IPC and 32% cache miss rate, but this is still quite enough to achieve a sizeable improvement over *cudd*.

2) *Modern hardware*: Next, we investigate whether we achieved our initial goal. That is, whether we improved the scalability of the BDD APPLY algorithm on modern hardware. As such, we repeat the experiment from Section IV-A with our new implementation. The results are shown in Fig. 6, which is directly comparable to Fig. 3.

Here, we see that the speed-up is again diminishing with growing BDD size. However, we also see that the overall improvement is greater and more predictable compared to the other implementations. In particular, while we see a slight slow-down in some of the experiments, it is generally within

the 5% measurement noise tolerance established earlier. For the whole dataset, we see an average speed-up of $1.72\times$, which is comparable to *lib-bdd* and *sylvan-1*. However, for the 200 largest benchmarks, we still have a speed-up of $1.26\times$, as opposed to slow-down for *lib-bdd* ($0.88\times$) and *sylvan-1* ($0.61\times$). Finally, zooming in on the 10 largest benchmarks, we have a small average slow-down of $0.95\times$, which is again (barely) within our 5% measurement tolerance.

3) *Memory consumption and memory layout*: Due to the prototype nature of our implementation, we have not thoroughly evaluated the memory consumption yet. However, we observed no significant differences between the packages when the memory was limited to 32GB. Specifically, each package ran out of memory on roughly the same handful of largest benchmark problems. We thus do not consider our method to be significant advantaged nor disadvantaged in this regard.

Finally, we should stress that in the presented setting, our method supersedes [29]. It benefits from the temporal ordering of BDD nodes, but further reduces the number of necessary memory accesses and collisions through the use of the prefix trie. However, while our method also *benefits* when the in-memory ordering of BDD nodes matches the temporal ordering, the in-memory ordering is not strictly required: as opposed to [29], the critical aspects of the temporal ordering are essentially stored in the prefix tries.

This begs the question: How important is the in-memory node ordering for our method? To test this, we prepared an experiment where we compare our method on BDDs pre-processed with three possible in-memory orderings: DFS pre-order, DFS post-order, and randomly shuffled.

We find that when comparing pre-order and post-order, pre-order is slightly faster, but the difference is $< 5\%$ and diminishes with increasing benchmark size. However, comparing post-order and shuffled node ordering, we observe that post-order is on average 26% faster, and almost 40% faster for the 200 largest benchmarks. However, this lead then diminishes for the largest 10 queries, where it is less than 10%, suggesting that the ordering is the most important for medium-sized BDDs that are “close” to the L3 cache capacity.

Hence we see that the improvements of our method are not completely dependent on the in-memory layout of the BDD nodes. The method is still better than *sylvan-4*, but not as good as *sylvan-8* for the largest benchmarks. However, the choice of memory ordering does measurably influence the outcome. Consequently, this information can inform the implementation of garbage collection algorithms for BDDs. It is not uncommon for garbage collection methods to reorder objects to improve memory locality [23]. Our results thus show this to be an important consideration for BDDs.

V. CONCLUSION

In this paper, we demonstrated the impact of memory latency on the performance of BDD packages. Specifically, we show that a more “modern” CPU does not necessarily guarantee improved performance once the size of the problem no longer fits into the L3 cache of the CPU.

We then proposed improvements to the node and cache table used within the APPLY algorithm with the goal of increasing locality and reducing the number of memory accesses overall. We demonstrate that with these improvements, our implementation significantly outperforms classical BDD packages like *cudd*, and is better or comparable to parallelization of the same task to 8 CPU cores using the package *sylvan*. Importantly, we also show that it is the only implementation in our testing that exhibits a consistent improvement in performance when comparing a modern and a legacy CPU.

However, we should stress that the results of this paper do not argue *against* parallelization of BDD operations. We simply use parallelism as a meaningful comparison that gives an alternative way of speeding up BDD operations. In fact, we believe that similar performance benefits can also translate to parallel BDD algorithms if appropriate data structures are developed based on our observations.

Additionally, we should note that the presented approach does not in any way prevent dynamic variable reordering: to swap two adjacent variables, we can swap the BDD nodes in-place, such as in [19]. We then update the parent-local trie for each affected node accordingly. This requires a delete operation on the trie structure, but such operation does not differ from deletion on normal tries.

Finally, we observe that latest hardware developments open new interesting propositions for improving BDD performance. First, several new CPUs utilize silicon die stacking or other advanced packaging techniques to significantly increase the L3 cache capacity or add another layer of L4 cache.

Second, we are currently experiencing a resurgence of task-specific hardware in the field of statistical machine learning and a great increase in capabilities of field programmable gate arrays (FPGAs). Perhaps an in-hardware APPLY implementation tuned for out-of-order exploration of the product graph can be designed such that it sufficiently hides the large latency of modern random access memory.

ACKNOWLEDGMENTS

This work was supported by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 101034413 and the “VAMOS” grant ERC-2020-AdG 101020093.

REFERENCES

- [1] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44, 2010.
- [2] Junaid Babar, Gianfranco Ciardo, and Andrew Miner. CESRBDDs: binary decision diagrams with complemented edges and edge-specified reductions. *International Journal on Software Tools for Technology Transfer*, 24(1):89–109, 2022.
- [3] Jiří Barnat, Jakub Chaloupka, and Jaco Van De Pol. Distributed algorithms for SCC decomposition. *Journal of Logic and Computation*, 21(1):23–44, 2011.
- [4] Nikola Beneš, Luboš Brim, Jakub Kadlec, Samuel Pastva, and David Šafránek. AEON: attractor bifurcation analysis of parametrised Boolean networks. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I* 32, pages 569–581. Springer, 2020.
- [5] Nikola Beneš, Luboš Brim, Samuel Pastva, and David Šafránek. Computing bottom SCCs symbolically using transition guided reduction. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I* 33, pages 505–528. Springer, 2021.
- [6] Nikola Beneš, Luboš Brim, Samuel Pastva, and David Šafránek. Symbolic coloured SCC decomposition. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II* 27, pages 64–83. Springer, 2021.
- [7] David Bergman, Andre A Cire, Willem-Jan Van Hove, and John Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.
- [8] Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991.
- [9] Randal E Bryant. Graph-based algorithms for Boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.
- [10] Randal E Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 236–243. IEEE, 1995.
- [11] Luigi Capogrosso, Luca Geretti, Marco Cristani, Franco Fummi, and Tiziano Villa. HermesBDD: A multi-core and multi-platform binary decision diagram package. *arXiv preprint arXiv:2305.00039*, 2023.
- [12] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Alexander Svozil. Symbolic time and space tradeoffs for probabilistic verification. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021.
- [13] ChipsAndCheese. AMD’s 7950X3D: Zen 4 gets VCache. <https://chipsandcheese.com/2023/04/23/amds-7950x3d-zen-4-gets-vcache/>, Apr 2023. Accessed: 2023-05-01.
- [14] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.
- [15] Mahmoud Elbayoumi, Michael S Hsiao, and Mustafa ElNainay. A novel concurrent cache-friendly binary decision diagram construction for multi-core platforms. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1427–1430. IEEE, 2013.
- [16] Stefan Ellmauthaler, Sarah Alice Gaggli, Dominik Rusovac, and Johannes P Wallner. Representing abstract dialectical frameworks with binary decision diagrams. In *Logic Programming and Nonmonotonic Reasoning: 16th International Conference, LPNMR 2022, Genova, Italy, September 5–9, 2022, Proceedings*, pages 177–189. Springer, 2022.
- [17] Jeff Erickson. *Algorithms*. 2019.
- [18] Eric Felt, Gary York, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Dynamic variable reordering for BDD minimization. In *Proceedings of EURO-DAC 93 and EURO-VHDL 93-European Design Automation Conference*, pages 130–135. IEEE, 1993.
- [19] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54. IEEE, 1991.
- [20] Daochuan Ge, Meng Lin, Yanhua Yang, Ruoxing Zhang, and Qiang Chou. Quantitative analysis of dynamic fault trees using improved sequential binary decision diagrams. *Reliability Engineering & System Safety*, 142:289–299, 2015.
- [21] Justin E Harlow III and Franc Brglez. Design of experiments in BDD variable ordering: Lessons learned. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 646–652, 1998.
- [22] Tobias Heß, Chico Sundermann, and Thomas Thüm. On the scalability of building binary decision diagrams for current feature models. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*, pages 131–135, 2021.
- [23] Xianglong Huang, Stephen M Blackburn, Kathryn S McKinley, J Eliot B Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. *ACM SIGPLAN Notices*, 39(10):69–80, 2004.
- [24] Martin Jonáš and Jan Strejček. Solving quantified bit-vector formulas using binary decision diagrams. In *Theory and Applications of Satisfiability Testing—SAT 2016: 19th International Conference, Bordeaux, France, July 5–8, 2016, Proceedings 19*, pages 267–283. Springer, 2016.

- [25] Donald Ervin Knuth. *The art of computer programming, volume 3: Sorting and searching*, volume 3. Pearson Education India, 1973.
- [26] Lukas Kohutka and Peter Pistek. Faster synthesis of combinational logic based on multiplexer trees and binary decision diagrams. In *2014 IEEE 12th IEEE International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 239–244. IEEE, 2014.
- [27] Jitendra Kumar, Yukio Miyasaka, Asutosh Srivastava, and Masahiro Fujita. Formal verification of integer multiplier circuits using binary decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.
- [28] Casper Abild Larsen, Simon Meldahl Schmidt, Jesper Steensgaard, Anna Blume Jakobsen, Jaco van de Pol, and Andreas Pavlogiannis. A truly symbolic linear-time algorithm for SCC decomposition. In *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023*, pages 353–371. Springer, 2023.
- [29] David E Long. The design of a cache-friendly BDD library. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 639–645, 1998.
- [30] Memotut. Multiplicative hash is not perfect. <https://memotut.com/en/aecfa085417b7134f793/>.
- [31] Mercari. Knuth multiplicative hash is the least complete hash function. <https://engineering.mercari.com/blog/entry/2017-08-29-115047/>, Oct 2017. Accessed: 2023-05-01.
- [32] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, pages 272–277, 1993.
- [33] AMY Miyasaka and M Fujita. A simple BDD package without variable reordering and its application to logic optimization with permissible functions. In *Proc. Int. Workshop Log. Synth*, pages 1–8, 2019.
- [34] Jim Newton and Didier Verna. A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. *ACM Transactions on Computational Logic (TOCL)*, 20(1):1–36, 2019.
- [35] Wytse Oortwijn, Tom van Dijk, and Jaco van de Pol. Distributed binary decision diagrams for symbolic reachability. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 21–30, 2017.
- [36] Samuel Pastva, David Safranek, Nikola Benes, Lubos Brim, and Thomas Henzinger. Repository of logically consistent real-world boolean network models. *bioRxiv*, pages 2023–06, 2023.
- [37] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 254–264, 2011.
- [38] Steffan Christ Sølvsten, Jaco van de Pol, Anna Blume Jakobsen, and Mathias Weller Berg Thomasen. Adiar binary decision diagrams in external memory. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022*, pages 295–313. Springer, 2022.
- [39] Fabio Somenzi. CUDD: CU decision diagram package release 3.0.0. URL: <http://vlsi.colorado.edu/fabio/CUDD>, 4(3), 2015.
- [40] Tom van Dijk, Ernst Moritz Hahn, David N Jansen, Yong Li, Thomas Neele, Mariëlle Stoelinga, Andrea Turrini, and Lijun Zhang. A comparative study of BDD packages for probabilistic symbolic model checking. In *Dependable Software Engineering: Theories, Tools, and Applications: First International Symposium, SETTA 2015, Nanjing, China, November 4-6, 2015, Proceedings 1*, pages 35–51. Springer, 2015.
- [41] Tom Van Dijk and Jaco Van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19:675–696, 2017.
- [42] Miroslav N Velez and Ping Gao. Efficient parallel gpu algorithms for BDD manipulation. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 750–755. IEEE, 2014.
- [43] Alexander Von Rhein, Sven Apel, and Franco Raimondi. Introducing binary decision diagrams in the explicit-state verification of Java code. In *Proc. Java Pathfinder Workshop*, volume 82, page 2, 2011.
- [44] Liudong Xing, Ola Tannous, and Joanne Bechta Dugan. Reliability analysis of nonrepairable cold-standby systems using sequential binary decision diagrams. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 42(3):715–726, 2011.
- [45] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *Innovations in Systems and Software Engineering*, 7:141–150, 2011.


Proofs for Incremental SAT with Inprocessing

Benjamin Kiesl-Reiter 

Amazon Web Services

Munich, Germany

Email: benkiesl@amazon.com

Michael W. Whalen 

Amazon Web Services and University of Minnesota

Minneapolis, MN, United States

Email: mww@amazon.com

Abstract—Incremental SAT solvers are automated-reasoning tools that efficiently solve sequences of related logic problems, making them a go-to tool for inherently incremental applications such as model checking, planning, or test generation. Recent advances in incremental solving using inprocessing have led to substantial performance improvements, but it has remained unclear how the resulting solvers could produce verifiable proofs of unsatisfiability. Here we provide a simple approach that enables inprocessing solvers to produce proofs for incremental results. The approach extends the standard DRAT format with clause restoration steps. These are later removed during post-processing, yielding a standard DRAT proof. Our empirical evaluation shows that our approach is sound and efficient. Proofs can be generated much faster compared to re-solving a problem non-incrementally, but the resulting proofs tend to be larger. Nevertheless, even when taking proof checking into account, our approach is still slightly faster on average. In addition, our technique has the advantage of guaranteeing proof production whereas the non-incremental approach can time out on hard problems.

Index Terms—Automated reasoning, SAT solving, incremental solving, proof.

I. INTRODUCTION

Incremental SAT solving is a key technique for the formal analysis of software and hardware. Instead of solving each input problem independently, an incremental SAT solver retains state between solver calls, allowing it to rely on previously learned information when faced with new problems. This reuse of learned information can dramatically boost performance for applications that produce sequences of closely related SAT problems, such as automated planning [1], lazy SMT solving [2], test-case generation [3], [4], and bounded model checking [5]–[7].

In practice, a user of an incremental SAT solver initializes the solver and provides it with an initial input formula. After solving the formula, the user can then extend the formula before sending another solve request to the solver. The resulting extend-and-solve loop can be repeated arbitrarily many times until eventually the user decides to release the solver. While this incremental feature makes solvers efficient and easy-to-use, the formula modifications performed between solver calls add an additional layer of complexity that can render several commonly-used reasoning techniques unsound. In particular, many preprocessing and inprocessing techniques that are crucial to the performance of non-incremental solvers cannot be used straightforwardly in an incremental context.

This is a pity since inprocessing-based solvers have won the top spots in the yearly SAT competitions since 2020 [8].

To harvest at least some of the performance gains offered by inprocessing techniques, several ways of using them in a restricted way have been suggested in the literature [6], [7], [9]. In 2019, a breakthrough was made by Fazekas, Biere, and Scholl [10], who introduced a calculus that allows the unrestricted use of inprocessing techniques *during* incremental solver runs, given that additional reasoning steps—so-called *clause restorations*—are performed *ahead* of the runs. The implementation of their approach on top of the award-winning SAT solver CaDiCaL [11] has led to impressive performance gains, showing that inprocessing and incremental solving can coexist in practice.

One key capability of non-incremental solving, however, has still failed to enter the picture—*proof*. While virtually all modern non-incremental solvers can produce independently verifiable proofs of unsatisfiability (usually in the DRAT format [12] required by SAT competitions), it has remained unclear how an incremental solver with unrestricted inprocessing could produce such proofs. In this paper, we address this issue by presenting a surprisingly simple approach for extracting a verifiable DRAT proof from an incremental solver relying on the calculus by Fazekas et al. [11].

Our approach requires a solver to augment its proof trace with additional proof steps whenever it performs clause restorations. Once the solver finishes, we transform the augmented proof trace into a verifiable DRAT proof. The key idea is to remove deletions from the proof corresponding to clauses that need to be restored. To demonstrate the feasibility of our approach in practice, we modified the solver CaDiCaL and implemented our proof-transformation algorithm as a separate tool. The resulting version of CaDiCaL is thus the first incremental SAT solver that combines unrestricted inprocessing with proof production.

Despite the theoretical simplicity of our approach, its actual implementation required us to make several careful changes to the solver, which we explain in detail. This should provide solver developers with sufficient background to implement our approach on top of other solvers. We performed an evaluation with 300 benchmarks from the 2017 Hardware Model Checking competition [13], demonstrating that the overhead of our approach is small and that all resulting proofs can be verified with the existing proof checker DRAT-trim [14].

The main contributions of this paper are as follows:

- An approach for extracting verifiable DRAT proofs from an incremental SAT solver that performs unrestricted inprocessing.
- Implementation on top of the SAT solver CaDiCaL.
- Empirical evaluation on a comprehensive benchmark set, demonstrating the feasibility of our approach.

The rest of this paper is structured as follows. In Section II, we present the background required to understand the rest of the paper. In Section III, we describe the general problem of producing proofs for incremental SAT solving with inprocessing. In Section IV, we present our algorithm for generating proofs and establish its soundness before discussing implementation details in Section V. Finally, we present the results of our empirical evaluation in Section VI and conclude with a summary and an outlook for future work in Section VII.

II. BACKGROUND AND RELATED WORK

We first cover basics of SAT solving before giving a high-level overview of the relationship between incremental SAT solving and inprocessing.

A. SAT Solving Basics

The Boolean satisfiability problem (SAT) asks whether a formula of propositional logic can be satisfied by some assignment of truth values (*true* and *false*) to its variables. An overview can be found in [15].

As is common in practical SAT solving, we concern ourselves with formulas in *conjunctive normal form* (CNF). Formulas are built from *literals*, which are either variables (x) or their negations (\bar{x}). These are called *positive literals* and *negative literals* respectively. We write $\text{var}(l)$ to refer to the variable of the literal l ($\text{var}(x) = x$ and $\text{var}(\bar{x}) = x$). The *complement* \bar{l} of a literal l is defined as $\bar{l} = \bar{x}$ if $l = x$ and as $\bar{l} = x$ if $l = \bar{x}$. A *clause* is a finite disjunction of literals of the form $(l_1 \vee l_2 \vee \dots \vee l_n)$. A *formula* is a finite conjunction of clauses of the form $C_1 \wedge C_2 \wedge \dots \wedge C_m$. For example, $(\bar{x} \vee y \vee z) \wedge (y \vee \bar{z}) \wedge (x)$ is a formula with three clauses, where the last clause is called a *unit clause* because it contains only one literal. Formulas can be viewed as sets of clauses, which can be viewed as sets of literals.

A *truth assignment* (or *assignment* for short) is a function from a set of variables to the truth values 1 (*true*) and 0 (*false*). A literal is *satisfied* by an assignment α if l is positive and $\alpha(\text{var}(l)) = 1$ or if l is negative and $\alpha(\text{var}(l)) = 0$. A literal l is *falsified* by α if \bar{l} is satisfied by α . An assignment α can be viewed as the set $\{l \mid l \text{ is satisfied by } \alpha\}$ of literals. A clause is satisfied by an assignment if the assignment satisfies at least one of its literals. A formula is satisfied by an assignment if the assignment satisfies all of its clauses.

A formula is *satisfiable* if there exists an assignment that satisfies it, otherwise it is *unsatisfiable*. Two formulas are *logically equivalent* if they are satisfied by the same assignments; they are *equisatisfiable* if they are either both satisfiable or both unsatisfiable.

Incremental SAT. An incremental SAT problem is a sequence $\langle \Delta_0, A_0 \rangle, \dots, \langle \Delta_n, A_n \rangle$ of pairs, where each Δ_i is

DIMACS					DRAT				
p	cnf	4	8						
1	-2		0				-3	0	
	2		-4	0		1	2	0	
1	2		4	0			-1	0	
-1		-3		0			-3	0	
1		-3		0				1	0
-1		3		0	d	1	2	0	
1		3	-4	0					
1		3	4	0				0	

Fig. 1. DIMACS formula and corresponding proof in DRAT format.

a set of clauses and each A_i is a set of literals called *assumptions*. In each solving phase $i \in 0, \dots, n$, the task is to determine satisfiability of the union of the first i sets of clauses under the single set A_i of assumptions, i.e., to determine satisfiability of $\Delta_0 \cup \dots \cup \Delta_i \cup \{(l) \mid l \in A_i\}$.

File Formats and Proofs. In non-incremental SAT, formulas are typically specified in the DIMACS format. DIMACS files feature a header of the form ‘p cnf #variables #clauses’ followed by a list of clauses. Each clause is represented by a list of integers, with a 0 denoting the end of a clause. For example, the clause $(x_1 \vee \bar{x}_2 \vee x_3)$ is represented as ‘1 -2 3 0’. An example formula in DIMACS format is given in Fig. 1.

The current standard format for proofs is DRAT (short for *Deletion Resolution Asymmetric Tautology*) [12]. A DRAT file specifies a sequence of proof statements, which are either clause *additions* or clause *deletions*. Formally, a DRAT proof of a formula F can be seen as a sequence $\langle s_1, C_1 \rangle, \dots, \langle s_m, C_m \rangle$, where each $s_i \in \{a, d\}$ and each C_i is a clause. A proof of F gives rise to an *accumulated formula* as follows (where $F_0 := F$):

$$F_i = \begin{cases} F_{i-1} \cup \{C_i\} & \text{if } s_i = a \\ F_{i-1} \setminus \{C_i\} & \text{if } s_i = d \end{cases}$$

Each added clause C_i must have the *RAT property* [16] with respect to F_{i-1} . RAT (short for *Resolution Asymmetric Tautology*) is a non-monotonic syntactic property that is checkable in polynomial time and that guarantees that the clause addition preserves satisfiability. Further details of RAT are not essential to our paper, we refer the interested reader to [16] for more information.

Deletions can remove arbitrary clauses from the accumulated formula; they clearly preserve satisfiability. A valid DRAT proof of unsatisfiability ends with the addition of the empty clause. Because the empty clause is trivially unsatisfiable (and since each proof step preserves satisfiability) the unsatisfiability of the original formula F can be concluded. DRAT has a plain-text format and a more compact binary format. An example plain-text DRAT proof is given in Fig. 1 (note that deletions are preceded by a *d* symbol whereas additions are not preceded by any symbol; a 0 marks the end of a statement).

Example 1. The DRAT proof on the right of Fig. 1 is a proof of the DIMACS file on the left. It derives all clauses from earlier

clauses via a RAT derivation step. For example, the first line of the DRAT proof, clause -3 , can be derived using resolution (which satisfies RAT) from the fourth and fifth clauses of the DIMCAS file: $-1 -3$ and $1 -3$. Similarly, the next clause $1 2$ can be derived via resolution from the second and third clauses: $2 -4$ and $1 2 4$. The remaining clauses can be derived via similar resolution steps. The deletion step $d 1 2$ removes the clause $1 2$ from the accumulated formula, reducing the proof search space. As the derivation ends with the empty clause, unsatisfiability of the original formula is proved.

B. Inprocessing in Incremental SAT Solving

Inprocessing in SAT relies on adding and deleting *redundant* clauses both *before* and *during* solving. A clause C is considered redundant with respect to a formula F if F and $F \wedge C$ are equisatisfiable. An overview of common inprocessing techniques is given in [8]. In practice, solvers often delete clauses that are not implied but still redundant. When it comes to clause additions, however, they usually only add clauses that are implied (clauses that are not implied are only learned by special techniques like extended resolution [17], blocked-clause addition [18], and satisfaction-driven clause learning [19], which most solvers don't use by default). Fazekas et al. [10] have recently presented a calculus to capture the inner workings of modern incremental SAT solvers. Intuitively, a solver whose solving process can be expressed by the calculus is guaranteed to return correct results. We refer the interested reader to the paper for details. Since our work builds on their results, we give a high-level overview here.

The calculus consists of seven derivation rules that operate over triples $\langle \varphi, \rho, \sigma \rangle$, where φ is called the set of *irredundant clauses*, ρ is the set of *redundant clauses*, and σ is the so-called *reconstruction stack*, which we describe in detail later.

Most practical techniques in non-incremental SAT solving without inprocessing are captured by four derivation rules called LEARN⁻, STRENGTHEN, FORGET, and DROP.¹ Intuitively, the calculus starts out with all input clauses in the irredundant set φ and adds new implied clauses to the redundant set ρ via the LEARN⁻ rule. Clauses from the redundant set ρ can be moved to the irredundant set φ via the unconditional STRENGTHEN rule. The FORGET rule enables the unconditional deletion of clauses from ρ , and the DROP rule enables the deletion of clauses from φ if they are implied.

The calculus contains three more rules—WEAKEN⁺, ADDCLAUSES, and RESTORE—that enable the sound combination of inprocessing and incremental solving. These three rules interact with the reconstruction stack σ , which in practice is a crucial ingredient for solvers that perform non-trivial preprocessing or inprocessing. Formally, the reconstruction stack is a sequence $(\omega_1 : C_1), \dots, (\omega_n : C_n)$, where each C_i is a clause and each ω_i is a set of literals (called the *witness*) such

that $C_i \cap \omega_i \neq \emptyset$; $(\omega_i : C_i)$ is also called a *witness-labeled clause*.

Using the WEAKEN⁺ rule, clauses of the irredundant set φ can be removed if they are determined to be equisatisfiability-redundant. When such clauses are deleted during solving, however, the solver might later find an assignment that satisfies the resulting formula but not the deleted clauses. To efficiently recover a satisfying assignment of the original formula, the solver stores these clauses on the reconstruction stack to later perform *model reconstruction*. When performing model reconstruction, the solver starts with the assignment α and iterates over the reconstruction stack in reverse order, checking for each witness-labeled clause $(\omega : C)$ whether C is satisfied by α . If C is satisfied, it can be skipped, otherwise α is modified by making all literals in ω true (denoted by $\alpha \circ \omega$).

Formally, the reconstruction function \mathcal{R} , which maps an assignment and a reconstruction stack to a new assignment, is defined as follows (ϵ denotes the empty sequence; concatenation of sequences σ and σ' is denoted by $\sigma \cdot \sigma'$):

$$\begin{aligned} \mathcal{R}(\alpha, \epsilon) &= \alpha, \\ \mathcal{R}(\alpha, \sigma \cdot (\omega : C)) &= \begin{cases} \mathcal{R}(\alpha, \sigma) & \text{if } \alpha(C) = 1 \\ \mathcal{R}(\alpha \circ \omega, \sigma) & \text{otherwise} \end{cases} \end{aligned}$$

Since $C \cap \omega \neq \emptyset$, making ω true also makes C true, but the solver must ensure that making ω true does not falsify any of the other clauses. For non-incremental SAT solving, all state-of-the-art inprocessing techniques generate witness-labeled clauses in such a way that this is guaranteed, and in most cases (like bounded variable elimination [20], pure-literal elimination, or blocked-clause elimination [21]), ω consists of only a single literal.

When incremental solving comes into play, however, things get tricky because the deletion of non-implied clauses can weaken a formula. The naive addition of clauses later on during incremental calls can then lead to unsound results. This observation led Fazekas et al. to the introduction of the rules ADDCLAUSES and RESTORE in their calculus. Before explaining the two rules, we give an example to illustrate the problem.

Example 2 (from [10]). *Consider the Boolean formula $F = (a \vee b) \wedge (\bar{a} \vee \bar{b})$. This formula is clearly satisfiable, and there are inprocessing techniques (e.g., blocked-clause elimination) that would delete the clause $(\bar{a} \vee \bar{b})$ to obtain $F' = (a \vee b)$, which is also satisfiable. Now, assume that at the next incremental call, the unit clauses (a) and (b) are added. The formula $F \wedge (a) \wedge (b) = (a \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a) \wedge (b)$ is then unsatisfiable whereas $F' \wedge (a) \wedge (b) = (a \vee b) \wedge (a) \wedge (b)$ is satisfiable. The deletion of $(\bar{a} \vee \bar{b})$ thus weakened the formula too much, leading to unsound results.*

The key insight for incremental solving from [10] is that whenever a new set Δ_i of clauses is added to the problem at an incremental solver call, the solver can ensure soundness by moving some of the previously-deleted clauses back from the reconstruction stack σ to the set φ of irredundant clauses.

¹The minus symbol in the rule LEARN⁻ and the plus symbol in the later rule WEAKEN⁺ are used because the rules are modified versions of rules LEARN and WEAKEN from an earlier calculus described in [16].

```

1: function RESTOREADDCLAUSES( $\Delta$ : clauses,  $\sigma$ : stack)
2:    $(\omega_1 : C_1), \dots, (\omega_n : C_n) := \sigma$ 
3:   for  $i$  from 1 to  $n$  do
4:     if exists  $l \in \omega_i$  where  $\bar{l}$  occurs in  $\Delta$  then
5:        $\Delta := \Delta \cup \{C_i\}, \sigma := \sigma \setminus (\omega_i : C_i)$ 
6:   return  $\langle \Delta, \sigma \rangle$ 

```

Fig. 2. Algorithm RestoreAddClauses to restore clauses.

This process is called *clause restoration*, and it is based on the notion of a *clean* clause.

Definition 1 (Clean Clause [10]). *A clause C is clean with respect to a sequence of witness-labeled clauses σ if for all $(\omega : C') \in \sigma$, we have that $\{\bar{l} \mid l \in C\} \cap \omega = \emptyset$.*

A clause is thus clean with respect to σ if it does not contain the negations of literals that serve as witnesses in σ . Cleanliness of a clause ensures that whenever model reconstruction takes place at the end of incremental solving, the truth of clean clauses is not affected because making a witness ω true cannot falsify clean clauses.

Given that satisfiability is preserved for clean clauses, when we add a set Δ of new clauses, we must ensure that they are clean with respect to the current reconstruction stack. The ADDCLAUSES rule thus has the precondition that only clean clauses can be added. Taken at face value, this would seem to make it very difficult to perform inprocessing with incremental solving: what if we need to add clauses that are not clean? This situation is where the RESTORE rule comes into play. The RESTORE rule allows moving a clause from the reconstruction stack back to the set of irredundant clauses as long as the clause is clean with respect to the subsequent portion of the reconstruction stack:

$$\text{RESTORE: } \frac{\langle \varphi, \rho, \sigma \cdot (\omega : C) \cdot \sigma' \rangle}{\langle \varphi \wedge C, \rho, \sigma \cdot \sigma' \rangle} \quad (C \text{ is clean w.r.t. } \sigma')$$

Thus, whenever we want to add unclean clauses at an incremental solver call, we can turn them into clean clauses by first restoring all labeled clauses of the reconstruction stack that would make them unclean. This is achieved with the algorithm RestoreAddClauses [10] shown in Fig. 2. The algorithm iterates over the stack starting at the bottom, continuously restoring clauses that would prevent cleanliness of the new clauses. To make sure that the precondition of the RESTORE rule (C is unclean w.r.t. σ') is fulfilled, it also restores clauses that would prevent cleanliness of previously restored clauses.

With this background, we can now state the problem we are trying to solve.

III. PROBLEM STATEMENT

The work in [10] presents a sound calculus for solvers to perform incremental solving with inprocessing, but it does not define how a solver based on that calculus could efficiently produce an independently-checkable proof. This is also the reason why the SAT solver CaDiCaL, which in [10] was

augmented with the RestoreAddClauses algorithm, does not produce valid proofs when using inprocessing during incremental solving.

Our goal is to obtain a valid DRAT proof from a solver based on the incremental inprocessing calculus in [10]. The calculus requires that all clauses derived by a solver are implied. Strictly speaking, this would allow the addition of implied clauses that do not necessarily have the RAT property. In practice, however, all state-of-the-art CDCL solvers derive only clauses with the so-called RUP (short for *Reverse Unit Propagation*) property—a simpler property guaranteeing that the clauses are implied *and* have the RAT property (see [22], [23] for details on RUP). Hence, we require that solvers derive clauses with the RUP property, meaning that our approach applies to virtually all existing solvers. RUP is monotonic: If a clause has the RUP property with respect to a formula F , it also has it with respect to each superset of F .

Proof checkers for DRAT require as input both a formula and a corresponding proof. We produce proofs in such a way that they can be checked against the formula consisting of all incrementally added clauses, plus a unit clause for each literal in the final assumption. More formally, let $\langle \Delta_0, A_0 \rangle, \dots, \langle \Delta_n, A_n \rangle$ be an unsatisfiable incremental problem. We then produce a DRAT proof of the formula $\Delta_0 \cup \dots \cup \Delta_n \cup \{(l \mid l \in A_n)\}$.

The four derivation rules LEARN[−], STRENGTHEN, FORGET, and DROP can be accommodated in a straightforward way in DRAT: LEARN[−] corresponds to a clause addition (of the learned clause), STRENGTHEN is not reflected in the proof (as DRAT does not distinguish between redundant and irredundant clauses), and both FORGET and DROP correspond to clause deletions in DRAT.

Applications of ADDCLAUSES don't need to be represented in the proof trace because we simply consider all clauses that are added incrementally part of the initial formula. Thus, in the DRAT proof trace, everything that was derived using these clauses can still be derived because they are part of the proof's accumulated formula.

The problem is how to express applications of WEAKEN⁺ and corresponding applications of RESTORE. One option is to represent WEAKEN⁺ by clause deletion. However, if we then naively express RESTORE by clause addition, we run into soundness problems. Specifically, a clause that had the RAT property at the point of WEAKEN⁺/deletion (which is the case for most clauses deleted during practical inprocessing) might not have the RAT property at the point of restoration anymore. This would render the clause addition in the proof invalid. Intuitively, this is because the RAT property is influenced by additions and deletions happening between a clause's initial deletion and its restoration. For readers familiar with the details of the RAT property, the following example demonstrates this on a concrete formula:

Example 3. *Consider the formula $(\bar{x} \vee y) \wedge (x \vee \bar{z}) \wedge (\bar{y} \vee z) \wedge (\bar{x} \vee z)$. The clause $(\bar{y} \vee z)$, which has the RAT property (as an interested reader might convince themselves of) can be*

deleted and put on the reconstruction stack. The clause $(\bar{x} \vee z)$ can be subsequently deleted because it has the RAT property with respect to the remaining clauses. We end up with the formula $(\bar{x} \vee y) \wedge (x \vee \bar{z})$. Now assume that we want to restore the clause $(\bar{y} \vee z)$, e.g., because we want to solve under the assumption (y) . If we tried to add the clause to a DRAT proof, the proof would become invalid because $(\bar{y} \vee z)$ does not have the RAT property with respect to the remaining formula—the clause $(x \vee \bar{z})$ was crucial for establishing the RAT property, but it has been deleted.

Because solvers can use restored clauses to derive further clauses, the restored clauses need to enter the accumulated formula of the proof, otherwise those derivations become invalid. We thus need another way to express RESTORE in a DRAT proof.

IV. ALGORITHM FOR PROOF PRODUCTION

To handle applications of RESTORE in DRAT proofs and thus support proofs for incremental solving with inprocessing, we propose the following approach:

- We first produce an *augmented* proof trace (which itself is not a valid DRAT proof) where all applications of WEAKEN⁺ are logged as deletions and all applications of RESTORE are logged with a new dedicated proof rule.
- In a post-processing step, we then make use of the restoration information to convert the augmented proof trace into a valid DRAT proof.

The idea is surprisingly simple: Instead of trying to add restored clauses to the proof via clause additions, we act as if they hadn't been deleted in the first place.

Specifically, whenever a solver applies the RESTORE rule to restore a clause C from the reconstruction stack, we add to the proof trace a novel statement of the form $\langle r, C \rangle$. The resulting augmented proof trace is a sequence $\langle s_1, C_1 \rangle, \dots, \langle s_m, C_m \rangle$, where each $s_i \in \{a, d, r\}$ and each C_i is a clause. Similar to a normal DRAT proof trace, we define an accumulated formula as follows:

$$F_0 = \text{clauses in the original problem}$$

$$F_i = \begin{cases} F_{i-1} \cup \{C_i\} & \text{if } s_i \in \{a, r\} \\ F_{i-1} \setminus \{C_i\} & \text{if } s_i = d \end{cases}$$

Given that, as assumed earlier, all clauses derived by the solver are RUP (and thus RAT) clauses with respect to the accumulated formula, this holds for all clause additions in the augmented proof as well. Only the clause restorations are not justified, but we deal with them during post-processing.

The post-processing, which we describe in detail below, is formalized via the post-processing function Φ , which takes

as arguments an augmented proof trace P together with an (initially empty) set R of restored clauses.

$$\Phi(\epsilon, R) = \epsilon$$

$$\Phi(P \cdot \langle s, C \rangle, R) = \begin{cases} \Phi(P, R) \cdot \langle a, C \rangle & \text{if } s = a \\ \Phi(P, R \cup \{C\}) & \text{if } s = r \\ \Phi(P, R \setminus \{C\}) & \text{if } s = d \wedge C \in R \\ \Phi(P, R) \cdot \langle d, C \rangle & \text{if } s = d \wedge C \notin R \end{cases}$$

Intuitively, Φ traverses the augmented proof trace in reverse order, statement by statement. When the algorithm encounters a clause addition, it simply adds it to the new proof trace. When it encounters a clause restoration, it adds the clause to the set R but does not add anything to the new proof trace. Finally, when it encounters a deletion, it checks whether the deleted clause is contained in R , and if so, removes the deleted clause from R without changing the proof trace, otherwise it just adds the deletion to the new proof trace. The resulting proof trace is a subsequence of the original proof P .

Example 4. Consider the augmented proof trace $P = \langle a, x \vee y \rangle, \langle d, y \vee z \rangle, \langle d, \bar{z} \vee u \rangle, \langle r, y \vee z \rangle, \langle a, x \vee \bar{u} \rangle$. At the beginning, $R := \emptyset$. The post-processing function Φ traverses P in reverse order, starting with $\langle a, x \vee \bar{u} \rangle$, which it appends to the result of the recursive call for the remainder of the list: $\Phi(\langle a, x \vee y \rangle, \langle d, y \vee z \rangle, \langle d, \bar{z} \vee u \rangle, \langle r, y \vee z \rangle, \emptyset) \cdot \langle a, x \vee \bar{u} \rangle$. It next encounters the restoration $\langle r, y \vee z \rangle$ and thus sets $R := \{y \vee z\}$ for the recursive call. When it next encounters the deletion $\langle d, \bar{z} \vee u \rangle$, it adds it to the processed proof because the clause is not in the restore set R : $\Phi(\langle a, x \vee y \rangle, \langle d, y \vee z \rangle, \{y \vee z\}) \cdot \langle d, \bar{z} \vee u \rangle, \langle a, x \vee \bar{u} \rangle$. However, for the next deletion, $\langle d, y \vee z \rangle$, the clause is already in R . Thus the deletion is not added to the processed proof but instead, the clause is removed from R . Finally, the addition $\langle a, x \vee y \rangle$ is also added to the processed proof, so we end up with the proof $\langle a, x \vee y \rangle, \langle d, \bar{z} \vee u \rangle, \langle a, x \vee \bar{u} \rangle$.

As we can see in Example 4, the clause $(y \vee z)$ was restored during solving and thus $\langle r, y \vee z \rangle$ was part of the augmented proof trace. In the final proof trace, however, the clause is never even deleted. Intuitively, the consequence for the accumulated formula is that it always contains all the clauses required to make further derivations. More specifically, consider the augmented proof trace P and the processed proof P' , where the latter is a subsequence of the former. If we compare the accumulated formulas of each proof after any proof statement $\langle s_i, C_i \rangle$ contained in both P' and P , we observe that the accumulated formula with respect to P' contains all clauses of the accumulated formula with respect to P .

Since we assume all added clauses are RUP (and thus RAT) clauses, and since RUP is monotonic, we obtain a valid DRAT proof in which all clause additions fulfill the RAT property.

V. IMPLEMENTATION

We implemented our approach on top of the incremental inprocessing SAT solver CaDiCaL. In particular, we added

the capability to produce augmented proofs with restore statements. As restorations only occur during incremental solving with inprocessing, our changes to CaDiCaL do not impact non-incremental solving. Additionally, we implemented proof post-processing in a dedicated tool. The tool traverses a proof backwards, printing all proof statements immediately (instead of prepending them to an internal data structure) to keep the memory requirements low. Since this leads to a reversed proof, we reverse it again at the end to get a valid DRAT proof. Our toolchain can produce DRAT proofs in both the plain-text format and the binary format.

In order to obtain valid proofs, we made changes to CaDiCaL to maintain the invariant that each clause restoration is preceded by a corresponding deletion. This invariant, which we used when arguing about correctness of our approach, is guaranteed to hold if a solver records all applications of the WEAKEN⁺ rule as deletions in the proof. In practice, however, there are additional subtleties that need to be taken care of to make sure this is the case. We now explain the three most important changes to CaDiCaL to provide solver developers with guidance for implementing our approach on top of other solvers.

A. Deletion of Binary Clauses

When CaDiCaL logically deletes a clause during solving, it immediately marks the clause as deleted but only later, during garbage collection, really removes it from memory. For non-binary clauses, it logs a deletion statement to the proof immediately after logical deletion. For binary clauses, however, the deletion is only logged once the clause is really removed during garbage collection. For binary clauses, this could lead to the case where a clause is first marked deleted and then restored, but the deletion is only logged in the proof trace after restoration. The fix is simple: ensure garbage collection is triggered before restoration; thus all deletions occur in the proof before their corresponding restorations.

B. Proper Handling of Equivalent-Literal Substitution

CaDiCaL performs an inprocessing technique called *equivalent-literal substitution* [24] (“decompose” in CaDiCaL’s code). The technique identifies equivalent literals in the binary implication graph of a formula and then substitutes a single representative literal for each equivalence class of literals. For example, if x and y are identified as equivalent, CaDiCaL might replace all occurrences of y by x . To later reconstruct a proper model for the removed literals, CaDiCaL adds to the reconstruction stack an equivalence $(\bar{x} \vee y) \wedge (x \vee \bar{y})$ for each removed y and representative x . These clauses are never explicitly added or deleted and thus they are not represented in the proof. However, as they are on the reconstruction stack, CaDiCaL may restore them. To deal with this situation, we add the equivalences to the proof (which is allowed because they are trivially RUP) and then immediately delete them again. This allows us to remove the deletions during post-processing, ensuring proper derivation of the equivalences in the proof.

C. Internal and External Representations of Literals

CaDiCaL maintains both an internal and an external representation of literals, together with mappings between these representations. This is because the solver sometimes removes literals and then remaps the remaining ones to save memory (e.g., when performing equivalent-literal substitution as above). This can lead to problems when restoring clauses. For example, when a clause is restored and immediately simplified (because some of its literals are falsified or satisfied at the top level), the solver deletes the original clause and adds the simplified clause to the proof. In particular, the solver first maps the restored clause to an internal representation before remapping it to an external representation when performing the deletion—this “round trip” can lead to a different external representation than the one originally deleted. In our implementation, we modified the corresponding code to ensure that the deleted and restored clauses match in the proof.

VI. EVALUATION

To evaluate our approach in practice, we plugged our proof-producing version of CaDiCaL into CaMiCaL [10], a SAT-based bounded model checker that runs on AIGER models [25] used in the hardware model checking competition (HWMCC) [13]. Following the experiment in [10], we ran CaMiCaL on the 300 models of the single safety property track of HWMCC’17 [13], up to bound 1000 with a time limit of 3600 seconds per model. In [10], it was shown that on these models, enabling inprocessing with clause restoration leads to a significant performance improvement compared to disabling inprocessing or running it only in a restricted way, e.g., by *freezing* [6] certain variables.

We ran our experiments on an Amazon EC2 m5d.metal instance, which has an AWS-custom Intel Xeon Scalable (Skylake) processor with 96 vCPUs, 384 GiB memory, and four 900 GB SSDs, running Amazon Linux 2. We ran 24 benchmark processes in parallel.

Our primary goal was to demonstrate that our approach produces valid DRAT proofs. For each model of the benchmark set, we performed a CaMiCaL run and generated a DRAT proof for the highest unsatisfiable bound solved by CaMiCaL within the time limit. This means that for unsatisfiable problems, we took the highest solved bound whereas for satisfiable problems, we took the second highest solved bound (as it yields an UNSAT result). There were four problems that were satisfiable at the first bound and another two problems for which CaMiCaL timed out while attempting to solve the first bound, leaving 294 problems with an UNSAT result.²

To check the correctness of the resulting DRAT proof for each problem, we extracted from CaMiCaL a DIMACS file including all clauses that were added incrementally to CaDiCaL during solving as well as unit clauses for the assumptions of the final unsatisfiable call (i.e., one unit clause per assumption). We then passed the DIMACS file together with

²The problems satisfiable at the first bound are 6s389b02.aig, bob-miterbm1or.aig, bobsynth13.aig, and bobtuint24.aig; the problems for which CaMiCaL timed out at the first bound are 6s128.aig and 6s398b09.aig.

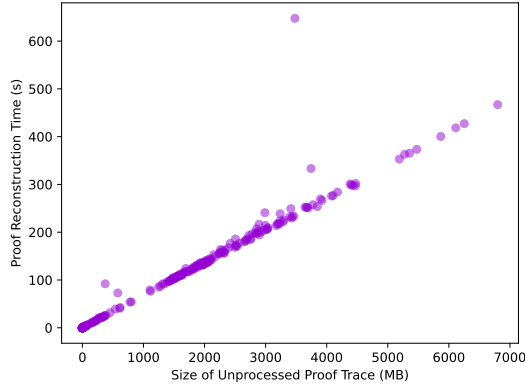


Fig. 3. Proof Reconstruction Time vs. Size of Proof Trace.

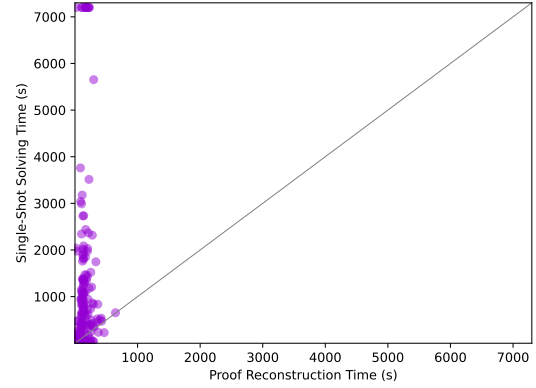


Fig. 5. Proof Reconstruction vs. Single-Shot Solving.

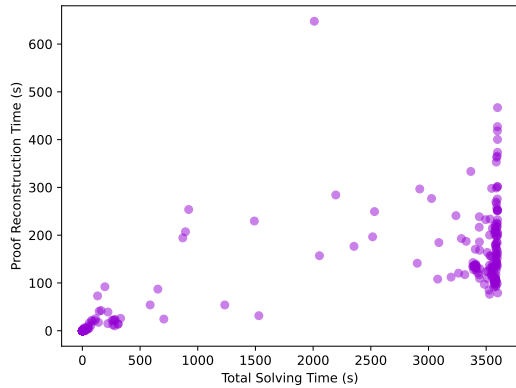


Fig. 4. Proof Reconstruction Time vs. Total Solving Time.

the corresponding DRAT proof—obtained by post-processing the original proof trace with our tool—to the proof checker DRAT-trim [12]. For all 294 problems, DRAT-trim confirmed that the proofs were correct.

To get an idea of the overhead introduced by our post-processing approach, we measured the time spent on post-processing as compared to solving. On average, the overhead of post-processing was 5.3% of the actual solving time (i.e., the time spent inside the SAT solver, not the whole time spent by the model checker). The time to post-process is linear in the size of the proof trace, as shown in Fig. 3. The figure shows that our implementation takes a little more than a minute to post-process 1 GB of proof. While fast, the overhead varies considerably between problems, as the proof size is not a function of the solving time, i.e., two solver runs that take the same time might produce proofs of different sizes, as is illustrated in Fig. 4.

Next, we present performance measurements from our experiments to give an indication of the performance of our approach as compared to an alternate approach to generating a proof by solving the corresponding bound with CaDiCaL monolithically from scratch using non-incremental solving

with proof generation enabled. For example, if CaMiCaL was able to solve n bounds of an incremental problem, we take the propositional formula corresponding to bound n and solve it non-incrementally with a new CaDiCaL instance that produces a proof. We then compare the time it takes CaDiCaL to produce that proof with the time it takes to post-process the incremental proof with our approach, to see which approach is more efficient. In this experiment, we gave the non-incremental CaDiCaL a timeout of 7200 seconds, i.e., twice the CaMiCaL timeout. While we believe these numbers are informative, we note that they are not representative for general incremental SAT solving as they only apply to our restricted benchmark set (which we chose primarily to demonstrate soundness, as it triggers many clause restorations).

Fig. 5 compares the two approaches. Clearly, post-processing an incremental proof trace is much more efficient than re-solving a formula from scratch. In particular, there were 13 instances for which CaDiCaL timed out when trying to solve them in a single shot. This is not a surprise: there is no guarantee that a problem that is solvable incrementally can also be solved in a single shot, whereas post-processing a proof takes a small overhead that can be estimated based on the size of the unprocessed proof trace.

The results are less clear when we consider the sum of time spent on post-processing/single-shot solving and on checking the resulting proofs. For this comparison, we excluded the 13 instances for which CaDiCaL timed out when solving them in a single shot; Fig. 6 shows the results. Although our approach is 13% faster on average, there are several problems where the monolithic approach is faster. One reason for this is that incremental proofs are usually larger than the ones produced by single-shot solving, as illustrated in Fig. 7. To summarize, on the models of the single safety property track of HWMCC’17, post-processing incremental proofs is faster than producing proofs from scratch in a single shot, with the latter carrying the risk of timeouts. On the flip side, if single-shot solving succeeds, it tends to produce shorter proofs, which can in turn be checked faster.

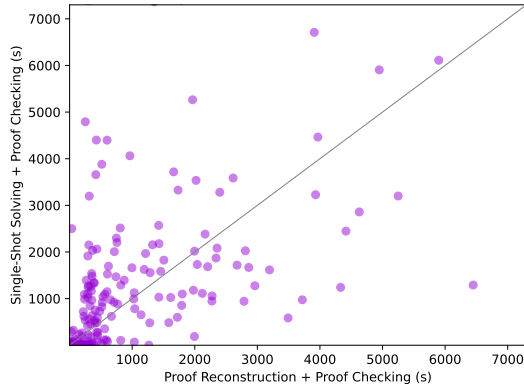


Fig. 6. Proof Reconstruction vs. Single-Shot Solving (incl. Checking).

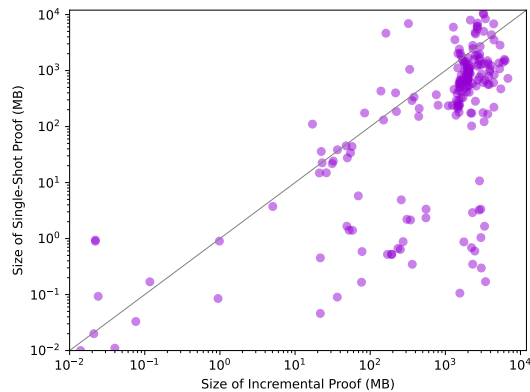


Fig. 7. Proof Size: Incremental vs. Single-Shot Solving (log scale).

VII. CONCLUSION AND FUTURE WORK

We have presented an efficient approach to generate proofs for inprocessing incremental solvers based on the calculus from [10]. We augment the DRAT [12] proof format by adding *restore* steps to the proof. These steps can be efficiently removed during post-processing, yielding a standard DRAT proof. We implemented the approach on top of the CaDiCaL [26] solver and demonstrated its soundness and efficiency against the benchmark suite from HWMCC’17 [13]. For these benchmarks, proof generation adds 2.9% average overhead to solving. Post-processing adds 5.3% average overhead.

We compared our approach to a monolithic one in which we solved the final formula in a single shot and generated proofs during this step. If solving times are compared, our approach is faster, as it does not require a “from scratch” solve of the entire problem. As our incremental proofs are larger than the DRAT proofs generated by the monolithic solve, the picture including proof-checking times is mixed. Although our approach is faster on average when measuring the sum of solving plus proof checking times, there are examples where the situation is reversed.

There are two important advantages of our approach vs. resolving the final problem monolithically. First, it always yields a solution, whereas the monolithic approach sometimes times out on our benchmark set. Second, it provides a foundation that can be adapted in future work towards efficiency and robustness improvements for inprocessing incremental solvers as discussed in the following.


When the LRAT support for CaDiCaL described in a future paper [27] is integrated into the main repository, we will convert the augmented DRAT format to LRAT (the deletions and restorations are the same). In the results in [27], converting, trimming, and checking proofs is approximately 5.5x faster than for DRAT; this will lower the overhead for proof checking in our approach relative to the solving time for the monolithic problem. We also plan to examine how to use augmented proof traces to migrate state of incremental solvers, extending the work from [28] to support incremental use-cases. By combining the migration approach with our proof approach, we can migrate solver state between multiple platforms and later combine the resulting incremental proof fragments. This support allows us to better utilize cloud resources and build solvers that can be restarted after machine failures. Finally, we can adapt our approach for use in distributed incremental solving, as was earlier demonstrated for monolithic solving [29].


REFERENCES


- [1] S. Gocht and T. Balyo, “Accelerating SAT based planning with incremental SAT solving,” in *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, L. Barbulescu, J. Frank, Mausam, and S. F. Smith, Eds. AAAI Press, 2017, pp. 135–139. [Online]. Available: <https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/view/15580>
- [2] R. Sebastiani, “Lazy satisfiability modulo theories,” *J. Satisf. Boolean Model. Comput.*, vol. 3, no. 3-4, pp. 141–224, 2007. [Online]. Available: <https://doi.org/10.3233/sat190034>
- [3] P. Mishra and M. Chen, “Efficient techniques for directed test generation using incremental satisfiability,” in *Proceedings of the 2009 22nd International Conference on VLSI Design*, ser. VLSID ’09. USA: IEEE Computer Society, 2009, p. 65–70. [Online]. Available: <https://doi.org/10.1109/VLSI.Design.2009.72>
- [4] A. Yamada, T. Kitamura, C. Artho, E.-H. Choi, Y. Oiwa, and A. Biere, “Optimization of combinatorial testing by incremental sat solving,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.
- [5] A. Biere, “Bounded model checking,” in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 739–764. [Online]. Available: <https://doi.org/10.3233/FAIA201002>
- [6] N. Eén and N. Sörensson, “Temporal induction by incremental SAT solving,” *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003. [Online]. Available: [https://doi.org/10.1016/S1571-0661\(05\)82542-3](https://doi.org/10.1016/S1571-0661(05)82542-3)
- [7] S. Kupferschmid, M. Lewis, T. Schubert, and B. Becker, “Incremental preprocessing methods for use in BMC,” *Formal Methods Syst. Des.*, vol. 39, no. 2, pp. 185–204, 2011. [Online]. Available: <https://doi.org/10.1007/s10703-011-0122-4>
- [8] A. Biere, M. Jarvisalo, and B. Kiesl, “Preprocessing in SAT solving,” *Handbook of Satisfiability*, vol. 336, pp. 391–435, 2021.
- [9] A. Nadel, V. Ryzhkin, and O. Strichman, “Preprocessing in incremental SAT,” in *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 256–269. [Online]. Available: https://doi.org/10.1007/978-3-642-31612-8_20

- [10] K. Fazekas, A. Biere, and C. Scholl, "Incremental inprocessing in SAT solving," in *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, ser. Lecture Notes in Computer Science, M. Janota and I. Lynce, Eds., vol. 11628. Springer, 2019, pp. 136–154. [Online]. Available: https://doi.org/10.1007/978-3-030-24258-9_9
- [11] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.
- [12] M. J. H. Heule, "The DRAT format and drat-trim checker," *CoRR*, vol. abs/1610.06229, 2016. [Online]. Available: <http://arxiv.org/abs/1610.06229>
- [13] A. Biere, T. van Dijk, and K. Heljanko, "Hardware model checking competition 2017," in *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, p. 9. [Online]. Available: <https://doi.org/10.23919/FMCAD.2017.8102233>
- [14] N. Wetzler, M. J. Heule, and W. A. H. Jr., "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds., vol. 8561. Springer, 2014, pp. 422–429. [Online]. Available: https://doi.org/10.1007/978-3-319-09284-3_31
- [15] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185. [Online]. Available: <http://dblp.uni-trier.de/db/series/faia/faia185.html>
- [16] M. Järvisalo, M. J. Heule, and A. Biere, "Inprocessing rules," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370. [Online]. Available: https://doi.org/10.1007/978-3-642-31365-3_28
- [17] G. Audemard, G. Katsirelos, and L. Simon, "A restriction of extended resolution for clause learning SAT solvers," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, M. Fox and D. Poole, Eds. AAAI Press, 2010. [Online]. Available: <http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1811>
- [18] O. Kullmann, "On a generalization of extended resolution," *Discret. Appl. Math.*, vol. 96-97, pp. 149–176, 1999. [Online]. Available: [https://doi.org/10.1016/S0166-218X\(99\)00037-2](https://doi.org/10.1016/S0166-218X(99)00037-2)
- [19] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere, "Pruning through satisfaction," in *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017. Proceedings*, ser. Lecture Notes in Computer Science, O. Strichman and R. Tzoref-Brill, Eds., vol. 10629. Springer, 2017, pp. 179–194. [Online]. Available: https://doi.org/10.1007/978-3-319-70389-3_12
- [20] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005. Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75. [Online]. Available: https://doi.org/10.1007/11499107_5
- [21] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 129–144. [Online]. Available: https://doi.org/10.1007/978-3-642-12002-2_10
- [22] E. I. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*. IEEE Computer Society, 2003, pp. 10 886–10 891. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/DATE.2003.10008>
- [23] A. V. Gelder, "Verifying RUP proofs of propositional unsatisfiability," in *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. [Online]. Available: http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf
- [24] M. Heule, M. Järvisalo, and A. Biere, "Efficient CNF simplification based on binary implication graphs," in *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, ser. Lecture Notes in Computer Science, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 201–215. [Online]. Available: https://doi.org/10.1007/978-3-642-21581-0_17
- [25] A. Biere, T. van Dijk, and K. Heljanko, "Aiger 1.9 and beyond," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstrasse 69, 4040 Linz, Austria, FMV Reports Series, 2011.
- [26] A. Biere, "CaDiCaL at the SAT Race 2019," in *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.
- [27] F. Pollitt, M. Fleury, and A. Biere, "Efficient proof checking with lrat in cadical (work in progress)," in *26th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2023, Freiburg, Germany, March 23-23, 2023*, A. Biere and D. GroÅŸe, Eds. VDE, 2023, pp. 64–67, accepted.
- [28] A. Biere, M. S. Chowdhury, M. J. Heule, B. Kiesl-Reiter, and M. Whalen, "Migrating solver state," in *SAT 2022, 2022*. [Online]. Available: <https://www.amazon.science/publications/migrating-solver-state>
- [29] D. Michaelson, D. Schreiber, M. J. Heule, B. Kiesl-Reiter, and M. Whalen, "Unsatisfiability proofs for distributed clause-sharing sat solvers," in *TACAS 2023, 2023*. [Online]. Available: <https://www.amazon.science/publications/unsatisfiability-proofs-for-distributed-clause-sharing-sat-solvers>

Verified Encodings for SAT Solvers

Cayden R. Codel 
 Computer Science Department
 Carnegie Mellon University
 Pittsburgh, USA
 ccodel@cs.cmu.edu

Jeremy Avigad 
 Department of Philosophy
 Carnegie Mellon University
 Pittsburgh, USA
 avigad@cmu.edu

Marijn J. H. Heule 
 Computer Science Department
 Carnegie Mellon University
 Pittsburgh, USA
 marijn@cmu.edu

Abstract—Satisfiability (SAT) solvers are versatile tools that can solve a wide array of problems, and the models and proofs of unsatisfiability emitted by SAT solvers can be checked by verified software. In this way, the SAT toolchain is trustworthy. However, many applications are not expressed natively in SAT and must instead be *encoded* into SAT. These encodings are often subtle, and implementations are error-prone. Formal correctness proofs are needed to ensure that implementations are bug-free.

In this paper, we present a library for formally verifying SAT encodings, written using the Lean interactive theorem prover. Our library currently contains verified encodings for the parity, at-most-one, and at-most- k constraints. It also contains methods of generating fresh variable names and combining sub-encodings to form more complex ones, such as one for encoding a valid Sudoku board. The proofs in our library are general, and so this library serves as a basis for future encoding efforts.

I. INTRODUCTION

Satisfiability (SAT) solvers are powerful and versatile tools. They solve hardware and software verification tasks [1, 2], they are used in satisfiability modulo theory solvers [3, 4], and they are instrumental in resolving longstanding open problems in mathematics [5, 6]. Impressed by their strength and utility, Donald Knuth called SAT solvers a “killer app” [7].

Modern SAT solvers are also *trustworthy*. Since the SAT problem is in NP [8], models can be efficiently checked. When no model exists, solvers emit a certificate of unsatisfiability, which is a step-by-step proof written in a formal proof system [9, 10], the de facto standard being DRAT [11]. Proof checkers can then check that the certificate is correct [12, 13]. Proof checkers are simple pieces of software; several checkers have been formally verified [14–17].

Many applications are not expressed natively in SAT and must instead be *encoded* into SAT. The challenge is that encodings are often subtle, and so it is easy to make off-by-one errors and other bugs when implementing an encoding. Detecting errors in an encoded formula is even more challenging: in extreme cases, encodings can contain hundreds of thousands of variables and clauses. A single wrong variable or clause can render the entire formula useless. One way to get rid of bugs in encodings is to verify them in a proof assistant.

In this paper, we present a library for verifying SAT encodings, written with the proof assistant Lean 3 [18]. So far, we have verified encodings for the parity, at-most-one, and

at-most- k constraints. These encodings are common and are used in applications such as cryptography [19, 20], haplotype inference [21], and approximate model counting [22].

In addition to our correctness proofs, we discuss the techniques we developed in our library. One major contribution in our library is a way of introducing and managing fresh variables, which are commonly used to minimize the number of clauses in an encoding. Another contribution is a method of composing constraints and encodings together to form more complex ones while keeping correctness proofs short. We demonstrate how these operations are used in an encoding of Sudoku in Section VI.

II. PRELIMINARIES

Boolean variables range over the classical truth values true (\top) and false (\perp). *Boolean literals* are positive or negative forms of boolean variables, written as x and \bar{x} , respectively. *Truth assignments* τ give truth values to sets of boolean variables. When $\tau(x) = \top$, then $\tau(\bar{x}) = \perp$. If F is a propositional formula that evaluates to true under τ , written as $\tau(F) = \top$, then we say that τ *satisfies* F . If there exists a τ that satisfies F , then F is *satisfiable*.

Let $\text{vars}(\cdot)$ be the set of variables contained in a propositional formula. We overload $\text{vars}(\cdot)$ for τ to mean the set of variables that τ is defined over. If τ and τ' are truth assignments such that $\text{vars}(\tau) \subseteq \text{vars}(\tau')$ and $\tau(x) = \tau'(x)$ whenever $x \in \text{vars}(\tau)$, then τ' *extends* τ .

Most modern SAT solvers only accept formulas in *conjunctive normal form* (CNF). A formula is in CNF if it is a conjunction of clauses, with each clause a disjunction of literals. Unless otherwise noted, when we refer to a formula F , we assume it is in CNF.

Any problem may admit many encodings. From a mathematical point of view, the choice of encoding doesn’t matter as long as each is correct, but in practice, solvers perform better on encodings with fewer variables and clauses [23, 24]. Generally, compact encodings introduce fresh variables to reduce the overall number of clauses, but at the cost of added complexity.

In this paper, we focus on encodings of n -ary boolean constraints. Let $X = x_1, \dots, x_n$ represent the inputs to an n -ary boolean constraint C , and let F be any propositional formula. When $\text{vars}(F) \subseteq X$, then F encodes C if and only if it *defines* it: for every full assignment τ on X , we require

$$C(\tau(x_1), \dots, \tau(x_n)) \leftrightarrow \tau(F) = \top.$$

This work was partially supported by the Hoskinson Center for Formalized Mathematics and by NSF grant CCF-2108521.

Yet F may use additional variables. The following definition handles the more general case.

Definition 1 (Encoding a boolean constraint): Let C be a boolean constraint, F be any propositional logic formula, and $X = x_1, \dots, x_n$ be variables representing the inputs to C . Then F *encodes* C if and only if: for every assignment τ on X , $C(\tau(x_1), \dots, \tau(x_n))$ if and only if F is satisfied by some assignment that extends τ .

Using the language of quantified propositional logic, this amounts to saying that C is defined by $\exists y_1, \dots, y_m F$, where the y_i are the additional variables appearing in F . It may help to think of the y_i as auxiliary objects that are required to satisfy their descriptions in F .

In our library, an *encoding function* for a constraint C takes an input list of boolean literals and returns an encoding for C on those literals.¹ An encoding function is *correct* for C if the formulas it produces encode C on all valid inputs. We will see in Section IV that this notion of correctness will need to be augmented to account for fresh variable generation.

III. THE CONSTRAINTS AND THEIR ENCODINGS

We now discuss the constraints and encodings that appear in our proof library and develop the intuition for why the encodings are correct. These intuitions form the basis for the correctness proofs presented in Section V.

A. The parity constraint

The n -ary *parity constraint* is encountered in problems from a wide range of domains, such as cryptography [19, 20], approximate model counting [22], the creation of matrix multiplication schemes [25, 26], and the construction of set membership filters [27]. Many encodings for this constraint have been proposed [26, 28–30], and to the best of our knowledge, these encodings remain unverified. In our library, we prove the correctness of two particular encodings: the direct encoding and a recursive encoding.

The parity constraint concerns the true-false parity of a set of boolean variables. Let $\text{PARITY}(X)$ be satisfied iff an odd number of the x_i are true. One way to write PARITY in propositional logic is with the XOR (\oplus) connective, where $x \oplus y = \top$ iff exactly one of x and y are true. We can thus write $\text{PARITY}(X)$ as $x_1 \oplus \dots \oplus x_n$.

The first encoding we examine is the *direct* (or *naive*) *encoding*. Every boolean constraint has a direct encoding that is essentially a spelled-out truth table. Direct encodings are sometimes chosen because they are simple to implement, since they don't introduce fresh variables, but they often produce formulas with many clauses, and so they are not preferred on large inputs. In the case of the parity constraint, its direct encoding produces a formula with 2^{n-1} clauses. Such a formula quickly becomes intractable for solvers.

¹It is convenient for encoding functions to accept lists of *literals* as input rather than lists of variables since that allows the inputs to be negated. This is useful when implementing several encodings, especially recursive ones.

Definition 2 (Direct encoding of PARITY): The direct encoding of PARITY on boolean literals $X = x_1, \dots, x_n$ is

$$\text{DIRECT}_{\text{PARITY}}(X) = \bigwedge_{\text{even \# of negations}} \left(\bigvee_{i=1}^n \pm x_i \right).$$

To see how the encoding works, consider any assignment τ that does not satisfy PARITY . We know by the definition of PARITY that an even number of the x_i must be true under τ . Since the direct encoding includes every clause with an even number of negations, we can find the clause that negates exactly those x_i that are true under τ . That clause evaluates to false under τ . Thus, the only truth assignments that satisfy every clause are those that set an odd number of the x_i to true, which are precisely the assignments that satisfy PARITY .

The second encoding is a recursive one loosely based on the Tseitin transformation [31]. The Tseitin transformation takes a propositional logic formula F and produces an equisatisfiable CNF formula that has length linear in the size of F by recursively introducing fresh literals via if-and-only-if relations with sub-formulas. This method of introducing fresh variables is used in the recursive encoding.

We first fix a *cutting number* $k \geq 3$ to determine how to split $x_1 \oplus \dots \oplus x_n$ into two sub-constraints. We then replace the first $k-1$ literals with a fresh literal and recurse:

$$\begin{aligned} R_k(X) &= (\text{PARITY}(X_{[1,k]}) \leftrightarrow y) \wedge (y \oplus R_k(X_{[k,n]})) \\ &= \text{DIRECT}_{\text{PARITY}}(X_{[1,k]}, \bar{y}) \wedge R_k(y, X_{[k,n]}), \end{aligned}$$

where we get the second line by rearranging the variables (recall that $a \leftrightarrow b$ is equivalent to $a \oplus \bar{b}$) and by replacing the left instance of the parity constraint with the direct encoding.

Note that because \oplus is commutative, we have a choice of where to place y in the recursive step. In practice, it is common to place y in either the leftmost or rightmost position. Encodings that use the former method are called *linear*; the latter, *pooled*. We prove the more general result that the encoding is correct for any permutation of x_k, \dots, x_n and y .

The choice of where to place y in the recursive transformation can have a big impact on solver performance. For example, consider an assignment that satisfies the parity constraint but falsifies the two clauses containing the literals x_1 and x_n in both encodings. The linear encoding requires $O(n)$ updates to its fresh literals to make the formula evaluate to true, while the pooled encoding only requires $O(\log n)$ updates [26].

The choice of cutting number is also critical for solver performance. When k is larger, each encoding introduces fewer fresh variables but at a cost of larger direct encoding sub-formulas. Applications are known for which cutting numbers of $k = 4, 6$, and 7 are optimal [22, 26, 28]. In our correctness proof, the cutting number is arbitrary.

Definition 3 (Recursive encoding for PARITY): Fix $k \geq 3$, and let p be a function that permutes lists. Then the recursive encoding for the PARITY on literals $X = x_1, \dots, x_n$ is

$$R_k(X) = \text{DIRECT}_{\text{PARITY}}(X_{[1,k]}, \bar{y}) \wedge R_k(p(y, X_{[k,n]})),$$

where y is fresh. When $n \leq k$, the direct encoding is used instead. The linear encoding places y in the leftmost position, while the pooled encoding places y in the rightmost position.

Both encodings presented in this section encode the *positive* form of the parity constraint. To encode its negation, which is satisfied iff an even number of the x_i are true, one can either encode $\text{PARITY}(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ or introduce a fresh variable z and add a unit clause to ensure that z is set to true in any satisfying assignment: $z \wedge \text{PARITY}(z, x_1, \dots, x_n)$.

B. The at-most-one constraint

Pseudo-boolean constraints appear in many applications for the SAT and maximum satisfiability problems, from scheduling to haplotype inference [21, 32–35]. One important class of pseudo-boolean constraint is the *cardinality constraint*, which can be written as

$$\sum_{i=1}^n a_i x_i \leq k \quad \text{or} \quad \sum_{i=1}^n a_i x_i \geq k,$$

where k is a fixed constant, $a_i \in \{\pm 1\}$, and $x_i = 1$ when $\tau(x_i) = \top$ and $x_i = 0$ otherwise. In our library, we assume that all $a_i = 1$, but we allow writing cardinality constraints in terms of boolean literals, so the two systems are equivalent.

The *at-most-one constraint* (AMO) is an especially common cardinality constraint. As its name indicates, it specifies that at most one of the x_i can evaluate to true. Many AMO encodings have been proposed [36–40]. In our library, we prove the correctness of the direct and sequential counter encodings.

Like for PARITY , the direct encoding for AMO is its CNF definition. It consists of binary clauses $(\bar{x}_i \vee \bar{x}_j)$ that disallow truth assignments that set both x_i and x_j to true. The encoding produces $\binom{n}{2} \in O(n^2)$ clauses and uses no fresh variables.

Definition 4 (Direct encoding of AMO): The direct encoding of the AMO constraint on boolean literals $X = x_1, \dots, x_n$ is

$$\text{DIRECT}_{\text{AMO}}(X) = \bigwedge_{1 \leq i < j \leq n} (\bar{x}_i \vee \bar{x}_j).$$

The *sequential counter encoding* [40] is a popular linear-sized encoding. It produces $3n - 4 \in O(n)$ clauses and introduces $n - 1$ *signal variables* that propagate the truth value of any true x_i to other signal variables to ensure that all later x_j remain false. Figure 1 shows the encoding under a satisfying truth assignment.

Definition 5 (The sequential counter AMO encoding): The sequential counter encoding for the AMO constraint on boolean literals $X = x_1, \dots, x_n$ is

$$\text{SC}(X) = \bigwedge_{i=1}^{n-1} \left((\bar{x}_i \vee s_i) \wedge (\bar{s}_i \vee s_{i+1}) \wedge (\bar{s}_i \vee \bar{x}_{i+1}) \right),$$

where the s_i are fresh and pairwise distinct.

In our library, we omit the clause $(\bar{s}_{n-1} \vee s_n)$ because s_n doesn't appear in any other clause. Omitting the clause keeps the number of signal variables at $n - 1$.

There are three kinds of clauses in the encoding. They are logically equivalent to

$$(x_i \rightarrow s_i) \quad \wedge \quad (s_i \rightarrow s_{i+1}) \quad \wedge \quad (s_i \rightarrow \bar{x}_{i+1}).$$

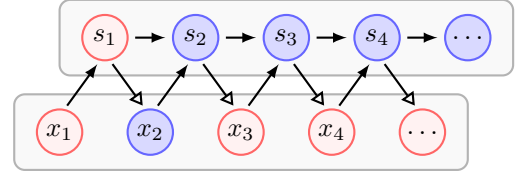


Fig. 1. The sequential counter AMO encoding under a satisfying truth assignment. Blue means the literal is true and red means the literal is false. The hollow arrow heads indicate a negated implication. Notice how the signal variables propagate that x_2 is true, enforcing that all later x_i must be false.

Writing the clauses like this makes it easier to see how the encoding works. A true x_i sets all following signal variables to true, which then forces all following x_j to false.

C. The at-most-k constraint

The sequential counter encoding can be generalized into an encoding of the *at-most-k constraint* (AMK). It introduces a $(k + 1) \times n$ matrix of signal variables and produces $O(nk)$ clauses. Clauses similar to those in the AMO encoding ensure that the matrix tracks the cumulative number of the x_i that are true. A unit clause containing the last signal variable disallows truth assignments that set more than k of the x_i to true.

Let $s_{i,j}$ be the signal variable on the i th row and j th column of the matrix. The encoding ensures that $s_{i,j}$ is set to true when at least i of x_1, \dots, x_j are true. One can think of j as defining the $X_{[1,j]}$ sub-array and i as the truth counter.

Definition 6 (The sequential counter AMK encoding): Let $k \geq 2$ be given. The sequential counter AMK encoding on literals $X = x_1, \dots, x_n$ is

$$\begin{aligned} \text{SC}_k(X) = & \left(\bigwedge_{j=1}^n (\bar{x}_j \vee s_{1,j}) \right) \wedge \left(\bigwedge_{i=1}^{k+1} \bigwedge_{j=1}^{n-1} (\bar{s}_{i,j} \vee s_{i,j+1}) \right) \\ & \wedge \left(\bigwedge_{i=1}^k \bigwedge_{j=1}^{n-1} (\bar{x}_{j+1} \vee \bar{s}_{i,j} \vee s_{i+1,j+1}) \right) \wedge \bar{s}_{k+1,n}, \end{aligned}$$

where the $s_{i,j}$ are fresh and pairwise distinct.

There are three types of clauses in the encoding. The first two appear in the AMO encoding. The third kind, the ternary clause, is logically equivalent to $(x_{j+1} \wedge s_{i,j}) \rightarrow s_{i+1,j+1}$. Whenever $s_{i,j}$ is true, meaning that at least i of x_1, \dots, x_j are true, and x_{j+1} is true, then $s_{i+1,j+1}$ is set to true. In other words, the ternary clause propagates the truth counter up a row in the signal variable matrix when a new x_{j+1} is set to true.

Figure 2 shows the encoding under a satisfying truth assignment.

IV. LIBRARY FOR VERIFIED ENCODINGS

In this section, we present our library for verifying SAT encodings. We used the interactive theorem prover Lean 3 [18] (hereafter called Lean), and our library depends on Lean's community proof library *mathlib* [41]. Our library is open-source, and all proofs and compilation instructions can be found at the following URL:

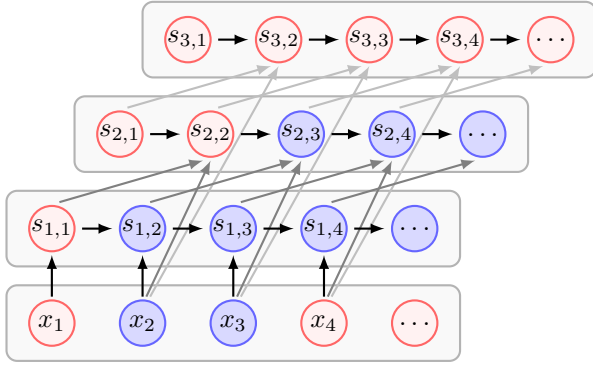


Fig. 2. The sequential counter AMK ($k = 2$) encoding under a satisfying truth assignment. Blue means that the literal is true and red means the literal is false. Notice how x_3 being true sets a new row in the signal variable matrix to true. A unit clause containing the top-rightmost signal variable disallows any assignment that sets the top row of the matrix to true.

<https://github.com/ccodel/verified-encodings>.

Lean’s axiomatic foundation is a dependent type theory with inductive types and a type of propositions. While Lean’s core logic is constructive, proofs in `mathlib` make use of classical logic. Our proofs do not depend on the specifics of Lean beyond basic facts on natural numbers, functions, lists, and sets.

Variable and theorem names in this paper may differ from those in our library since they can be verbose or cryptic. This is due to our following Lean’s naming convention. While we have edited names for readability, we have hyperlinked definitions and theorems to their counterparts in our library.

A. Library preliminaries

We start our tour of the proof library by covering the basic objects and operations we use. Most definitions are natural and correspond to general intuition about CNF formulas, clauses, and truth assignments and the interactions between them.

To avoid using a specific type for boolean variables, we use an arbitrary type V that is equipped with a computable test for equality. Literals are a sum type that are either positive (`Pos v`) or negative (`Neg v`). Truth assignments are functions from V to `bool`. Clauses and CNF formulas are represented by lists of literals and clauses, respectively.

We define operations on these types, such as evaluation and `vars(·)`, and we prove theorems about those operations. All the operations in our library are computable, meaning that Lean can execute them on explicit instances of clauses and formulas. As an example, consider the evaluation of clauses under a truth assignment τ and the statement that a clause evaluates to true under τ it has some literal that evaluates to true under τ .

```
def eval (τ : assignment V) (c : clause V) :=
  c.foldr (λ l b, b || l.eval τ) ff
theorem clause_eval_tt_iff {τ} {c} :
  c.eval τ = tt ↔ ∃ l ∈ c, l.eval τ = tt
```

The `foldr` function folds a binary operation over the elements of a list, and `l.eval τ` evaluates literal l under τ . True and false in Lean are written as `tt` and `ff`, respectively.

(The expressions `c.foldr` and `l.eval` are in Lean’s *anonymous projection notation*. Because Lean infers `l` to have type `literal`, it interprets `l.eval τ` as `literal.eval l τ`, inserting `l` as the first explicit argument of the correct type. Similarly, because the type clause V reduces to `list V`, Lean interprets `c.foldr` as `list.foldr c`. We use this notation often.)

One consequence of our decision to represent truth assignments as maps from V to `bool` is that any assignment in our library is a *full* assignment. However, we have defined in this paper that assignments are (potentially) *partial* maps on sets of variables. Having assignments be full maps makes it easier to construct and combine them, but it adds a small amount of overhead in correctness proofs to manage the sets on which the assignments are “defined.” It also requires us to modify some definitions in Section II. For example, instead of saying that τ_2 extends τ_1 , we say that τ_2 agrees with τ_1 on a specified set of variables V (`agree_on`). Thus, when τ_1 and τ_2 agree on the variables in a clause, a formula, etc., evaluation and other operations are equivalent under the two assignments.

A common pattern in our library is to start with an assignment τ_1 that satisfies a property on a set of variables V , and then “extend” it to a new assignment by setting explicit truth values for variables not in V . One way to construct such assignments is to use `aite` (short for “assignment if-then-else”). Then, as long as the object under consideration only has variables in (or not in) V , the `aite` assignment can be reduced back to one of τ_1 or τ_2 .

```
def aite (V : finset V) (τ₁ τ₂) :=
  λ v, if v ∈ V then τ₁ v else τ₂ v
theorem aite_pos {V} {v} :
  v ∈ V → ∀ τ₁ τ₂, (aite V τ₁ τ₂) v = τ₁ v
```

B. Fresh variable generation and management

Almost all compact SAT encodings introduce *auxiliary* or *fresh variables*, which are variables that don’t appear in the input. For mathematicians (and most computer scientists), generating fresh variables is easy: one assumes that there exists a set with enough fresh variables and that these variables can be chosen at will. But we have no such *a priori* assumption when we use Lean. So, we took inspiration from de Bruijn indices [42] and `gensym` objects [43] (such as in Lisp) to create our own `gensym` object that generates fresh variables.

In our library, a `gensym` object is a pointer n on the natural number line and an injective function $f : \mathbb{N} \rightarrow \alpha$ for α an arbitrary type. The `gensym`’s pointer starts by default at $n = 0$, but it can be initialized to a higher value, perhaps to avoid variables already present in a formula. The `fresh` operation provides a fresh variable under f and an updated `gensym` with an incremented pointer. Batches of fresh variables can be acquired with `nfresh`. Because f is injective, the generated variables are all distinct.

A useful notion for a `gensym` is its *stock*, the set of variables the `gensym` can produce. The stock S of a `gensym` g with offset n is $S(g) := \{x : \alpha \mid \exists d \in \mathbb{N}, f(n + d) = x\}$.

To ease proof burdens when proving encodings correct, we provide lemmas that state how an updated `gensym`'s stock and generated fresh variables relate to the original stock. For example, here are two lemmas we use often.

```
lemma fresh_stock_subset (g : gensym V) :
  g.fresh.2.stock  $\subseteq$  g.stock
lemma fresh_not_mem_fresh_stock (g) :
  g.fresh.1  $\notin$  g.fresh.2.stock
```

The `fresh` operation returns a pair of a variable and an updated `gensym` object. In Lean, the components of a pair are accessed through the `.1` and `.2` notation, which are abbreviations for `.fst` and `.snd`.

Sometimes it is more convenient to index into a `gensym`'s stock rather than request fresh variables. The `nth` operation takes a number i and returns the fresh variable that would have been generated after i calls to `fresh`, but without updating the `gensym`. Using `nth` makes proving correctness more challenging, however, since many lemmas associated with `fresh` cannot be applied to `nth`, so `fresh` is the recommended operation.

An equivalent definition for `gensym` is to only manage the injective function f , where calls to `fresh` would give back $f := (\lambda n, f(n+1))$. We chose the offset representation because it is easier to reason about natural numbers in Lean than anonymous lambda functions. For example, it is easy to prove that $g.nth\ i \neq g.nth\ j$ when $i \neq j$ due to the injectivity of f , whereas the proof for the alternate definition would be a more roundabout induction proof.

C. Encodings and correctness

We now have enough tools and machinery at hand to discuss how the encodings and their proofs of correctness appear in our library. We start by defining when a formula encodes a constraint. In our library, a constraint is a function from `list bool` to `bool`. Using `agree_on` in place of assignment extension, we represent Definition 1 like so:

```
def encodes (C) (F) (l : list (literal V)) :=
   $\forall \tau, (C.eval\ \tau\ l = tt) \leftrightarrow \exists \sigma, F.eval\ \sigma = tt$ 
   $\wedge (agree\_on\ \tau\ \sigma\ (vars\ l))$ 
```

Encoding functions take lists of literals to CNF formulas. In our library, we require a `gensym` to generate fresh variables, so we add the `gensym` as an explicit input and output.

```
def enc_fn (V : Type*) := list (literal V)  $\rightarrow$ 
  gensym V  $\rightarrow$  cnf V  $\times$  gensym V
```

The definition of correctness follows naturally from the one in Section II: that for any input list of literals l , the resulting formula produced by the encoding function encodes the constraint on l . We must also add the assumption that the variables in l and the stock of the provided `gensym` are disjoint, ensuring that the fresh variables are actually fresh.

```
def is_correct (C) (e : enc_fn V) :=
   $\forall \{l\} \{g\}, disjoint\ (vars\ l)\ g.stock \rightarrow$ 
  encodes C (e l g).1 l
```

Often, proving correctness is insufficient. Many encodings are comprised of sub-encodings, and to prove that these composite encodings are correct, we need to know that the sub-encoding functions “play nice” with the `gensym` object as it passes from one to the next. Otherwise, the combination of two encoding functions may result in unexpected behavior. For example, fresh variables could be taken from the `gensym`'s stock without updating the `gensym`, leading to variable clash when sub-formulas are combined.

To solve this problem, we introduce a notion of well-behavedness. Intuitively, an encoding function is *well-behaved* if the variables of its formulas either come from its input list l or from the `gensym`'s stock, and its output `gensym` is updated to avoid those fresh variables. All the encodings in this paper are well-behaved.

```
def is_wb (e : enc_fn V) :=  $\forall \{l\} \{g\},$ 
  disjoint (vars l) g.stock  $\rightarrow$ 
  (e l g).2.stock  $\subseteq$  g.stock  $\wedge$ 
  (e l g).1.vars  $\subseteq$  (vars l)  $\cup$ 
  (g.stock  $\setminus$  (e l g).2.stock)
```

Well-behaved encoding functions can be combined together safely. We define an append operation, written as `++`, to run one encoding function and then the other on the same list of input literals. If each well-behaved encoding function encodes a constraint, then their combination is also well-behaved and encodes the boolean-AND of the two constraints.

```
def append (e1 e2) : enc_fn V :=  $\lambda\ l\ g,$ 
  let  $\langle F_1, g_1 \rangle := e_1\ l\ g$  in
  let  $\langle F_2, g_2 \rangle := e_2\ l\ g_1$  in  $\langle F_1 ++ F_2, g_2 \rangle$ 
```

We also define a `fold` operation that folds `append` over a list of encodings from left to right in the natural way. Analogous `append` and `fold` operations are defined for constraints as well, where `append` is the boolean-AND of the outputs.

V. PROVING THE ENCODINGS CORRECT

In this section, we present the correctness proofs for the encodings we presented in Section III. The proofs generally follow the intuition of correctness given with the definition of the encodings, but we report challenges, quirks, or surprises.

A. The parity encodings

We represented `PARITY` by folding \oplus (written as `bxor` in Lean) across the input. A lemma states that the constraint is satisfied iff an odd number of inputs are true.

```
def parity :=  $\lambda\ l, l.foldr\ bxor\ ff$ 
lemma parity_eq_bodd : parity.eval  $\tau\ l =$ 
  bodd (clause.count_tt  $\tau\ l)$ 
```

We implemented the direct encoding by adding either x_1 or its negation to each of the 2^{n-1} (ordered) clauses on x_2, \dots, x_n . The proof of correctness is short (only about 30 lines) and proceeds along the already given intuition: By specifying that all clauses in the encoded formula have an even number of negations, any falsifying assignment for `PARITY` has a corresponding clause in the formula that it does not satisfy.

The recursive encoding and its correctness proof are more interesting. In Lean, we take a cutting number k and a permutation function p and implement the encoding recursively.

```
def recursive_parity {k} (hk : k ≥ 3) {p}
  (hp : ∀ l, perm l (p l)) : enc_fn V
| l g := if l.length ≤ k then
  direct_parity l g else
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨lhd, ltl⟩ := l.split (k - 1) in
  let ⟨Frec, g₂⟩ := recursive_parity (
    p (Pos y :: ltl) ) g₁ in
  ⟨(direct_parity (lhd ++ [Neg y]) g₁).1
    ++ Frec, g₂⟩
```

The `perm` relation specifies whether two lists are permutations of each other. The `split` operation returns two halves of the list, split at the specified index.

The proof of correctness proceeds by strong induction on the input list l . Let τ be the truth assignment in the encodes judgment. The reverse direction (that if there exists an assignment σ that agrees with τ on the variables in l and satisfies the encoded formula, then τ satisfies PARITY) is almost trivial. Applying the correctness proof for the direct encoding and the induction hypothesis on the recursive sub-formula gives two satisfied PARITY constraints. Dropping the fresh variable y in both gives a single satisfied PARITY constraint.

The forward direction is more involved. To use the induction hypothesis, we must show that $\text{PARITY}(y, X_{[k,n]})$ is satisfied under some truth assignment. We construct such a truth assignment ν by extending τ to include a truth value for the fresh variable y . If $\text{PARITY}(X_{[k,n]})$ is satisfied under τ , then y is set to false in ν , and true otherwise. The induction hypothesis on the sub-formula returns an assignment σ that satisfies the sub-formula and that agrees with ν on $\{y\} \cup X_{[k,n]}$. Combining σ with τ on $X_{[1,k]}$ via `aite` finishes the proof.

The general takeaway is that for recursively-defined encodings, the proof of correctness proceeds by (strong) induction on the input list and requires the explicit setting of truth values for one or more variables in an extended assignment, especially among the fresh variables. Lemmas that manipulate and reduce `aite` constructions are helpful, but the management of hypotheses about set membership ultimately remains tedious.

B. The at-most encodings

We defined the AMK constraint with Lean's `list.count` operation, which counts the number of elements in a list that match a given element. The AMO constraint is `amk 1`. The at-least- k constraint `ALK` and the at-least-one constraint `alo` are defined analogously.

```
def amk (k : nat) := λ l, l.count tt ≤ k
```

We implemented the direct encoding for AMO as a recursive function. Because the direct encoding doesn't require any fresh variables, we defined a base function `direct_amo'` to produce the formula. The actual encoding function passes the `gensym` through untouched.

```
def direct_amo' : list (literal V) → cnf V
| [] := []
```

```
| (lit :: ls) := (ls.map (λ m,
  [lit.flip, m.flip])) ++ (direct_amo' ls)
```

The correctness proof is straightforward and proceeds by relating both the AMO constraint and the clauses in the direct encoding to the truth value of any two elements in distinct positions in the list via a distinct proposition we defined.

```
def distinct {α} (a₁ a₂ : α) (l : list α) :=
  ∃ (i j : nat) (Hi : i < l.length)
    (Hj : j < l.length), i < j ∧
  l.nth_le i Hi = a₁ ∧ l.nth_le j Hj = a₂
```

We now discuss the sequential counter encodings, starting with the AMO encoding. Unlike for the recursive encoding for PARITY, the AMO encoding isn't inherently recursive, but the three clauses have the same form for each i , so a recursive implementation is possible. However, a non-recursive implementation may allow for lemmas that better capture the global behavior of the signal variables. We implemented both a recursive and non-recursive encoding function in our library to compare the proof efforts of the two methods.

The non-recursive `sc_amo` uses three helper functions that each generate one of the three types of clauses. We provide one of them, `xi_to_si`, below as an example. We omit the cases where the input list has cardinality at most one.

```
def sc_amo : enc_fn V
| l g := let n := length l in
  ⟨join (map_with_index (λ idx lit,
    xi_to_si g n idx lit ++
    si_to_next_si g n idx ++
    si_to_next_xi g idx lit) l),
    (g.nfresh (n - 1)).2⟩

def xi_to_si (g) (n i : nat) (lit) : cnf V :=
  if i < n - 1 then
    [[lit.flip, Pos (g.nth i)]] else []
```

The function `map_with_index` applies a function to each element in a list along with its index in the list. We use `map_with_index` to access the corresponding signal variable for each literal in l . The function `join` flattens a list of lists into a single list.

In the recursive implementation `sc_rec`, we generate a fresh signal variable y at each recursive level. We also generate a second fresh variable z since we need to produce a clause with two adjacent signal variables, but the `gensym` given to the recursive call is the one that was produced by a single call to `fresh`. We once again omit the trivial cases.

```
def sc_rec : enc_fn V
| [l₁, l₂] g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩
| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F', g₂⟩ := sc_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
    [Neg y, l₂.flip]] ++ F', g₂⟩
```

The correctness proof for `sc_rec` proceeds by induction on the input list l . For the forward direction, the proof strategy is

similar to the one for `recursive_parity`. An assignment that extends the given τ is constructed by setting the truth value for signal variable s_1 to $\tau(x_1)$. If s_1 is set to true, then the assignment that sets all signal variables to true is proved to satisfy the formula. Otherwise, s_1 is set to false in the extended assignment, and the induction hypothesis gives back an assignment σ that satisfies the recursive sub-formula. An `aite` construction finishes the proof.

Now the reverse direction. If $\tau(x_1)$ is true, then we use a lemma stating that because τ satisfies the formula, all other x_i must be false under τ (due to all signal variables being true). Otherwise, the induction hypothesis gives that $\text{AMO}(x_2, \dots, x_n)$ is satisfied under τ , and since $\tau(x_1)$ is false, the full constraint is satisfied.

The correctness proof for the non-recursive `sc_amo` required engineering that was not present in the proof for `sc_rec`. The additional engineering contributed to the proof being 50% larger (300 vs. 200 lines).

In the forward direction, we supply an explicit truth assignment `sc_tau` that provides the truth values for all signal variables at once. A helper function `var_idx` extracts the index i for each provided signal variable. Proving that `sc_tau` satisfies the encoded formula was straightforward but tedious.

For the reverse direction, if no x_i is true under τ , then of course the AMO constraint is satisfied. Otherwise, the proof uses lemmas showing that the signal variables propagate the truth value of x_i appropriately. Because the lemmas were stated in terms of “if x_i is true, then all later x_j are false,” `distinct` was used to finish the proof.

There are two main takeaways. The first is that the recursive implementation and its correctness proof are more compact than the non-recursive version’s. Because the fresh variables generated at each recursive level are largely independent, the induction hypothesis can be leveraged effectively.

The second takeaway is that much of the proof overhead in the non-recursive case came from managing hypotheses about set membership among the fresh variables (i.e., that they are disjoint from the literals in `l` and the updated stock, and that they were distinct from each other). As we developed our library, we added more lemmas to ease these proof burdens, but ultimately, proving facts like `g.nth i ∉ l.take j` when `g.stock` and `l` are disjoint will persist. Future work could address this burden by automating the proving of these facts.

We implemented the AMK encoding in a non-recursive manner analogous to `sc_amo`. Since the implementation and correctness proof are so similar, we omit the details.

VI. ENCODING SUDOKU

To demonstrate the use of our proof library, we implemented an encoding for the Sudoku problem using AMO sub-encodings. Because the encoding was formed by composing well-behaved, correct sub-encodings, its correctness proof was only 15 lines. In addition, it uses an abstract sub-encoding `amo_enc` that can be defined to be any correct sub-encoding, which happily agrees with the mathematical view of encodings.

4		3						
					7	9		
			6					
			1	4		5		
9							1	
2								6
				7	2			
	5					8		
			9					

1	4	7	3	8	9	2	6	5
5	8	6	2	1	4	7	9	3
3	9	2	6	5	7	1	8	4
8	7	3	1	4	6	5	2	9
9	6	4	7	2	5	3	1	8
2	1	5	9	3	8	4	7	6
6	3	8	5	7	2	9	4	1
7	5	9	4	6	1	8	3	2
4	2	1	8	9	3	6	5	7

Fig. 3. A 9-by-9 Sudoku puzzle (left) and its unique solution (right). Every row, column, and 3-by-3 subgrid that compose the grid must have exactly one each of the numbers 1 through 9.

Sudoku is a classic Japanese puzzle where one must fill in a number between 1 and 9 in every cell in a 9-by-9 grid such that every row, column, and 3-by-3 subgrid comprising the grid must contain each number 1 through 9 exactly once. Numbers present in the grid from the start define a single unique solution to the puzzle. Figure 3 depicts a difficult Sudoku puzzle and its solution. The general Sudoku problem is parameterized by n , the side length for a single subgrid. In the figure, $n = 3$.

One encoding for Sudoku is to use AMO constraints for each cell, row, column, and square, along with an ALO constraint on each cell. Let $X = \{x_{r,c,k}\}$ be the set of boolean variables used in the encoding. Setting $x_{r,c,k}$ to true means placing number k in the cell in row r and column c . Each of r , c , and k run from 1 to n^2 , making a total of n^6 variables. For example, the AMO constraint on the rows would look like

$$\bigwedge_{r=1}^{n^2} \bigwedge_{k=1}^{n^2} \text{AMO}(x_{r,1,k}, \dots, x_{r,n^2,k}).$$

Implementing the Sudoku encoding in Lean using the machinery we’ve discussed so far seems challenging. At first blush, it looks like the `append` operation could combine all the sub-encodings together, but `append` combines encodings that share an input list of literals. In the Sudoku encoding discussed above, each ALO and AMO constraint is expressed on a *different subset* of the n^6 variables in X . So `append` won’t work without some modification.

Our solution is to compose each ALO and AMO encoding function with a function `filter_by_idx` that filters out indexes in a list that don’t satisfy a given predicate. This way, all of the encoding functions can be folded together, since each will extract the literals it needs. An example of one predicate we use, `is_cell_lit`, returns whether a list index corresponds to one of the literals associated with a particular cell. The filter functions help define the sub-constraints on each cell, row, column, and square on the Sudoku board.

```
def is_cell_lit (n row col : nat) := λ idx,
  idx ∈ (range (n^2)).map (λ num,
    (row * n^4) + (col * n^2) + num)
```

The function `range k` returns a list $[0, 1, \dots, k - 1]$.

(The presentation of `is_cell_lit` above elides many bookkeeping details. For instance, we use Lean’s `fin` type instead of `nat`. However, the logical core is the same.)

In the spirit of Wadler’s “Theorems for Free!” [44], the correctness and well-behavedness of encoding functions are preserved under operations on arbitrary lists, such as permutation and element copying and deletion, since our notion of correctness refers only to the values in the list and not the list itself. We use the theorem that `filter_by_idx`, which returns a sublist of its input, preserves an encoding function’s correctness to prove the Sudoku encoding correct.

We can now define the Sudoku constraint. We note that while the constraint isn’t the most efficient one possible, it is the most natural, and the redundant clauses in the encoded formula help SAT solvers in practice. The encoding is defined almost identically by swapping in the abstract encoding function `amo_enc` in place of the sub-constraints.

```
def is_valid_sudoku (n : nat) :=
  let L := cart_prod (n^2) (n^2) in
  fold (L.map (λ ⟨r, c⟩, is_cell_valid r c)) ++
  fold (L.map (λ ⟨c, k⟩, is_row_valid c k)) ++
  fold (L.map (λ ⟨r, k⟩, is_col_valid r k)) ++
  fold ((cart_prod n n).zip
    (fin.range (n^2))).map
    (λ ⟨⟨sr, sc⟩, k⟩, is_subgrid_valid sr sc k)
```

We omit a check by the function `len_check`, which ensures that the constraint only accepts lists of appropriate length (of length n^6). The function `cart_prod` returns a list of pairs representing the Cartesian product of the universe of fintypes. For example, `cart_prod 2 3 = [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]`.

To demonstrate Lean’s ability to produce these encodings, we had Lean generate and save the Sudoku encoding to a file in DIMACS format, which can then be sent to conventional SAT solvers. The file can be found in our library.

VII. RELATED WORK

We are not the first to verify SAT encodings using a proof assistant. Our work is most similar to Giljegård and Wennerbreck’s library for verified SAT encodings [45], written in CakeML [46]. In their library, they verified the correctness of the naive AMO encoding, several encodings of pseudo-boolean constraints, and the Tseitin transformation for turning arbitrary propositional logic formulas into CNF. They also provide a way to translate mathematical objects (e.g., unordered sets and natural numbers) into SAT analogues, which helps with the writing of encodings. They then applied their verified encodings to logic puzzles like the n -queens problem and Sudoku variants.

Our work improves on Giljegård and Wennerbreck’s library in two main regards. The first is our richer set of operations and theorems on CNF objects. For example, constructing composite truth assignments with `aite` is crucial for proving the correctness of encodings that introduce fresh variables.

The second improvement is our library’s management of fresh variables. While Giljegård and Wennerbreck’s library also generates fresh variables to implement the Tseitin transformation, their reasoning about fresh variables is specialized to the

Tseitin transformation, as it is the only place in their library where fresh variables are used. What’s more, they state that they did not implement more-efficient encodings for pseudo-boolean constraints due to the challenges of fresh variable management. Our library solves this problem.

Our work also shares similarities with Luís Cruz-Filipe, et al.’s work on an end-to-end verification of the encoding of the Pythagorean triples problem [47]. They verified the encoding, and additional symmetry breaking techniques, in Coq [48]. Their types for literals, clauses, and CNF formulas are identical to ours, and their notion of encoding correctness agrees with our definition. However, because the encoding they verified did not introduce fresh variables, they did not develop any infrastructure for managing fresh variables.

Other verification efforts are domain-specific and mainly translate other logical systems into SAT [49–51]. For example, Ishii and Fujii verify an encoding of SAT-based model checking in Coq. They formalize methods such as k -induction and property-directed reachability to check the safety of state-transition systems, and then they prove the soundness of converting safety properties expressed using these methods into SAT. By taking in a fixed number of transitions k , they express the safety properties in terms of a finite logical formula, which is then converted into propositional logic.

Our work has already seen application beyond this paper. Holliday, Norman, and Pacuit [52] used our library to verify their SAT encoding of problems in voting theory.

VIII. CONCLUSION AND FUTURE WORK

Our library has laid the groundwork for formally verifying SAT encodings. The encodings we verify are efficient in practice, even at large scales, and we have made efforts to develop a framework that is general, extensible, and easy to use in practice. So far, we have verified the correctness of encodings for the parity, at-most-one, and at-most- k constraints and an encoding of Sudoku. To do so, we developed methods of generating fresh variables, constructing extended assignments, and combining sub-encodings, and we proved lemmas that allow us to reason about these operations.

Despite our progress, there is still much left to do. We plan to upgrade our library to Lean version 4, which offers better automation and linking to SAT and satisfiability modulo theory solvers. We will also rewrite our `gensym` in terms of a state monad to simplify writing encodings. Finally, many encodings are still unverified. For example, a wide number of SAT-solving applications require efficient encodings of cardinality constraints [40, 53], pseudo-boolean constraints [54, 55], and symmetry-breaking predicates [56, 57].

In the long run, our goal is to provide tools so that any claim established with a SAT solver can be fully verified from start to finish. Increasingly-complex mathematical theorems and claims about hardware and software are being reduced to propositional search problems, and these reductions are becoming more subtle and involved. Interactive theorem proving therefore has an important role to play, and our goal is to develop a library that can adequately support the task.


REFERENCES


- [1] F. Copt, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi, “Benefits of bounded model checking at an industrial setting,” in *CAV*, pp. 436–453, Springer, 2001.
- [2] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, “Efficient SAT-based bounded model checking for software verification,” *Theoretical Computer Science*, vol. 404, no. 3, pp. 256–274, 2008.
- [3] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 171–177, Springer, 2011.
- [4] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.
- [5] J. Brakensiek, M. J. H. Heule, J. Mackey, and D. Narváez, “The Resolution of Keller’s Conjecture,” in *Automated Reasoning* (N. Peltier and V. Sofronie-Stokkermans, eds.), (Cham), pp. 48–65, Springer International Publishing, 2020.
- [6] M. J. H. Heule, O. Kullmann, and V. W. Marek, “Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer,” in *Theory and Applications of Satisfiability Testing – SAT 2016* (N. Creignou and D. Le Berre, eds.), (Cham), pp. 228–245, Springer International Publishing, 2016.
- [7] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands: IOS Press, 2009.
- [8] S. A. Cook, “The complexity of theorem-proving procedures,” in *STOC ’71: Proceedings of the third annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 151–158, ACM, 1971.
- [9] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, “Efficient generation of unsatisfiability proofs and cores in SAT,” in *LPAR*, pp. 16–30, 2008.
- [10] A. Van Gelder, “Producing and verifying extremely large propositional refutations - have your cake and eat it too,” *Ann. Math. Artif. Intell.*, vol. 65, no. 4, pp. 329–372, 2012.
- [11] M. J. H. Heule, W. A. Hunt, Jr., and N. Wetzler, “Verifying refutations with extended resolution,” in *International Conference on Automated Deduction (CADE)*, vol. 7898 of *LNAI*, pp. 345–359, Springer, 2013.
- [12] E. I. Goldberg and Y. Novikov, “Verification of proofs of unsatisfiability for CNF formulas,” in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 10886–10891, IEEE, 2003.
- [13] M. J. H. Heule, W. A. Hunt, Jr., and N. Wetzler, “Trimming while checking clausal proofs,” in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 181–188, IEEE, 2013.
- [14] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, “Efficient certified RAT verification,” in *Automated Deduction – CADE 26* (L. de Moura, ed.), (Cham), pp. 220–236, Springer International Publishing, 2017.
- [15] L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp, “Efficient certified resolution proof checking,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 10205, pp. 118–135, 2017.
- [16] M. J. H. Heule, W. Hunt, M. Kaufmann, and N. Wetzler, “Efficient, verified checking of propositional proofs,” in *Interactive Theorem Proving* (M. Ayala-Rincón and C. A. Muñoz, eds.), (Cham), pp. 269–284, Springer International Publishing, 2017.
- [17] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, “cake_lpr: Verified propagation redundancy checking in CakeML,” in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II* (J. F. Groote and K. G. Larsen, eds.), vol. 12652 of *Lecture Notes in Computer Science*, pp. 223–241, Springer, 2021.
- [18] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, “The Lean theorem prover (system description),” in *Conference on Automated Deduction (CADE) 2015* (A. P. Felty and A. Middeldorp, eds.), pp. 378–388, Springer, Berlin, 2015.
- [19] A. Leventi-Peetz, O. Zendel, W. Lennartz, and K. Weber, “CryptoMiniSat switches-optimization for solving cryptographic instances,” in *Proceedings of Pragmatics of SAT 2015 and 2018* (D. L. Berre and M. Järvisalo, eds.), vol. 59 of *EPiC Series in Computing*, pp. 79–93, EasyChair, 2019.
- [20] M. Soos, K. Nohl, and C. Castelluccia, “Extending SAT solvers to cryptographic problems,” in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings* (O. Kullmann, ed.), vol. 5584 of *Lecture Notes in Computer Science*, pp. 244–257, Springer, 2009.
- [21] A. Graça, I. Lynce, J. Marques-Silva, and A. L. Oliveira, “Efficient and accurate haplotype inference by combining parsimony and pedigree information,” in *Algebraic and Numeric Biology* (K. Horimoto, M. Nakatsui, and N. Popov, eds.), pp. 38–56, Springer Berlin Heidelberg, 2012.
- [22] M. Soos and K. S. Meel, “BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting,” in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference,*


- IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pp. 1592–1599, AAAI Press, 2019.
- [23] J. P. Marques-Silva and I. Lynce, “Towards robust CNF encodings of cardinality constraints,” in *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings* (C. Bessiere, ed.), vol. 4741 of *Lecture Notes in Computer Science*, pp. 483–497, Springer, 2007.
 - [24] N. Manthey, M. J. H. Heule, and A. Biere, “Automated reencoding of Boolean formulas,” in *Hardware and Software: Verification and Testing* (A. Biere, A. Nahir, and T. Vos, eds.), (Berlin, Heidelberg), pp. 102–117, Springer Berlin Heidelberg, 2013.
 - [25] M. J. H. Heule, M. Kauers, and M. Seidl, “New ways to multiply 3×3 -matrices,” *J. Symb. Comput.*, vol. 104, pp. 899–916, 2021.
 - [26] W. Nawrocki, Z. Liu, A. Fröhlich, M. J. H. Heule, and A. Biere, “XOR local search for Boolean brent equations,” in *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings* (C. Li and F. Manyà, eds.), vol. 12831 of *Lecture Notes in Computer Science*, pp. 417–435, Springer, 2021.
 - [27] S. A. Weaver, H. J. Roberts, and M. J. Smith, “XOR-satisfiability set membership filters,” in *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings* (O. Beyersdorff and C. M. Wintersteiger, eds.), vol. 10929 of *Lecture Notes in Computer Science*, pp. 401–418, Springer, 2018.
 - [28] G. V. Bard, N. T. Courtois, and C. Jefferson., “Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over $\text{GF}(2)$ via SAT-solvers.” Cryptology ePrint Archive, Report 2007/024, 2007.
 - [29] M. Gwynne and O. Kullmann, “A framework for good SAT translations, with applications to CNF representations of XOR constraints,” 2014.
 - [30] M. Gwynne and O. Kullmann, “On SAT representations of XOR constraints,” in *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings* (A. Dediu, C. Martín-Vide, J. L. Sierra-Rodríguez, and B. Truthe, eds.), vol. 8370 of *Lecture Notes in Computer Science*, pp. 409–420, Springer, 2014.
 - [31] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” in *Automation of Reasoning 2* (J. Siekmann and G. Wrightson, eds.), pp. 466–483, Springer, 1983.
 - [32] W. Küchlin and C. Sinz, “Proving consistency assertions for automotive product data management,” *Journal of Automated Reasoning*, vol. 24, no. 1, pp. 145–163, 2000.
 - [33] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners, “Radio link frequency assignment,” *Constraints*, vol. 4, no. 1, pp. 79–89, 1999.
 - [34] M. Jose and R. Majumdar, “Cause clue clauses: Error localization using maximum satisfiability,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’11*, p. 437–446, Association for Computing Machinery, 2011.
 - [35] R. Asín Achá and R. Nieuwenhuis, “Curriculum-based course timetabling with SAT and MaxSAT,” *Annals of Operations Research*, vol. 218, no. 1, pp. 71–91, 2014.
 - [36] O. Bailleux and Y. Bouffkhad, “Efficient CNF encoding of Boolean cardinality constraints,” in *Principles and Practice of Constraint Programming - CP 2003* (F. Rossi, ed.), (Berlin, Heidelberg), pp. 108–122, Springer Berlin Heidelberg, 2003.
 - [37] A. Frisch, T. Peugniez, A. Doggett, and P. Nightingale, “Solving non-Boolean satisfiability problems with stochastic local search: A comparison of encodings,” *Journal of Automated Reasoning*, vol. 35, pp. 143–179, 01 2005.
 - [38] W. Klieber and G. Kwon, “Efficient CNF encoding for selecting 1 from N objects,” in *Fourth Workshop on Constraint in Formal Verification (CFV)*, 2007.
 - [39] V.-H. Nguyen and S. T. Mai, “A new method to encode the at-most-one constraint into SAT,” in *Proceedings of the Sixth International Symposium on Information and Communication Technology, SoICT 2015, (New York, NY, USA)*, p. 46–53, Association for Computing Machinery, 2015.
 - [40] C. Sinz, “Towards an optimal CNF encoding of Boolean cardinality constraints,” in *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings* (P. van Beek, ed.), vol. 3709 of *LNCS*, pp. 827–831, Springer, 2005.
 - [41] The mathlib community, “The Lean mathematical library,” in *Certified Programs and Proofs (CPP) 2020* (J. Blanchette and C. Hritcu, eds.), pp. 367–381, ACM, 2020.
 - [42] N. de Bruijn, “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem,” *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5, pp. 381–392, 1972.
 - [43] A. Filinski, “Normalization by evaluation for the computational lambda-calculus,” in *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications, TLCA’01, (Berlin, Heidelberg)*, pp. 151–165, Springer-Verlag, 2001.
 - [44] P. Wadler, “Theorems for free!,” in *Conference on Functional Programming Languages and Computer Architecture*, 1989.
 - [45] S. Giljegård and J. Wennerbreck, “Puzzle solving with proof,” Master’s thesis, Chalmers University of Technology, University of Gothenburg, 2021.
 - [46] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens,

- “CakeML: A verified implementation of ML,” in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’14, (New York, NY, USA), pp. 179–191, Association for Computing Machinery, 2014.
- [47] L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp, “Formally verifying the solution to the Boolean Pythagorean triples problem,” *J. Autom. Reason.*, vol. 63, no. 3, pp. 695–722, 2019.
 - [48] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.
 - [49] K. Anuarul Hoque, O. A. Mohamed, S. Abed, and M. Boukadoum, “An automated SAT encoding-verification approach for efficient model checking,” in *2010 International Conference on Microelectronics*, pp. 419–422, 2010.
 - [50] D. Ishii and S. Fujii, “Formalizing the soundness of the encoding methods of SAT-based model checking,” in *International Symposium on Theoretical Aspects of Software Engineering, TASE 2020, Hangzhou, China, December 11-13, 2020* (T. Aoki and Q. Li, eds.), pp. 105–112, IEEE, 2020.
 - [51] M. Abdulaziz and F. Kurz, “Formally verified SAT-based AI planning,” 2020.
 - [52] W. H. Holliday, C. Norman, and E. Pacuit, “Voting theory in the Lean theorem prover,” in *Logic, Rationality, and Interaction: 8th International Workshop, Lori 2021, Xi’an, China, October 16-18, 2021, Proceedings* (S. Ghosh and T. Icard, eds.), pp. 111–127, Springer Verlag, 2021.
 - [53] J.-C. Régin, “Generalized arc consistency for global cardinality constraint,” in *14th National Conference on Artificial Intelligence (AAAI 1996)*, vol. 1, pp. 209–215, AAAI Press / The MIT Press, 1996.
 - [54] O. Bailleux, Y. Boufkhad, and O. Roussel, “A translation of pseudo-Boolean constraints to SAT,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 183–192, 2006. Special Issue on SAT 2005 competition and evaluations.
 - [55] N. Eén and N. Sörensson, “Translating Pseudo-Boolean Constraints into SAT,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–25, 2006. Special Issue on SAT 2005 competition and evaluations.
 - [56] F. A. Aloul, K. A. Sakallah, and I. L. Markov, “Efficient symmetry breaking for Boolean satisfiability,” *IEEE Trans. Computers*, vol. 55, no. 5, pp. 549–558, 2006.
 - [57] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, “Symmetry-breaking predicates for search problems,” in *Proc. KR’96, 5th Int. Conf. on Knowledge Representation and Reasoning*, pp. 148–159, Morgan Kaufmann, 1996.

SAT-Based Quantified Symmetric Minimization of the Reachable States of Distributed Protocols

Katalin Fazekas 
 TU Wien
 Vienna, Austria
 katalin.fazekas@tuwien.ac.at

Aman Goel 
 Amazon Web Services[§]
 Seattle, USA
 goelaman@amazon.com

Karem A. Sakallah 
 University of Michigan
 Ann Arbor, USA
 karem@umich.edu

Abstract—Most of the recent published work on the automated verification of distributed protocols has been concerned with deriving an inductive invariant that implies a safety specification. In this paper we argue that the inherent structural symmetry of protocols strongly suggests the existence of a unique property-independent formula r_{min} that describes a protocol's reachable states as a minimum-cost conjunction of quantified first-order logic predicates. We show, for finite instances, that these predicates correspond to symmetry orbits of prime implicants, and show how they are derived using a novel SAT-based logic minimization algorithm which relies on the connection between symmetry and quantification as complementary ways of representing these orbits. We also present empirical data showing that the minimum-cost orbits derived for increasing protocol sizes converge syntactically, reaching a fixed point at a relatively small critical size. Our findings, thus, confirm earlier observations about the cutoff and saturation phenomenon of parameterized systems. To our knowledge, our approach is the first to algorithmically derive quantified first-order logic formulas for the reachable states of unbounded parameterized systems, enabling the verification of any safety property.

Index Terms—Distributed protocols, logic minimization, invariant inference, symmetry, quantifier inference.

I. INTRODUCTION

Driven by the availability of modern Satisfiability Modulo Theories (SMT) solvers [1], [2], the last few years have seen increasing interest in finding ways to automate the analysis and verification of distributed protocol specifications. Most of the recent published work [3]–[9] has been concerned with deriving an inductive invariant in quantified first-order logic (FOL) that serves as a proof certificate of a protocol's safety property.

In this paper we argue that (an enhanced version of) classical logic minimization adds a new perspective that furthers our understanding of protocol behavior. Specifically we show, for a restricted class of protocol specifications, that it is possible to algorithmically derive a formula r_{min} that encodes the reachable states as an *exact minimum-cost conjunction of quantified FOL invariants*. For this purpose, we define the cost of a quantified invariant in prenex normal form (PNF) to be the sum of the number of quantifiers in its prefix and the number of literals in its matrix.

Key to deriving these minimum-cost formulas for the reachable states is the inherent structural symmetries of proto-

col specifications as well as the recently-established connection between symmetry and quantification [6]. Applied to finite protocol instances, our proposed *Quantified Symmetric Minimization (QSM)* algorithm preserves these symmetries in both the prime implicant (PI) generation and set covering phases of the classical Quine-McCluskey (QM)¹ algorithm. In addition, it replaces the unscalable tabular procedures in QM with scalable alternatives based on incremental SAT solving [10], [11]. Empirically, we also show that the finitely-quantified reachable state formulas generated by *QSM* at increasing protocol sizes reach a *syntactic fixed point* at a critical *cutoff* size and yield the minimum formula for the reachable states of the unbounded protocol. We believe this to be a direct consequence of the restrictions (elaborated later) on the class of protocols we consider, but leave a rigorous formal proof as future work.

The invariants in the r_{min} formula will be shown to be *prime implicate symmetry orbits* of the reachable states and that they represent the complete set of strengthening assertions needed to establish the validity of *any* safety property S . Intuitively, if $r_{min} \rightarrow S$ is valid, then S holds. However, the simplest explanation of why S holds might be a minimal subset of r_{min} 's orbits that acts as its strongest strengthening assertion.

Our key contributions include:

- A novel symmetry-aware SAT-based logic minimization algorithm that utilizes structural symmetry and its connection with quantification to derive an exact minimum-cost representation of the original structurally-symmetric formula as a finitely-quantified FOL formula.
- A novel forward-reachability algorithm that derives the strongest complete property-independent quantified formula r_{min}^k representing the set of reachable states for a protocol instance of size k .
- A simple property-independent procedure that derives $r_{min}^1, r_{min}^2, \dots$ for protocol instances with increasing sizes until reaching convergence at a critical cutoff size k^* , where $r_{min}^{k^*}$ syntactically converges to $r_{min}^{k^*+1}$. At the cutoff size, $r_{min}^{k^*}$ represents the strongest and complete inductive invariant that summarizes all protocol behaviors for any size.

¹The fact that *QSM* and QM have two identical initials is purely coincidental.

[§]Work does not relate to Aman Goel's position at Amazon

- The first empirical demonstration of the cutoff phenomenon for a collection of distributed protocols based on deriving a quantified FOL formula r_{min} that encodes the protocol's reachable states for all sizes.

The paper is organized as follows: Section II provides preliminaries and context. Section III details the *QSM* algorithm. Section IV shows how *QSM*, applied to increasing protocol sizes, reaches cutoff. Section V presents our experimental evaluation with Section VI giving a brief survey of related work. Section VII concludes the paper with future work directions.

II. PRELIMINARIES AND CONTEXT

We assume familiarity with the basics of 2-valued Boolean algebra including literals, minterms, prime implicants, and prime implicants of an n -variable function $f(x_1, \dots, x_n)$ as well as basic notions from group theory including permutation groups, cycle notation, orbits, etc., which can be readily found in standard textbooks on Abstract Algebra [12]. We use $primes(f)$ to denote the disjunction of f 's prime implicants, i.e., its *complete sum*. On the other hand, the *complete product*² of f is the conjunction of its prime implicants and can be expressed as $\neg primes(\neg f)$. $primes(f)$ can also be viewed as a set and we define $\#lits(\rho)$ for $\rho \in primes(f)$ to be the number of literals in ρ .

A. Exact Two-Level Minimization

Exact two-level sum-of-product (SOP/DNF) minimization is an optimization problem seeking to find a minimum-cost subset of $primes(f)$ that covers all of f 's minterms. Mathematically, the problem can be stated as finding a Boolean assignment to a set of *selector* variables $z_\rho \in \{0, 1\}$, for $\rho \in primes(f)$, that represents a solution to the following set covering problem [13]:

$$\begin{aligned} & \text{minimize} \quad \sum_{\rho \in primes(f)} cost(\rho) \times z_\rho \\ & \text{subject to} \quad \left(\bigvee_{\rho \in primes(f)} z_\rho \wedge \rho \right) = f \end{aligned} \quad (1)$$

where $cost(\rho) \triangleq \#lits(\rho)$. This formulation can also be used to find a minimum-cost product-of-sums (POS/CNF) solution by applying De Morgan's law to the minimum-cost SOP solution of $\neg f$.

B. The Quine-McCluskey Algorithm

The classical Quine-McCluskey (QM) algorithm [14]–[16] solves this problem by first deriving $primes(f)$ using a tabular procedure starting from f 's minterms, followed by a branch-and-bound search to find the optimal solution to (1). Both steps assume an explicit listing of f 's minterms. In particular, the set covering problem is represented as a 2-dimensional $\{0, 1\}$ *prime implicant chart* whose rows and columns correspond,

respectively, to f 's minterms and prime implicants. A 1 (resp. 0) entry in row μ and column ρ indicates that minterm μ is (resp. is not) covered by prime implicant ρ . In this encoding, the optimization objective is stated as finding a minimum-cost set of columns that covers all the rows.

C. Distributed Protocols

Our focus is the verification of distributed protocol *specifications*, i.e., protocols described at an abstraction level that hides code implementation details that model network topology and the effects of message interleaving, message loss, node failures, etc. Such specifications are typically encoded in FOL in such languages as TLA+ [17] or Ivy [18].

We specifically consider the class of multi-sorted *data-independent* protocol specifications [19], [20] that satisfy the following three requirements:

- The protocol sorts are unbounded sets of interchangeable *structurally-symmetric* elements.
- The protocol actions are atomic and asynchronous, i.e., they occur one at a time and interleave arbitrarily.
- The protocol encoding is in the empty theory of FOL, namely equality with uninterpreted functions.

This class encompasses a wide range of common protocols and should be considered a starting point that does not exclude future extensions to other types of protocols such as ones with totally-ordered sorts.

For purposes of illustration, and without loss of generality, in this paper we consider a protocol \mathcal{P} defined over a single unbounded sort $node \triangleq \{n_0, n_1, n_2, \dots\}$ along with a) a finite set of *relations*³ on $node$ that serve as \mathcal{P} 's state variables, and b) a finite set of *actions* that capture \mathcal{P} 's state transitions. The elements of $node$ are referred to as its *constants* and are assumed to be indistinguishable; they can be arbitrarily permuted without changing \mathcal{P} 's behavior.

A predicate Ψ on \mathcal{P} 's state variables is a closed quantified FOL expression. In prenex normal form (PNF) it can be expressed as $\Psi \triangleq Q_1 X_1 Q_2 X_2 \dots Q_n X_n. \psi(X_1, X_2, \dots, X_n)$ where $Q_i \in \{\forall, \exists\}$, $X_i \in node$ and ψ is a quantifier-free Boolean formula over \mathcal{P} 's relations. Following standard practice, we define $prefix(\Psi)$ as the string of quantifiers and bound variables, and refer to ψ as $matrix(\Psi)$. In the context of minimization, we further define the quantified cost of Ψ as

$$qCost(\Psi) = \#Q(prefix(\Psi)) + \#lits(matrix(\Psi)) \quad (2)$$

where $\#Q$ is the number of Ψ 's quantified variables.

We use \mathcal{P}^k to denote a finite instance of \mathcal{P} defined over $node^k \triangleq \{n_0, n_1, \dots, n_{k-1}\}$ for $k \geq 1$. Instantiating \mathcal{P}^k 's relations with all possible combinations of its constants yields \mathcal{P}^k 's state variables, denoted $vars^k$, whose cardinality is

$$|vars^k| = \sum_{h \in relations} k^{arity(h)} \quad (3)$$

\mathcal{P}^k is *structurally symmetric*; its behavior remains invariant under the action of $Sym(node^k)$, the group of permutations

²Sum and product are commonly used in the hardware logic design literature. They are synonymous with disjunction and conjunction.

³The arity of these relations is typically between 1 and 4.

TABLE I: Sample explicit clause orbits and their implicit encoding by finitely-quantified FOL formulas

Partial Protocol Spec \mathcal{P}	Domain: $\text{node} \triangleq \{n_0, n_1, n_2, \dots\}$ Relations: $a : \text{node} \mapsto \{0, 1\}, b : \text{node} \mapsto \{0, 1\}$		
Finite Protocol Instance \mathcal{P}^3	$\text{node}^3 \triangleq \{n_0, n_1, n_2\}$ $\text{vars}^3 = \{a(n_0), a(n_1), a(n_2), b(n_0), b(n_1), b(n_2)\}$ $\text{Sym}(\text{node}^3) : \{(), (n_0 n_1), (n_0 n_2), (n_1 n_2), (n_0 n_1 n_2), (n_0 n_2 n_1)\}$		
Explicit Clause Orbit	$(a(n_0) \vee b(n_1)) \wedge (a(n_0) \vee b(n_2)) \wedge$ $(a(n_1) \vee b(n_0)) \wedge (a(n_1) \vee b(n_2)) \wedge$ $(a(n_2) \vee b(n_0)) \wedge (a(n_2) \vee b(n_1))$	$(b(n_0) \vee b(n_1) \vee b(n_2))$	$(a(n_0) \vee b(n_0) \vee b(n_1) \vee b(n_2)) \wedge$ $(a(n_1) \vee b(n_0) \vee b(n_1) \vee b(n_2)) \wedge$ $(a(n_2) \vee b(n_0) \vee b(n_1) \vee b(n_2))$
Implicitly-Quantified Orbit	$\forall^3 N, M : (N = M) \vee a(N) \vee b(M)$ $qCost = 2 + 3 = 5$	$\exists^3 N : b(N)$ $qCost = 1 + 1 = 2$	$\forall^3 N, \exists^3 M : a(N) \vee b(M)$ $qCost = 2 + 2 = 4$
	(a) \forall^3 Quantification	(b) \exists^3 Quantification	(c) Mixed $\forall^3 \exists^3$ Quantification

on a k -element set. In particular, $\text{Sym}(\text{node}^k)$ partitions \mathcal{P}^k 's variables, as well as any Boolean expressions on them (conjunctions, disjunctions, etc.) into equivalence classes or *orbits*. Given any finite set S^k of syntactically “similar” formulas on the variables of \mathcal{P}^k and any $f \in S^k$ we define

$$\text{orbit}^k(f) \triangleq \{g \in S^k \mid \exists \pi \in \text{Sym}(\text{node}^k) : \pi(f) = g\} \quad (4)$$

where $\pi(f)$ is the result of applying the permutation π to f . The set of orbits in S^k will be denoted as $\text{orbs}(S^k)$.

We are particularly interested in clausal orbits and their compact encoding as finitely-quantified FOL predicates. Table I illustrates this concept with three example clausal orbits. We assume that our generic protocol \mathcal{P} has two unary relations labeled a and b . Its finite instantiation with 3 nodes creates 6 variables and has $3! = 6$ structural symmetries (identity, 3 swaps, and 2 rotations) expressed as node permutations in standard cycle notation. The example orbits in the table represent a 6-clause orbit in column (a), a 1-clause orbit in column (b), and a 3-clause orbit in column (c). Note that the set of clauses in each of these orbits remains unchanged under the action of the 6 permutations of node^3 . The effect of these permutations is to simply reorder the literals and clauses in each orbit while preserving logical equivalence. Logical invariance, in fact, is a direct consequence of two properties of conjunction and disjunction: idempotency ($x \wedge x = x, x \vee x = x$) and commutativity ($x \wedge y = y \wedge x, x \vee y = y \vee x$).

The last row in Table I shows the finitely-quantified FOL formulas that encode these clause orbits. To emphasize that the quantification is over the finite node^3 set, we use the convention of annotating the universal and existential quantifiers with a “3” superscript. Each of these formulas are derived by the mechanical quantifier inference procedure from [6]. This procedure is based on a syntactic analysis of any clause in the orbit (basically the number and distribution of sort constants in the clause’s relations) and guarantees that instantiating the universal and existential quantifiers in these formulas over node^3 yields the exact set of clauses in the corresponding explicit orbits, except possibly for potential duplicates and tautologies. The correspondence between an explicit orbit orbit_i^k and its finitely-quantified encoding Ψ_i^k can be expressed by a pair of

related functions as

$$\begin{aligned} \Psi_i^k &= qInf(\text{orbit}_i^k) \\ \text{orbit}_i^k &= qIns(\Psi_i^k) \end{aligned} \quad (5)$$

where $qInf$ performs finite quantifier *inference* whereas $qIns$ performs finite quantifier *instantiation*.

D. An Example r_{min} Formula

Before describing the steps for deriving r_{min} , let’s illustrate it for a specific example. Consider the TLA+ specification [21] of the *Transaction Commit* (TC) protocol [22]. This protocol is based on a single sort for representing resource managers and four unary relations *working*, *prepared*, *committed*, and *aborted*. Denoting these relations by their initials, the minimum formula produced by QSM for the protocol’s reachable states converged syntactically at a finite instance with 2 resource managers yielding the following eight-orbit expression:

$$\begin{aligned} r_{min}(\text{TC}) &= \bigwedge_{1 \leq i \leq 8} \Psi_i \\ \Psi_1 &= \forall R. (a(R) \rightarrow \neg w(R)) \\ \Psi_2 &= \forall R. (a(R) \rightarrow \neg p(R)) \\ \Psi_3 &= \forall R. (a(R) \rightarrow \neg c(R)) \\ \Psi_4 &= \forall R. (p(R) \rightarrow \neg w(R)) \\ \Psi_5 &= \forall R. (c(R) \rightarrow \neg p(R)) \\ \Psi_6 &= \forall R. (c(R) \rightarrow \neg w(R)) \\ \Psi_7 &= \forall R. (w(R) \vee p(R) \vee c(R) \vee a(R)) \\ \Psi_8 &= \forall R_1, R_2. \\ &\quad c(R_1) \wedge \neg c(R_2) \wedge \neg p(R_2) \rightarrow (R_1 = R_2) \end{aligned} \quad (6)$$

An unbounded quantified SMT query showed that this formula is indeed an inductive invariant for TC. Checking if TC satisfies *any* desired safety property S can now be achieved by showing that $r_{min}(\text{TC}) \rightarrow S$ is valid. More interestingly, the shortest strengthening assertion that explains why S holds can be seen as a minimal subset of the eight orbits in (6). Denoting this subset by $A_{min}(\text{TC}, S)$ it can be found using a minimal unsatisfiable subset (MUS) extractor, such as MARCO [23], from the UNSAT CNF formula

$$A_{min}(\text{TC}, S) = MUS[(r_{min}(\text{TC}) \wedge S) \wedge T \wedge \neg(r'_{min}(\text{TC}) \wedge S')]$$

where T is the transition relation, the primes indicate a variable's next state, and the clauses of $r_{min}(TC)$ are **highlighted** to emphasize that they are treated as soft clauses by the MUS extractor. For example, given the following two safety properties for TC,

$$\begin{aligned} S_1 &= \forall R_1, R_2. (\neg a(R_1) \vee \neg c(R_2)) \\ S_2 &= \forall R_1, R_2. (\neg w(R_1) \vee \neg c(R_2)) \end{aligned}$$

we can show that their shortest respective proof certificates/strengthening assertions are:

$$\begin{aligned} A_{min}(TC, S_1) &= \Psi_2 \\ A_{min}(TC, S_2) &= \Psi_4 \end{aligned}$$

III. QSM: SAT-BASED QUANTIFIED SYMMETRIC MINIMIZATION

The *QSM* minimization algorithm seeks to derive a minimum-cost finitely-quantified formula for r^k , the set of reachable states of \mathcal{P}^k . To achieve this, it takes advantage of two features of these formulas. The first, obvious, feature is the structural symmetry of \mathcal{P}^k . *QSM* preserves this symmetry by operating on prime implicant *orbits* rather than on individual prime implicants. The second, less obvious, feature is that the number of \mathcal{P}^k 's reachable states is almost always much smaller than the number of its unreachable states. This suggests seeking a minimum-cost CNF, rather than DNF, solution. Before delving into the detailed description of *QSM*, it is helpful to understand its operation at a very high level as

$$\mathcal{P}^k \xrightarrow{QSM} r_{min}^k = \bigwedge_{1 \leq i \leq l} \Psi_i^k \quad (7)$$

In other words, *QSM* produces a minimum-cost conjunction of l finitely-quantified FOL formulas where each Ψ_i^k captures an orbit of r^k 's prime implicants.

In contrast to (1), the minimization problem for r^k can now be stated as finding a Boolean assignment to a set of selector variables z_{ω^k} , for $\omega^k \in \text{orbs}(\text{primes}(\neg r^k))$, that represents a solution of the set covering problem

$$\begin{aligned} \min \quad & \sum_{\omega^k \in \text{orbs}(\text{primes}(\neg r^k))} qCost(qInf(\neg \omega^k)) \times z_{\omega^k} \\ \text{s.t.} \quad & \left(\bigvee_{\omega^k \in \text{orbs}(\text{primes}(\neg r^k))} z_{\omega^k} \wedge \omega^k \right) = \neg r^k \end{aligned} \quad (8)$$

Viewed as a formula, each such orbit ω^k is a disjunction of symmetric prime implicants; thus, its negation $\neg \omega^k$ is a conjunction of symmetric prime implicants, i.e., a clausal orbit. This explains the particular choice of the cost metric in (8).

The derivation of r_{min}^k in *QSM* is a deterministic mechanical procedure consisting of the following four steps:

- 1) A BDD-based forward image computation [24] to produce a DNF representation of r^k .
- 2) A SAT-based procedure to generate the set of prime implicant orbits of $\neg r^k$.

Algorithm 1 Symmetry-Aware Enumeration of PI Orbits

```

1 procedure EnumeratePIOrbits( $\neg r^k$ )
2    $\neg r_D^k \leftarrow \text{dualRail}(\neg r^k)$ 
3    $i \leftarrow 1, m \leftarrow |\text{vars}^k|, \text{primeOrbits} \leftarrow \emptyset$ 
4   while  $i \leq m$  do
5      $(\text{found}, \rho_D) \leftarrow \text{SAT}^?[\neg r_D^k \wedge \sum_{1 \leq j \leq m} (x_j^p + x_j^n) \leq i]$ 
6     if found then
7        $\omega \leftarrow \text{orbit}^k(\text{singleRail}(\rho_D))$ 
8        $\text{primeOrbits} \leftarrow \text{primeOrbits} \cup \{\omega\}$ 
9        $\neg r_D^k \leftarrow \neg r_D^k \bigwedge_{\rho \in \omega} \text{dualRail}(\neg \rho)$ 
10    else
11       $i \leftarrow i + 1$ 
12  return primeOrbits

```

- 3) A quantifier-inference procedure *qInf* from [6] that outputs a finitely-quantified FOL formula for each prime implicate orbit of r^k along with its *qCost*.
- 4) A branch-and-bound set covering procedure that finds the minimal number of prime implicate orbits that cover r^k using their quantified cost as the minimization objective.

A. Symmetry-Aware Enumeration of Prime Implicant Orbits

The PI enumeration algorithm operates on the CNF formula representing $\neg r^k$ and is based on a *dualRail* encoding of the state variables [25], [26]. Specifically, each state variable x is encoded using two fresh variables x^p and x^n according to

x^p	x^n	x
0	0	d
0	1	0
1	0	1
1	1	invalid

where d stands for *don't-care*. The *dualRail* version of $\neg r^k$ is obtained by replacing all positive (resp. negative) appearances of x with x^p (resp. x^n) and by adding the clause $(\neg x^p \vee \neg x^n)$ to exclude the invalid combination. This encoding is reversible: given any conjunction (model) or disjunction (clause) of $\neg r^k$ we use *dualRail*($\neg r^k$) to denote the above encoding, and *singleRail*(*dualRail*($\neg r^k$)) to recover the $\neg r^k$ formula based on the original state variables.

This encoding makes it possible to interpret the complete assignments produced by a SAT solver for the x^p and x^n variables as partial assignments (i.e., assignments with don't-cares) for the original x variables. Assuming that $|\text{vars}^k| = m$, a prime implicant consisting of l literals corresponds to (i.e., covers) 2^{m-l} states and can be found by checking the satisfiability of the conjunction of *dualRail*($\neg r^k$) with the following pseudo-Boolean (cardinality) constraint [27]:

$$\sum_{1 \leq j \leq m} (x_j^p + x_j^n) \leq l \quad (9)$$

The orbit enumeration procedure is depicted in Algorithm 1. The procedure accepts a CNF representation of $\neg r^k$ and returns the complete set of prime orbits. The primes are found, in increasing literal size, by executing the SAT query

on line 5 for $i = 1, \dots, m$ using a single incremental SAT solver instance based on an incremental encoding [28] of the cardinality constraint (9). If satisfiable, the solution to the query is an i -literal prime ρ_D in *dualRail* encoding. The orbit of the *singleRail* encoding of this prime, computed by applying the appropriate structural symmetry permutations to its sort constants (line 7), is then added to *primeOrbits* (line 8) and eliminated from further consideration (line 9) for all subsequent SAT queries. When the query is unsatisfiable (i.e., when there are no i -literal primes or all i -literal primes have been found), i is incremented to find primes with $i+1$ literals.

B. Symmetry-Aware Set Covering

Our *QSM* algorithm is an adaptation of the standard textbook branch-and-bound (BnB) logic minimization procedure that uses an explicit matrix encoding of the covering constraints. Specifically, it is based on the BCP procedure for unate and binate covering in [29]. This procedure has three parts: a) a reduction step that uses column and row dominance rules to identify essential and covered (dominated) primes, b) a termination check to accept or reject a complete solution by comparing its cost to the best seen so far, and c) a depth-first BnB search when the “reduced” covering constraints become cyclic. *The QSM algorithm closely follows this computational flow but replaces the column and row dominance rules with queries to an incremental SAT solver using an implicit CNF encoding of the covering constraints.*

To simplify the description of *QSM*, let’s assume that the prime orbits are numbered from 1 to n , i.e., $orbs(primes(\neg r^k)) = \{\omega_1^k, \dots, \omega_n^k\}$, and let $[n] \triangleq \{1, 2, \dots, n\}$. The covering constraints can now be captured by the CNF formula

$$\varphi^k \triangleq \bigwedge_{i \in [n]} (\neg z_i \vee \neg \omega_i^k) \quad (10)$$

which can be queried by an incremental SAT solver under different assumptions involving the literals of the formula. Specifically, the SAT query

$$SAT?[\varphi^k, \text{assume } chosenLiterals(\varphi^k)] \quad (11)$$

checks the satisfiability of φ^k assuming that all literals in *chosenLiterals*(φ^k) are set to True. These literals can include the protocol state variables as well as the selection variables. In particular, it is convenient to define the orbit selection formula

$$Z(sel) \triangleq \left(\bigwedge_{i \in sel} z_i \right) \wedge \left(\bigwedge_{i \notin sel} \neg z_i \right) \quad (12)$$

which can serve as an assumption in (11) to activate the prime orbits specified by the set $sel \subseteq [n]$ and to deactivate the remaining orbits.

During the search we use $sol \subseteq [n]$ to represent the set of prime orbits in the current partial solution and $pnd \subseteq [n]$ for the prime orbits that are *pending*, i.e., the orbits that may or may not be needed to complete the solution. sol becomes a complete solution when $pnd = \emptyset$.

Identifying Essential Orbits: A pending prime orbit ω_i^k is essential if it covers some states that are not covered by the union of a) the remaining pending orbits and b) the orbits in the current partial solution; otherwise it is not essential. This can be checked by the SAT query

$$\begin{aligned} isEssential(\omega_i^k) &\triangleq SAT?[\neg(\omega_i^k \rightarrow \bigvee_{j \in sol \cup pnd \setminus \{i\}} \omega_j^k)] \\ &= SAT?[\omega_i^k \wedge \bigwedge_{j \in sol \cup pnd \setminus \{i\}} \neg \omega_j^k] \end{aligned}$$

Since ω_i^k is a disjunction of primes, the formula in this query is not in CNF. By symmetry, however, it is sufficient to check the essentiality of any prime $\rho \in \omega_i^k$ to conclude if the whole orbit is or is not essential. This allows the above query to be re-expressed as

$$isEssential(\omega_i^k) = SAT?[\rho(\omega_i^k) \wedge \bigwedge_{j \in sol \cup pnd \setminus \{i\}} \neg \omega_j^k]$$

where, with a slight notational abuse, $\rho(\omega_i^k)$ is used to assert an arbitrary prime (a conjunction of protocol literals) from the ω_i^k orbit. The SAT query to check whether or not the ω_i^k orbit is essential can now be expressed as

$$\boxed{isEssential(\omega_i^k) = SAT?[\varphi^k, \text{assume } \rho(\omega_i^k) \wedge Z(sol \cup pnd \setminus \{i\})]} \quad (13)$$

Identifying Covered and Partially-Covered Orbits: The coverage of a pending orbit is the number of states it covers that are not already covered by the current partial solution and can be found as the solution of this #SAT [30] query:

$$coverage(\omega_i^k) \triangleq \#SAT?[\neg(\omega_i^k \rightarrow \bigvee_{j \in sol} \omega_j^k)]$$

Exact coverage can, thus, be expressed as

$$\boxed{coverage(\omega_i^k) = \#SAT?[\varphi^k, \text{assume } \rho(\omega_i^k) \wedge Z(sol)]} \quad (14)$$

Coverage is used to remove completely covered orbits from *pnd* (when *coverage* = 0) and to rank partially-covered orbits for the branching step (when *coverage* > 0). Our implementation uses an approximation of #SAT since the exact number of solutions to (14) is not needed. The coverage estimate of pending orbits is stored in an array *cov*.

The pseudo-code of *QSM* is shown in Algorithm 2. Initially, $pnd = [n]$, $sol = \emptyset$, the entries in the *cov* array are uninitialized, and *UB* (the upper bound on the cost of the solution) is set to $1 + \sum_{i \in [n]} qCost(\omega_i^k)$.

At each invocation, *QSM* performs the following steps:

- Line 2: It updates the current covering requirements (encoded by *pnd*, *sol*, and *cov*) by calling *reduce* to identify essential and covered primes, if any.
- Lines 3-8: It checks if a complete solution has been found and
 - Lines 4-6: returns this solution and updates *UB* to its cost if it is cheaper than the best seen so far.

Algorithm 2 Quantified Symmetric Minimization

```

1 procedure QSM(pnd, sol, cov, UB)
2   (pnd, sol, cov)  $\leftarrow$  reduce(pnd, sol, cov)
3   if pnd =  $\emptyset$  then
4     if qCost(sol) < UB then
5       UB  $\leftarrow$  qCost(sol)
6       return sol
7     else
8       return NOSOLUTION
9   LB  $\leftarrow$  qCost(sol)
10  if LB  $\geq$  UB then
11    return NOSOLUTION
12  i  $\leftarrow$  chooseOrbit(pnd, cov)
13  Swith-i  $\leftarrow$  QSM(pnd \ {i}, sol  $\cup$  {i}, cov, UB)
14  if qCost(Swith-i) = LB then
15    return (Swith-i)
16  Swithout-i  $\leftarrow$  QSM(pnd \ {i}, sol, cov, UB)
17  return BESTSOLUTION(Swith-i, Swithout-i)

18 procedure reduce(pnd, sol, cov)
19   (existEss, pnd, sol)  $\leftarrow$  addEssentials(pnd, sol)
20   (existCov, pnd, cov)  $\leftarrow$  removeCovered(pnd, sol, cov)
21   if existEss  $\vee$  existCov then
22     (pnd, sol, cov)  $\leftarrow$  reduce(pnd, sol, cov)
23   return (pnd, sol, cov)

24 procedure addEssentials(pnd, sol)
25   essentials  $\leftarrow$   $\emptyset$ 
26   for each orbit  $\in$  pnd do
27     if isEssential(orbit, pnd, sol) then
28       essentials  $\leftarrow$  essentials  $\cup$  {orbit}
29   sol  $\leftarrow$  sol  $\cup$  essentials
30   pnd  $\leftarrow$  pnd \ essentials
31   return (essentials > 0, pnd, sol)

32 procedure removeCovered(pnd, sol, cov)
33   covered  $\leftarrow$   $\emptyset$ 
34   for each orbit  $\in$  pnd do
35     cov[orbit]  $\leftarrow$  coverage(orbit, sol)
36     if cov[orbit] = 0 then
37       covered  $\leftarrow$  covered  $\cup$  {orbit}
38   pnd  $\leftarrow$  pnd \ covered
39   return (covered > 0, pnd, cov)

```

- Lines 7-8: returns “no solution” (i.e., backtracks) if the cost is higher than the best seen so far.
- Lines 9-11: It sets the lower bound *LB* to be the cost of the current *partial* solution and backtracks if that cost is greater than the current upper bound.
- Line 12: It ranks the pending orbits by their estimated coverage and chooses the orbit with the highest coverage for the next branching decision breaking ties arbitrarily. In addition, it ranks orbits that are not parameterized by sort constants (i.e., they are independent of “*k*”) higher than other orbits. The intuition behind this heuristic is that such orbits are more likely to be in the minimum solution since they are *size- independent*.
- Lines 13-17: It recursively calls itself to search for a solution that *includes* the chosen orbit and returns that

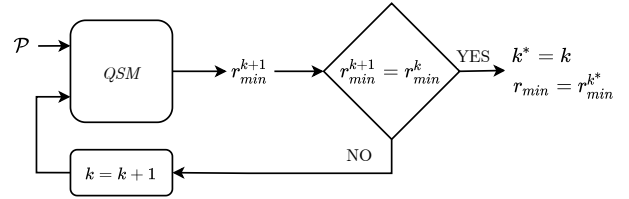


Fig. 1: *QSM* Syntactic Fixed Point Convergence

solution if its cost is equal to the lower bound. Otherwise, it recursively calls itself to search for a solution that *excludes* the chosen orbit and returns the cheaper of the two solutions.

The computational core of *QSM* is in the *reduce*, *addEssentials*, and *removeCovered* procedures. The *reduce* procedure repeatedly calls *addEssentials* and *removeCovered* until all essential and covered orbits have been processed and *pnd*, *sol*, and *cov* updated. Finally, the *addEssentials* and *removeCovered* procedures implement the SAT queries corresponding to (13) and (14).

IV. FROM BOUNDED TO UNBOUNDED MINIMIZATION

Applying the *QSM* algorithm to $\mathcal{P}^1, \mathcal{P}^2, \dots$ generates a corresponding sequence of minimum solutions⁴ $r_{min}^1, r_{min}^2, \dots$. An interesting empirical observation is that this sequence reaches a *syntactic* fixed point (Figure 1) at some value k^* defined as:

$$\begin{aligned}
 \bullet \quad r_{min}^{k^*} &= \bigwedge_{1 \leq i \leq l} \Psi_i^{k^*} \\
 \bullet \quad r_{min}^{k^*+1} &= \bigwedge_{1 \leq i \leq l} \Psi_i^{k^*+1} \\
 \bullet \quad \forall i \in [1, l] : \text{prefix}(\Psi_i^{k^*+1}) &= \text{prefix}(\Psi_i^{k^*}) \\
 \bullet \quad \forall i \in [1, l] : \text{matrix}(\Psi_i^{k^*+1}) &= \text{matrix}(\Psi_i^{k^*})
 \end{aligned} \tag{15}$$

Another way of saying this is that the minimum clausal orbits “converge” and that additional orbits that might be produced at values of k larger than k^* become redundant and do not introduce new behaviors beyond k^* . This is reminiscent of the *cutoff* [31], [32] and *data saturation* [20] phenomena in the model checking literature and suggests that the finite quantification can be replaced with unbounded quantification yielding an exact minimum formula

$$r_{min} = \bigwedge_{1 \leq i \leq l} \Psi_i \tag{16}$$

for the unbounded protocol \mathcal{P} . Our contribution can be seen as the culmination of these earlier efforts by showing that the incorporation of minimization a) yields the natural quantified forms of the r_{min} orbits and b) “explains” how saturation happens.

V. EXPERIMENTAL EVALUATION

We evaluated *QSM* on a set of 17 protocols from [4], [5], [33]. This set includes fairly complex high-level descriptions of

⁴In practice, the initial base size usually starts at $i > 1$.

TABLE II: *QSM* Experimental Results[†]

Protocol	Memory MB	CPU Time, sec					Number of				r_{min} Orbits	Asseertions	
		Total	BDD	PI Gen	qInf	Cov	vars	r^k cubes	$\neg r^k$ PIs	Orbits		Human	IC3PO
tla-consensus	59	9	4	0	4	0*	3	3	3	1	1	0	0
tla-tcommit	67	9	4	0	4	0	8	12	26	13	8	2	1
tla-twophase	67	7197	4	1	4	7188	17	29	252	134	?	11	11
distai-ricart-agrawala	67	9	4	0	4	0*	10	4	10	6	4	2	2
i4-lock-server	67	17	8	1	8	1*	8	16	14	3	3	1	1
pyv-sharded-kv	104	58	22	10	17	10*	42	324	420	15	8	4	5
pyv-sharded-kv-no-lost-keys	106	68	22	15	18	14	42	324	422	16	9	1	1
ex-simple-decentralized-lock	67	18	8	1	8	1*	24	80	198	18	18	4	3
pyv-firewall	67	56	6	1	7	42	15	23	348	62	5	1	2
pyv-lockserv	67	9	4	0	4	0*	13	10	46	13	13	8	8
ex-lockserv-automaton	67	9	4	0	4	0*	13	10	46	13	13	1	8
ex-toy-consensus	108	29	14	2	12	2	27	138	510	19	4	2	2
ex-naive-consensus	67	21	9	2	8	2*	27	189	396	14	3	3	3
ex-simple-election	47056	3812	3773	2	11	27	27	220	849	55	7	2	3
pyv-toy-consensus-forall	67	31	13	3	13	3	23	1072	518	19	6	3	3
pyv-toy-consensus-epr	48912	3622	3607	1	11	3	24	94	534	35	6	3	3
pyv-consensus-epr	913	7401	4675	74	32	2621	72	1318	19818	743	16	6	5

[†] The memory and time statistics capture the runs of *QSM* from the initial finite size to the one larger than the cutoff size. The number of variables, cubes, PIs, and Orbits are at the cutoff size.

*For these protocols, r_{min} was found without any branch-and-bound search.

mutual-exclusion and consensus algorithms, including protocols such as sharded key-value store, two-phase commit, asynchronous lock server, Ricart-Agrawala, etc. Several studies [4], [5], [18], [34]–[36] have indicated the challenges involved in verifying these protocols.

We assessed the performance of each step in *QSM* and contrasted its derivation of r_{min} (which is inferred independently of any protocol property) to human-written and automatically-derived *property-driven strengthening assertions*. For each protocol in Table II we made a sequence of *QSM* runs from an initial *base* size to the converged k^* *cutoff* size (details on these sizes are shown in Table III) and report the cumulative time and maximum memory usage for all these runs. The total time for these runs is broken down into the following stages:

- **BDD**: the time to generate the DNF table (as a set of *cubes*) of r^k using BDD-based forward image computation.
- **PI Gen**: the time for the SAT-based procedure to enumerate the prime implicants of $\neg r^k$ and to partition them into symmetry orbits.
- **qInf**: the time to perform quantifier inference on all prime implicate orbits.
- **Cov**: the time for the SAT-based branch-and-bound set covering minimization problem that yields r_{min}^k .

The table also shows, at the cutoff size, the number of variables, cubes, prime implicants/implicates, and orbits. Column “ r_{min} Orbits” gives the number of (invariant) orbits in the final unbounded formula r_{min} ; these formulas were independently confirmed to be inductive using Ivy [18]. Note that r_{min} can be instantiated for any arbitrary protocol size k and can be independently confirmed to be logically-equivalent to the set of reachable states of \mathcal{P}^k . The final two columns give the number of manually-written and automatically-derived strengthening assertions using IC3PO [6], [37] for the protocol’s specified safety property.

We can make the following observations about these results.

- Except for 4 cases, the total time for deriving r_{min} is less than a couple of CPU minutes.
- For 8 protocols, r_{min} was found without any branch-and-bound search (indicated with * in the **Cov** column).
- Except for the tla-twophase protocol, the derivation of r_{min} was completed and the solution was unique. The minimization step timed out for tla-twophase. Interestingly though, the complete set of orbits at sizes 2 and 3 were identical and found to be inductive. Thus, even in this case the complete product was unique.
- In 3 cases, the BDD image computation had a large memory footprint and dominated the total run time.

A preliminary analysis of these results identified the causes for the observed computational bottlenecks. Specifically, the current BDD front-end does not account for symmetry causing a huge memory blow-up and the attendant increase in run time. A natural solution would be to preserve the protocol’s structural symmetry in the forward image computation in order to produce a set of *cube orbits*, rather than individual cubes⁵, for r^k . The excessive run time of the covering step in tla-twophase and pyv-consensus-epr was a direct consequence of the large number of PIs and PI orbits and the failure of the branching heuristic to identify good candidate orbits that can guide the search to close-to-minimal initial solutions. We noticed that many of the PI orbits in these, as well as other, protocols are “similar” (involving the same literals) and can be merged as disjoint *sub-orbits* into larger *super orbits*. As an example, Table IV shows 5 sub-orbits from the ex-simple-decentralized-lock protocol and their merger into a single super orbit with a much smaller *qCost*. Identifying super orbits

⁵The current BDD front-end limited the set of protocols our prototype can support.

TABLE III: Finite instance sizes from the initial *base* size to the converged *cutoff* size

Protocol	Finite instance sizes $\dagger \ddagger$
tla-consensus	$\text{value} = 2 \mapsto 3$
tla-tcommit	$\text{resource-manager} = 2$
tla-twophase	$\text{resource-manager} = 2$
distai-ricart-agrawala	$\text{node} = 2$
i4-lock-server	$\text{client} = 2 \mapsto 3, \text{server} = 1 \mapsto 2$
pyv-sharded-kv	$\text{key} = 2, \text{node} = 2 \mapsto 3, \text{value} = 2 \mapsto 3$
pyv-sharded-kv-no-lost-keys	$\text{key} = 2, \text{node} = 2 \mapsto 3, \text{value} = 2 \mapsto 3$
ex-simple-decentralized-lock	$\text{node} = 2 \mapsto 4$
pyv-firewall	$\text{node} = 2 \mapsto 3$
pyv-lockserv	$\text{node} = 2 \mapsto 3$
ex-lockserv-automaton	$\text{node} = 2 \mapsto 3$
ex-toy-consensus	$\text{node} = 2 \mapsto 4, \text{quorum} = 1 \mapsto 4, \text{value} = 2 \mapsto 3$
ex-naive-consensus	$\text{node} = 3 \mapsto 4, \text{quorum} = 3 \mapsto 4, \text{value} = 3$
ex-simple-election	$\text{acceptor} = 2 \mapsto 3, \text{proposer} = 2 \mapsto 3, \text{quorum} = 1 \mapsto 3$
pyv-toy-consensus-forall	$\text{node} = 2 \mapsto 4, \text{quorum} = 1 \mapsto 4, \text{value} = 2 \mapsto 3$
pyv-toy-consensus-epr	$\text{node} = 2 \mapsto 3, \text{quorum} = 1 \mapsto 3, \text{value} = 2 \mapsto 3$
pyv-consensus-epr	$\text{node} = 2 \mapsto 4, \text{quorum} = 1 \mapsto 4, \text{value} = 2 \mapsto 3$

$\dagger \mathbf{s} = x$ denotes sort \mathbf{s} has both initial size and final cutoff size x

$\ddagger \mathbf{s} = x \mapsto y$ denotes sort \mathbf{s} has initial size x and final cutoff size y

during the PI generation step will yield a much smaller number of orbits for the subsequent cover minimization step.

These initial results provide strong support to our thesis that the structural symmetries of a protocol enable the derivation of a minimal conjunctive FOL formula for its reachable states.

VI. RELATED WORK

Notwithstanding the undecidability result of Apt and Kozen [38], many efforts to *automatically* infer quantified inductive invariants for distributed protocols have been reported with the pace increasing in recent years [3]–[9]. All these works, however, perform a property-dependent analysis of the distributed protocol and aim to derive an inductive invariant specific to a given safety property. In contrast, our work attempts to derive an FOL encoding of the exact set of reachable states of a distributed protocol, which can be utilized to check the validity of any safety property.

Several manual or semi-automatic verification techniques based on interactive theorem proving have been proposed for deriving system-level proofs [18], [34], [39]–[43]. However, unlike fully-automatic verification, all these methods require a detailed understanding of the intricate inner workings of the protocol and entail significant manual effort to guide proof development.

Verification of parameterized systems using SMT solvers is further explored in MCMT [44], Cubicle [45], and paraVerifier [46]. Our work is closest in spirit to *view abstraction*, proposed in [47], which computes the reachable set for finite

instances using forward reachability until cutoff is reached. Our technique further builds on these works with the ability to automatically derive a quantified FOL encoding of the set of reachable states by utilizing a novel symmetry-aware SAT-based logic minimization algorithm.

In the context of logic minimization, the implicit encoding of the covering constraints in *QSM* is similar, at least in spirit but not details, to the procedure in [48]. Finally, it is worth noting, as an interesting historical fact, that McCluskey [16] considered the incorporation of *Boolean* symmetry in his tabular method for deriving the set of prime implicants.

VII. CONCLUSIONS AND FUTURE WORK

We proposed *QSM*, a novel forward-reachability algorithm that combines the relationship between symmetry and quantification in a SAT-based logic minimization procedure to derive a compact quantified FOL formula r_{min} representing the set of reachable states of a distributed protocol. We empirically demonstrate the ability of our prototype to derive such quantified representations of the reachable states, independent of the protocol size, on a restricted class of distributed protocols. The derivation of r_{min} is property-independent, enables checking the validity of *any* protocol safety property and compactly summarizes all protocol behaviors for any size.

In its current form, our *QSM* prototype is limited to protocol specifications based on unbounded symmetric sorts. Structural symmetry is a manifestation of what can be called *spatial regularity* which leads to boundedness in the *spatial dimension*.

TABLE IV: Illustrating the merger of sub-orbits into a single super orbit for the ex-simple-decentralized-lock protocol

Sub-orbits	invariant [pi19] forall N1, N2, N3.	($\sim \text{has_lock}(N1) \mid \sim \text{message}(N3, N2) \mid (N1 = N2) \mid (N1 = N3) \mid (N2 = N3)$)
	invariant [pi25] forall N1, N2.	($\sim \text{has_lock}(N1) \mid \sim \text{message}(N2, N2) \mid (N1 = N2)$)
	invariant [pi31] forall N1, N2.	($\sim \text{has_lock}(N1) \mid \sim \text{message}(N1, N2) \mid (N1 = N2)$)
	invariant [pi37] forall N1, N2.	($\sim \text{has_lock}(N1) \mid \sim \text{message}(N2, N1) \mid (N1 = N2)$)
	invariant [pi43] forall N1.	($\sim \text{has_lock}(N1) \mid \sim \text{message}(N1, N1)$)
Equivalent Super Orbit	invariant [pi19_pi43] forall N, S, D.	($\sim \text{has_lock}(N) \mid \sim \text{message}(S, D)$)

An important extension would be to derive r_{min} for protocols that also include *totally-ordered* sorts. We do not foresee conceptual difficulties for such an extension since totally-ordered sorts introduce another type of regularity, namely *temporal regularity* which leads to boundedness in the *temporal dimension*, as explored in [49] and applied to automatically prove the safety of Paxos [50] and Bakery [51] protocols. Intuitively, while a totally-ordered sort causes the state space of the protocol to expand without bound, that expansion must be characterized by a repeating pattern since, otherwise, it would not be captured by a finite set of quantifiers. Thus, as observed in [49], we expect a cutoff/saturation phenomenon conceptually similar to that exhibited by symmetry but different in implementation details. We also plan to augment *QSM* with the MARCO MUS extractor [23] to automatically derive subsets of the minimum orbits of r_{min} that can serve as minimum strengthening assertions for given safety properties.

The experimental results strongly hint that the r_{min} formula produced by *QSM* is unique. We conjecture that this must follow from symmetry and the particular cost function used in the set covering step. However, we do not have a formal proof that this is always the case and we plan to develop such a proof since solution uniqueness is critical for syntactic convergence. More speculatively, the possibility that a unique quantified formula for r_{min} can be mechanically derived, even when it contains predicates that violate known decidable FOL classes, suggests perhaps the existence of a new decidable fragment of FOL.

Finally, the limited experiments we reported highlighted the need for several optimizations to our current prototype implementation of *QSM* including the incorporation of symmetry in BDD-based forward image computation, the identification of super orbits in the PI enumeration step, and improving the accuracy of coverage estimates during the branch-and-bound search for the minimum cover. Specifically, one simple modification of the #SAT query in (14) is to multiply its answer by the number of primes in the orbit to get a more accurate orbit coverage.

ACKNOWLEDGMENTS

This work was done in part while the authors were participating in a program at the Simons Institute for the Theory of Computing. The research was funded in part by the Austrian Science Fund (FWF) under project No. T-1306.


REFERENCES


- [1] C. Barrett and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.
- [2] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” www.SMT-LIB.org, 2016.
- [3] A. Karbyshev, N. Björner, S. Itzhaky, N. Rinetzy, and S. Shoham, “Property-directed inference of universal invariants or proving their absence,” *J. ACM*, vol. 64, no. 1, Mar. 2017. [Online]. Available: <https://doi.org/10.1145/3022187>
- [4] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, “I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols,” in *The 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, Huntsville, Ontario, Canada, October 2019, pp. 370–384.
- [5] J. R. Koenig, O. Padon, N. Immerman, and A. Aiken, “First-order quantified separators,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 703–717. [Online]. Available: <https://doi.org/10.1145/3385412.3386018>
- [6] A. Goel and K. A. Sakallah, “On Symmetry and Quantification: A New Approach to Verify Distributed Protocols,” in *13th Annual NASA Formal Methods Symposium (NFM 2021)*, Langley, Virginia, May 2021, pp. 131–150. [Online]. Available: <https://arxiv.org/abs/2103.14831>
- [7] T. Hance, M. Heule, R. Martins, and B. Parno, “Finding invariants of distributed systems: It’s a small (enough) world after all,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 115–131. [Online]. Available: <https://www.usenix.org/conference/nsdi21/presentation/hance>
- [8] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, “Distai: Data-driven automated invariant learning for distributed protocols,” in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 405–421.
- [9] J. Yao, R. Tao, R. Gu, and J. Nieh, “{DuoAI}: Fast, automated inference of inductive invariants for verifying distributed protocols,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 485–501.
- [10] N. Eén and N. Sörensson, “An extensible sat-solver,” in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518. [Online]. Available: https://doi.org/10.1007/978-3-540-24605-3_37
- [11] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.
- [12] J. B. Fraleigh, *A First Course in Abstract Algebra*, 6th ed. Reading, Massachusetts: Addison Wesley Longman, 2000.
- [13] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
- [14] W. V. Quine, “The problem of simplifying truth functions,” *The American mathematical monthly*, vol. 59, no. 8, pp. 521–531, 1952.
- [15] —, “A way to simplify truth functions,” *The American mathematical monthly*, vol. 62, no. 9, pp. 627–631, 1955.
- [16] E. J. McCluskey, “Detection of group invariance or total symmetry of a boolean function,” *The Bell System technical journal*, vol. 35, no. 6, pp. 1445–1453, 1956.
- [17] L. Lamport, “The temporal logic of actions,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.
- [18] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: safety verification by interactive generalization,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 614–630.
- [19] P. Wolper, “Expressing interesting properties of programs in propositional temporal logic,” in *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1986, pp. 184–193.
- [20] C. Norris IP and D. L. Dill, “Better verification through symmetry,” *Formal Methods in System Design*, vol. 9, no. 1, pp. 41–75, Aug 1996. [Online]. Available: <https://doi.org/10.1007/BF00625968>
- [21] “A TLA+ specification of the Transaction Commit protocol,” <https://github.com/tlaplus/Examples/blob/master/specifications/transaction.commit/TCCommit.tla>.
- [22] J. Gray and L. Lamport, “Consensus on transaction commit,” *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 133–160, 2006.
- [23] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva, “Fast, flexible mus enumeration,” *Constraints*, vol. 21, no. 2, pp. 223–250, 2016.
- [24] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, “Symbolic Model Checking: 10^{20} States and Beyond,” in *Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990, pp. 428–439.
- [25] V. M. Manquinho, P. F. Flores, J. P. M. Silva, and A. L. Oliveira, “Prime implicant computation using satisfiability algorithms,” in *9th International Conference on Tools with Artificial Intelligence, ICTAI ’97, Newport Beach, CA, USA, November 3-8, 1997*. IEEE Computer Society, 1997, pp. 232–239. [Online]. Available: <https://doi.org/10.1109/TAI.1997.632261>

- [26] S. Jabbour, J. Marques-Silva, L. Sais, and Y. Salhi, "Enumerating prime implicants of propositional formulae in conjunctive normal form," in *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Fermé and J. Leite, Eds., vol. 8761. Springer, 2014, pp. 152–165. [Online]. Available: https://doi.org/10.1007/978-3-319-11558-0_11
- [27] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce, "Incremental cardinality constraints for maxsat," in *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, ser. Lecture Notes in Computer Science, B. O'Sullivan, Ed., vol. 8656. Springer, 2014, pp. 531–548. [Online]. Available: https://doi.org/10.1007/978-3-319-10428-7_39
- [28] R. Martins, S. Joshi, V. Manquinho, and I. Lynce, "Incremental cardinality constraints for maxsat," in *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings* 20. Springer, 2014, pp. 531–548.
- [29] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2007.
- [30] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting," in *Handbook of satisfiability*. IOS press, 2021, pp. 993–1014.
- [31] A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2001, pp. 82–97.
- [32] T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. Zuck, "Parameterized verification with automatically computed inductive assertions," in *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 221–234.
- [33] "A collection of distributed protocol verification problems," <https://github.com/aman-goel/ivybench>.
- [34] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 1–17.
- [35] Y. M. Feldman, J. R. Wilcox, S. Shoham, and M. Sagiv, "Inferring inductive invariants from phase structures," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 405–425.
- [36] I. Berkovits, M. Lazić, G. Losa, O. Padon, and S. Shoham, "Verification of threshold-based distributed algorithms by decomposition to decidable logics," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 245–266.
- [37] A. Goel and K. A. Sakallah, "IC3PO: IC3 for Proving Protocol Properties," <https://github.com/aman-goel/ic3po>.
- [38] K. R. Apt and D. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Inf. Process. Lett.*, vol. 22, no. 6, pp. 307–309, 1986.
- [39] S. Owre, J. M. Rushby, and N. Shankar, "Pvs: A prototype verification system," in *International Conference on Automated Deduction*. Springer, 1992, pp. 748–752.
- [40] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "Verifying safety properties with the tla+ proof system," in *International Joint Conference on Automated Reasoning*. Springer, 2010, pp. 142–148.
- [41] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 357–368. [Online]. Available: <http://doi.acm.org/10.1145/2737924.2737958>
- [42] J. Hoenicke, R. Majumdar, and A. Podelski, "Thread modularity at many levels: a pearl in compositional verification," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 473–485, 2017.
- [43] K. v. Gleissenthall, R. G. Kıcı, A. Bakst, D. Stefan, and R. Jhala, "Pretend synchrony: synchronous verification of asynchronous distributed programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.
- [44] S. Ranise and S. Ghilardi, "Backward reachability of array-based systems by smt solving: Termination and invariant synthesis," *Logical Methods in Computer Science*, vol. 6, 2010.
- [45] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi, "Cubicle: A parallel smt-based model checker for parameterized systems," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 718–724.
- [46] Y. Li, J. Pang, Y. Lv, D. Fan, S. Cao, and K. Duan, "Paraverifier: An automatic framework for proving parameterized cache coherence protocols," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2015, pp. 207–213.
- [47] P. Abdulla, F. Haziza, and L. Holík, "Parameterized verification through view abstraction," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 5, pp. 495–516, 2016.
- [48] M. J. Ghazala, "Irredundant disjunctive and conjunctive forms of a boolean function," *IBM J. Res. Dev.*, vol. 1, no. 2, pp. 171–176, 1957. [Online]. Available: <https://doi.org/10.1147/rd.12.0171>
- [49] A. Goel and K. A. Sakallah, "Regularity and Quantification: A New Approach to Verify Distributed Protocols," *Innovations in Systems and Software Engineering (ISSE)*, pp. 1–19, September 2022.
- [50] —, "Towards an Automatic Proof of Lamport's Paxos," in *Formal Methods in Computer-Aided Design (FMCAD)*, R. Piskac and M. W. Whalen, Eds., New Haven, Connecticut, October 2021, pp. 112–122.
- [51] A. Goel, S. Merz, and K. A. Sakallah, "Towards an automatic proof of the bakery algorithm," in *Formal Techniques for Distributed Objects, Components, and Systems*, M. Huisman and A. Ravara, Eds. Cham: Springer Nature Switzerland, 2023, pp. 21–28.

BIG Backbones

Nils Froleys* 
nils.froleys@jku.at

Emily Yu* 
emily.yu2019@gmail.com

Armin Biere† 
biere@cs.uni-freiburg.de

*Johannes Kepler University, Linz, Austria

†University of Freiburg, Freiburg, Germany

Abstract—The backbone of a satisfiable formula is the set of literals that hold true in every model. In this paper we introduce Single Unit Resolution Backbone (SURB) which names both a polynomial-time algorithm for backbone extraction and a class of propositional formulas on which it is complete. We show that this class is a superset of the polynomial-time solvable SLUR formulas. The presented algorithm meets a lower bound on time complexity under the strong exponential-time hypothesis. As a second contribution, we present a version that operates on the binary implication graph (BIG) and implement it as a preprocessor in the recently introduced backbone extractor CADiBACK. Experiments on a large number of SAT competition benchmarks show that our implementation results in faster BIG backbone extraction by an order of magnitude. Additionally, incorporating it as a preprocessor enables CADiBACK to identify up to four times as many backbone literals early on.

I. INTRODUCTION

Backbone extraction has been put forward as an effective technique for a wide variety of applications including chip verification, specifically fault localization [1], [2], [3], and interactive configuration [4]. The concept of the backbone, which refers to the set of literals that hold true in all models of a satisfiable formula, was initially studied when investigating the hardness of (random) propositional formulas [5], [6], [7]. Since then, a number of practical applications for backbone extraction have been discovered. One notable example is the improved performance in SAT solving itself [8], [9]. In fact, the proposed algorithm in this paper has been implemented as a cheaper version of failed literal elimination in SAT solvers developed by one of the authors [10]. Other related areas like maximum satisfiability [11], [12], [13], [14] have been found to benefit from early knowledge of backbone literals.

In these applications, it is highly advantageous to promptly access as many backbone literals as possible. This can be due to two key reasons: either the backbone computation is bound by a time limit or the identification of a backbone literal triggers additional computations that can be executed in parallel. As a result, our focus shifts to the time taken to identify individual backbone literals rather than the completion of the entire backbone extraction.

The state-of-the-art in backbone extraction has remained unchanged for a long time, until recently CADiBACK [15] was introduced, exhibiting significantly better performance. This was achieved by using the modern SAT solver CADICAL and tightly integrating features currently not found in any other SAT solver. Our contribution presented in this paper is orthogonal to that. Instead of using an exponential approach

based on incremental SAT solving, we use a polynomial time algorithm to extract the backbone from a subset of the formula.

In SAT solving, Single Look-ahead Unit Resolution (SLUR) [16], independently discovered as Backtrack-once in [17], defines a class of formulas that are solvable in polynomial time. Similar to that, we define a simple polynomial algorithm called SURB and use it to define a subclass of propositional formulas on which the backbone can be extracted in polynomial time. We formally show that SURB is a strict superset of SLUR. As another novel contribution, we present a practical algorithm that exhibits considerably better performance in our experimental evaluation than SURB. The algorithm finds all backbone literals in the binary implication graph. We implement it as a preprocessor, extending the recently introduced backbone extractor CADiBACK [15]. Results show that our implementation outperforms the previous state-of-the-art by an order of magnitude. Furthermore, our extension enables CADiBACK to identify a subset of all backbone literals within a fraction of the time required to find an initial model.

II. PRELIMINARIES

We consider satisfiable SAT formulas in conjunctive normal form (CNF). A formula \mathcal{F} is defined over a fixed set of variables \mathcal{V} or their literals $\mathcal{L} = \mathcal{V} \cup \neg\mathcal{V}$, where $\neg\mathcal{V} = \{\neg\ell \mid \ell \in \mathcal{V}\}$ is the set of negative literals over \mathcal{V} . It consists of a set of clauses, which are sets of literals. A clause is *unit* if it contains only one literal. We use $|\mathcal{F}|$ to denote the number of literal occurrences in \mathcal{F} , the number of distinct literals $|\mathcal{L}|$ will commonly be referred to as n .

An *assignment* $\sigma \subset \mathcal{L}$ can also be interpreted as the conjunction of its literals and we use $\mathcal{F}_\sigma = \mathcal{F} \bigwedge_{\ell \in \sigma} \ell$ to denote a formula \mathcal{F} under assignment σ . The *unit-clause rule* [18] picks a unit clause $\{\ell\}$, removes all clauses containing ℓ and removes $\neg\ell$ from all clauses. We write $\mathcal{F} \vdash_1 \ell$ and say ℓ is derived from \mathcal{F} by *unit propagation*, if a unit clause $\{\ell\}$ can be picked during repeated application of this rule. If both $\mathcal{F} \vdash_1 \ell$ and $\mathcal{F} \vdash_1 \neg\ell$ we say a conflict is derived and for convenience write $\mathcal{F} \vdash_1 \bot$ (note that \bot is not a literal). The assignment resulting from unit propagation under σ until fixpoint is $\sigma' = \{k \mid \mathcal{F}_\sigma \vdash_1 k\}$. If σ' contains conflicting literal we use the same notation $\bot \in \sigma'$. If no conflict is encountered, we can set $\sigma = \sigma'$, extend the assignment, and repeat the computation until a full assignment is reached. The entire process can be implemented in $\mathcal{O}(|\mathcal{F}|)$ [19].

The Binary Implication Graph (BIG) of \mathcal{F} has a node for each literal in \mathcal{L} and two edges $(\neg u, v)$ and $(\neg v, u)$ for each binary clause $\{u, v\}$ [20]. By contraposition, if there is a path from u to v , there is also a path from $\neg v$ to $\neg u$ [21]. Equivalent Literal Substitution (ELS) identifies all cycles in the BIG and replaces the corresponding literals with a single representative. Failed Literal Elimination (FLE) [22] identifies literals ℓ with $\mathcal{F}|_{\ell} \vdash_1 \bot$ (called *failed*) and adds $\neg \ell$ as a unit clause. This is done repeatedly until a fixpoint is reached.

Algorithm 1 (SLUR) [16] may return unsatisfiability, a satisfying assignment, or give up. If it succeeds for any variable ordering (line 3), the formula is in the SLUR class [23]. Notable subsets of SLUR include 2-CNF which contain only binary clauses, and Horn-3-CNF that contain length 3 clauses with at most one positive literal.

```

SLUR (CNF  $\mathcal{F}$ )
1   $\sigma \leftarrow \{k \mid \mathcal{F} \vdash_1 k\}$ 
2  if  $\bot \in \sigma$  then return UNSAT
3  for  $v \in \mathcal{V}$ 
4     $\sigma^+ \leftarrow \{k \mid \mathcal{F}|_{\sigma \wedge v} \vdash_1 k\}$ 
5     $\sigma^- \leftarrow \{k \mid \mathcal{F}|_{\sigma \wedge \neg v} \vdash_1 k\}$ 
6    if  $\bot \in \sigma^+$  and  $\bot \in \sigma^-$  then
7      return GIVE-UP
8    if  $\bot \in \sigma^+$  then  $\sigma \leftarrow \sigma^+$ 
9    else  $\sigma \leftarrow \sigma^-$ 
10 return SAT,  $\sigma$ 

```

Algorithm 1: Single Look-ahead Unit Resolution. Success depends on the formula and the variable order chosen in line 3.

III. SINGLE UNIT RESOLUTION BACKBONE

This section, introduces the algorithm SURB (Single Unit Resolution Backbone) for finding backbone literals and defines a subclass of formulas with the same name.

```

SURB (CNF  $\mathcal{F}$ )
1   $\mathcal{B} \leftarrow \emptyset$ 
2  for  $\ell \in \mathcal{L}$ 
3    if  $\mathcal{F}|_{\mathcal{B} \wedge \ell} \vdash_1 \bot$  then
4       $\mathcal{B} \leftarrow \{k \mid \mathcal{F}|_{\mathcal{B} \wedge \neg \ell} \vdash_1 k\}$ 
5  return  $\mathcal{B}$ 

```

Algorithm 2: Single Unit Resolution Backbone identifies a subset of the backbone. The order of literals chosen in line 2 is non-deterministic and can influence which backbone literals are identified.

The algorithm is sound, since the negation of failed literals are backbone literals and only previously identified backbone literals are added to the propagation. In the following we introduce the SURB subclass based on Algorithm 2.

Definition 1. A formula \mathcal{F} is in SURB if the algorithm identifies the entire backbone for any order of literal selection.

The relation of SURB and other classes can be summarized as the following, where FLBE is defined later in Def. 2.

$$2\text{-CNF} \subsetneq \text{SLUR} \subsetneq \text{SURB} \subsetneq \text{FLBE}$$

Similar to SLUR, running Algorithm 2 does not indicate the membership in the class. Deciding if a formula is in SLUR is co-NP-complete [24]. We leave a similar proof for SURB to future work. In practice, this means that without additional knowledge about the formula, it is unknown if the backbone extends beyond the literals identified by SURB. We now formally prove the subset relations from above.

Theorem 1. $\text{SLUR} \subset \text{SURB}$

Proof. Assume a satisfiable formula \mathcal{F} has a backbone literal $\neg \ell$ that is not identified by SURB, we show SLUR can fail on \mathcal{F} . Let ℓ be the first variable that is decided by SLUR. By the assumption $\mathcal{F}|_{\mathcal{B} \wedge \ell} \not\vdash_1 \bot$ for some set \mathcal{B} and therefore especially for $\mathcal{B} = \emptyset$. SLUR chooses σ^+ to proceed and will eventually give up since $\neg \ell$ is a backbone literal. \square

The example shows not all formulas in SURB are in SLUR.

Example 1. Consider the formula $\mathcal{F} = (\neg a \vee b \vee \neg c \vee d) \wedge (\neg a \vee b \vee \neg c \vee \neg d) \wedge (\neg a \vee b \vee c \vee d) \wedge (\neg a \vee b \vee c \vee \neg d)$.

SLUR fails for the variable order $[a, b, c, d]$. However, \mathcal{F} has neither failed nor backbone literals.

Definition 2. Failed Literal Backbone Equivalent (FLBE) is the class of formulas on which the negation of every backbone literal is a failed literal.

This class defines the upper bound on which SURB is complete, if it had an oracle to determine the optimal ordering of literals. Without the correct ordering, SURB would need to be executed up to n times to identify the entire backbone of a formula in FLBE. The following example illustrates this.

Example 2. Consider the formula

$$\mathcal{F} = (\neg a \vee \neg b) \wedge (\neg a \vee b) \wedge (a \vee \neg c \vee \neg d) \wedge (a \vee \neg c \vee d).$$

If c is propagated before a , only $\neg a$ will be found by SURB. However, both a and c are failed literals and there are no further backbone literals, thus \mathcal{F} is in FLBE.

Now we proceed to discuss the time complexity of SURB. It performs up to n propagations and therefore has a worst-case complexity of $\mathcal{O}(n \cdot |\mathcal{F}|)$. Järvisalo and Korhonen [25] suggest that any algorithm to find even a single backbone literal in a Horn-3-CNF has worst-case complexity of $\mathcal{O}(n \cdot |\mathcal{F}|)$ under the strong exponential time hypothesis [26]. Since SURB subsumes the problem and is complete on a superset of Horn-3-CNF, it is unlikely that we can achieve a better worst-case complexity than what this simple algorithm offers.

The same asymptotic time complexity is also shared by SLUR [16]. However, while SLUR continuously extends an assignment and uses it for future propagations, SURB only saves backbone literals. As in the end both algorithms propagate each literal at least once, keeping more literals assigned can lead to a faster overall runtime. We exploit this idea in the design of Algorithm 3 in the next section.

IV. BIG BACKBONES

As the algorithm presented in the previous section is generally not guaranteed to identify the entire backbone of a formula, it can only serve as part of the backbone search. Applying SURB to the entire formula would be too slow, even with the highly optimized implementations of unit propagation in modern SAT solvers. It is also not possible to efficiently identify the subset of a formula that is in SLUR [24]. We therefore focus on the binary clauses where propagation can be implemented more efficiently and SURB is complete. The following proposition justifies focusing on a subset.

Proposition 1. The backbone found on a subset of a satisfiable formula \mathcal{F} is a subset of the backbone of \mathcal{F} .

In Algorithm 3, we present KB3, a version of SURB, which is only valid for 2-CNFs and avoids re-propagation by keeping an assignment between propagations.

KB3 (2-CNF \mathcal{F})

```

1   $\mathcal{B} \leftarrow \emptyset, \quad \Lambda \leftarrow \mathcal{L}$ 
2  while  $\Lambda \neq \emptyset$ 
3     $\sigma \leftarrow \mathcal{B}, \quad \Delta \leftarrow \emptyset$ 
4    for  $\ell \in \Lambda$  // next candidate
5      if  $\neg\ell \in \sigma$  then continue
6       $\Delta \leftarrow \Delta \cup \{\ell\}$ 
7      if  $\ell \in \sigma$  then continue
8       $\sigma' \leftarrow \{k \mid \mathcal{F}_{|\sigma \wedge \ell} \vdash_1 k\}$ 
9      if  $\frac{1}{2} \in \sigma'$  then
10        $\mathcal{B} \leftarrow \mathcal{B} \cup \{k \mid \mathcal{F}_{|\neg\ell} \vdash_1 k\}$ 
11        $\Delta \leftarrow \Delta \cup \mathcal{B} \cup \neg\mathcal{B}$ 
12        $\sigma \leftarrow \sigma \cup \mathcal{B}$ 
13     else  $\sigma \leftarrow \sigma'$ 
14    $\Lambda \leftarrow \Lambda \setminus \Delta$ 
15 return  $\mathcal{B}$ 

```

Algorithm 3: Keep assignment BIG Backbone (KB3) is only defined on 2-CNFs, for which it is complete regardless of the literal selection order (in line 4).

The example in Figure 1 illustrates why we can keep literals assigned without encountering spurious conflicts. Specifically, running the KB3 algorithm for this formula, when c is picked as the first candidate (line 4), all candidates with a path to $\neg c$ are blocked until the assignment is reset in line 3.

Theorem 2. Algorithm 3 is sound and complete on 2-CNF.

Proof. Since 2-CNF is a subset of SURB we can rely on the completeness of Algorithm 2 for any variable ordering. We can therefore assume the set \mathcal{B} to be empty for every candidate ℓ in line 3. Every literal is either immediately eliminated in line 7 or eventually propagated in line 8. Note that propagation under a bigger assignment is only more likely to derive a conflict. Any eliminated literal has been assigned by a previous propagation that did not lead to a conflict.

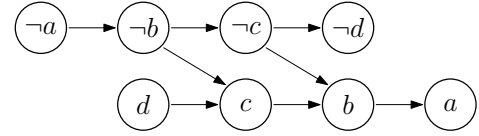


Fig. 1: BIG of $(a \vee \neg b) \wedge (b \vee \neg c) \wedge (b \vee c) \wedge (c \vee \neg d)$.

To show soundness, consider a conflict derived in line 8. Let c and $\neg c$ be any pair of conflicting literals and ℓ the current candidate. We show neither c nor $\neg c$ are in σ and therefore $\mathcal{F}_{|\ell} \vdash_1 \frac{1}{2}$ which implies that $\neg\ell$ is a backbone literal. It is impossible for c and $\neg c$ to both be in σ since the conflict would have prevented the assignment from being updated in line 13. Without loss of generality assume c to be in σ and the propagation of ℓ to imply $\neg c$. Since ℓ implies $\neg c$, there is a path from ℓ to $\neg c$ in the BIG and by contraposition there is also a path from c to $\neg\ell$. The set σ is the result of propagation therefore every literal implied by c is included. But if $\neg\ell \in \sigma$ the current candidate would have been skipped in line 5. \square

By this proof, ℓ is a failed literal in the original formula. Therefore a resolution proof for $\neg\ell$ being in the backbone can be found by resolving the clauses corresponding to the paths from ℓ to c and ℓ to $\neg c$ in the BIG.

The example below shows that Algorithm 3 does not extend to Horn-3-CNF.

Example 3. Consider the formula $(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$. Both $\neg a, b, c$ and $a, \neg b, \neg c$ satisfy the formula, the backbone is therefore empty. However, if the candidates are picked in the order $[a, b, \dots]$ literal $\neg b$ is identified as part of the backbone.

V. RELATED TECHNIQUES

We now discuss some previous work on extracting backbones from BIGs [21], as well as other techniques used in failed literal extraction and how they relate to KB3. We refer to Figure 1 for an illustration of the following discussions. The algorithm Van Gelder describes in [21] is essentially equivalent to SURB with a depth-first search order, instead of the usual breadth-first propagation. Whenever the BFS propagation of a literal $\neg a$ causes the assignment of conflicting literals c and $\neg c$, the BIG does not only contain a path from $\neg a$ to c and $\neg a$ to $\neg c$ but by contraposition also a path from c to a . Thus, with a DFS order the first conflicting literal b , is always in the backbone. Furthermore, b is the highest such literal in the search tree, so propagating it will identify all other backbones that can be found for this conflict. To emulate this desirable property with BFS, we can explicitly store the search tree and identify the first UIP[27] after a conflict is encountered.

Stamping [28], [29] prevents a literal to be considered as a candidate, if it has been propagated since the last backbone literal was identified. However, such a literal must still be re-propagated if it is encountered during another propagation, as the previous example demonstrates for the candidate order $[d, \neg a, \dots]$. KB3 subsumes this technique, since any candidate that is not propagated due to stamping would still be assigned and added to Δ in line 6, at the first time it is encountered.

Moreover, while stamping is reset when a conflict is encountered, KB3 still maintains part of the assignment.

Roots [21], [30], [31] only propagates a candidate if it has no predecessor in the BIG. Removing the negation of an identified backbone literal can add new roots. This optimization is part of Van Gelder’s algorithm and also used in failed literal elimination. To maintain completeness, it is necessary to run ELS until fixpoint. Note that if only the BIG is considered, one round of ELS is sufficient. Since a root cannot be implied by another candidate, this technique also subsumes Stamping. Note that combining this technique with KB3 increases the size of the unkept assignment when a conflict is encountered and can therefore also have negative effects.

We present two scalable examples of 2-CNF formulas. Figure 2a is lifted from [21]. They used the example to show that their algorithm expands $\mathcal{O}(n^3)$ edges and is therefore not more efficient than computing the two-closure. In contrast, our algorithm expands each edge exactly once and is thus in $\mathcal{O}(n^2)$. We can therefore achieve a speedup of n times and reach the lower bound complexity of performing a single propagation. However, as the example in Figure 2b shows, the worst case complexity has not changed. Each of the $\mathcal{O}(n)$ roots in group R expands the $\mathcal{O}(n^2)$ edges in P and the at-most-one constraint prevents any of the propagations from being reused.

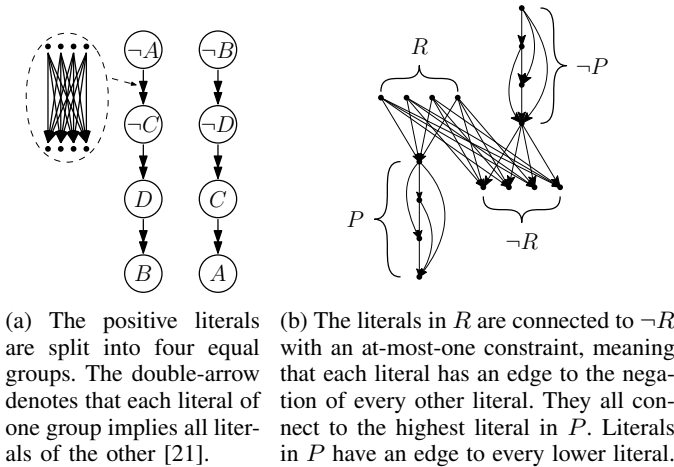


Fig. 2: Depicted are the BIGs of two scalable SAT formulas. On (a) KB3 achieves a linear speed up over the previous state of the art. However, (b) shows that KB3 does not improve upon the worst case performance.

VI. IMPLEMENTATION AND EVALUATION

We implement the new algorithm KB3 [32] and the base version SURB with various optimizations as preprocessors for CADiBACK [15]. For each configuration we tested both DFS and BFS for propagation. The binary clauses are extracted after some basic preprocessing has been performed by CADICAL and stored as an adjacency array. All backbone literals in the BIG are then extracted and added as unit clauses before the first call to a SAT solver. To increase trust, we checked that all configurations identify the same backbone on close to a

	SURB		KB3	
	BFS	DFS	BFS	DFS
Base	21136.11	21287.25	647.53	728.46
ELS	20523.81	20756.81	640.43	733.47
ELS+Roots	18164.47	18756.10	643.57	721.09
stamp	19205.73	19636.01		
ELS+Stamping	18947.99	19392.12		
ELS+Roots+UIP	822.49			

Fig. 3: The time in seconds to run backbone extraction on the BIG until completion accumulated over all satisfiable benchmarks from the last 19 SAT competitions (2004-2022). The time to run ELS on the entire benchmark set is 50.59 seconds and included for the algorithms which use it.

billion randomly generated 2-CNF. We use a cluster with 20 nodes each running two AMD EPYC 7313 at 3.7Ghz under Ubuntu 22.04 LTS. Memory is limited to 15GB per instance.

For benchmarking, we collected formulas from SAT competitions 2004-2022, and removed duplicates to obtain a large and representative set. We ran Kissat 3.0.0 [33] for 5,000 seconds to identify satisfiable benchmarks. This left us with 1798 benchmarks (available at <https://cca.informatik.uni-freiburg.de/sc04to22sat.zip> (6 GB) and [34]).

Table 3 presents the comparison of the different configurations. Even though the source code from [21] is not available, the configuration of SURB with DFS and ELS+Roots in our implementation is equivalent to what they describe in their paper and we use it as a representation of the previous state of the art. The results show that the new algorithm clearly outperforms the configuration of [21], being more than 29 times faster. Three benchmarks are particularly hard for SURB. Only the BFS configurations without stamping solve them within the time limit of 5000 seconds, whereas KB3 takes less than a second to solve them. Furthermore, the additional optimizations work well for the base version, however, as discussed in the previous section, KB3 does not seem to benefit from them as much.

As argued before, KB3 subsumes stamping and the combination is therefore not presented. Similarly, the UIP technique is not necessary when a depth first order is used for propagation and has not been implemented for SURB.

In the second part of the evaluation we investigate how the best configuration of KB3 (BFS and ELS) performs as a preprocessor for the complete backbone extractor CADiBACK. We log the time of identifying a backbone literal for the 533 benchmarks from the past three SAT competitions (2020-2022) and present their accumulation over time in Figure 4. Even though we limit the run time to 1000 seconds, still more than 10 Million backbone literals are identified. The version with KB3 holds the biggest absolute advantage at around 210 seconds, where it identified 5.5 Million backbone literals, 4.5 times as many as the base version has found at that point.

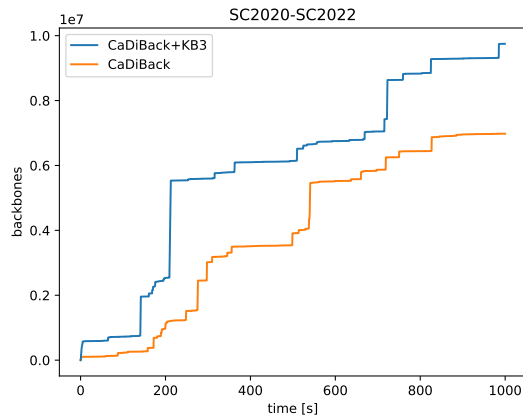


Fig. 4: Presented is the number of backbone literals identified over time. We compare default CADIBACK to a version with added preprocessing performed by KB3.

VII. CONCLUSION

We proposed a new algorithm for backbone extraction from the binary implication graph of a formula. The new algorithm exhibits a significant performance advantage over the previous state-of-the-art approach. Furthermore, we have integrated our algorithm into the backbone extractor CADIBACK as a preprocessor, yielding remarkable improvements, particularly in the early identification of backbone literals.



Acknowledgements: This work is supported by the Austrian Science Fund (FWF) under projects W1256-N23 and S11408-N23, and the LIT AI Lab funded by the State of Upper Austria.

REFERENCES

- [1] C. S. Zhu, G. Weissenbacher, and S. Malik, "Post-silicon fault localisation using maximum satisfiability and backbones," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2011, pp. 63–66.
- [2] C. S. Zhu, G. Weissenbacher, D. Sethi, and S. Malik, "SAT-based techniques for determining backbones for post-silicon fault localisation," in *2011 IEEE International High Level Design Validation and Test Workshop*, Nov. 2011, pp. 84–91.
- [3] C. S. Zhu, G. Weissenbacher, and S. Malik, "Silicon fault diagnosis using sequence interpolation with backbones," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2014, pp. 348–355.
- [4] M. Janota, "SAT solving in interactive configuration," Ph.D. dissertation, University College Dublin, 2010.
- [5] P. C. Cheeseman, B. Kanefsky, W. M. Taylor *et al.*, "Where the really hard problems are," in *Ijcai*, vol. 91, 1991, pp. 331–337.
- [6] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky, "Determining computational complexity from characteristic 'phase transitions'," *Nature*, vol. 400, pp. 133–137, Jul. 1999.
- [7] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman, "Generating satisfiable problem instances," *AAAI/IAAI*, vol. 2000, pp. 256–261, 2000.
- [8] T. Al-Yahya, M. E. B. A. Menai, and H. Mathkour, "Boosting the Performance of CDCL-Based SAT Solvers by Exploiting Backbones and Backdoors," *Algorithms*, vol. 15, no. 9, p. 302, Sep. 2022.
- [9] O. Dubois and G. Dequen, "A backbone-search heuristic for efficient solving of hard 3-SAT formulae," in *IJCAI*, vol. 1, 2001, pp. 248–253.
- [10] A. B. M. Fleury, "GIMSATUL, ISASAT, KISSAT," *SAT COMPETITION 2022*, p. 10.
- [11] M. El Bachir Menai, "A Two-Phase Backbone-Based Search Heuristic for Partial MAX-SAT – An Initial Investigation," in *Innovations in Applied Artificial Intelligence*, ser. Lecture Notes in Computer Science, M. Ali and F. Esposito, Eds. Berlin, Heidelberg: Springer, 2005, pp. 681–684.
- [12] W. Zhang, A. Rangan, M. Looks *et al.*, "Backbone guided local search for maximum satisfiability," in *IJCAI*, vol. 3, 2003, pp. 1179–1186.
- [13] G. Zeng, C. Zheng, Z. Zhang, and Y. Lu, "An Backbone Guided Extremal Optimization Method for Solving the Hard Maximum Satisfiability Problem," in *2012 International Conference on Computer Application and System Modeling*. Atlantis Press, Aug. 2012, pp. 1301–1304.
- [14] W. Zhang, "Configuration landscape analysis and backbone guided local search.: Part I: Satisfiability and maximum satisfiability," *Artificial Intelligence*, vol. 158, no. 1, pp. 1–26, Sep. 2004.
- [15] A. Biere, N. Froleys, and W. Wang, "CadiBack: Extracting Backbones with CaDiCaL," in *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Mahajan and F. Slivovsky, Eds., vol. 271. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 3:1–3:12. [Online]. Available: <https://drops.dagstuhl.de/opus/volltexte/2023/18465>
- [16] J. S. Schlipf, F. S. Annexstein, J. V. Franco, and R. P. Swaminathan, "On Finding Solutions for Extended Horn Formulas," *Inf. Process. Lett.*, vol. 54, no. 3, pp. 133–137, 1995.
- [17] A. del Val, "On 2-SAT and renamable Horn," pp. 279–284, 2000.
- [18] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960. [Online]. Available: <https://doi.org/10.1145/321033.321034>
- [19] I. P. Gent, "Optimal Implementation of Watched Literals and More General Techniques," *Journal of Artificial Intelligence Research*, vol. 48, pp. 231–252, Oct. 2013.
- [20] B. Aspvall, M. F. Plass, and R. E. Tarjan, "A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas," *Inf. Process. Lett.*, vol. 8, no. 3, pp. 121–123, 1979.
- [21] A. Van Gelder, "Toward leaner binary-clause reasoning in a satisfiability solver," *Annals of Mathematics and Artificial Intelligence*, vol. 43, no. 1, pp. 239–253, Jan. 2005.
- [22] D. L. Berre, "Exploiting the real power of unit propagation lookahead," *Electron. Notes Discret. Math.*, vol. 9, pp. 59–80, 2001.
- [23] J. Franco and A. Van Gelder, "A perspective on certain polynomial-time solvable classes of satisfiability," *Discrete Applied Mathematics*, vol. 125, no. 2, pp. 177–214, Feb. 2003.
- [24] O. Čepek, P. Kučera, and V. Vlček, "Properties of SLUR Formulae," in *SOFSEM 2012: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán, Eds. Berlin, Heidelberg: Springer, 2012, pp. 177–189.
- [25] M. Järvisalo and J. H. Korhonen, "Conditional Lower Bounds for Failed Literals and Related Techniques," in *Theory and Applications of Satisfiability Testing – SAT 2014*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds. Cham: Springer International Publishing, 2014, pp. 75–84.
- [26] R. Impagliazzo, R. Paturi, and F. Zane, "Which Problems Have Strongly Exponential Complexity?" *Journal of Computer and System Sciences*, vol. 63, no. 4, pp. 512–530, Dec. 2001.
- [27] A. Darwiche and K. Pipatsrisawat, "Complete algorithms," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 101–132. [Online]. Available: <https://doi.org/10.3233/FAIA200986>
- [28] A. Biere, M. Järvisalo, and B. Kiesl, "Preprocessing in SAT solving," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 391–435. [Online]. Available: <https://doi.org/10.3233/FAIA200992>
- [29] P. Simons, I. Niemelä, and T. Soinen, "Extending and implementing the stable model semantics," *Artificial Intelligence*, vol. 138, no. 1-2, pp. 181–234, 2002.
- [30] R. Gershman and O. Strichman, "Cost-Effective Hyper-Resolution for Preprocessing CNF Formulas," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds. Berlin, Heidelberg: Springer, 2005, pp. 423–429.

- [31] A. Biere, “CaDiCaL, Lingeling, Plingeling, Treengeling and Yalsat entering the SAT Competition 2018,” *Proceedings of SAT Competition*, pp. 14–15, 2017.
- [32] Froleys, Nils, “KB3: Keep Big Backbones,” 2023, <http://fmv.jku.at/kb3>.
- [33] A. Biere and M. Fleury, “Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022,” in *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, M. Iser, M. Jarvisalo, and M. Suda, Eds., vol. B-2022-1. University of Helsinki, 2022, pp. 10–11.
- [34] A. Biere, N. Froleys, and W. Wang, “Sampled and Normalized Satisfiable Instances from the main track of the SAT Competition 2004 to 2022,” Mar. 2023. [Online]. Available: <https://doi.org/10.5281/zenodo.7750076>

Local Search For SMT On Linear and Multi-linear Real Arithmetic

Bohan Li , Shaowei Cai 

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences

School of Computer Science and Technology, University of Chinese Academy of Sciences

Beijing, China

{libh, caisw}@ios.ac.cn

Abstract—Satisfiability Modulo Theories (SMT) has significant application in various domains. In this paper, we focus on quantifier-free Satisfiability Modulo Real Arithmetic, referred to as SMT(RA), including both linear and non-linear real arithmetic theories. As for non-linear real arithmetic theory, we focus on one of its important fragments where the atomic constraints are multi-linear. We propose the first local search algorithm for SMT(RA), called LocalSMT(RA), based on two novel ideas. First, an interval-based operator is proposed to cooperate with the traditional local search operator by considering the interval information. Moreover, we propose a tie-breaking mechanism to further evaluate the operations when the operations are indistinguishable according to the score function. Experiments are conducted to evaluate LocalSMT(RA) on benchmarks from SMT-LIB. The results show that LocalSMT(RA) is competitive with the state-of-the-art SMT solvers, and performs particularly well on multi-linear instances.

I. INTRODUCTION

Satisfiability Modulo Theories (SMT) is the problem of checking the satisfiability of a first order logic formula with respect to certain background theories. It has been applied in various areas, including program verification and termination analysis [1], [2], symbolic execution [3] and test-case generation [4], etc.

In this paper, we focus on the theory of quantifier-free real arithmetic, consisting of atomic constraints in the form of polynomial equalities or inequalities over real variables. The theory can be divided into two categories, namely *linear real arithmetic* (LRA) and *non-linear real arithmetic* (NRA), based on whether the arithmetic atomic constraints are linear or not. As for NRA, this paper concerns one of its important fragments where the atomic constraints are multi-linear. The SMT problem with the background theory of LRA and NRA is to determine the satisfiability of the Boolean combination of respective atomic constraints, referred to as SMT(LRA) and SMT(NRA). In general, we refer to the SMT problem on the theory of real arithmetic as SMT(RA).

A. Related Work

The mainstream approach for solving SMT(RA) is the *lazy* approach [5], [6], also known as DPLL(T) [7], which relies on the interaction of a SAT solver with a theory solver. Most state-of-the-art SMT solvers supporting the theory of real arithmetic are mainly based on the *lazy* approach, including Z3 [8], Yices2 [9], SMT-RAT [10], cvc5 [11], OpenSMT [12] and

MathSAT5 [13]. In the DPLL(T) framework, the SMT formula is abstracted into a Boolean formula by replacing arithmetic atomic constraints with fresh Boolean variables. A SAT solver is employed to reason about the Boolean structure, while a theory solver is invoked to receive the set of theory constraints determined by the SAT solver, and solve the conjunction of these theory constraints, including consistency checking of the assignments and theory-based deduction.

The efforts in the *lazy* approach are mainly devoted to designing effective decision procedures, serving as theory solvers to deal with the conjunction of theory constraints. The core reasoning module for LRA integrated in DPLL(T) is a variant of the *simplex* algorithm dedicated for SMT solving, proposed in [14]. Another approach for solving LRA constraint systems is the *Fourier-Motzkin* variable elimination [15], which often shows worse performance than the *simplex* algorithm.

As for non-linear real arithmetic, the *cylindrical algebraic decomposition* (CAD) [16] is the most widely used decision procedure, and CAD is adapted and embedded as theory solver in the SMT-RAT solver [10] with improvement since [17]. Other well-known methods use Gröbner bases [18] or the realization of sign conditions [19]. Incomplete methods include a theory solver [20] based on virtual substitution [21], and techniques based on interval constraint propagation [22] proposed in [23], [24].

Moreover, Constructing Satisfiability (MCSat) calculus is also an efficient framework for solving SMT(RA). It was proposed in [25] for solving SMT(LRA), and an elegant variation of CAD method is instantiated in the model-constructing satisfiability calculus framework of Z3 [26] for solving SMT(NRA).

Local search is an incomplete method playing a significant part in many combinatorial problems [27]. It has been successfully applied to the Boolean Satisfiability (SAT) problem [28], [29], [30], [31], [32] and can rival CDCL solvers on certain types of instances.

Local search for SMT, however, has only received very little amount of attention. The idea of integrating local search solvers with theory solvers has been previously explored to solve SMT(LRA), where a local search SAT solver WalkSAT is used to solve the Boolean skeleton of the SMT formula [33]. A local search solver called LocalSMT was recently employed to SMT on integer arithmetic [34], [35]. Moreover, local search

algorithm has been applied to Bit-vectors [36], [37], [38]. However, we are not aware of any local search algorithm for SMT on real arithmetic.

B. Contributions

In this paper, for the first time, we design a local search algorithm for SMT(RA), namely LocalSMT(RA), based on the following novel strategies. Note that LocalSMT(RA) is implemented as a fragment of LocalSMT [35], which is our local search solver dedicated for SMT.

First, we propose the *interval-based* operator to enhance the conventional local search operator by taking interval information into account. Specifically, we observe that assigning the real-value variable to any value in a given interval would make the same amount of currently falsified clauses become satisfied. Hence, the *interval-based* operator evaluates multiple values inside the interval as the potential value of the operation, rather than only assign it to a fixed value (e.g. the threshold value to satisfy a constraint).

Moreover, we observe that there frequently exist multiple operations with the same best score when performing local search, and thus a tie-breaking mechanism is proposed to further distinguish these operations.

Experiments are conducted to evaluate LocalSMT(RA) on 2 benchmark sets, namely SMT(LRA) and SMT(NRA) benchmarks from SMT-LIB. Note that unsatisfiable instances are excluded, and we only consider multi-linear instances from SMT(NRA) benchmark. We compare LocalSMT(RA) with the top 4 SMT solvers in the relevant logics (QF_LRA, QF_NRA) according to the SMT-COMP 2022¹, excluding the portfolio and derived solvers. Specifically, as for SMT(LRA), we compare LocalSMT(RA) with OpenSMT, Yices2, cvc5 and Z3, while for SMT(NRA), the competitors are Z3, cvc5, Yices2 and SMT-RAT. Experimental results show that LocalSMT(RA) is competitive and complementary with state-of-the-art SMT solvers, especially on multi-linear instances. Moreover, the ablation experiment confirms the effectiveness of our proposed novel strategies.

Note that multi-linear instances are comparatively difficult to solve by existing solvers. For example, Z3, perhaps the best solver for satisfiable SMT(NRA) instances according to SMT-COMP 2022, can solve 90.5% QF(NRA) instances, while it can only solve 77.5% multi-linear instances. However, multi-linear instances are suitable for local search, since without high order terms, the operation can be efficiently calculated.

C. Paper Organization

In section II, preliminary knowledge is introduced. In section III, we propose a novel *interval-based operator* to enrich the traditional operator by considering the interval information. In section IV, a *tie-breaking mechanism* is proposed to distinguish multiple operations with the same best score. Based on the two novel strategies, our local search for SMT(RA) is proposed in section V. Experiments are presented in section VI. Conclusion and future work are given in section VII.

¹<https://smt-comp.github.io/2022/>

II. PRELIMINARIES

A. Basic Definitions

A *monomial* is an expression of the form $x_1^{e_1} \dots x_m^{e_m}$ where $m > 0$, x_i are variables and e_i are exponents, $e_i > 0$ for all $i \in \{1 \dots m\}$, and $x_i \neq x_j$ for all $i, j \in \{1 \dots m\}, i \neq j$. A monomial is linear if $m = 1$ and $e_1 = 1$.

A *polynomial* is a linear combination of monomials, that is, an arithmetic expression of the form $\sum_i a_i m_i$ where a_i are coefficients and m_i are monomials. If all its monomials are linear in a polynomial, indicating that the *polynomial* can be written as $\sum_i a_i x_i$, then it is *linear*, and otherwise it is *non-linear*. A special case of non-linear polynomial is *multi-linear* polynomial, where the highest exponent for all variables is 1, indicating that each monomial is in the form of $x_1 \dots x_m$.

Definition 1: The atomic constraints of the theory of real arithmetic are polynomial inequalities and equalities, in the form of $\sum_i a_i m_i \bowtie k$, where $\bowtie \in \{=, \leq, <, \geq, >\}$, m_i are monomials consisting of real-valued variables, k and a_i are rational constants.

The formulas of the SMT problem on the theory of real arithmetic, denoted as SMT(RA), are Boolean combinations of atomic constraints and propositional variables, where the sets of real-valued variables and propositional variables are denoted as X and P . The SMT problem on the theory of linear real arithmetic (LRA) and non-linear arithmetic (NRA) are denoted as SMT(LRA) and SMT(NRA), respectively. As for NRA, this paper only considers one important fragment where the polynomials in atomic constraints are multi-linear, denoted as *MRA* in this paper.

Example 1: Let $X = \{x_1, x_2, x_3, x_4, x_5\}$ and $P = \{p_1, p_2\}$ denote the sets of integer-valued and propositional variables respectively. A typical SMT(LRA) formula F_{LRA} and SMT(MRA) formula F_{MRA} is shown as follows:

$$F_{LRA}: (p_1 \vee (x_1 + 2x_2 \leq 2)) \wedge (p_2 \vee (3x_3 + 4x_4 = 2) \vee (-x_2 - x_3 < 3))$$

$$F_{MRA}: (p_1 \vee (x_1 x_2 \leq 2)) \wedge (p_2 \vee (3x_3 x_4 + 4x_4 = 2) \vee (-x_2 - x_3 < 3))$$

In the theory of real arithmetic, a positive, infinitesimal real number is denoted as δ .

A literal is an atomic constraint or a propositional variable, or their negation. A *clause* is the disjunction of a set of literals, and a formula in *conjunctive normal form* (CNF) is the conjunction of a set of clauses. For an SMT(RA) formula F , an assignment α is a mapping $X \rightarrow R$ and $P \rightarrow \{false, true\}$, and $\alpha(x)$ denotes the value of a variable x under α . A *complete assignment* is a mapping which assigns to each variable a value. A literal is true if it evaluates to true under the given assignment, and false otherwise. A clause is *satisfied* if it has at least one true literal, and *falsified* if all literals in the clause are false. A complete assignment is a *solution* to an SMT(RA) formula iff it satisfies all the clauses.

B. Local Search

When local search is performed on the SMT problem, the search space is comprised of all complete assignments, each of

which represents a candidate solution. Typically, a local search algorithm begins with a complete assignment and repeatedly updates it by modifying the value of variables in order to find a *solution*.

Given a formula F , the *cost* of an assignment α , denoted as $cost(\alpha)$, is the number of falsified clauses under α . In dynamic local search algorithms which use clause weighting techniques [39], [30], $cost(\alpha)$ denotes the total weight of all falsified clauses under an assignment α (The weight is computed according to the PAWS scheme which will be described in detail in Section V-B).

A key component of a local search algorithm is the set of *operators*, which define how to modify the current solution. When an operator is instantiated by specifying the variable to operate and the value to assign, an *operation* is obtained. The operation to assign variable x to value v is denoted as $op(x, v)$. The critical move operator for SMT on linear integer arithmetic proposed in [34] is defined as follows.

Definition 2: The critical move operator, denoted as $cm(x, \ell)$, assigns an integer variable x to the threshold value making literal ℓ true, where ℓ is a falsified literal containing x .

For example, given a falsified literal $\ell = (x + 1 \leq 0)$ where x is currently assigned to 0, the corresponding operation $cm(x, \ell)$ will assign x to -1 .

Local search algorithms usually choose an operation among candidate operations according to some scoring function. Given a formula and an assignment α , the most commonly used scoring function of an operation op is defined as

$$score(op) = cost(\alpha') - cost(\alpha)$$

where α' is the resulting assignment by applying op to α . An operation op is said to be *decreasing* if $score(op) > 0$.

Another property used for evaluating an operation is its *make value*.

Definition 3: Given an operation op , the make value of op , denoted as $make(op)$, is the number of falsified clauses that would become satisfied after performing op .

III. INTERVAL-BASED OPERATION

Critical move satisfies falsified clauses by modifying one variable in a false literal to make it true. This operator can still be used in the context of SMT(RA), and it is also used in our algorithm. However, an issue of accuracy arises when applying the critical move operator in the context of Real Arithmetic. Recalling that we need to calculate the threshold value for a literal to become true, when solving a strict inequality, there is no threshold value. Instead, the value depends on what accuracy we intend to maintain. In this section, we propose an operator for SMT(RA), which considers the interval information and is more flexible than critical move.

A. Satisfying Domain

An important fact on linear or multi-linear inequality of real-value variables is that, when all variables but one in

the inequality are fixed, there is a domain for the remaining variable whose coefficient is not 0, such that assigning the variable to any value in the domain makes the inequality hold. Thus, given a falsified literal ℓ in the form of an atomic constraint and a variable x in it, it can be satisfied by assigning x to any value in the corresponding domain, called *Satisfying Domain*. For example, consider a literal $\ell : (x - y > 4)$ where the current assignment is $\alpha = \{x = 0, y = 0\}$, then obviously assigning x to any value in $(4, +\infty)$ satisfies the inequality, and thus the *Satisfying Domain* is $(4, +\infty)$.

We further extend the definition of *Satisfying Domain* to the clause level, defined as follows.

Definition 4: Given an assignment α , for a false literal ℓ and a variable x appearing in ℓ , the **satisfying domain of x for literal ℓ** is $SD_l(x, \ell) = \{v | \ell \text{ becomes true if assigning } x \text{ to } v\}$; for a falsified clause c and a variable x in c , the **satisfying domain of x for clause c** is $SD_c(x, c) = \bigcup_{\ell \in c} SD_l(x, \ell)$.

Since the false literal ℓ is in the form of linear or multi-linear inequality, $SD_l(x, \ell)$ is either in the form of $(-\infty, u]$ or $[l, \infty)$. Thus, as the union of $SD_l(x, \ell)$, $SD_c(x, c)$ may contain $(-\infty, u]$ whose upper bound is defined as $UB(x, c) = u$, or $[l, \infty)$ whose lower bound is defined as $LB(x, c) = l$, or both kinds of intervals. For simplicity, interval $(-\infty, u)$ or (l, ∞) are denoted as $(-\infty, u - \delta]$ or $[l + \delta, \infty)$ respectively.

Example 2: Given a clause $c = \ell_1 \vee \ell_2 \vee \ell_3 = (a - b > 4) \vee (2a - b \geq 7) \vee (2a - c \leq -5)$ and the current assignment $\alpha = \{a = 0, b = 0, c = 0\}$, for variable a , the satisfying domains to the three literals are $SD_l(a, \ell_1) = [4 + \delta, \infty)$, $SD_l(a, \ell_2) = [3.5, \infty)$ and $SD_l(a, \ell_3) = (-\infty, -2.5]$ respectively. The *Satisfying Domain* to clause c is $SD_c(a, c) = (-\infty, -2.5] \cup [3.5, \infty)$, and the corresponding upper bound and lower bound are $UB(a, c) = -2.5$ and $LB(a, c) = 3.5$.

B. Equi-make Intervals

Based on the variables' *satisfying domain* to clauses, we observe that operations assigning the variable to any value in a given interval would satisfy the same amount of falsified clauses, that is, they have the same *make value*. We call such interval as *equi-make interval*.

Definition 5: Given an SMT(RA) formula F and an assignment α to its variables, for a variable x , an **equi-make interval** is a maximal interval I such that all operations $op(x, v)$ with $v \in I$ have the same make value.

We can divide $(-\infty, +\infty)$ into several equi-make intervals w.r.t. a variable.

Example 3: Consider a formula $F : c_1 \wedge c_2$ where both clauses are falsified under the current assignment and variable a appears in both clauses. Suppose $SD_c(a, c_1) = [3, +\infty)$ and $SD_c(a, c_2) = [5, +\infty)$, then we can divide $(-\infty, +\infty)$ into three intervals as $(-\infty, 3)$, $[3, 5)$ and $[5, +\infty)$. Operations assigning a to any value in $(-\infty, 3)$ results in a make value of 0, those assigning a to a value in $[3, 5)$ results in a make value of 1, while those corresponding to $[5, +\infty)$ results in a make value of 2.

Thus, we can enrich the traditional *critical move* operator by considering the interval information. The intuition is to find

the equi-make intervals, and then consider multiple values in such interval as the options for future value of operations, rather than only consider the threshold value.

We focus on the variables appearing in at least one falsified clause. Here we describe a procedure to partition $(-\infty, +\infty)$ into equi-make intervals for such variables.

- First, we go through the falsified clauses. For each falsified clause c , we calculate for each real-valued variable x in c the corresponding *satisfying domain* to c , $SD_c(x, c)$, as well as the upper bound $UB(x, c)$ and lower bound $LB(x, c)$ if they exist.
- Then, for each real-valued variable x appearing in falsified clauses, all its UB s are sorted in the ascending order, while LB s sorted in the descending order. After sorting, these bounds are labeled as $UB^1(x), \dots, UB^n(x)$ and $LB^1(x), \dots, LB^m(x)$, where $UB^n(x)$ and $LB^m(x)$ denotes the maximum of UB and minimum of LB for x respectively². For convenience in description, we denote $UB^0(x) = -\infty$ and $LB^0(x) = \infty$. These bounds are listed in order: $UB^0(x) < UB^1(x) < \dots < UB^n(x) < LB^m(x) < \dots < LB^1(x) < LB^0(x)$.
- Finally, for each variable x , we obtain an interval partition $IP(x) = \bigcup_{0 \leq i \leq n} \{(UB^{i-1}(x), UB^i(x))\} \cup (UB^n, LB^m) \cup \bigcup_{0 \leq j \leq m} \{(LB^j(x), LB^{j-1}(x))\}$

Formally, given a real variable x and an interval I from $IP(x)$, $\forall v_1, v_2 \in I$, $make(op(x, v_1)) = make(op(x, v_2))$. As a slight abuse of notation, for an interval I from $IP(x)$, we define its *make value* as the make value of any operation $op(x, v)$ with $v \in I$. Note that all intervals in $IP(x)$ have positive make values except (UB^n, LB^m) , whose make value is 0.

Example 4: Given a formula $F : c_1 \wedge c_2 = (a - b > 4 \vee 2a - b \geq 7 \vee 2a - c \leq -5) \wedge (a - c \geq 2 \vee a - d \leq -1)$ and the current assignment is $\alpha = \{a = 0, b = 0, c = 0\}$. For variable a , $SD_c(a, c_1) = (-\infty, -2.5] \cup [3.5, \infty)$ and $SD_c(a, c_2) = (-\infty, -1] \cup [2, \infty)$. Then, $UB^0(a) = -\infty$, $UB^1(a) = UB(a, c_1) = -2.5$, $UB^2(a) = UB(a, c_2) = -1$, $LB^2(a) = LB(a, c_2) = 2$, $LB^1(a) = LB(a, c_1) = 3.5$ and $LB^0(a) = \infty$.

Therefore, interval partition for x is $IP(x) = I_1 \cup I_2 \cup I_3 \cup I_4 \cup I_5 = (-\infty, -2.5] \cup (-2.5, -1] \cup (-1, 2) \cup [2, 3.5) \cup [3.5, +\infty)$, as shown in Fig 1. For these intervals w.r.t. x , the make value is 2, 1, 0, 1, 2 respectively.

C. Candidate Values for Operations

Since assigning a variable x to any value in an equi-make interval would satisfy the same amount of falsified clauses, after choosing an equi-make interval, we can consider more values in the interval as the option for the future value of operation, rather than only the threshold.

²Note that $UB^n(x) < LB^m(x)$, since otherwise the current assignment is either in the interval $[LB^m(x), \infty)$ or $(-\infty, UB^n(x)]$. Suppose the falsified clauses corresponding to the intervals $[LB^m(x), \infty)$ and $(-\infty, UB^n(x)]$ are c_1 and c_2 , if the current assignment is either in the first or second interval, then according to the definition, either c_1 or c_2 has already been satisfied, which contradicts the definition of UB^n and LB^m .

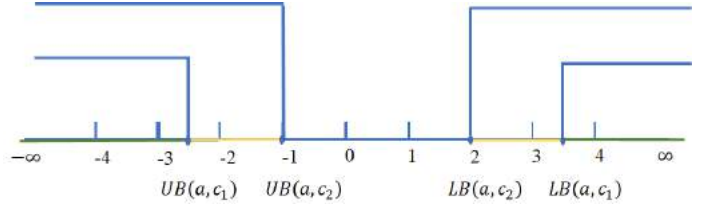


Fig. 1: Interval example

The motivation for the candidate future values is based on the following 3 intuitions: First, we have to restrict the value in the given interval. Moreover, we hope the denominator of corresponding value to be relatively small, in order to avoid exploring complex search space as will be explained in the next section. Finally, the value should be easy to calculate, in order to improve the efficiency of algorithm.

In this work, we only consider the intervals with a positive make value, and thus the interval (UB^n, LB^m) is omitted. Thus, the interval for consideration is of the form $(UB^{i-1}(x), UB^i(x)]$ or $[LB^i(x), LB^{i-1}(x))$. For such an interval, we consider the following values for the operation:

- Assign x to the threshold $UB^i(x)$ or $LB^i(x)$.
- Assign x to the median of the interval, that is $(UB^{i-1}(x) + UB^i(x))/2$ or $(LB^i(x) + LB^{i-1}(x))/2$ (when one of the bounds is ∞ or $-\infty$, the median will not be considered).
- if there are integers in the interval $(UB^{i-1}(x), UB^i(x))$ or the interval $(LB^i(x), LB^{i-1}(x))$, assign x to the largest or smallest integer in the respective open interval; Otherwise, suppose that the open interval can be written as $(\frac{a}{b}, \frac{c}{d})$, then assign x to $\frac{a+c}{b+d}$.

The first option is the same as critical move, and thus *critical move* can be regarded as a special case of our interval-based operator. The second option is a simple choice among intervals. The third option aims to find a rational value limited in the open interval with small denominator, and it is easy to calculate.

Note that it is difficult and even impractical to compute the operations leading to the largest local decrease in the score, based on the following reasons: An operation consists of the variable to modify and the future value to assign. However, the variable can be assigned with arbitrary real number, which is inexhaustible. Although this can be reduced to enumerable intervals, it is still too time-consuming to enumerate all possible operations. Moreover, a literal ℓ contains different variables, and changing one such variable would affect all other literals (not only ℓ) containing the variable. Thus, to calculate the score of an operation, we need to go through all literals containing the variable to modify, which is time-consuming. These two reasons make it a very time-consuming procedure to compute such an operation leading to the largest local decrease in the score.

Thus, our algorithm does not tempt to compute such an operation, but chooses an operation with good score from sampled candidate operations with those candidate values. In

the future work, we will further enrich the sampled candidate values by considering more random values with small denominator in the interval.

IV. A TIE-BREAKING MECHANISM

We notice that there often exist different operations with the same best *score* during local search, and thus tie-breaking is also important to guide the search.

To confirm our observation, we conduct a pre-experiment on 100 randomly selected instances. On each instance, we execute a simple local search algorithm which selects an operation with the best *score* for 10000 iterations, and we count the number of steps where k operations with the same best *score* are found, denoted as $step(k)$.

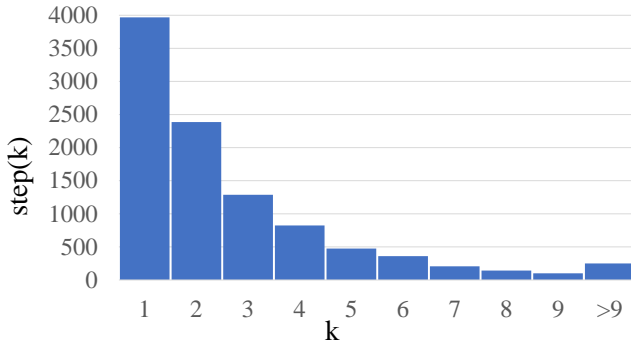


Fig. 2: Average $step(k)$ distribution

As shown in Fig. 2, the steps where more than one operations have the same best *score* take up 61.2% of the total steps. Thus, a tie-breaking heuristic is required to further distinguish these operations with same best *score*.

First, we consider that assigning real-valued variable to values with large denominator can lead the algorithm to a more complex search space where variables are assigned to real number with extremely large denominators, leading to more complicated computation and possible errors, because the local search solver needs to perform factorization of large numbers and numerical approximation to handle these values during the iteration. Thus, we prefer operations that assign variable to a value with a small denominator.

Moreover, we consider that assigning variables to values with large absolute value can lead to the assignment with an extraordinarily large value, deviating the algorithm from finding a possible solution. Thus, we prefer operations assigning variables to values with small absolute value.

Based on the above observation and intuition, we propose a selection rule for picking operations, described as follows.

Selection Rules: Select the operation with the greatest *score*, breaking ties by preferring the one assigning the corresponding variable to a value with the smallest denominator. Further ties are broken by picking the operation assigning the variable with the smallest absolute value.

V. LOCALSMT(RA) ALGORITHM

Our local search algorithm adopts a two-mode framework, which switches between Real mode and Boolean mode. This framework has been used in the local search algorithm LS-LIA for integer arithmetic theories [34].

A. Local search Framework

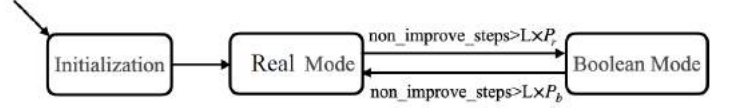


Fig. 3: An SMT Local Search Framework

LocalSMT(RA) is running a local search algorithm starting from an arbitrary model with all real-value variables assigned to 0 and all Boolean variables assigned to false. As depicted in Fig. 3, after the initialization, the algorithm switches between Real mode and Boolean mode. In each mode, an operation on a variable of the corresponding data type is selected to modify the current solution. The two modes switches between each other when the number of non-improving steps (denoted as *non_improve_steps*) of the current mode reaches a threshold. *non_improve_steps* is increased by one when the algorithm fails to find a better solution, otherwise, it is reset as 0. The threshold is set to $L \times P_b$ for the Boolean mode and $L \times P_r$ for the Real mode, where P_b and P_r denote the proportion of Boolean and real-variable literals to all literals in falsified clauses, and L is a parameter.

B. Local Search in Real and Boolean Mode

The algorithm for the Real mode of LocalSMT(RA) is described in Algorithm 1: if the current assignment α satisfies the given formula F , then the solution is found (Line 2). The algorithm tries to find a decreasing *interval-based* operation according to the **Selection Rule** (Line 3–4).

If there exists no such decreasing operation, this is an indication that the algorithm falls into the local optimum. We first update the clause weights according to the probabilistic version of the Pure Additive Weighting Scheme (PAWS) [39], [30] (Line 6), and then randomly sample K interval-based operations into the set Set_{op} (Line 7), where K is a relatively small parameter. The best operation is picked according to **Selection Rules** among Set_{op} (Line 8). Note that since the *interval-based* operation can satisfy at least one clause, picking the best one among few randomly sampled *interval-based* operation can be regarded as a diversification operation.

The probabilistic version of the PAWS scheme works as follows. In the beginning, the weight of each clause is initialized as 1. During the local search process, with probability $1 - sp$, the weight of each falsified clause is increased by one, and with probability sp , for each satisfied clause whose weight is greater than 1, the weight is decreased by one.

As for the Boolean mode focusing on the subformula consisting of Boolean variables, LocalSMT(RA) works in the same way as the Boolean mode of LS-LIA. By putting the

Algorithm 1: Real Mode of LocalSMT(RA)

Input: formula F

Output: A satisfying assignment α of F , or
“UNKNOWN”

```
1 while non_impr_steps <  $L \times P_r$  do
2   if  $\alpha$  satisfies  $F$  then return  $\alpha$ ;
3   if  $\exists$  decreasing interval-based operation then
4      $op :=$  such an operation selected according to
      Selection Rules;
5   else
6     update clause weights according to the PAWS
      scheme;
7     randomly sample  $K$  interval-based operations
      into the set  $Set_{op}$ ;
8      $op :=$  the best operation among  $Set_{op}$  picked
      by Selection Rules;
9    $\alpha := \alpha$  with  $op$  performed;
10 return “UNKNOWN”;
```

Boolean mode and the Real mode together, we propose our local search algorithm for SMT(RA) denoted as LocalSMT(RA).

VI. EXPERIMENTS

We conducted experiments to evaluate LocalSMT(RA) on 2 benchmark sets from SMT-LIB, and compare it with state-of-the-art SMT solvers and local search solvers. Moreover, ablation experiments are conducted to analyze the effectiveness of the proposed strategies.

A. Experiment Preliminaries

Implementation: LocalSMT(RA) is implemented in C++ and compiled by g++ with ‘-O3’ option. There are 3 parameters in LS-LRA: L for switching phases, K for the number of sampled operation and sp (the smoothing probability) for the PAWS scheme. The parameters are tuned according to suggestions from the literature [34], [30] and our preliminary experiments on 20% sampled instances. Preliminary experiments show that LocalSMT(RA) is not sensitive to the parameter setting in a considerable range. Parameters are set as follows: $L = 20$, $K = 3$, $sp = 0.0003$ for all benchmarks. In our implementation, to escape from local optimum, LocalSMT(RA) restarts every 500000 iterations. We use a fixed value for the infinitesimal real number δ , which is $\min(\frac{1}{256}, \frac{1}{c_{max}})$, where c_{max} denotes the maximum absolute value of coefficients in the formula.

Note that in contrast to previous local search for SMT on Linear Integer Arithmetic [34], our solver is able to handle arbitrary Boolean structure of formulas, including “ite” operator, by using the Tseitin encoding [40].

Although any formula with high exponents can be rewritten as multi-linear formula by introducing fresh variables (for example, $x^2 > 4$ can be rewritten as $(x = y) \wedge (x * y > 4)$), however, in that case, we cannot efficiently find the correct feasible solution of x because of the relationship of both

variables. In our implementation, we simplify the formula by eliminating equations in form of $x = (a * y)$, where x and y are real-value variables and a is coefficient. Specifically, we will replace x with $(a * y)$, and after simplifying, we only reserve those instances which are still multi-linear.

Competitors: In the context of SMT(LRA), we compare LocalSMT(RA) with the top 4 state-of-the-art SMT solvers according to SMT-COMP 2022, namely OpenSMT (version 2.4.2)³, Yices2 (version 2.6.2)⁴, cvc5 (version 1.0.0)⁵, and Z3 (version 4.8.17)⁶. While in the context of SMT(NRA), the top 4 competitors are as follows, cvc5 (version 1.0.0), Yices2 (version 2.6.2), Z3 (version 4.8.17) and SMT-RAT-MCSAT (version 22.06)⁷. The binaries of all competitors are downloaded from their websites. Note that portfolio and derived solvers are excluded.

Benchmarks: Experiments are carried out on 2 benchmark sets from SMT-LIB.

- SMTLIB-LRA: The benchmark set contains SMT(LRA) instances from SMT-LIB⁸. As LocalSMT(RA) is an incomplete solver, UNSAT instances are excluded, resulting in a benchmark with 1044 unknown and satisfiable instances.
- SMTLIB-MRA: The benchmark set contains SMT(NRA) instances with multi-linear atomic constraints from SMT-LIB⁹. UNSAT instances are also excluded, resulting in a benchmark with 979 unknown and satisfiable instances.

Experiment Setup: All experiments are carried out on a server with AMD EPYC 7763 64-Core 2.45GHz and 2048G RAM under the system Ubuntu 20.04.4. Each solver is executed once with a cutoff time of 1200 seconds (as in the SMT-COMP) for each instance in both benchmark sets, as they contain sufficient instances (1044 for SMTLIB-LRA and 979 for SMTLIB-MRA). We compare the number of instances where an algorithm finds a model (“#solved”), as well as the run time. Bold values in table emphasize the solver with greatest “#solved”.

B. Results on SMTLIB-LRA Benchmark

1) *Comparison with DPLL(T) solvers:* As shown in Table I, LocalSMT(RA) can solve 900 out of 1044 instances, which is competitive but still cannot rival its competitors. We also perform a runtime comparison between LocalSMT(RA) and each competitor on instances from SMTLIB-LRA in Fig 4, which shows that LocalSMT(RA) is complementary to the competitors.

One explanation for the fact that LocalSMT(RA) cannot rival its DPLL(T) competitors is that 54.5% of the instances in SMTLIB-LRA contain Boolean variables, while the Boolean mode of LocalSMT(RA) is not good at exploiting the relations

³<https://github.com/usi-verification-and-security/opensmt>

⁴<https://yices.csl.sri.com>

⁵<https://cvc5.github.io/>

⁶<https://github.com/Z3Prover/z3/>

⁷<https://github.com/th-s-rwth/smtat>

⁸https://clg-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_LRA

⁹https://clg-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NRA

TABLE I: Results on instances from SMTLIB-LRA

	#inst	cvc5	Yices	Z3	OpenSMT	LocalSMT(RA)
2017-Heizmann	8	4	3	4	4	7
2019-ezsm	84	61	61	53	62	35
check	1	1	1	1	1	1
DTP-Scheduling	91	91	91	91	91	91
LassoRanker	271	232	265	256	262	240
latendresse	16	9	12	1	10	0
meti-tarski	338	338	338	338	338	338
miplib	22	14	15	15	15	4
sal	11	11	11	11	11	11
sc	108	108	108	108	108	108
TM	24	24	24	24	24	11
tropical-matrix	10	1	6	4	6	0
tta	24	24	24	24	24	24
uart	36	36	36	36	36	30
Total	1044	954	995	966	992	900

TABLE II: Comparison with WalkSMT on instances from SMTLIB-LRA

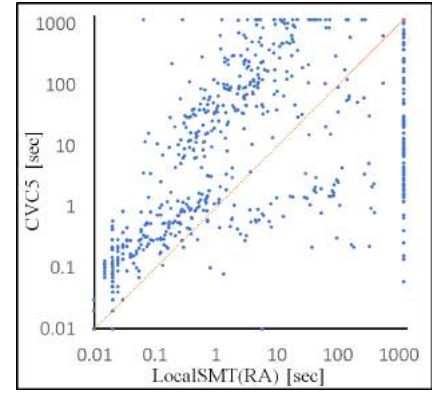
	sc	uart	sal	TM	tta	miplib	Total
#inst	108	36	11	24	24	22	225
WalkSMT UBCSAT	78	35	10	14	9	3	149
WalkSMT UBCSAT++	63	14	8	19	10	2	116
LocalSMT(RA)	106	29	8	9	24	3	179

among Boolean variables, similar to previous local search for SMT on Linear Integer Arithmetic [34]. Another possible reason accounting for the poor performance of LocalSMT(RA) on this benchmark set is that our algorithm is not complete in the sense of probabilistically approximately complete (PAC), because the candidate value selection of *interval-based operation* can miss the possible satisfying solution where the values of variables are not considered.

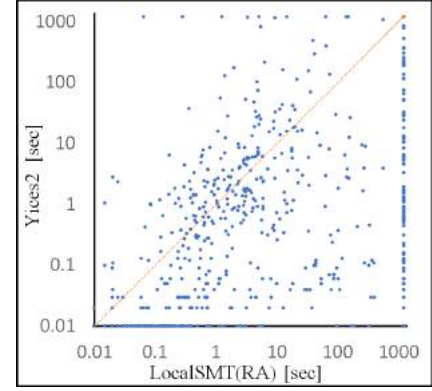
2) *Comparison with Local Search Solvers*: We compare LocalSMT(RA) with a previous local search solver dedicated for SMT(LRA) [33], called WalkSMT. The best two configurations of WalkSMT, namely “WalkSMT UBCSAT” and “WalkSMT UBCSAT++”, are adopted for comparison. Since WalkSMT only supports the earlier version of SMT-LIB standard, which has been deprecated, we perform experiments using the same experiment setup in their paper [33], where the cutoff is set as 600s, and we compare with the result presented in the original paper. The results are shown in Table II, indicating that LocalSMT(RA) can significantly outperform both configurations of WalkSMT, especially on the “sc” type.

C. Results on SMTLIB-MRA Benchmark

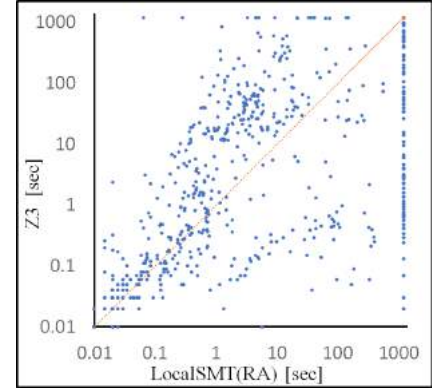
LocalSMT(RA) can solve more multi-linear instances than all competitors on this benchmark set (solving 891 out of 979 instances), which is shown in Table III. Moreover, LocalSMT(RA) can uniquely solve 28 instances in this benchmark set. The time comparison between LocalSMT(RA) and its competitors is shown in Fig 5, confirming that LocalSMT(RA) is more efficient than all competitors in SMTLIB-MRA. As shown in Table IV, LocalSMT(RA) shows better performance with smaller cutoff. Specifically, Lo-



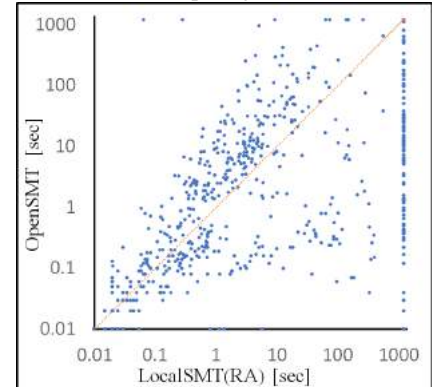
(a) Comparing with cvc5



(b) Comparing with Yices2



(c) Comparing with Z3



(d) Comparing with OpenSMT

Fig. 4: Run time comparison on instances from SMTLIB-LRA

TABLE III: Results on instances from SMTLIB-MRA

	#inst	cvc5	Yices	Z3	SMT-RAT	LocalSMT(RA)
20170501-Heizmann	51	1	0	4	0	17
20180501-Economics	28	28	28	28	28	28
2019-ezsmt	32	31	32	32	21	28
20220314-Uncu	12	12	12	12	12	12
LassoRanker	347	312	124	199	0	297
meti-tarski	423	423	423	423	423	423
UltimateAutomizer	48	34	39	46	18	48
zankl	38	24	25	28	30	38
Total	979	865	683	772	532	891

TABLE IV: Results on instances from SMTLIB-MRA with different cutoff

Cutoff	cvc5	Yices	Z3	SMT-RAT	LocalSMT(RA)
1s	535	567	595	495	733
5s	607	609	674	507	796
10s	649	623	714	508	813

calSMT(RA) can solve 138, 122 and 99 more instances than the best of competitors respectively with the cutoff of 1s, 5s, and 10s.

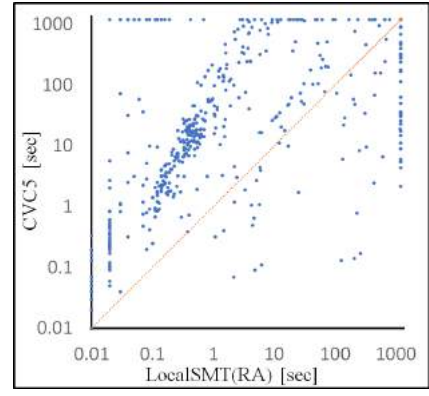
LocalSMT(RA) works particularly well on instances from “zankl” and “UltimateAutomizer” type, which are industrial instances generated in the context of software verification. On these types, LocalSMT(RA) can solve all instances, outperforming all the competitors. Moreover, LocalSMT(RA) can exclusively solve 13 instances from “20170501-Heizmann” type, which implements a constraint-based synthesis of invariants[41].

The explanation for the superiority of LocalSMT(RA) on SMTLIB-MRA are as follows. In contrast to LRA, the theory solver for NRA constraints requires complex calculation, which reduces the performance of these competitors, while LocalSMT(RA) can trivially determine the operations in multi-linear constraints, and thus LocalSMT(RA) can efficiently explore the search space. Moreover, In SMTLIB-MRA benchmarks, 3.4% instances have Boolean variables. In contrast to SMTLIB-LRA, whose counterpart is 54.5%, SMTLIB-MRA has simpler Boolean structures.

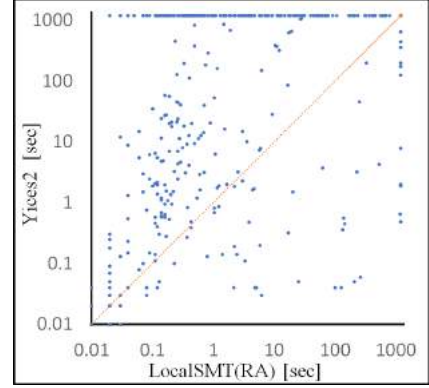
D. Effectiveness of Proposed strategies

To analyze the effectiveness of the strategies in LocalSMT(RA), we modify LocalSMT(RA) to obtain 3 alternative versions as follows.

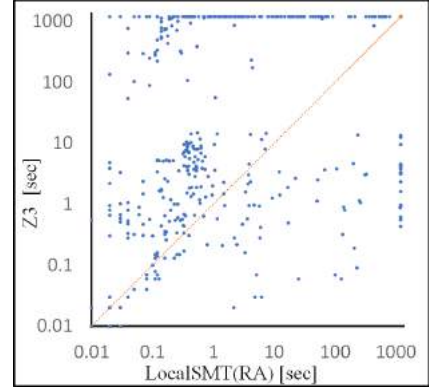
- To analyze the effectiveness of *interval-based* operator, LocalSMT(RA) is modified by replacing the operator with traditional *cm* operator, leading to the version *v_cm*.
- To analyze the effectiveness of the *tie-breaking mechanism*, we modify LocalSMT(RA) by evaluating operation based on *score* without considering the **Selection Rules**, that is to randomly pick one operation with greatest *score*, leading to the version *v_score*.



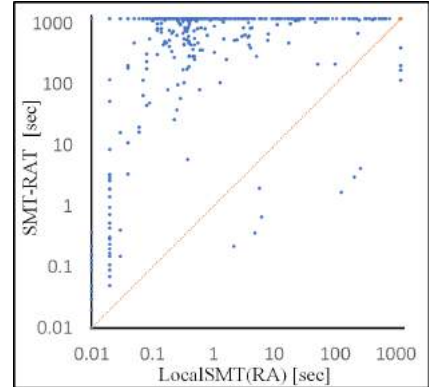
(a) Comparing with cvc5



(b) Comparing with Yices2

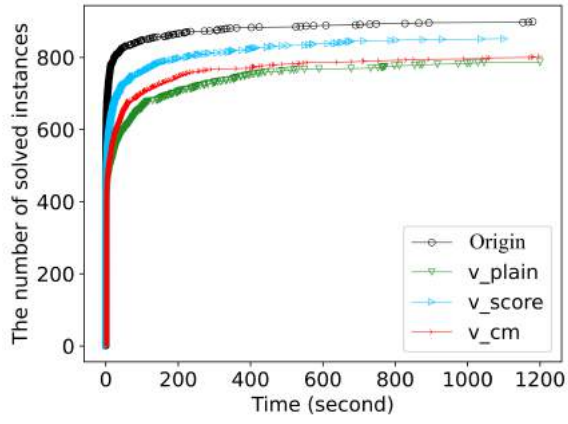


(c) Comparing with Z3

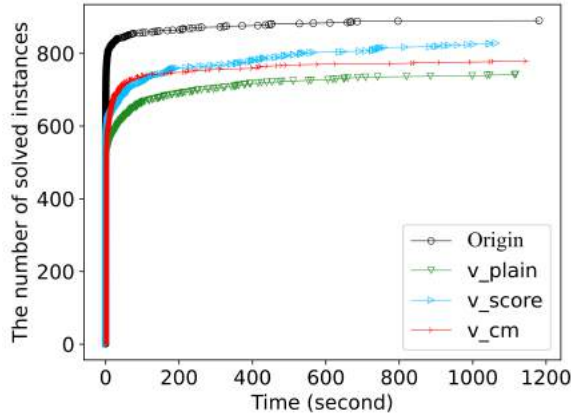


(d) Comparing with SMT-RAT

Fig. 5: Run time comparison on instances from SMTLIB-MRA



(a) Comparison on SMTLIB-LRA



(b) Comparison on SMTLIB-MRA

Fig. 6: Run time distribution comparison

- We also implement a plain version which adopts neither *interval-based* operator nor *tie-breaking mechanism*, denoted as *v_plain*.

The original version of LocalSMT(RA), denoted as *Origin*, is compared with these modified versions on both benchmarks. The runtime distribution of LocalSMT(RA) and its modified versions is presented in Fig. 6, which confirms the effectiveness of our proposed strategies.

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose the first local search algorithm for SMT on Real Arithmetic based on the following novel ideas. First, the *interval-based* operation is proposed to enrich the traditional critical move operator, by considering the interval information. Moreover, a *tie-breaking* mechanism is proposed to distinguish operations with same best *score*. Experiments on SMT-LIB show that our solver is competitive and complementary with state-of-the-art SMT solvers, especially for those multi-linear instances.

The future direction is to extend LocalSMT(RA) to support all SMT(NRA) instances and deeply combine our local search algorithm with the DPLL(T) SMT solver, resulting in a hybrid solver that can make the most of respective advantages.

Moreover, we will enrich the sampled candidate values of *interval-based* operation by considering more random values with small denominator.

VIII. ACKNOWLEDGEMENTS

This work is supported by NSFC Grant 62122078.

REFERENCES

- [1] S. Lahiri and S. Qadeer, "Back to the future: revisiting precise program verification using smt solvers," *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 171–182, 2008.
- [2] N. S. Björner, K. L. McMillan, and A. Rybalchenko, "Program verification as satisfiability modulo theories," *SMT@ IJCAR*, vol. 20, pp. 3–11, 2012.
- [3] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.
- [4] J. Peleska, E. Vorobev, and F. Lapschies, "Automated test case generation with smt-solving and abstract interpretation," in *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*. Springer, 2011, pp. 298–312.
- [5] R. Sebastiani, "Lazy satisfiability modulo theories," *JSAT*, vol. 3, no. 3-4, pp. 141–224, 2007.
- [6] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking*. Springer, 2018, pp. 305–343.
- [7] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t)," *Journal of the ACM*, vol. 53, no. 6, pp. 937–977, 2006.
- [8] L. De Moura and N. Björner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 337–340.
- [9] B. Dutertre, "Yices 2.2," in *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 2014, pp. 737–744.
- [10] F. Corzilius, U. Loup, S. Junges, and E. Ábrahám, "Smt-rat: An smt-compliant nonlinear real arithmetic toolbox: (tool presentation)," in *Theory and Applications of Satisfiability Testing–SAT 2012: 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings 15*. Springer, 2012, pp. 442–448.
- [11] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli et al., "cvc5: A versatile and industrial-strength smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022. Proceedings, Part I*. Springer, 2022, pp. 415–442.
- [12] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The opensmt solver," in *TACAS*, vol. 6015. Springer, 2010, pp. 150–153.
- [13] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*. Springer, 2013, pp. 93–107.
- [14] B. Dutertre and L. De Moura, "A fast linear-arithmetic solver for dpll (t)," in *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*. Springer, 2006, pp. 81–94.
- [15] N. Björner, "Linear quantifier elimination as an abstract decision procedure," in *Automated Reasoning: 5th International Joint Conference, IJ-CAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*. Springer, 2010, pp. 316–330.
- [16] G. E. Collins, "Quantifier elimination for real closed fields by cylindrical algebraic decomposition," in *Automata Theory and Formal Languages: 2nd GI Conference Kaiserslautern, May 20–23, 1975*. Springer, 1975, pp. 134–183.

- [17] U. Loup, K. Scheibler, F. Corzilius, E. Ábrahám, and B. Becker, “A symbiosis of interval constraint propagation and cylindrical algebraic decomposition,” in *Automated Deduction—CADE-24: 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings 24*. Springer, 2013, pp. 193–207.
- [18] S. Junges, U. Loup, F. Corzilius, and E. Ábrahám, “On gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers,” in *Algebraic Informatics: 5th International Conference, CAI 2013, Porquerolles, France, September 3-6, 2013. Proceedings 5*. Springer, 2013, pp. 186–198.
- [19] S. Basu, “Algorithms in real algebraic geometry: a survey,” *arXiv preprint arXiv:1409.1534*, 2014.
- [20] F. Corzilius and E. Ábrahám, “Virtual substitution for smt-solving,” in *Fundamentals of Computation Theory: 18th International Symposium, FCT 2011, Oslo, Norway, August 22-25, 2011. Proceedings 18*. Springer, 2011, pp. 360–371.
- [21] V. Weispfenning, “Quantifier elimination for real algebra—the quadratic case and beyond,” *Applicable Algebra in Engineering, Communication and Computing*, vol. 8, pp. 85–101, 1997.
- [22] P. Van Hentenryck, D. McAllester, and D. Kapur, “Solving polynomial systems using a branch and prune approach,” *SIAM Journal on Numerical Analysis*, vol. 34, no. 2, pp. 797–827, 1997.
- [23] S. Gao, S. Kong, and E. M. Clarke, “dreal: An smt solver for nonlinear theories over the reals,” in *Automated Deduction—CADE-24: 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings 24*. Springer, 2013, pp. 208–214.
- [24] S. Schupp, E. Ábrahám, P. Rossmanith, and D.-I. U. Loup, “Interval constraint propagation in smt compliant decision procedures,” *Master’s thesis, RWTH Aachen*, 2013.
- [25] D. Jovanovic, C. Barrett, and L. De Moura, “The design and implementation of the model constructing satisfiability calculus,” in *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 173–180.
- [26] D. Jovanović and L. De Moura, “Solving non-linear arithmetic,” *ACM Communications in Computer Algebra*, vol. 46, no. 3/4, pp. 104–105, 2013.
- [27] H. H. Hoos and T. Stützle, *Stochastic local search: Foundations and applications*, 2004.
- [28] C. M. Li and Y. Li, “Satisfying versus falsifying in local search for satisfiability,” in *Proc. of SAT 2012*, 2012, pp. 477–478.
- [29] A. Balint and U. Schöning, “Choosing probability distributions for stochastic local search and the role of make versus break,” in *Proc. of SAT 2012*, 2012, pp. 16–29.
- [30] S. Cai and K. Su, “Local search for boolean satisfiability with configuration checking and subscore,” *Artificial Intelligence*, vol. 204, pp. 75–98, 2013.
- [31] S. Cai, C. Luo, and K. Su, “CCAnr: A configuration checking based local search solver for non-random satisfiability,” in *Proc. of SAT 2015*, 2015, pp. 1–8.
- [32] A. Biere, “Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2016,” *Proc. of SAT Competition 2016*, pp. 44–45, 2016.
- [33] A. Griggio, Q.-S. Phan, R. Sebastiani, and S. Tomasi, “Stochastic local search for smt: combining theory solvers with walksat,” in *International Symposium on Frontiers of Combining Systems*. Springer, 2011, pp. 163–178.
- [34] S. Cai, B. Li, and X. Zhang, “Local search for smt on linear integer arithmetic,” in *International Conference on Computer Aided Verification*. Springer, 2022, pp. 227–248.
- [35] —, “Local search for satisfiability modulo integer arithmetic theories,” *ACM Transactions on Computational Logic*, 2022.
- [36] A. Niemetz, M. Preiner, and A. Biere, “Propagation based local search for bit-precise reasoning,” *Formal Methods Syst. Des.*, vol. 51, no. 3, pp. 608–636, 2017. [Online]. Available: <https://doi.org/10.1007/s10703-017-0295-6>
- [37] A. Niemetz and M. Preiner, “Ternary propagation-based local search for more bit-precise reasoning,” in *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 214–224. [Online]. Available: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29
- [38] A. Fröhlich, A. Biere, C. M. Wintersteiger, and Y. Hamadi, “Stochastic local search for satisfiability modulo theories,” in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 1136–1143. [Online]. Available: <http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9896>
- [39] J. Thornton, D. N. Pham, S. Bain, and V. F. Jr., “Additive versus multiplicative clause weighting for SAT,” in *Proc. of AAAI 2004*, 2004, pp. 191–196.
- [40] S. D. Prestwich, “Cnf encodings,” *Handbook of satisfiability*, vol. 185, pp. 75–97, 2009.
- [41] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, “Linear invariant generation using non-linear constraint solving,” in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 420–432.

Mariposa: Measuring SMT Instability in Automated Program Verification

Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, Bryan Parno
Carnegie Mellon University, Pittsburgh, PA, USA
{yeet, jaybosamiya, ytakashi, jgli, marijn, parno}@cmu.edu

Abstract—Program verification has been successfully applied to increasingly large and complex systems. Much of this recent success can be attributed to the automation provided by dispatching verification condition queries via SMT solvers. However, multiple teams anecdotally report that this style of automated verification is plagued by *proof instability*, where semantically irrelevant changes to the query can have large effects on the SMT solver’s response.

In this work, we present Mariposa, a tool to detect and quantify instability. To better understand the status quo of instability, we apply Mariposa to a set of 17,043 SMT queries from six existing program verification projects. We discover that SMT solver upgrades often make projects less stable, and that the most recent SMT solver version is unstable on 2.6% of the queries. For individual projects, the unstable ratio can grow to 5.0%. Based on our experimental results, we curate the Mariposa benchmark, which we hope will help measure and incentivize stability improvements in SMT-based program verification.

I. INTRODUCTION

Software verification can statically guarantee a program’s correctness, reliability, and/or security. In recent years, we have seen significant progress scaling software verification up to large, practical programs, both in academia [1–8] and industry [9–11].

Much of this success relies on Satisfiability Modulo Theories (SMT) solvers [12–15]. The developer writes specifications, proofs, and code, which are transformed into a verification condition [16], expressed as a query in the SMT-LIB [13] format. The SMT solver then does the heavy lifting by checking the verification condition, essentially verifying that the code meets its specification. In practice, this process is iterative: when a query fails, the developer adjusts the specifications, proofs, and/or code until the updated query is accepted and the developer moves on to the next code region.

Unfortunately, automated program verification suffers from *proof instability* [11, 17, 18], where seemingly *irrelevant* changes to the verification condition can cause notable variation in SMT solver performance. For example, simply renaming a source-level variable may cause a verified procedure to take orders of magnitude longer to verify, or even to fail to verify at all. In either case, the developer must tediously supply additional proof hints

that attempt to steer the SMT solver back towards a fast and successful verification result.

Instability poses a significant challenge for large-scale, industrial-level program verification. Concretely, in the verification projects we study, we find up to 5% of queries to be unstable with the most recent SMT solver version (Section IV-D). For developers, such instability disrupts their iterative workflow, as it substantially lengthens their code-prove-debug cycle. Moreover, spurious failures may require developers to fix issues that arise in code or proofs they did not write and may not even understand. In a large team of developers, this problem is amplified, as independent and concurrent changes to the codebase potentially create instability that is only visible after changes are merged. In short, instability impedes monotonic progress in developing a verified codebase.

While the program verification community has recognized the issue of instability [11, 17, 18], popular automated verification tools like Dafny [19] and F* [20] only offer heuristic options to identify it [21, 22]. Furthermore, our results show that these heuristics only capture a fraction of the problem (Section IV-D).

In the SMT community, SMT-COMP [23], the annual competition for SMT solvers, does not include any benchmarks for evaluating stability. Possibly as a result, the stability of some program verification projects actually *deteriorates* with solver upgrades (Section IV-D).

We believe there is a need for a systematic study of the instability phenomenon, where concrete data and statistical analysis can inform both the program verification and SMT communities. A robust measurement methodology can help program verification frameworks adapt their query-generation strategy to avoid issuing unstable queries. For SMT solvers, a benchmark for measuring instability would help evaluate strategies for mitigating it, as well as help prevent stability regressions. In this work we fill this need with the following contributions.

- We present a methodology (and a concrete tool named Mariposa¹) to detect and quantify SMT-based proof instability (Section III).

¹Mariposa is Spanish for butterfly. The name is inspired by the butterfly effect, where small changes can have large effects.

- We perform a detailed empirical study analyzing the (in)stability of six projects written in three program verification frameworks across multiple SMT solver versions (Section IV). The study generates over three million SMT queries that consume ~ 578 CPU days.
- We distill our study’s queries into the Mariposa benchmark to facilitate future research on measuring and mitigating instability (Section V).
- Our study quantifies anecdotal reports of SMT instability, showing that it affects a non-trivial number of queries and often grows worse with new solver versions. We also find that multiple mutation methods are needed to uncover unstable proofs.

The SMT queries and results from our experiments, the source code for the Mariposa tool, and the Mariposa benchmark are all publicly available: <https://github.com/secure-foundations/mariposa>.

II. RELATED WORK

To the best of our knowledge, the problem of SMT proof instability was first reported by the developers of Ironclad Apps [17], who noticed instability in certain non-linear integer arithmetic queries. In the later Komodo work [24], instability was described as “the most frustrating recurring problem.” More recently, Galois highlighted the “fragility of proofs” as a challenge in formally verified industry cryptography [11].

Leino and Pit-Claudel studied the problem of SMT instability in the specific context of Dafny quantifier instantiation [18]. They investigated trigger loops as a possible source of instability, improved algorithms for trigger selection, and then used ad hoc instability measures to evaluate the impact of their algorithms.

The SAT Competition [25] and SMT-COMP [23] may perform benchmark scrambling before evaluating the solvers’ performance. Scrambling involves syntactic transformations similar to our query mutations (Section III). However, scrambling is not sufficient (nor intended) to characterize stability. Prior work has examined the impact of scrambling on competition results [26, 27].

Most work on testing SMT solvers focuses on finding unsoundness bugs [28–33]. One exception is Janus [34], which finds incompleteness bugs, where a query unexpectedly returns `unknown`, placing it closer to our work. However, Janus does not offer a metric for instability, nor does it target program verification queries.

III. METHODOLOGY

In this section, we outline our methodology for characterizing proof instability. At a high level, our goal is to answer two main questions for a given query Q and solver S : (1) Is Q stable or unstable under S ? and (2) How stable or unstable is it?

Intuitively, instability means that the performance of S diverges when seemingly irrelevant mutations are applied to Q . Our methodology, detailed below, follows this intuition. First, we characterize the queries of interest, drawn from prior program verification projects (Section III-A). Next, we describe the mutations chosen for our study and the rationales behind the choices (Section III-B). We then propose a scheme to differentiate stable and unstable queries (Section III-C), addressing question (1) above. Finally, we elaborate on metrics used to quantify stability (Section III-F), addressing question (2).

A. Characterizing Program Verification Queries

This study focuses on queries from automated verification projects, where instability is problematic. Here, we describe their general characteristics, which might differ from those in other domains, such as symbolic execution or model checking. We discuss the specific verification projects chosen for our study in Section IV-B.

Relevant Logics. Program verification queries involve a mixture of bit-vector, integer arithmetic, and uninterpreted functions, typically with quantifiers. There is no single SMT-LIB logic (e.g. `QF_UF` or `NIA`) that captures these at the same time, and thus program verification queries commonly use the `ALL` logic.

Expected Query Result. The goal of program verification is to prove that a property holds in all cases. Therefore, the SMT query is formulated as the *negation* of the desired property, such that a successful proof is indicated via an `unsat` result. Intuitively, if the result is `sat`, then the property is violated in at least one case (the satisfying assignment). For this study, the expected result is always `unsat`, which means that the property holds in all cases (i.e., the program verifies).

Expected Response Time. As discussed in Section I, the process of developing verified software is iterative. Given that the developer is blocked while the solver is running, the solver’s run time should be in the responsive range of human interaction. For most of the projects in our study, the solver time limits used during development are under 30 seconds.

B. Mutation Methods

In this study, we focus on mutation methods that yield queries that are both *semantically equivalent* and *syntactically isomorphic*; i.e., the original query Q and its mutated version Q' share the same semantic meaning and syntactic structures. Hence it seems reasonable to expect similar performance from the solver on both queries.

Semantic Equivalence. Q and Q' are semantically equivalent when there is a bijection between the set of proofs for Q and those for Q' . In other words, a proof of Q can be transformed into a proof of Q' , and vice versa.

Syntactic Isomorphism. Q and Q' are syntactically isomorphic if there exists a one-to-one correspondence

between the symbols (e.g., variables) and commands (e.g., assertions). In other words, each symbol or command in Q has a counterpart in Q' , and vice versa.

For our concrete experiments, we are interested in mutations that also correspond to common developer practices. Specifically, we consider the following three mutation methods:

- **Assertion Shuffling.** Reordering of source-level lemmas or code methods is a common practice when developing verified software. Such reordering roughly corresponds to shuffling the order of commands in the generated SMT query. Specifically, SMT queries introduce constraints using the `assert` command. Shuffling the order in which the constraints are declared guarantees syntactic isomorphism. Further, the order within a local context is irrelevant to the query’s semantics.
- **Symbol Renaming.** It is common to rename source-level methods, types, or variables, which roughly corresponds to α -renaming the symbols in the SMT queries. Renaming preserves semantic equivalence and syntactic isomorphism, as long as the symbol names are used consistently.
- **Randomness Reseeding.** SMT solvers optionally take as input a random seed, which is used in some of their non-deterministic choices. Changing the seed has no effect on the query’s semantics but is known to affect the solver’s performance. Historically, some verification tools have attempted to use reseeding to measure instability: Dafny² and F* have options to run the same query multiple times with different random seeds and report the number of failures encountered.

When a mutation method is exhaustively applied to a query Q , it produces a set of mutated queries M_Q , which also includes Q itself. Consider assertion shuffling as an example. If Q contains 100 assertions, then M_Q would have $100! \approx 9 \times 10^{157}$ permutations of Q . We refer to Q as the *original* query and members of M_Q as *mutants*.

C. Detecting Stability

Intuitively, stability is a performance property over M_Q . That is, whether a query-solver pair (Q, S) is stable or not depends on how the mutants perform. To simplify the discussion, we assume for now that a single mutation method, such as assertion shuffling, is used. In Section III-E, we discuss how to aggregate results from multiple mutation methods.

Mutant Success Rate. A natural performance metric is the success rate of solver S over M_Q . More precisely, it is the percentage of queries in M_Q that are proven (i.e., that return the expected `unsat` result).

²Dafny has recently started to perform shuffling and renaming. The option has changed from `randomSeedIterations` to `randomizeVcIterations`.

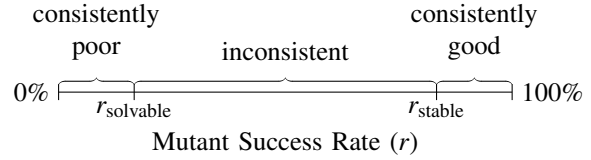


Fig. 1. **Intuition for Our Stability Categories.** (S, Q) is a solver-query pair. r is the mutant success rate. When $r < r_{\text{solvable}}$, Q is not solvable under S . When $r > r_{\text{stable}}$, Q is stable under S . Otherwise, Q is unstable under S .

The success rate, which we denote by r , reflects performance consistency. A low r indicates consistently poor results; a high r indicates consistently good results; and a moderate r indicates inconsistent results, i.e., instability. This intuition is illustrated in Figure 1.

We thus define four stability categories using the success rate r . The scheme includes two additional parameters: r_{solvable} and r_{stable} , which correspond respectively to the lower and upper bounds of the success rate range for unstable queries.

- **unsolvable.** Q is too difficult for solver S ($r < r_{\text{solvable}}$). For example, if S gives up and returns `unknown` for all members of M_Q , we may conclude that S is unable to solve Q or any version of it.
- **unstable.** S cannot consistently find a proof in the presence of mutations to Q ($r_{\text{solvable}} \leq r < r_{\text{stable}}$).
- **stable:** S proves M_Q consistently ($r \geq r_{\text{stable}}$).
- **inconclusive:** statistical tests do not result in enough confidence to draw a conclusion.

Mutant Sampling. In practice, it is often intractable to enumerate all members of M_Q (recall the $100!$ mutants from our shuffling example), so r is generally unknown. Therefore we use statistical tests to estimate r from a sample of mutants. We use \hat{M}_Q and \hat{r} to denote sample mutants and sample success rate, respectively.

Our scheme is based on comparing proportions, so we use the Z-test [35], which is a commonly used statistical test to make inferences about the true proportion of a population based on a set of samples. The test is parameterized by the alpha level, which specifies confidence in its result. We use an alpha level of 0.05 (i.e., 95% confidence), which is a standard choice.

Figure 2 shows our proposed workflow for categorizing the stability of a query-solver pair. For a statistical test (shown as a trapezium shape), if we reject the null hypothesis (H_0), there is enough confidence to conclude that the alternative hypothesis (H_A) is true. For example, in the Instability Test, if we reject H_0 , we are 95% sure that H_A is true, i.e., $r < r_{\text{stable}}$. However, failing to reject H_0 simply means the result is not statistically significant. That is, failing the Instability Test does *not* imply stability. Hence, we test again using the opposite hypothesis. If the test is still not significant, we do not have a conclusive result.

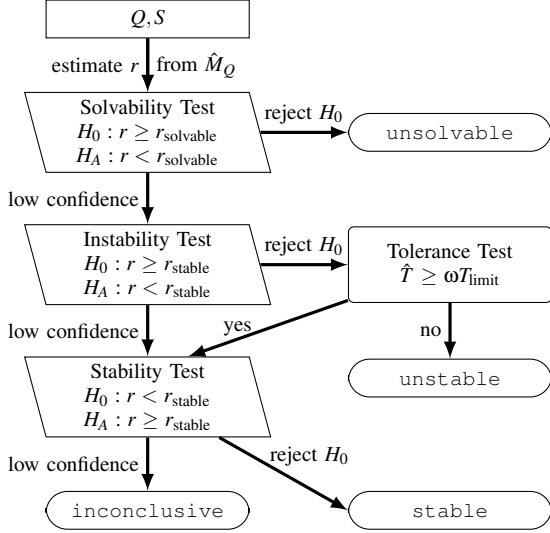


Fig. 2. **Flowchart for Stability Categorization.** We output a stability category based on the performance of the mutants through a series of hypothesis tests. An additional tolerance test is used to filter out queries that are finishing close to the time limit T_{limit} .

A natural goal to make is to pick a sufficiently large sample size such that very few cases are inconclusive. As a sanity check, we expect to conclude *unsolvable* ($r < r_{\text{solvable}}$) if no sample mutant succeeds ($\hat{r} = 0\%$) using an alpha level of 0.05.

We thus calculate the required sample sizes for different values of r_{solvable} . For $r_{\text{solvable}} = 1\%$, we need 269 mutants to be 95% sure that the true success rate is less than 1%. On the other hand, if $r_{\text{solvable}} = 5\%$, 60 mutants are more than enough. Similarly, we expect to conclude *stable* if the sample success rate \hat{r} is 100%. We note this is symmetric to the previous scenario, and thus, to conclude *stable* ($r \geq 95\%$), 60 sample mutants all succeeding ($\hat{r} = 100\%$) is sufficient.

For our experiments, we use $r_{\text{solvable}} = 5\%$, $r_{\text{stable}} = 95\%$, and 60 mutants for each mutation method.

D. Accounting for Time Limits

Since there is no guarantee that a solver will terminate, we impose a time limit T_{limit} on all of our experiments. Solvers may allow the user to bound the solver execution with a resource limit (r_{limit}) instead of a time limit, in an effort to make results more consistent across machines with different computational abilities. However, the resource tracking often counts only some of the resources used (e.g., it may ignore resources spent inside a theory solver). Further, there is no guarantee of consistency across solver versions, let alone across different solvers. Hence, in this work Mariposa uses execution time as a more universal measure.

In the categorization scheme, a mutant that times out is considered a verification failure. However, when the

expected response time of M_Q is close to the time limit, small deviations in the response time can push some mutants into failure. This might give a false impression of instability, while in reality the solver behaves stably given enough time.

To address this issue, we further parameterize the categorization scheme with a tolerance factor ω between 0 and 1. When mixed results are observed in the samples \hat{M}_Q , we estimate the expected response time for M_Q using the mean response time of successful samples, denoted as \hat{T} . If the latter is close to the time limit, i.e., $\hat{T} \geq \omega T_{\text{limit}}$, the failures may be due to an insufficient T_{limit} . In that case, we take a conservative approach and do not label (Q, S) as *unstable*.

Figure 2 shows the tolerance test in the workflow. In our experiments, we use $\omega = 0.8$, and $T_{\text{limit}} = 60\text{s}$. Section IV gives a more detailed analysis of the impact of T_{limit} .

E. Results from Different Mutation Methods

The discussion about the workflow thus far has been based on a single mutation method. In our study, we consider **shuffling**, **renaming**, and **reseeding**, each of which outputs a stability category through our scheme. We use the following procedure to combine the results.

- 1) If the results are unanimously *inconclusive*, output *inconclusive*.
- 2) Remove *inconclusive* results. If the rest are unanimously *X*, output *X*.
- 3) Otherwise output *unstable*.

Note that if the mutation methods disagree on the categories, the procedure returns *unstable*. For example, if **shuffling** outputs *stable*, but **reseeding** outputs *unsolvable*, then the final result is *unstable*. In Section IV we show how mutation methods differ in their ability to detect instability.

F. Quantifying (In)stability

Given a query-solver pair (Q, S) , we use the categorization scheme to answer the question of whether the pair is *stable*. To quantify the instability of an *unstable* pair, we simply use the **Mutant Success Rate** (from Section III-C) as a metric, where higher values are preferable.

To quantify the stability of *stable* queries, we use the **Standard Deviation of Mutant Response Times**. As discussed in Section I, increased response time impedes the iterative development cycles. Therefore, even if a query-solver pair is consistently producing the same verification result, a large variation in response time is still undesirable to the developer. Moreover, such variation is indicative of potential instability: if the time limit is shortened by a small amount, some mutants may fail to finish in time. Therefore, the larger the standard deviation, the less *stable* (Q, S) actually is.

IV. EXPERIMENTS

We have presented a general methodology to detect and quantify SMT-based proof instability. To better understand the status quo of instability, we implement our methodology in the Mariposa tool and use it to perform experiments on existing program verification projects.

In this section, we first describe the experimental setup, which includes an overview of the Mariposa tool (Section IV-A), the verification projects studied (Section IV-B), and the configurations used (Section IV-C). We then present the experimental results, which are organized as a series of research questions (Section IV-D).

A. The Mariposa Tool

We implement our methodology in Mariposa, a tool for SMT stability testing. In its basic use case, Mariposa inputs a query-solver pair (Q, S) , performs mutations on Q , runs S on the mutants, analyzes the performance data, and outputs the stability category and metrics.

For efficient manipulation of queries, the mutations are implemented using Rust (~ 200 LoC). The scripts for running the mutants, recording performance, and analyzing data are implemented in Python ($\sim 2K$ LoC).

Mariposa is extensible, so new mutation methods can be easily added. Mariposa is also configurable, allowing the user to specify parameters such as the number of mutants, the time limit, etc.

B. Projects Under Study

We experiment with prior automated program verification projects. For verification tools, we mainly focus on F^* [20] and Dafny [19], since (1) they have been used to develop complex verified systems; (2) each has an active community of users; (3) they are actively maintained. We then select the following projects and extract all of the SMT verification queries they generate.

- **Komodo_D**. Komodo [24] is a security hypervisor verified and implemented in Dafny, a general-purpose program verifier that often generates undecidable queries.
- **Komodo_S**. Another research team reimplemented parts of Komodo using the Serval framework [2], which requires developers to work within a decidable fragment of first-order logic. For example, recursive functions and loops must be statically bounded. The goal is for developers to write fewer proofs, but one might also conjecture that using a simpler logic would lead to greater query stability.
- **VeriBetrKV_D**. VeriBetrKV [3] is a key-value store based on a B^e tree [36], implemented and verified in Dafny. VeriBetrKV_D uses Dafny’s standard dynamic frames [37] for heap reasoning.
- **VeriBetrKV_L**. In a follow-up study [38], researchers modified the VeriBetrKV code base to use a customized Dafny version that employs linear types for

Project	Source LoC	Query Count
Komodo _D	26K	2,054
Komodo _S	4K	773
VeriBetrKV _D	44K	5,325
VeriBetrKV _L	49K	5,600
DICE _F [*]	25K	1,536
vWasm _F	15K	1,755

TABLE I
BASIC STATISTICS ON PROJECTS USED IN OUR EXPERIMENTS

heap reasoning. They found that using linear types results in faster queries. We explore whether linear types also result in more stable queries.

- **DICE_F^{*}**. DICE is an industry standard measured boot protocol [39]. DICE^{*} [40] is a provably-correct implementation of the protocol in F^* .
- **vWasm_F**. WebAssembly (Wasm) is a portable binary instruction format for web applications [41]. vWasm [42] is a provably-safe sandboxing compiler from Wasm to native code, implemented in F^* .

These project all exhibit non-trivial complexity. The source lines of code (LoC) and query counts for each project are summarized in Table I.

C. Experiment Configurations

We run the experiments on machines with an Intel Core i9-9900K (max 5.00 GHz) CPU, 128 GB of RAM, and the Ubuntu 20.04.3 LTS operating system. Recapitulating earlier parameter settings, we set $T_{\text{limit}} = 60s$; 60 samples per mutation method; an alpha level of 0.05; $\omega = 0.8$; $r_{\text{solvable}} = 5\%$; and $r_{\text{stable}} = 95\%$.

For our experiments, we focus on the Z3 SMT solver [14], which all of our experiment projects were developed with, except for Komodo_S, which used both Z3 and CVC4 [43]³. We are interested in both the current and historical status of SMT stability. Therefore, in addition to the latest Z3 solver (version 4.12.1, as of this writing), we include seven legacy versions of Z3, with the earliest released in 2015. In particular, for each project we include its *artifact solver*, which is the version used in the project’s official artifact.

D. Experimental Results

We organize our experimental results around a series of research questions (RQs). Where necessary for space, we present the results from a subset of projects here and defer the rest to a technical report [44].

³We had initially planned to run our experiments with cvc5 [15] too. However, our preliminary experiments showed the projects are overfitted to Z3. Without intervention, cvc5 cannot solve any of the Dafny or F^* queries, since it cannot even parse the SMT queries these program verification tools produce, due to their use of various bits of Z3-specific syntax and features. After we converted the queries to a format cvc5 understands, it could only solve $\sim 14\%$ of the queries in Komodo_D. We consulted with the cvc5 developers for option tuning and tried cvc5’s automated configuration script for SMT-COMP, but it did not significantly improve the number of queries solved.

RQ1: Do Solver Upgrades Improve Stability?

For each query-solver pair (Q, S) , we run Mariposa, which outputs a stability category. In Figure 3, each stacked bar shows the proportions of categories in a project-solver pair. In all project-solver pairs, the majority of queries are stable. However, a non-trivial amount of instability persists as well.

We observe different trends in each project as newer solver versions are used. The `unstable` proportion of `vWasmF` and `KomodoS` remain consistently small across the tested solver versions. On the other hand, we observe signs of projects that “overfit” to their artifact solver, in that they become less stable with solver upgrades.

Specifically, all of the Dafny-based projects in our study show more instability in newer Z3 versions, with a noticeable gap between Z3 4.8.5 and Z3 4.8.8. The difference in the stability performance is perhaps expected, as these projects were all developed using (now) outdated Z3 solver versions. As of the time of writing, `F*` continues to use Z3 4.8.5, which is approximately four years old, while Dafny only transitioned away from that version earlier this year.

Commit Bisection. We perform further experiments to narrow down the Z3 git commits that may have caused the increase in instability. In the six experiment projects, 285 queries are `stable` under Z3 4.8.5 but `unstable` under Z3 4.8.8. For each query in this set, we run `git bisect` (which calls Mariposa) to find the commit to blame, i.e., where the query first becomes `unstable`.

Table II shows the the bisection results for the 285 queries. Note `git bisect` might not be able to find a unique commit to blame. For example, when the binary search narrows the problem down to a region where commits do not compile, all commits in that region are potentially to blame. We indicate such cases as N/A in the table.

There are a total of 1,453 commits between the two versions, among which we identify two commits that have the most impact. Out of the 285 queries, 115 (40%) are blamed on commit 5177cc4. Another 77 (27%) of the queries are blamed on 1e770af. The remaining queries are dispersed across the other commits.

These two most significant commits are small and localized: 5177cc4 has 2 changed files with 8 additions and 2 deletions; 1e770af has only 1 changed file with 18 additions and 3 deletions. Both commits are related to the order of flattened disjunctions. 1e770af, the earlier of the two, sorts the disjunctions, while 5177cc4 adds a new term ordering for ASTs, which it uses to replace the previous sorting order of disjunctions.⁴

⁴We contacted the Z3 developers after our paper submission. Coincidentally, they were investigating regressions in `F*` query success, and they identified the same two commits as having the most significant impact. Their fix is now merged into the Z3 master branch.

hash	blames	commit message
5177cc4	115	change lt
1e770af	77	local sort
db87f2a	16	separate rewriter...
ff6b330	12	remove incorrect ...
7f073a0	7	fix #2452 fix #...
8b23a17	3	move flatten func...
c70e9af	3	fix #3734
dd452e0	3	eq
762f265	3	merge with master
001ddef	3	fix #2749
3774d6d	2	fix #2890
3ef05ce	2	tuning
80994f7	1	redirect to the n...
d23230e	1	fix declaration s...
e5dffe8	1	fix #2365
ad55a1f	1	Update release.ym...
06ee09a	1	Update README.md
38ad66c	1	update hash #257...
9cccfb9	1	Take one on addin...
ba40a57	1	better branching ...
1e92165	1	branch selection ...
bba2cf9	1	fix #3163
2a1f8ac	1	revert normalizin...
N/A	28	
total	285	

TABLE II
COMMIT BISECTION RESULTS

RQ2: Do Projects Differ in Stability?

`KomodoD` and `KomodoS` are two implementations of the Komodo security hypervisor in Dafny and Serva respectively. The `unstable` proportion of both projects is small using their artifact solvers. However, `KomodoD` shows a significant increase in instability using newer versions of Z3, while `KomodoS` remains stable. Note that `KomodoS` implements a subset of the features in `KomodoD`. If we exclude the attestation-related queries from `KomodoD`, which are not present in `KomodoS`, the `unstable` proportion of `KomodoD` is reduced to 4.27% (from 5.01%) using the latest Z3. The proportion is still much higher than `KomodoS`’s (0.52%). The gap may be attributable to other differences in features and proof goals, but it may also indicate that restricting queries to a *decidable* logic (as `KomodoS` does) improves stability.

`VeriBetrKVL` and `VeriBetrKVD` are two implementations of `VeriBetrKV` in Dafny with different approaches to heap reasoning, where the authors of `VeriBetrKVL` report better query times by adopting linear types. However, their result does not appear to generalize to stability performance: `VeriBetrKVL` is only slightly more stable than `VeriBetrKVD` when using their artifact solvers, and both suffer similar stability regressions on later solvers.

We notice that `vWasmF` is remarkably stable: the `unstable` proportion of `vWasmF` is almost negligible across all solver versions. We contacted the authors of `vWasmF`, and they confirmed that they put significant

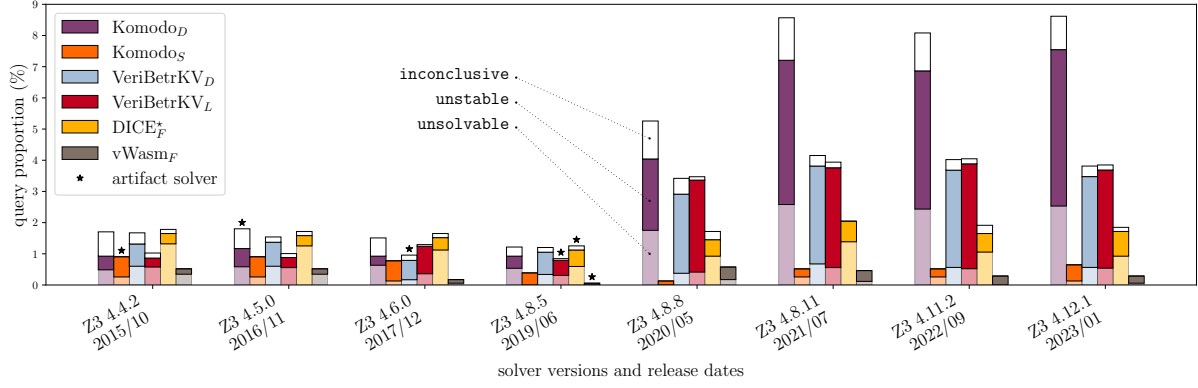


Fig. 3. **Overall Stability Status.** From bottom to top, each stacked bar shows the proportions of unsolvable (lightly shaded), unstable (deeply shaded), and inconclusive (uncolored) queries. The remaining portion of the queries (stacking each bar to 100%), not shown, are stable. The solver version used for each project’s artifact is marked with a star (*).

manual engineering effort into making the queries stable [45]. They attribute the stability of their queries to a disciplined usage of multiple empirically developed techniques. Globally, they disable the non-linear arithmetic solver (anecdotally prone to instability), reduce F^* ’s fuel/ifuel settings (which control unrolling of recursive functions and inductive data types), and minimize the use of ambient lemmas (that tend to bloat solver context). They also minimize the use of (user-introduced, F^* -level) quantifiers, and manually pick good quantifier triggers. Particularly complex proofs necessitated even more drastic measures: using F^* ’s tactic framework to perform manually-controlled normalization of terms before verification condition generation. They note that neither the original un-normalized nor the fully-normalized forms were amenable to stable proofs; only the manually controlled normalization worked.

While few projects can afford this level of manual effort, these results suggest that developers and program verification frameworks can potentially shape their queries to minimize instability.

RQ3: Do Longer Time Limits Mitigate Instability?

As we discussed in Section III-D, the choice of time limit T_{limit} could impact our experimental results. Indeed, one might expect that unstable queries will eventually turn into stable ones given large enough time limits. To test this hypothesis, we extended the experiments using the most recent Z3 (version 4.12.1) with a time limit of 150s ($2.5 \times 60s$).

In Figure 4 we report the proportion of unsolvable and unstable queries for each T_{limit} in KomodoD and VeriBetrKV_D. We observe that the unsolvable proportion drops as T_{limit} increases. This is expected, as a query might only become solvable with a longer time.

However, the unstable proportion stays remarkably consistent after initial fluctuations. That is, certain unstable queries *remain* unstable, even with a

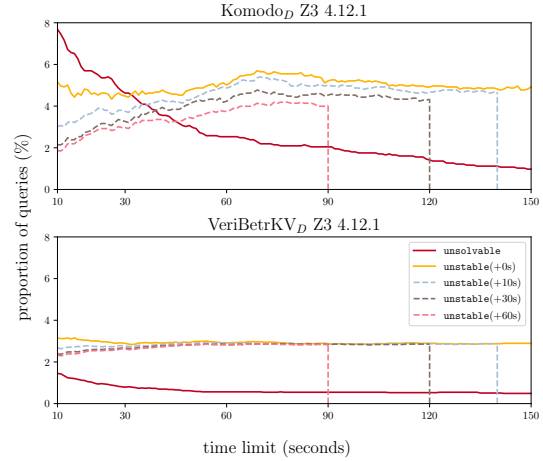


Fig. 4. **Comparing Time Limit Choices.** The proportion of unstable queries stays around the same level after some fluctuations.

longer time limit. To analyze this further, we report the intersection of unstable queries at T_{limit} and $T_{\text{limit}} + \text{step}$, for steps of 10, 30 and 60 seconds. One can interpret a $T_{\text{limit}} + \text{step}$ curve as follows: if some queries are unstable at T_{limit} , it reports how many of them will remain unstable at $T_{\text{limit}} + \text{step}$.

We observe that for a step of 10s, the difference is small. This means that most unstable queries remain unstable if given 10 more seconds, which is expected. For a step size of 60s, the difference is larger but still not significant. In VeriBetrKV_D, it has almost no impact beyond 30s. Therefore, while a longer time limit could help mitigate instability, it is not a silver bullet.

RQ4: Do Results from Mutation Methods Overlap?

We covered multiple mutation methods in our study. A natural question is whether these methods are equally effective in detecting instability.

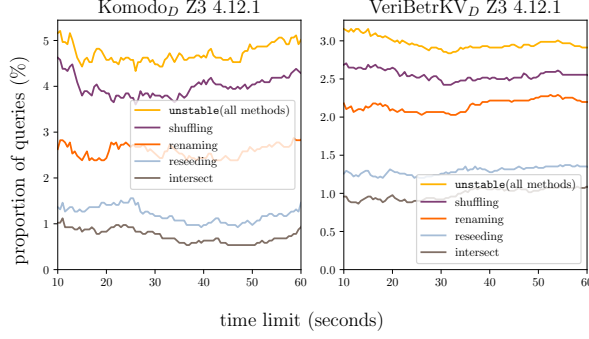


Fig. 5. **Comparing Mutation Methods.** Mutation methods differ in their ability to detect instability. Z3 4.12.1 has the most instability found through **shuffling**. The intersection of methods is also shown as a reference. Note that each sub-graph uses a different y-axis scale.

In Figure 5, we show the unstable proportions identified using each mutation method, along with the overall **unstable** proportion. Recall that the latter is a superset of the individual mutations, as discussed in Section III-E. Since the choice of T_{limit} may impact the categorization, we present results for different T_{limit} as well.

Our results indicate that the effectiveness of mutation methods differ. For example, in **KomodoD** and **VeriBetrKV_D**, the **unstable** proportion is the highest for **shuffling**, followed by **renaming**, then **reseeding**, regardless of T_{limit} . In fact, of the **unstable** queries in **KomodoD** at 60s, 36.9% are uniquely identified by **shuffling**, 6.8% by **renaming**, and 3.9% by **reseeding**.

RQ5: How Stable are Stable Queries?

The **Standard Deviation of Mutant Response Times** is a metric we introduced in Section III-F, where a large value indicates less actual stability, even if mutants consistently succeed. Figure 6 shows the distribution of standard deviation from **stable** queries, which are mostly less than 1s, but there are exceptions exceeding 10s, which is significant given the 60s limit.

RQ6: Is the Original Query Special?

In our methodology, the original query is treated as a member of the mutant set. It might be reasonable to ask how does the original query differ from its mutants in terms of performance.

In Figure 7, we show the verification time of the original query and the median of its mutants, using the data from our extended time limit experiment. In **KomodoD**, which has the highest **unstable** proportion among the six projects, the run time of the original and its mutants are generally within $\pm 50\%$ of each other. In **vWasm_F**, where the **unstable** proportion is the lowest, the two have nearly identical performance.

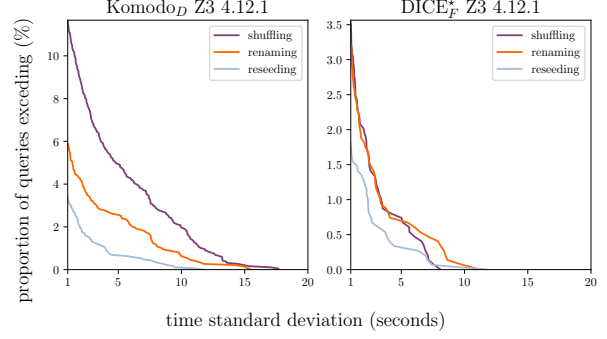


Fig. 6. **Degree of Stability in Stable Queries.** The proportions are taken over **stable** queries only, some of which still exhibit large standard deviations in time among mutants. Mutation methods also differ in their impact. Each sub-graph uses a different y-axis scale.

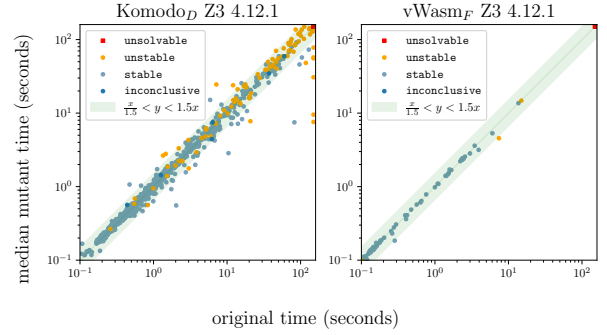


Fig. 7. **Comparing Original and Mutant Queries.** The original query has a run time similar to the median of the mutants. In **KomodoD**, a few cases have the original query time out while some mutants succeed.

V. THE MARIPOSA BENCHMARK

Our experiments over a total of 17,043 original queries generate more than 3 million mutants and take more than 578 CPU days to evaluate. To facilitate future research, we distill the experiment queries into the Mariposa benchmark set. We hope this first version of Mariposa will incentivize improvement and prevent regressions in SMT solver stability for program verification workloads.

The Mariposa benchmark includes both **unstable** and **stable** queries for the projects we experimented with, as shown in Table III. Each is further divided into a **core** and an **extension**, where the **core** contains fewer but more representative queries.

The **unstable** core set contains the queries from each project that are categorized as **unstable** in both the artifact solver and the latest solver. These queries have been consistently unstable, which might be indicative of a long-term problem. The **extension** set contains all the additional **unstable** queries in the latest Z3 version.

Project	Original	Unstable		Stable	
		core	ext.	core	ext.
Komodo _D	2,054	8	95	30	97
Komodo _S	773	2	2	30	9
VeriBetrKV _L	5,600	22	153	30	102
VeriBetrKV _D	5,325	25	130	30	81
DICE _F [*]	1,536	3	9	30	13
vWasm _F	1,755	0	4	30	0
Total	17,043	60	393	180	302

TABLE III
THE MARIPOSA BENCHMARK

The stable core set contains 30 randomly selected stable queries from each project, with mutant time standard deviation less than one second. This set is meant to prevent stability regression, since each member has a consistent result and run time. The extension set contains all the stable queries that have a standard deviation of more than six seconds. Given that the time limit is 60 seconds, such large standard deviations may indicate potential instability, as discussed in Section III-F.

VI. LIMITATIONS

Our experiments draw from six verification projects, which we cannot claim are fully representative of all the SMT-based program verification projects. Nevertheless, we believe our experiments offer valuable insights and serve as a starting point for future work.

Our experiments are performed only with Z3. As explained earlier, popular automated verification languages such as Dafny and F^{*} emit queries that are overfitted to Z3. Hence, our results may not extend to other solvers, such as cvc5.

Our mutation methods are not exhaustive. This study explores a few common mutations, but there are many other mutation methods that might be of interest. For example, mixing mutations may expose more instability, e.g., performing **shuffling** and **renaming** at the same time. We leave the exploration of additional mutation methods to future work.

Our results are dependent on our choice of configuration parameters, e.g., the time limit, the alpha level, etc. In our experiments and analysis, we have tried to analyze the impact of these choices (e.g., via our additional experiments with extended time limits). However we cannot guarantee that our results are not sensitive to our particular choice of configurations.

Our results likely under represent actual instability in the development process. We note that the projects we studied are the cleaned up final versions of the code. During development, although developers do not typically test for instability, they usually try to fix the most obnoxiously unstable proofs.

VII. CONCLUSION

In this work we have studied the phenomenon of SMT instability, specifically in program verification projects, where changes are expected, responsiveness is preferred, and stability is critical. We have proposed a new methodology for detecting and measuring instability, which can inform program verification tools of instability in generated queries. We have also constructed a new benchmark suite, which can be used by SMT solvers to evaluate and optimize for stability. We have applied our methodology to evaluate a number of existing verification projects on various solver versions. Our results show that:

- 1) Stability is the common case, but instability exists non-trivially: 2.6% of the queries in our experiment are unstable with the most recent Z3. In specific projects, this ratio can be as high as 5.0%.
- 2) Stability may deteriorate with solver upgrades: three out of six projects in our experiment show notably worse stability on newer solver versions.
- 3) Mutation methods differ in their effectiveness in detecting instability. Specifically, currently employed detection methods based on random seeds only capture a fraction of the problem.
- 4) Mutants of a given query can exhibit large run-time variance, even if consistent in verification results.
- 5) Source-level program changes may reduce instability, but this currently requires extensive manual engineering. For example, limiting the use of quantifiers, non-linear arithmetic, or undecidable theories may help.
- 6) Increasing the time limit for queries can improve stability, but it offers diminishing returns.

ACKNOWLEDGMENTS

Chris Hawblitzel and Doug Woos contributed to an initial exploration of SMT instability in 2016. Mariposa is a complete rewrite and a fresh set of experiments. We thank Andrew Reynolds for his help with cvc5 configuration; Jinjin Tian for her advice on our statistical analysis; Guido Martinez and Nikolaj Bjørner for sharing their work on patching Z3; and Joshua Gancher, Nikhil Swamy, and the anonymous reviewers for their helpful feedback on the paper.





This work was supported in part by the National Science Foundation (NSF) under grants 1901136, 2015445, and 2224279, grants from the Intel Corporation and Rolls-Royce, Amazon Research Awards (Fall 2022 CFP), the Prabhu and Poonam Goel Graduate Fellowship, and the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not reflect the views of these supporters.

REFERENCES

- [1] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, “IronFleet: Proving Practical Distributed Systems Correct,” in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.
- [2] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, “Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval,” in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
- [3] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, “Storage Systems are Distributed Systems (So Verify Them That Way!),” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [4] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin, “EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider,” in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020.
- [5] Y.-F. Fu, J. Liu, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang, “Signed Cryptographic Program Verification with Typed CryptoLine,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019. [Online]. Available: <https://doi.org/10.1145/3319535.3354199>
- [6] N. Swamy, T. Ramananandro, A. Rastogi, I. Spiridonova, H. Ni, D. Malloy, J. Vazquez, M. Tang, O. Cardona, and A. Gupta, “Hardening Attack Surfaces with Formally Proven Binary Format Parsers,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2022. [Online]. Available: <https://www.fstar-lang.org/papers/EverParse3D.pdf>
- [7] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeeown, and N. Foster, “P4v: Practical Verification for Programmable Data Planes,” in *Proceedings of ACM SIGCOMM*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 490–503.
- [8] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachler-Cohen, and M. Vechev, “Verx: Safety Verification of Smart Contracts,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2020.
- [9] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran *et al.*, “Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3,” in *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.
- [10] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran *et al.*, “Continuous Formal Verification of Amazon s2n,” in *Proceedings of the Conference on Computer Aided Verification (CAV)*, 2018.
- [11] M. Dodds, “Formally Verifying Industry Cryptography,” *IEEE Security and Privacy Magazine*, vol. 20, no. 3, 2022.
- [12] C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Springer, 2018.
- [13] C. Barrett, A. Stump, C. Tinelli *et al.*, “The SMTlib Standard: Version 2.0,” in *Proceedings of the Workshop on Satisfiability Modulo Theories*, 2010.
- [14] L. De Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Tools & Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
- [15] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, “cvc5: A Versatile and Industrial-Strength SMT Solver,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2022.
- [16] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” *Communications of the ACM*, vol. 12, no. 10, 1969.
- [17] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, “Ironclad Apps: End-to-End Security via Automated Full-System Verification,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.
- [18] K. R. M. Leino and C. Pit-Claudel, “Trigger Selection Strategies to Stabilize Program Verifiers,” in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, S. Chaudhuri and A. Farzan, Eds., 2016.
- [19] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds., 2010.
- [20] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, “Dependent Types and Multi-Monadic Effects in F*,” in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016.
- [21] “Debugging Unstable Verification,”

- <http://dafny.org/dafny/DafnyRef/DafnyRef.html#1365-debugging-unstable-verification>.
- [22] “Repeating Proofs with Quake,” http://www.fstar-lang.org/tutorial/book/under_the_hood/uth_smt.html#repeating-proofs-with-quake.
 - [23] C. Barrett, L. de Moura, and A. Stump, “SMT-COMP: Satisfiability modulo theories competition,” in *Computer Aided Verification*, 2005.
 - [24] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, “Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software,” in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.
 - [25] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, “SAT Competition 2020,” *Artificial Intelligence*, vol. 301, 2021.
 - [26] T. Weber, “Scrambling and Descrambling SMT-LIB Benchmarks,” in *SMT @ IJCAR*, 2016.
 - [27] A. Biere and M. Heule, “The Effect of Scrambling CNFs,” in *Proceedings of Pragmatics of SAT*, vol. 59, 2019.
 - [28] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, “StringFuzz: A Fuzzer for String Solvers,” in *Proceedings of the Conference on Computer Aided Verification (CAV)*, 2018.
 - [29] D. Winterer, C. Zhang, and Z. Su, “On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers,” vol. 4, no. OOPSLA. Association for Computing Machinery, Nov. 2020.
 - [30] J. Park, D. Winterer, C. Zhang, and Z. Su, “Generative Type-Aware Mutation for Testing SMT Solvers,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: <https://doi.org/10.1145/3485529>
 - [31] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, “Skeletal Approximation Enumeration for SMT Solver Testing,” in *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2021.
 - [32] A. Niemetz, M. Preiner, and C. Barrett, “Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers,” in *Computer Aided Verification*, 2022.
 - [33] A. Bugariu and P. Müller, “Automatically Testing String Solvers,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1459–1470.
 - [34] M. Bringolf, D. Winterer, and Z. Su, “Finding and Understanding Incompleteness Bugs in SMT Solvers,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3560435>
 - [35] W. Feller, *An Introduction to Probability Theory and its Applications, Volume 2*. John Wiley & Sons, 1991, vol. 81.
 - [36] G. S. Brodal and R. Fagerberg, “Lower Bounds for External Memory Dictionaries,” in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.
 - [37] I. T. Kassios, “Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions,” in *Proceedings on the International Symposium on Formal Methods (FM)*, 2006.
 - [38] J. Li, A. Lattuada, Y. Zhou, J. Cameron, J. Howell, B. Parno, and C. Hawblitzel, “Linear Types for Large-Scale Systems Verification,” in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, December 2022.
 - [39] A. Marochko, D. Mattoon, P. England, R. Aigner, R. Spiger (CELA), and S. Thom, “Cyber-Resilient Platforms Overview,” Microsoft Research, Tech. Rep. MSR-TR-2017-40, September 2017. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/cyber-resilient-platforms-overview/>
 - [40] Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. V. Thakur, “DICE*: A Formally Verified Implementation of DICE Measured Boot,” in *Proceedings of the USENIX Security Symposium*, Aug. 2021.
 - [41] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the Web up to Speed with WebAssembly,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.
 - [42] J. Bosamiya, W. S. Lim, and B. Parno, “Provably-Safe Multilingual Software Sandboxing using WebAssembly,” in *Proceedings of the USENIX Security Symposium*, August 2022.
 - [43] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2011.
 - [44] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, “Mariposa: Measuring SMT Instability in Automated Program Verification (Technical Report),” Carnegie Mellon University, Tech. Rep., August 2023. [Online]. Available: <https://doi.org/10.1184/R1/23905905>
 - [45] J. Bosamiya, W. S. Lim, and B. Parno, private communication, 2023.

A Procedure for SyGuS Solution Fitting via Matching and Rewrite Rule Discovery

Abdalkhman Mohamed[†], Andrew Reynolds^{*}, Clark Barrett[†], and Cesare Tinelli^{*}

[†]Stanford University, Stanford, CA, USA, ✉ abdalkhman@stanford.edu

^{*}The University of Iowa, Iowa City, IA, USA

Abstract—Syntax-guided synthesis (SyGuS) is a recent software synthesis paradigm in which an automated synthesis tool is asked to synthesize a term that satisfies both a semantic and a syntactic specification. We consider a special case of the SyGuS problem, where a term is already known to satisfy the semantic specification but may not satisfy the syntactic one. The goal is then to find an equivalent term that additionally satisfies the syntactic specification, provided by a context-free grammar. We introduce a novel procedure for solving this problem which leverages pattern matching and automated discovery of rewrite rules. We also provide an implementation of the procedure by modifying the SyGuS solver embedded in the CVC5 SMT solver. Our evaluation shows that our new procedure significantly outperforms the state of the art on a large set of SyGuS problems for standard SMT-LIB theories such as bit-vectors, arithmetic, and strings.

I. INTRODUCTION

Program synthesis is a powerful technique with many potential applications, including program optimization, loop invariance generation, and protocol synthesis [1], [2], [3], [4]. Syntax-guided Synthesis [5] (SyGuS) is a particular paradigm for program synthesis in which the goal is to generate correct functional code from a high-level description of the desired program behavior. This high-level description is typically represented as a set of *semantic constraints*, logical formulas expressing the properties the program should satisfy, and *syntactic constraints*, which dictate the structure and syntax of the program and are commonly encoded as a formal grammar.

Program synthesis is generally undecidable and is considered to be a challenging task, even in restricted settings. Nevertheless, in the past decade, several efficient SyGuS solvers have emerged [6], [7], [8], [9], [10]. These solvers are typically *enumerative*, with a few notable exceptions [6], [10]. In an enumerative approach to syntax-guided synthesis, the solver uses the provided grammar to generate a list of terms that meet the syntactic constraints. These terms are then passed to a backend reasoner, often an SMT solver, to determine whether or not they meet the semantic constraints as well.

We consider a special case of the SyGuS problem in which the semantic specification for the function $f(\vec{x})$ to synthesize is already known to be satisfied by a given solution term $t[\vec{x}]$ with free variables \vec{x} . The problem, which we call the *SyGuS solution fitting problem*, is to synthesize from t and a given grammar G a term t' that is equivalent to t and is generated by G . It can also be seen an instance of the SyGuS problem, where the semantic specification is the formula $\forall \vec{x}. f(\vec{x}) \approx t[\vec{x}]$. Procedures for solving this problem can be understood as

refinement procedures, where additional syntactic restrictions are imposed.

We propose a procedure for the SyGuS solution fitting problem that leverages both matching and rewrite rule synthesis to find a term in the language of G that is equivalent to the input term t . At a high level, the procedure matches t with terms generated by the production rules of the input grammar, thereby generating a set S of smaller terms to synthesize. We then augment S by using rewrite rule synthesis to find new terms that are equivalent to those in S . Finally, we use enumerative SyGuS to find derivations for a subset of S that is sufficient to construct a term equivalent to t .

One possible use case for our procedure is when a user has successfully solved a SyGuS conjecture for a (complete) semantic specification φ and a grammar G , and then needs a solution in the language of a revised grammar G' , which perhaps includes desirable and more stringent syntactic requirements. If the previous solution term t does not fit the updated grammar, our procedure can be used to construct from t an equivalent term that does. A second use case occurs within certain approaches for solving SyGuS problems [10] that focus solely on satisfying the semantic component of specifications. Our procedure can then be used to impose a posteriori syntactic constraints on the solution.

Contributions: We propose a novel approach for the SyGuS solution fitting problem. Our contributions include:

- A new procedure for this problem which combines matching, dynamic rewrite rule discovery, and enumerative SyGuS, and is parametric in the background theory of the semantic constraints.
- An implementation of this approach in the SyGuS solver of CVC5.
- A detailed evaluation of the procedure showing significant performance improvements over other SyGuS solvers on a set of crafted benchmarks, as well as a set of mutated benchmarks from the standard SyGuS library. Notably, the procedure scales well across multiple standard SMT-LIB theories, including bit-vectors, arithmetic, and strings.

We present related approaches and preliminaries in the rest of this section. Our approach and contributions are detailed in Section II. We evaluate our approach against other methods in Section III and conclude by outlining potential future directions in Section IV.

A. Related Work

Typical methods for solving SyGuS problems are enumerative in nature. However, some works have explored divide-and-conquer methods to solve restricted classes of SyGuS problems. For example, the SyGuS solver **STUN** [11] divides the input space of the function f to synthesize into subsets, enumerates expressions that are correct in these subsets, and then combines them using a *unification* operator (e.g., the if-then-else operator). **STUN** is fairly effective in solving general SyGuS problems. However, it requires domain knowledge to identify suitable unification operators and efficient program generation algorithms. Moreover, the desired unification operator may not be available in the provided grammar.

EUSolver [6] adopts a similar approach to **STUN** by using the ite operator and selected predicates to define the target function by cases, reformulating the SyGuS problem as a decision tree learning problem. This approach led to successful results in the SyGuS solver competition in 2016 and 2017 [12], [13]. A major limitation is that **EUSolver** requires point-wise specifications of the semantic constraints (relating an input point to its output, but not the outputs of different inputs) and a grammar that both contains the ite operator and can be decomposed into a term grammar and a predicate grammar.

Other work [10], implemented in the CVC4 SMT solver, addresses a class of synthesis problems called *single-invocation* problems, where the occurrences of the target function in the semantic constraints are all applied to the same input tuple. For these problems, CVC4 first looks for a solution satisfying just the semantic constraints and then tries to find an equivalent term within the grammar using an ad-hoc procedure based on matching against the grammar rules of the syntactic specification ([10], Section 5). This procedure is efficient but it often fails to find any equivalent term.

B. Technical Preliminaries

The goal in Program Synthesis is to generate functions that meet a set of specified constraints. These constraints can be formulated in terms of many-sorted second-order logic, using a set of non-empty sorts $S = \{\sigma_1, \sigma_2, \dots\}$. If the function we aim to synthesize has a rank of $\sigma_1 \cdots \sigma_n \sigma$, then we can express the synthesis problem as:

$$\exists f : \sigma_1 \cdots \sigma_n \sigma, \forall \vec{x} : \sigma_1 \cdots \sigma_n. P[f, \vec{x}]$$

where f is a second-order variable representing the target function, \vec{x} is a tuple of first-order variables representing f 's input, and P is a predicate that encodes the semantic constraints imposed on f within a particular background theory T . In the context of SyGuS, additional syntactic constraints are imposed, specifically that the body of the synthesized function f be in the language of a specified grammar G . We write $t \in \mathcal{L}(G)$ to denote that a term t is in the language $\mathcal{L}(G)$ generated by a grammar G . In that case, we will also say that t is *generated by G* .

In this paper, we consider a specific class of SyGuS problems in which a term $t[\vec{x}]$ over the free variables \vec{x} that satisfies the semantic part of the synthesis problem is also provided.

We can view this problem as the subclass of SyGuS problems in which the semantic specification has the form:

$$\exists f : \sigma_1 \cdots \sigma_n \sigma, \forall \vec{x} : \sigma_1 \cdots \sigma_n. f(\vec{x}) \approx t[\vec{x}].$$

The goal of our procedure, given t and G , is to find a term t' that is generated by G and is equivalent to t in the background theory T , denoted as $t' \approx_T t$.

Our procedure can be used in combination with approaches that are limited only to solving semantic specifications. We mentioned previous work [10] that introduced an efficient synthesis approach for single-invocation problems. This class of problems can be expressed by the conjecture:

$$\exists f : \sigma_1 \cdots \sigma_n \sigma, \forall \vec{x} : \sigma_1 \cdots \sigma_n. P[f(\vec{x}), \vec{x}] \quad (1)$$

where $P[f(\vec{x}), \vec{x}]$ is a first-order formula over the free variables f, \vec{x} . This conjecture is equivalent to the first-order formula:

$$\forall \vec{x} : \sigma_1 \cdots \sigma_n, \exists y : \sigma. P[y, \vec{x}] \quad (2)$$

A witness term $t[\vec{x}]$ for the quantifier in formula (1) can be constructed efficiently in practice from formula (2) by quantifier elimination (QE) if the theory T admits QE or, more generally, from a refutation in T of the negation of (2) [14]. However, finding then an equivalent term t' generated by a specified grammar G remains a challenge.

One approach is to use enumeration techniques developed for solving general SyGuS problems, however, those techniques do not scale well for large terms. Another option [10], is to match the structure of the term t against the rules in G to break down the synthesis problem into smaller, more manageable sub-problems. Unfortunately, this approach has its own limitations, since terms that slightly deviate from the grammar can be generated only by enumeration.

C. Motivating Example

The following SyGuS problem demonstrates some of the shortcomings discussed above. We explain how to address them efficiently using matching and rewriting.

Example 1. Consider the synthesis problem

$$\forall x, y, u : \text{Int}. f(x, y, u) \approx x - \text{ite}(y + u > 0, y + u, 0)$$

over the theory of integers and let t be the right-hand side of the equation. Assume our goal is to find an implementation of f whose body is generated by a grammar G with start symbol A and the following production rules:

$$\begin{aligned} A &\rightarrow 0 \mid 1 \mid x \mid y \mid u \mid 0 - A \mid A + A \mid \text{ite}(B, A, A) \\ B &\rightarrow A \approx A \mid A > A \end{aligned}$$

The minimal solution to this problem, as measured by the length of its shortest derivation in the grammar, is prohibitively large for enumerative approaches, which typically explore the solution space by increasing term size. We observed that such approaches check 5K terms or more before finding a solution in this example, due to the combinatorial explosion in the set of terms generated by G as a function of derivation length.

Alternatively, we can directly try to match t with the right-hand side of production rules of the grammar, treating

non-terminals like A and B above as match variables. This makes it easy to detect, for instance, if t is already in $\mathcal{L}(G)$. Unfortunately, this approach fails immediately in our example since $x - \text{ite}(y + u > 0, y + u, 0)$ does not match any right-hand sides in G 's production rules.¹

We can expand the possible patterns for term matching using rewriting modulo the background theory. Note, for example, that if we could add the rule $A \rightarrow A - A$ to the original grammar G , then matching t against $A - A$ would lead to a solution, as both x and $\text{ite}(y + u > 0, y + u, 0)$ in turn match a rule for A . Now, we cannot add this rule directly as it would change the language of the grammar.² However, we can simulate doing so by considering the production rule $A \rightarrow A + (0 - A)$, derived from $A \rightarrow A + A$ and $A \rightarrow 0 - A$. This is because every term of the form $t_1 + (0 - t_2)$ is equivalent to the term $t_1 - t_2$ in the theory, something that can be easily shown using simple theory-specific rewrite rules. Based on this reasoning, we thus conclude that $x + (0 - \text{ite}(y + u > 0, y + u, 0))$ is a solution for this example, being both equivalent to t and generated by the provided grammar.

Note how this process is driven by matching against the grammar rules. This is much more direct than having to wait for terms to be constructed with enumerative methods. In the above example, the matching is made more flexible by utilizing term rewriting, which effectively provides a controlled form of matching modulo the background theory, loosening the limitations of relying just on the rules in the input grammar. Our experimental evaluation shows that this flexibility boosts the effectiveness of the matching-based approach considerably, as we discuss in Section III.

For convenience, we will call an expression derivable in a grammar G (like the one in Example 1) a *pre-term* (generated by G) if it contains non-terminals. For instance, in Example 1, $0 - (A + A)$ is a pre-term, whereas $0 - (x + y)$ is not.

II. A PROCEDURE FOR SYGUS SOLUTION FITTING

In this section, we describe and discuss our procedure for solution fitting in SyGuS, starting with a high-level overview. We then sketch the invariants that are maintained by the procedure, and briefly discuss its properties. In particular, the procedure is *solution sound*: it returns only terms that are indeed a solution of the given synthesis problem. The procedure is also *relatively terminating*: it is guaranteed to terminate with a (correct) solution if its underlying enumerative approach terminates with a solution.

A. The Term Reconstruction Procedure

The reconstruction procedure, `rcons`, is described in Algorithm 1. It uses two main auxiliary procedures, `match` (Algorithm 2) and `markSolved` (Algorithm 3), to abstract the high-level structure from the finer details. We start by outlining the overall procedure and then elaborate on its key data structures and auxiliary functions. Finally, we illustrate

¹Note that it does not match with $0 - A$ because the terminal symbols x and 0 do not match.

²Observe that $A - A$ is not derivable from A in the original grammar.

Algorithm 1 `rcons`(G, t_0)

Require: t_0 : `typeOf`(N_0) and N_0 is start symbol of G

```

1:  $k_0 \leftarrow \text{newVar}(\mathcal{K}, N_0)$ 
2:  $Obs \leftarrow \{(k_0, t_0 \downarrow)\}$ 
3:  $Pool, CandSols, Sol \leftarrow \emptyset, \emptyset, \emptyset$ 
4:  $Targets \leftarrow \{(N_0, t_0 \downarrow)\}$ 
5: while  $k_0 \notin \text{dom}(Sol)$  do
6:   for non-terminal  $N$  of  $G$  do       $\triangleright$  Enumeration Phase
7:      $s \leftarrow \text{nextEnum}(N)$ 
8:     if  $\text{FV}(s) = \emptyset$  then
9:       if there is  $(k, t) \in Obs$  s.t.  $t \approx_T s$  then
10:         $\text{markSolved}(k, s)$ 
11:       else
12:         $k \leftarrow \text{newVar}(\mathcal{K}, N)$ 
13:         $Obs \leftarrow Obs \cup \{(k, s)\}$ 
14:         $\text{markSolved}(k, s)$ 
15:       end if
16:     else
17:        $Pool \leftarrow Pool \cup \{(N, s)\}$ 
18:        $Targets' \leftarrow \{\text{match}(t, s) \mid (N, t) \in Targets\}$ 
19:     end if
20:   end for
21:   while  $Targets' \neq \emptyset$  do       $\triangleright$  Match Phase
22:      $Targets \leftarrow Targets \cup Targets'$ 
23:      $Targets' \leftarrow$ 
24:        $\{\text{match}(t, s) \mid (N, t) \in Targets', (N, s) \in Pool\}$ 
25:   end while
26: end while
27: return  $Sol(k_0)$ 

```

through an example how the various components are integrated in `rcons`.

We assume two countably infinite sets \mathcal{K} and \mathcal{Z} of (meta-level) variables. The variables in \mathcal{K} are placeholders for terms that require reconstruction within the grammar. Those in \mathcal{Z} are matching variables; they represent holes in the patterns used for matching. Variables in \mathcal{K} and \mathcal{Z} are associated with two mappings, `nonTerminalOf` and `typeOf`, which respectively return a non-terminal in the given grammar G and a type for the variable. We denote by $\text{FV}(t)$ the set of variables occurring in term t . Note that for the purposes of the procedure, only elements of \mathcal{K} and \mathcal{Z} are considered variables. We also assume a *rewriting* procedure that rewrites each term t to a term $t \downarrow$, the *rewritten form* of t . We require that the procedure be *sound*, that is, $t \downarrow \approx_T t$ for the background theory T , but not that it be confluent. This means that $t \downarrow$ and $s \downarrow$ may be distinct for some T -equivalent terms t and s .

The procedure `rcons` updates the following *global* program variables.

- Obs stores a set of pairs of the form (k, t) , where k is a variable from \mathcal{K} and t is a term. We refer to variable k as an *obligation* and say it is *closed* if we succeed in finding a term t' equivalent to t and generated by `nonTerminalOf`(k).
- $Pool$ stores a set of pairs of the form (N, t) , where N

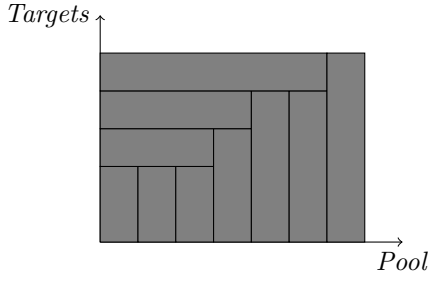


Fig. 1. The relationship between the set of patterns, *Pool*, and the set of terms to be reconstructed, *Targets*, can be depicted by the vertical and horizontal bars, respectively. The vertical bars represent the enumeration phases, while the horizontal bars represent the matching phases.

is a non-terminals of the grammar G and t is a term possibly with free variables from \mathcal{Z} . Those free variables in t are used as holes for matching. If a match from an enumerated term against t succeeds, then this match is used to generate further obligations.

- *CandSols* stores a set of pairs of the form (k, t) , where t is a potential solution to the obligation k . Terms t in this mapping may contain variables from \mathcal{K} , meaning they may be built from terms corresponding to obligations that have yet to be closed.
- *Sol* stores a mapping consisting of pairs $k \mapsto t$, where t is an actual solution to the obligation k for being a term generated by $\text{nonTerminalOf}(k)$. We write $\text{Sol}(s)$ for the result of applying *Sol* as a substitution to the term s .

The procedure *rcons* maintains a number of invariants over these sets, described in detail in Section II-D.

Procedure rcons: The main procedure, *rcons*, takes as input a term t_0 to be reconstructed in a grammar G . It first creates the main obligation to reconstruct, $(k_0, t_0 \downarrow)$, where k_0 is a variable used to refer to $t_0 \downarrow$ in the procedure. The set of terms to reconstruct, *Targets*, is initialized with $t_0 \downarrow$. The procedure is divided into two phases: the *enumeration phase* and the *match phase*.

In the *enumeration phase*, new patterns for each non-terminal in the grammar are enumerated using the iterator *nextEnum*. If a pattern s returned by *nextEnum* is a ground term (i.e., $\text{FV}(s) = \emptyset$), then we check if s closes any obligation k . If it does, we call $\text{markSolved}(k, s)$ to notify other candidate solutions that depend on k . Otherwise, a new obligation (k, s) is created on the fly, and s , for being in $\mathcal{L}(G)$, becomes the solution to this obligation. This is done in case an equivalent term is encountered in the future. If the pattern s is not a ground term, then it is matched against all the terms in *Targets*, and any potential subterms to reconstruct are saved in a new set *Targets'*. Additionally, s is added to the set *Pool* of patterns as it may be useful in matching against new terms to reconstruct in the match phase.

In the *match phase*, the new subterms in *Targets'* are appended to the set *Targets* and matched against the set of patterns that have been enumerated so far. Any subterms returned by this step are stored in *Targets'* and the process

Algorithm 2 $\text{match}(t, s)$

Require: $\text{FV}(t) = \emptyset$, $\text{FV}(s) \subseteq \mathcal{Z}$, $\text{typeOf}(t) = \text{typeOf}(s)$

```

1:  $\text{Targets}' \leftarrow \emptyset$ 
2: if  $\text{patternMatch}(s \downarrow, t) = \sigma$  then
3:    $\tau \leftarrow \emptyset$ 
4:   for  $(z, st) \in \sigma$  do
5:     if there is  $(sk, t') \in \text{Obs}$  s.t.  $t' \approx_T st$  then
6:        $\text{Obs} \leftarrow \text{Obs} \cup \{(sk, st)\}$ 
7:        $\tau \leftarrow \tau \cup \{z \mapsto sk\}$ 
8:     else
9:        $SN \leftarrow \text{nonTerminalOf}(z)$ 
10:       $sk \leftarrow \text{newVar}(\mathcal{K}, SN)$ 
11:       $\tau \leftarrow \tau \cup \{z \mapsto sk\}$ 
12:       $\text{Obs} \leftarrow \text{Obs} \cup \{(sk, st)\}$ 
13:       $\text{Targets}' \leftarrow \text{Targets}' \cup \{(SN, st)\}$ 
14:    end if
15:    let  $k$  be s.t.  $(k, t)$  in  $\text{Obs}$ 
16:     $\text{CandSols} \leftarrow \text{CandSols} \cup \{k \mapsto \tau(s)\}$ 
17:    if  $\text{FV}(\text{Sol}(\tau(s))) = \emptyset$  then
18:       $\text{markSolved}(k, \tau(s))$ 
19:    end if
20:  end for
21: end if
22: return  $\text{Targets}'$ 

```

is repeated until no new subterms are returned, indicating that all patterns have been matched against all terms.

Once the matching phase is complete, the solution status of the main obligation k_0 is evaluated. If it is solved, the reconstruction process is complete. If not, then the current pool of enumerated patterns cannot derive a solution, and the enumeration phase is resumed, to generate new patterns.

The alternating behavior of the enumeration and matching phases is depicted in Figure 1. The vertical bars represent the enumeration phases, during which new patterns are added to *Pool*. It is possible for multiple enumeration phases to occur before a match is successful. Once a match succeeds, terms to be reconstructed are added to *Targets*, and the matching phase begins. The horizontal bars represent the matching phases, during which each new term is matched against all the patterns in *Pool*.

Procedure newVar: This subprocedure can be invoked with either \mathcal{K} or \mathcal{Z} to obtain a fresh variable v for nonterminal N from \mathcal{K} or \mathcal{Z} , respectively. The procedure updates the mapping nonTerminalOf so that $\text{nonTerminalOf}(v) = N$.

Procedure nextEnum: This is a stateful procedure, called on line 7 of the main procedure *rcons*, parametrized by a non-terminal N of the grammar. Let $\mathcal{T}(N)$ be the set of all terms and pre-terms generated by N . Each call to $\text{nextEnum}(N)$ returns the next term in a (fair) enumeration of the set obtained from $\mathcal{T}(N)$ by replacing all occurrences of non-terminals with variables from \mathcal{Z} . For non-terminal A in the grammar of Example 1, this set of terms would include, for instance, $0 - z_1$, $z_1 + z_2$, and $\text{ite}(z_1, z_2, z_3)$ where $z_1, z_2, z_3 \in \mathcal{Z}$. In practice, *nextEnum* is implemented by modifying standard methods for

Algorithm 3 markSolved(k, s)

Require: $FV(s) = \emptyset$ and $s \approx_T t$ for some $(k, t) \in Obs$.

```
1:  $Sol \leftarrow Sol \cup \{(k, s)\}$ 
2:  $CandSols \leftarrow CandSols \cup \{(k, s)\}$ 
3: repeat
4:    $CandSols' \leftarrow CandSols$ 
5:    $CandSols \leftarrow \{(k, Sol(s)) \mid (k, s) \in CandSols'\}$ 
6:   for  $(k, s) \in CandSols$  do
7:     if  $k \notin dom(Sol)$  then
8:        $Sol \leftarrow Sol \cup \{k \mapsto s\}$ 
9:     end if
10:  end for
11: until  $CandSols' = CandSols$ 
```

fast term enumeration [9].

The terms returned by nextEnum serve two purposes in the main procedure: those with no free variables from \mathcal{Z} are used to discharge terms to synthesize, on lines 9-15 (as we will see later in subprocedure markSolved); those with free variables from \mathcal{Z} are used as new patterns for matching on lines 17-18.

Procedure match (Algorithm 2): This function takes in a term t to be reconstructed and a pattern s . The term t is assumed to be in rewritten form (i.e., $t = t\downarrow$). However, this is not necessarily the case for s as it must preserve the syntactic structure enforced by the grammar.

The first step is to rewrite s and attempt to match the structure of t . If the match succeeds, match returns the substitution σ (from variables in \mathcal{Z} to subterms from t) required to unify $s\downarrow$ and t . This substitution represents the subterms that must be synthesized before s can be marked as a solution. For each pair (z, st) in σ , match finds a corresponding sk in Obs or creates a new obligation sk and adds the pair (sk, st) to Obs . match then creates a new substitution τ from z to sk and applies it to s to construct a candidate solution, $\tau(s)$, for k (the variable whose obligation is t). If there are no subterms to synthesize, $\tau(s)$ is a solution to k , and markSolved($k, \tau(s)$) is called.

Example 2. Consider the following obligations and candidate solution sets: $Obs = \{(k_0, x - y), (k_1, x)\}$ and $CandSols = \{(k_1, x)\}$. If we invoke match with $s = z_0 + (0 - z_1)$ and $t = x - y$, then matching will succeed (assuming $s\downarrow = z_0 - z_1$) and return a substitution $\sigma = \{z_0 \mapsto x, z_1 \mapsto y\}$. The check at line 5 will determine that x is already in Obs , so there is no need to create a new obligation for it. However, y is not in Obs , so a new obligation (k_2, y) will be added.

As we process the substitutions in σ , we construct a new substitution $\tau = \{z_0 \mapsto k_1, z_1 \mapsto k_2\}$ and apply it to s to construct a candidate solution for k_0 , namely $k_1 + (0 - k_2)$. The match procedure returns $\{(A, y)\}$ as the set of subterms to reconstruct, resulting in the following updated sets:

$$Obs = \{(k_0, x - y), (k_1, x), (k_2, y)\}$$
$$CandSols = \{(k_1, x), (k_0, k_1 + (0 - k_2))\}$$

Procedure markSolved (Algorithm 3): The set $CandSols$ contains candidate solutions s to obligations k .

Specifically, term s is a solution if it does not contain variables. Whenever that is the case, markSolved is called to update all other potential solutions that depend on k . During this process, complete solutions for other obligations may be discovered. Thus, we repeat this step until no further such terms appear (i.e., a fixed point is reached). Algorithm 3 provides a basic implementation of this procedure.

Example 3. As an illustration, consider the following scenario, which uses the grammar from Example 1. Let $CandSols = \{(k_0, x + k_1), (k_1, 0 - k_2)\}$ and $Sol = \emptyset$. Invoking markSolved(k_2, y) results in:

$$CandSols = \{(k_0, x + k_1), (k_1, 0 - k_2),$$
$$(k_2, y), (k_1, 0 - y), (k_0, x + (0 - y))\}$$
$$Sol = \{k_0 \mapsto x + (0 - y), k_1 \mapsto 0 - y, k_2 \mapsto y\}$$

B. Rewrite Rule Discovery

There are two situations in which our overall procedure utilizes theory reasoning to synthesize new rewrite rules. The first instance occurs in line 9 of Algorithm 1, where we aim to establish if a ground term we derived from the grammar is T -equivalent to one of the target terms in Obs that we want to synthesize. The second instance occurs in line 5 of Algorithm 2 after a match against a pattern is successful and generates a substitution. We again utilize theory reasoning there to determine if there are any obligations that are T -equivalent to the substitution that match asks us to synthesize. This line creates an equivalence class of terms to synthesize, where synthesizing any term in the class amounts to synthesizing all of them.

We use an SMT solver to discover new rewrite rules. Calling the solver every time to find a term's equivalence class, however, is inefficient. Instead, we follow the approach in Nötzli et al. [15] and build a trie whose leaves are equivalence classes and whose nodes are points where the equivalence classes differ. Those points can be obtained by requesting a model when the SMT solver determines that two terms are not T -equivalent.

Note that calling the SMT solver is more costly than just calling the rewriter, but doing so leads to larger equivalence classes, thereby providing a greater selection of terms for reconstruction. In particular, simpler terms may become available, which can greatly accelerate the reconstruction process. Although the approach in Nötzli et al. [15] reduces the number of solver calls required, some calls may still take too long. In practice then, we set a time limit for each call and conservatively assume non-equivalence when a call times out.

C. Revisiting the Motivating Example

We return now to the motivating example from Section I-C to give a detailed run-through of the rcons procedure, which we simplify slightly to keep the presentation manageable. The

objective is to reconstruct the term $t_0 = x - \text{ite}(y > 0, y, 0)$ to an equivalent one in the language of the grammar below.

$$\begin{aligned} A &\rightarrow 0 \mid 1 \mid x \mid y \mid 0 - A \mid A + A \mid \text{ite}(B, A, A) \\ B &\rightarrow A \approx A \mid A > A \end{aligned}$$

`rcons` starts by initializing the data structure values with:

$$Obs = \{(k_0, t_0)\} \quad Pool = CandSols = Sol = \emptyset$$

where k_0 is a fresh variable. During the enumeration phase, `rcons` begins by enumerating the set of terminal symbols of the grammar: 0, 1, x , and y . None of them match with t_0 , but they are assigned as solutions to artificial obligations, in case they turn out to be equivalent to subterms of t_0 .

$$\begin{aligned} Obs &= \{(k_0, t_0), (k_1, 0), (k_2, 1), (k_3, x), (k_4, y)\} \\ CandSols &= \{(k_1, 0), (k_2, 1), (k_3, x), (k_4, y)\} \\ Sol &= \{k_1 \mapsto 0, k_2 \mapsto 1, k_3 \mapsto x, k_4 \mapsto y\} \end{aligned}$$

The procedure then proceeds to the next stage of enumeration, deriving more complex patterns from $\mathcal{T}(A)$ and $\mathcal{T}(B)$. The patterns that can be instantiated are stored in *Pool*. For brevity, we explicitly list only the significant patterns below.

$$Pool = \{(A, 0 - z_0), (A, z_1 + z_2), (A, \text{ite}(z_3, z_4, z_5)), (B, z_6 > z_7), \dots\}$$

`rcons` continues its process of deriving patterns from the grammar in an effort to reconstruct t_0 . At some point, it arrives at the pattern $z_8 + (0 - z_9)$, whose rewritten form is $z_8 - z_9$, matching t_0 . The match creates a substitution $\sigma = \{z_8 \mapsto x, z_9 \mapsto \text{ite}(y > 0, y, 0)\}$, mapping variables from \mathcal{Z} to subterms to reconstruct. Since we do not know how to reconstruct all subterms yet, `match` creates another substitution, $\tau = \{z_8 \mapsto k_3, z_9 \mapsto k_5\}$, which replaces the subterms with (potentially new) corresponding obligations. Substitution τ is then applied to s to create a candidate solution $\tau(z_8 + (0 - z_9)) = k_3 + (0 - k_5)$ for k_0 . Now we have:

$$\begin{aligned} Obs &= \{(k_0, t_0), (k_1, 0), (k_2, 1), (k_3, x), (k_4, y), \dots, (k_5, \text{ite}(y > 0, y, 0))\} \\ Pool &= \{\dots, (A, \text{ite}(z_3, z_4, z_5)), (B, z_6 > z_7), \dots, (A, z_8 + (0 - z_9))\} \\ CandSols &= \{(k_1, 0), (k_2, 1), (k_3, x), (k_4, y), \dots, (k_0, k_3 + (0 - k_5))\} \\ Sol &= \{k_1 \mapsto 0, k_2 \mapsto 1, k_3 \mapsto x, k_4 \mapsto y, \dots\} \end{aligned}$$

At this point, `match` returns $\{(A, \text{ite}(y > 0, y, 0))\}$, the new term to synthesize. Since *Targets'* is not empty, `rcons` enters the match phase for the first time, matching $\text{ite}(y > 0, y, 0)$ against all patterns stored in *Pool*. Matching against $\text{ite}(z_3, z_4, z_5)$ succeeds and generates the substitutions below.

$$\begin{aligned} \sigma &= \{z_3 \mapsto y > 0, z_1 \mapsto y, z_2 \mapsto 0\} \\ \tau &= \{z_3 \mapsto k_6, z_4 \mapsto k_4, z_5 \mapsto k_1\} \end{aligned}$$

`match` then adds $\tau(\text{ite}(z_3, z_4, z_5)) = \text{ite}(k_6, k_4, k_1)$ as a candidate solution for k_5 .

This process is then repeated for the term $y > 0$. This time, `rcons` matches $y > 0$ against $z_6 > z_7$ generating the candidate solution $k_4 > k_1$. Since both k_1 and k_4 are solved obligations, `markSolved`($k_6, k_4 > k_1$) is invoked to construct complete solutions for k_6 and any other obligations that depend on it, such as k_5 and k_0 .

$$\begin{aligned} Obs &= \{(k_0, t_0), (k_1, 0), (k_2, 1), (k_3, x), (k_4, y), \dots, (k_5, \text{ite}(y > 0, y, 0)), (k_6, y > 0)\} \\ CandSols &= \{(k_1, 0), (k_2, 1), (k_3, x), (k_4, y), \dots, (k_0, k_3 + (0 - k_5)), (k_5, \text{ite}(k_6, k_4, k_1)), (k_6, k_4 > k_1), \dots\} \\ Sol &= \{k_0 \mapsto x + (0 - \text{ite}(y > 0, y, 0)), \dots\} \end{aligned}$$

With the solution to k_0 contained in *Sol*, `rcons` exits the main loop and returns $Sol(k_0) = x + (0 - \text{ite}(y > 0, y, 0))$.

D. Properties

Procedure `rcons` maintains several invariants that are essential to its correctness. Those invariants are listed below. We say that a substitution σ *respects grammar* G if for all $v \mapsto t \in \sigma$, term t is generated by `nonTerminalOf`(v) of G .

Invariant 1. *Obs* is a set of pairs of the form (k, t) , where k is an obligation to reconstruct the term t in the grammar, such that:

- 1) $t : \text{typeOf}(k)$;
- 2) $\text{FV}(t) = \emptyset$;
- 3) $s \approx_T t$ for all $(k, s) \in Obs$.

Invariant 2. *Pool* is a set of pairs of the form (N, t) , where t is a pattern shared by some terms generated from the non-terminal N , such that:

- 1) $t : \text{typeOf}(N)$;
- 2) $\text{FV}(t) \subseteq \mathcal{Z}$;
- 3) the ground term $\sigma(t)$ is generated by N for all substitutions σ over \mathcal{Z} that respect grammar G .

Invariant 3. *CandSols* is a set of pairs of the form (k, t) , where t is a potential solution to the obligation k , such that:

- 1) $t : \text{typeOf}(k)$;
- 2) $\text{FV}(t) \subseteq \mathcal{K}$;
- 3) The ground term $\tau(t)$ is generated by `nonTerminalOf`(k) for all substitutions τ over \mathcal{K} that respect grammar G ;
- 4) if t is ground, $s \approx_T t$ for all $(k, s) \in Obs$.

Invariant 4. *Sol* is a mapping with pairs of the form $k \mapsto t$, where t is a solution to the obligation k , such that:

- 1) $t : \text{typeOf}(k)$;
- 2) $\text{FV}(t) = \emptyset$;
- 3) t is generated by `nonTerminalOf`(k);
- 4) $s \approx_T t$ for all $(k, s) \in Obs$.

Correctness: The main correctness property of procedure `rcons` can be summarized as follows.

Proposition 1. *If `rcons`(G, t_0) successfully terminates with a solution term t , then:*

- 1) $t \approx_T t_0$, and
- 2) t is generated by the start symbol N_0 of grammar G .

Thus, the returned term t is semantically equivalent to the input term t_0 and satisfies the syntactic restrictions G as well.

Proof sketch. One can show that each update in `rcons` of the globals sets maintains invariants 1–4. Points (1) and (2) above follow from those invariants. In particular, the returned solution t is $Sol(k_0)$, where $(k_0, t_0 \downarrow) \in Obs$ due to the initial value of Obs . By Invariant 4, $t \approx_T t_0 \downarrow \approx_T t_0$. Furthermore, by Invariant 3, we have that $Sol(k_0)$ is generated by $nonTerminalOf(k_0) = N_0$. \square

Termination: We briefly remark on the termination of `rcons` by examining the enumeration and match phases. We argue that the procedure terminates whenever a term $t \approx_T t_0$ can be produced by enumeration from the grammar, and the underlying theory T has a decision procedure for term equivalence. Even if the matching components of the procedure fail to make progress, the enumerator will still arrive at a solution since enumeration approaches terminate in this setting. So, it is sufficient to show that both phases of the procedure terminate. The enumeration phase terminates because T -equivalence checks terminate. In the match phase, every call to match returns smaller subterms as it only matches against the syntactic structure of the terms to synthesize. As a result, eventually, the algorithm will reach a point where matching fails or the terms can no longer be broken down further. In either case, match returns an empty set and the match phase terminates.

Unrealizability: It is possible that t_0 may be *unrealizable* with respect to G , meaning that there is no term $t \in \mathcal{L}(G)$ such that $t \approx_T t_0$. In its current version, `rcons` lacks a mechanism to detect and report unrealizability; it simply diverges. However, we can draw upon techniques from existing enumeration approaches, which `rcons` builds upon, to detect simple cases. An alternative approach would involve checking the realizability of syntactic constraints during the matching phase, potentially requiring a recursive invocation of SyGuS on sub-target terms. However, for performance reasons and due to `rcons` not being designed for handling this situation, we do not incorporate these mechanisms. For a more comprehensive treatment of unrealizability, we refer readers to the work by Hu et al. [16].

III. EXPERIMENTAL EVALUATION

We have implemented the `rcons` procedure as a module within CVC5’s synthesis solver. Our implementation is comprised of roughly 800 lines of C++ code. We evaluated it on two classes of SyGuS benchmarks: the first was designed to assess the procedure’s performance under a range of grammars and theories, while the second consists of benchmarks from the SyGuS 2019 competition for which the procedure directly applies. The benchmarks for both classes had a common structure consisting of four components:

- 1) Miscellaneous declarations and definitions.
- 2) The function f to synthesize.
- 3) Syntactic constraints on f via a grammar G .

- 4) One semantic constraint of the form $\forall \vec{x}. f(\vec{x}) \approx t[\vec{x}]$, where t is the solution we want to fit into G .

In evaluating the effectiveness of `rcons`, we compared its performance against the winning solvers of previous SyGuS competitions: **EUSolver**³ [6] and CVC4 [17] (and its latest iteration, CVC5 [18]). We used **EUSolver** in its default configuration. We considered purely enumerative configurations of CVC4 and CVC5 (referred to as **cvc4-enum** and **cvc5-enum**, respectively), the procedure from Section 5 of [10] (**cvc4-rcons**), as well as our new `rcons` procedure (**cvc5-rcons**). An additional configuration, **cvc5-match**, uses a modified version of `rcons`, with the enumeration phase disabled, and *Pool* only holding the patterns found in the given grammar. This configuration builds the solution solely through matching, serving as a baseline for our comparisons.

We ran our comparative evaluation on the StarExec platform [19], with a time limit of 30 minutes per instance.

A. Crafted Benchmarks

In the first set of experiments, we randomly generated SyGuS solution refinement benchmarks for three SMT-LIB theories of interest: bit-vectors, integer arithmetic, and strings. We followed the steps below to craft the benchmarks:

- 1) We constructed a reference grammar for each of the three theories, each comprising a majority of the symbols for that specific theory.⁴ We used the reference grammars to craft our set of benchmarks.
- 2) We developed a procedure to generate random terms from the reference grammars, with the derivation length of the terms adhering to a geometric distribution.
- 3) We used the reference grammars to construct random grammars containing the original non-terminals (with modified rules) and new non-terminals.

Using the simple grammar $A \rightarrow 1 \mid x \mid y \mid A + A$ as an example, the procedure used in Step 2 above works as follows. A string containing only the start symbol, A , is first generated. A coin is then flipped to determine whether or not to replace a non-terminal with a randomly selected rule containing non-terminals (only $A + A$ in our case). For example, if the first two coin flips yield heads, then A will be replaced with $A + A$ and either the first or second A (randomly chosen) with $A + A$. This would result in either $(A + A) + A$ or $A + (A + A)$. The process continues until a tails is seen, at which point all remaining non-terminals are replaced with randomly selected terminal symbols (0, x , or y in our case) and the resulting SyGuS term (e.g., $0 + (x + x)$) is returned. To ensure that the derivation length of the terms follows a geometric distribution, we crafted the reference grammars so that each non-terminal symbol has at least one terminal rule and at least one rule containing a non-terminal symbol.

³We updated **EUSolver** with bug fixes and added missing support for theory symbols to conform to the latest revisions of SMT-LIB’s standard and theories.

⁴Some redundant theory symbols were omitted for simplicity. Our reference grammars along with the benchmarks we considered are available at <https://github.com/cvc5/artifact-fmcad23-syguS>.

Solver\Fragment	bv (1K)	nia (1K)	slia (1K)	Total (3K)
cvc4-enum	518	90	248	856
cvc4-rcons	427	113	66	606
cvc5-enum	530	222	305	1057
cvc5-match	17	6	6	29
cvc5-rcons	955	810	604	2369
EUSolver	498	243	405	1146

TABLE I

THE TABLE SHOWS THE NUMBER OF SYNTHESIZED BENCHMARKS SOLVED BY EACH SOLVER CONFIGURATION.

The construction in Step 3 works as follows. First, rules containing non-terminals ($A + A$ for A in the example) are examined. A coin is flipped to decide whether or not to add this rule to the random rules for A in the new grammar. If the rule is added, each non-terminal within the rule is examined and either kept or replaced with a different (potentially new) non-terminal of the same type. We add at least one terminal rule for each non-terminal to ensure that the random grammar is well defined. Those two steps are repeated for each original and new non-terminal. The procedure is forced to terminate by making the probability of adding new non-terminals inversely proportional to the number of existing non-terminals. An example of a random grammar that can be generated from the grammar above is:

$$\begin{array}{ll}
 A \rightarrow x \mid A_1 + A_2 & A_1 \rightarrow 1 \mid A_2 + A_3 \\
 A_2 \rightarrow x \mid y \mid A_1 + A & A_3 \rightarrow x \mid A_1 + A_3
 \end{array}$$

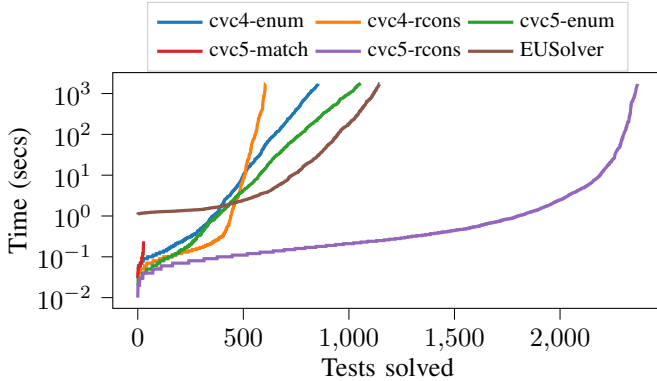


Fig. 2. Cactus plot comparing performance of solvers on crafted benchmarks.

Results: The results presented in Table I demonstrate a marked superiority of our new implementation compared to other solver configurations. Specifically, **cvc5-rcons** was able to successfully solve 846 benchmarks that were not solved by any other configuration. Conversely, only 44 benchmarks were solved by other configurations but not by **cvc5-rcons**. This disparity becomes even more pronounced when comparing **cvc5-rcons** against individual configurations, such as 1,334 vs. 22 uniquely solved benchmarks for **cvc5-enum** and 1,256 vs. 33 for **EUSolver**.

Furthermore, our rcons procedure consistently exhibits faster performance when solving commonly solved benchmarks. This is particularly pronounced in the case of bit-vector benchmarks, where significant speedups of one order of magnitude are observed (16, 17, and 74 times faster than

Solver\Fragment	General (877)
cvc4-enum	438
cvc4-rcons	205
cvc5-enum	672
cvc5-match	82
cvc5-rcons	721
EUSolver	640

TABLE II

THE TABLE SHOWS THE NUMBER OF SYGUS 2019 COMPETITION BENCHMARKS SOLVED BY EACH SOLVER CONFIGURATION.

EUSolver, **cvc4-rcons**, and **cvc5-enum** respectively). While more modest, still significant speedups are observed in the theories of integers (up to 21 times) and strings (up to 13 times), this is largely due to the other solvers timing out on most benchmarks in those fragments. The cactus plot presented in Figure 2 provides a summary of the results.

The number of benchmarks solved by **cvc5-match** is negligible when compared to that of the other configurations. This result shows that relying solely on the patterns provided in the grammar is not effective and generating new patterns through enumeration is critical for the success of rcons.

B. SyGuS Competition Benchmarks

We also evaluated rcons on a subset of the SyGuS benchmarks [20], a set of 877 benchmarks used in previous SyGuS competitions [21], [22], [12], [13], which come from a variety of user applications. The subset contains only SyGuS solution fitting problems, the focus of rcons.

The results are shown in Table II and Figure 3. Again, **cvc5-rcons** outperformed the competition. In particular, it managed to solve 49 and 81 more benchmarks than **cvc5-enum** and **EUSolver**, respectively. Overall, **cvc5-rcons** solved 42 benchmarks that were not solved by any other solver.

The cactus plot from Figure 3 provides further evidence of the robustness of our approach with respect to previous solutions. In particular, the graph shows that a previous approach for matching and enumeration (**cvc4-rcons**) is able to solve many benchmarks quickly, but is eventually eclipsed in performance by **cvc4-enum**. In contrast, **cvc5-rcons** solves an even larger percentage of benchmarks quickly and continues to compete with **cvc5-enum** for the entire 30 minute timeout. We note that **cvc5-enum** performs well in this set of benchmarks because 66% of it consists of circuit synthesis problems, which are not well-suited for unification and matching approaches. Nevertheless, **cvc5-rcons** still surpasses **cvc5-enum** by solving 25 additional problems in this category.

C. Key Insights

Our analysis reveals two factors that have a significant impact on the performance of rcons across both benchmark categories.

The first significant factor concerns the extent of rewrites supported by a particular theory. The presence of an increased number of rewrite rules often allows rcons to generate solutions that markedly differ from those produced by enumerative

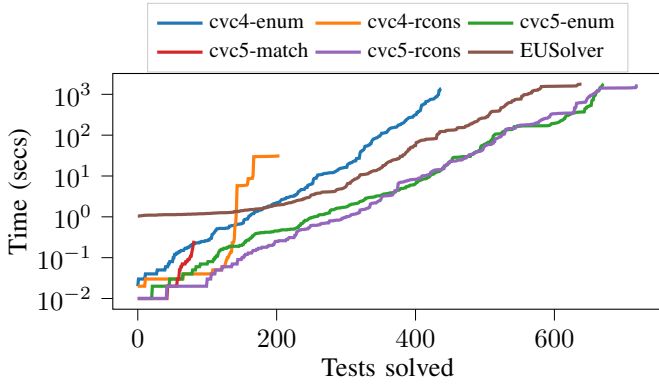


Fig. 3. Cactus plot comparing performance of solvers on SyGuS Competition benchmarks.

Parameter	General (877)
0.5	643
0.9	616
0.99	634
0.999	643
0.9999	721
0.99999	718

TABLE III

THE TABLE SHOWS THE NUMBER OF SYGUS 2019 COMPETITION BENCHMARKS SOLVED BY **CVC5-RCONS** WITH DIFFERENT PARAMETERS.

approaches. For instance, within Table I, we observe that **cvc5-rcons** outperforms alternative solver configurations in the bit-vector benchmarks. This enhanced performance is primarily attributed to the prevalence of rewrite rules within this theory. Conversely, when dealing with circuit synthesis benchmarks, where meaningful boolean rewrite rules are lacking, **cvc5-rcons** exhibits only marginal improvements over **cvc5-enum**, as it frequently converges to solutions identical to those found by **cvc5-enum**.

The second significant factor revolves around the number of patterns employed. The core principle guiding **rcons** is to mitigate the constraints imposed by the provided grammar rules by producing patterns that offer greater flexibility for matching purposes. Nonetheless, an excessive number of patterns can lead to extended durations in the match phase, thereby degrading overall performance. Our approach involves initially generating a substantial number of patterns and subsequently transitioning to enumerating ground terms. The optimal number of patterns depends on the specific grammar and theory. In our implementation, we leverage a geometric distribution to regulate pattern generation. Table III underscores the substantial impact of varying the geometric distribution parameter on the performance of **cvc5-rcons**.

IV. CONCLUSION AND FUTURE WORK

We have presented a novel procedure for the SyGuS solution fitting problem. The procedure enabled the development of an advanced enumerative solver that significantly outperforms other state-of-the-art SyGuS solvers. Our experimental results show that our procedure finds solutions efficiently by deriving complex patterns through enumeration, and using them for matching. The procedure is not restricted to a particular

background theory and can be used in combination with any theory solver that supports rewrites and equivalence checks.

We conjecture that the scalability of our procedure can be leveraged in synthesis problems involving optimization constraints. One class of problems is software optimization, as applied in compilers for embedded SQL queries, linear algebra operations, and circuit synthesis [23], [24], [25]. Current approaches based on synthesis rely on enumerative techniques to generate optimal programs, which does not scale well. A more practical approach could be to first synthesize an initial program, which may not be as efficient as the optimal one, and then gradually optimize it by optimizing its subterms. Although this does not always guarantee optimal performance, it is much more scalable for larger programs. We plan to investigate enhancements to the **rcons** procedure to handle weighted grammars and support this use case.

REFERENCES

- [1] R. Alur, M. M. K. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing finite-state protocols from scenarios and requirements," in *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings* (E. Yahav, ed.), vol. 8855 of *Lecture Notes in Computer Science*, pp. 75–91, Springer, 2014.
- [2] A. Solar-Lezama, "Program sketching," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5-6, pp. 475–495, 2013.
- [3] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006* (J. P. Shen and M. Martonosi, eds.), pp. 404–415, ACM, 2006.
- [4] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5-6, pp. 497–518, 2013.
- [5] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Dependable Software Systems Engineering* (M. Irlbeck, D. A. Peled, and A. Pretschner, eds.), vol. 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pp. 1–25, IOS Press, 2015.
- [6] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I* (A. Legay and T. Margaria, eds.), vol. 10205 of *Lecture Notes in Computer Science*, pp. 319–336, 2017.
- [7] K. Huang, X. Qiu, P. Shen, and Y. Wang, "Reconciling enumerative and deductive program synthesis," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020* (A. F. Donaldson and E. Torlak, eds.), pp. 1159–1174, ACM, 2020.
- [8] S. Padhi, R. Sharma, and T. D. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016* (C. Krutiz and E. D. Berger, eds.), pp. 42–56, ACM, 2016.
- [9] A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, and C. Tinelli, "cvc4sy: Smart and fast term enumeration for syntax-guided synthesis," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II* (I. Dillig and S. Tasiran, eds.), vol. 11562 of *Lecture Notes in Computer Science*, pp. 74–83, Springer, 2019.
- [10] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett, "Counterexample-guided quantifier instantiation for synthesis in SMT," in *Computer Aided Verification - 27th International Conference, CAV*

- 2015, San Francisco, CA, USA, July 18-24, 2015, *Proceedings, Part II* (D. Kroening and C. S. Pasareanu, eds.), vol. 9207 of *Lecture Notes in Computer Science*, pp. 198–216, Springer, 2015.
- [11] R. Alur, P. Cerný, and A. Radhakrishna, “Synthesis through unification,” in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II* (D. Kroening and C. S. Pasareanu, eds.), vol. 9207 of *Lecture Notes in Computer Science*, pp. 163–179, Springer, 2015.
 - [12] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, “SyGuS-Comp 2016: Results and analysis,” in *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016* (R. Piskac and R. Dimitrova, eds.), vol. 229 of *EPTCS*, pp. 178–202, 2016.
 - [13] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, “SyGuS-Comp 2017: Results and analysis,” in *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017* (D. Fisman and S. Jacobs, eds.), vol. 260 of *EPTCS*, pp. 97–115, 2017.
 - [14] A. Reynolds, V. Kuncak, C. Tinelli, C. W. Barrett, and M. Deters, “Refutation-based synthesis in SMT,” *Formal Methods Syst. Des.*, vol. 55, no. 2, pp. 73–102, 2019.
 - [15] A. Nötzli, A. Reynolds, H. Barbosa, A. Niemetz, M. Preiner, C. W. Barrett, and C. Tinelli, “Syntax-guided rewrite rule enumeration for SMT solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings* (M. Janota and I. Lynce, eds.), vol. 11628 of *Lecture Notes in Computer Science*, pp. 279–297, Springer, 2019.
 - [16] Q. Hu, J. Breck, J. Cyphert, L. D’Antoni, and T. W. Reps, “Proving unrealizability for syntax-guided synthesis,” in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (I. Dillig and S. Tasiran, eds.), vol. 11561 of *Lecture Notes in Computer Science*, pp. 335–352, Springer, 2019.
 - [17] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 171–177, Springer, 2011.
 - [18] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “cvc5: A versatile and industrial-strength SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I* (D. Fisman and G. Rosu, eds.), vol. 13243 of *Lecture Notes in Computer Science*, pp. 415–442, Springer, 2022.
 - [19] A. Stump, G. Sutcliffe, and C. Tinelli, “Starexec: A cross-community infrastructure for logic solving,” in *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings* (S. Demri, D. Kapur, and C. Weidenbach, eds.), vol. 8562 of *Lecture Notes in Computer Science*, pp. 367–373, Springer, 2014.
 - [20] S. Padhi, A. Reynolds, A. Udupa, and E. Polgreen, “SyGuS benchmarks.” <https://github.com/SyGuS-Org/benchmarks>, 2019.
 - [21] R. Alur, D. Fisman, S. Padhi, R. Singh, and A. Udupa, “SyGuS-Comp 2018: Results and analysis,” *CoRR*, vol. abs/1904.07146, 2019.
 - [22] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, “Results and analysis of SyGuS-Comp’15,” in *Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015* (P. Cerný, V. Kuncak, and P. Madhusudan, eds.), vol. 202 of *EPTCS*, pp. 3–26, 2015.
 - [23] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, vol. 2 of *The Kluwer International Series in Engineering and Computer Science*. Springer, 1984.
 - [24] S. Chaudhuri, “An overview of query optimization in relational systems,” in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA* (A. O. Mendelzon and J. Paredaens, eds.), pp. 34–43, ACM Press, 1998.
 - [25] M. Willsey, Y. R. Wang, O. Flatt, C. Nandi, P. Panckekha, and Z. Tatlock, “egg: Easy, efficient, and extensible e-graphs,” *CoRR*, vol. abs/2004.03082, 2020.

Partitioning Strategies for Distributed SMT Solving

Amalee Wilson^{*} , Andres Noetzi[†] , Andrew Reynolds[‡] , Byron Cook[§], Cesare Tinelli[‡] , and Clark Barrett^{*} 

^{*}Stanford University, Stanford, USA {amalee, barrett}@cs.stanford.edu

[†]Cubist, Inc., San Diego, USA n@cubist.dev

[‡]The University of Iowa, Iowa City, USA {andrew-reynolds, cesare-tinelli}@uiowa.edu

[§]Amazon Web Services, Seattle, USA byron@amazon.com

Abstract—For many users of Satisfiability Modulo Theories (SMT) solvers, the solver’s performance is the main bottleneck in their application. One promising approach for improving performance is to leverage the increasing availability of parallel and cloud computing. However, despite many efforts, the best parallel approach to date consists of running a portfolio of solvers, meaning that performance is still limited by the best possible sequential performance. In this paper, we revisit divide-and-conquer approaches to parallel SMT, in which a challenging problem is partitioned into several subproblems. We introduce several new partitioning strategies and evaluate their performance, both alone as well as within portfolios, on a large set of difficult SMT benchmarks. We show that hybrid portfolios that include our new strategies can significantly outperform traditional portfolios for parallel SMT.

I. INTRODUCTION

State-of-the-art Satisfiability Modulo Theories (SMT) solvers such as Bitwuzla [1], CVC5 [2], MathSAT [3], OpenSMT [4], Yices2 [5], and Z3 [6] are widely used as reasoning engines in the context of verification [7], model checking [8], security [9], synthesis [10], test case generation [11], scheduling [12], and optimization [13].

For many users of SMT solvers, the solver’s performance is a bottleneck for their application, and so improving solver performance continues to be a top priority for solver developers. Today, most of the aforementioned solvers remain single-threaded, and performance improvements have primarily been achieved through new solving techniques and heuristics. With the increasing availability of CPUs with large numbers of cores, high-performance computing (HPC), and cloud computing, a natural question is whether these resources together with parallel algorithms for SMT could be used to significantly

boost solver performance. Current research in this area can be divided into two main directions: *portfolio solving* and *divide-and-conquer solving*.

Portfolio solving is an approach in which multiple solvers (or different configurations of a solver), attempt to solve the same (or perturbed but equivalent) SMT problem in parallel [14]. It is well-known that SMT solvers are highly sensitive to small perturbations, which can dramatically improve or degrade their performance. While efforts to reduce this sensitivity are an interesting research direction, it is difficult because of the inherent instability of the heuristics used and the uneven nature of the search space. Portfolio solving aims to leverage this sensitivity by sampling from the possible configurations with the hope of finding one that performs well. In fact, this strategy *can* produce significant speed-ups and is currently considered the best way to leverage distributed computing resources to improve performance.

Of course, naive portfolio solving is always limited by the best possible sequential performance, meaning that beyond some point, additional parallel resources do not help. Also, portfolios are empirically ineffective for some classes of benchmarks. For these reasons, it is appealing to pursue the alternative “divide-and-conquer” strategy. In this approach, a problem is partitioned into independent subproblems in such a way that solving the subproblems provides a solution to the original problem. The hope is that because the subproblems have smaller search spaces, solving them in parallel will be faster than solving the original problem. Divide-and-conquer has the potential to dramatically outperform the best sequential performance, but only if an effective partitioning algorithm can be discovered. Such an algorithm has been elusive.

One promising direction is to adapt the *cube-and-conquer* [15] approach that has been successfully applied to the more basic problem of Boolean satisfiability (SAT). In this approach, a partitioning heuristic is used to select a set of n Boolean variables. Typically, a *lookahead* heuristic [16] is used, which chooses variables that, when assigned, most significantly prune the search space. These variables are used to partition the problem into 2^n independent subproblems, which are then solved in parallel. Unfortunately, attempts to adapt this approach for SMT have had limited success. In fact, for some cases, it has been observed that more partitioning is associated with larger, not smaller, runtimes [17].

In this paper, we introduce several new partitioning strate-

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0020347. Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. This work was also supported by an Amazon Research Award and the Stanford Center for Automated Reasoning.

gies which build on—but also go beyond—the basic cube-and-conquer approach. In particular, we look at different ways of combining *sources* for collecting atoms (the SMT version of variables) with ways of using those atoms to create different *partition types*. We evaluate an implementation of these strategies in CVC5 on a diverse set of benchmarks from the SMT-COMP cloud track and from previous work on parallel SMT. We show that a portfolio of partitioning strategies outperforms individual strategies, and we introduce the notion of a *graduated* portfolio which performs particularly well. We also show that *hybrid* portfolios combining partitioning and traditional portfolio strategies perform even better. Finally, we show that using a *multijob* scheduling algorithm for the partitioning portfolio accelerates performance even more. We also demonstrate that these approaches scale, i.e., we continue to get additional speedup with more parallelism.

In summary, our contributions are the following:

- the introduction of several novel partitioning strategies for parallel divide-and-conquer SMT solving;
- the introduction of graduated and hybrid partitioning portfolio strategies; and
- an implementation and evaluation of these strategies on a large set of benchmarks, including the first empirical results demonstrating a parallel solving technique that significantly outperforms a traditional non-partitioning portfolio and continues to do so as the number of partitions is increased.

II. PRELIMINARIES

We assume the standard many-sorted first-order logic setting with the usual notions of signature, term, and interpretation. A *theory* is a pair $\mathcal{T} = (\Sigma, \mathbf{I})$ where Σ is a signature and \mathbf{I} is a class of Σ -interpretations. For convenience, we assume a fixed background theory \mathcal{T} with signature Σ including the Boolean sort `BOOL`. We further assume that all terms are Σ -terms, that entailment (\models) is entailment modulo \mathcal{T} , equivalence is equivalence modulo \mathcal{T} , and that interpretations are \mathcal{T} -interpretations. An *atom* is a term of sort `BOOL` that does not contain any proper sub-terms of sort `BOOL`. A *literal* is either an atom or the negation of an atom. A *cube* is a conjunction of literals. A formula φ is a term of sort `BOOL` and is *satisfiable* (resp., *unsatisfiable*) if it is satisfied by some (resp., no) interpretation in \mathbf{I} . A formula whose negation is unsatisfiable is *valid*.

In this paper, we discuss partitioning algorithms that make use of the CDCL(\mathcal{T}) framework employed by modern SMT solvers. We give a brief overview of the CDCL(\mathcal{T}) framework and introduce the relevant terminology. Then, we describe how partitioning can be used for parallel SMT solving.

CDCL(\mathcal{T})-based SMT solvers solve problems via the cooperation of a SAT solver and one or more theory solvers [18]. The role of the SAT solver in this framework is to build a truth assignment M that satisfies the Boolean abstraction of the problem. Typically, M is built incrementally, and each time the SAT solver assigns a value to an atom, it

calls the theory solvers to check whether M is consistent with \mathcal{T} . Theory solvers return conflict clauses and optionally new lemmas to the SAT solver. A conflict clause is a valid disjunction of literals that is falsified by M , and a lemma is any other heuristically-chosen valid formula. When lemmas and conflict clauses are received by the SAT solver, they are added to the original problem. The process of finding satisfying assignments is repeated until one of two outcomes is achieved: either M is a complete SAT assignment and no conflicts are detected by the theory solvers, meaning the problem is satisfiable; or, an unrecoverable conflict is derived, and the problem is therefore unsatisfiable.

Parallel SMT Solving with Partitioning The satisfiability of a formula ϕ can be determined in parallel by dividing it into n independent *subproblems* ϕ_1, \dots, ϕ_n . Provided the disjunction $\phi_1 \vee \dots \vee \phi_n$ is equisatisfiable with ϕ , if any of the subproblems are satisfiable, then the original problem is satisfiable, and if all of the subproblems are unsatisfiable, then the original problem is unsatisfiable. In this simple scenario, no synchronization is necessary during solving, because the subproblems are independent.

A *partitioning strategy* constructs subproblems ϕ_i of the form $\phi \wedge p_i$. We call each p_i a *partitioning formula* and refer to each subproblem as a *partition*. Though not required for correctness, it is generally desirable for the partitions to be disjoint (i.e., for each $i \neq j$, the formula $\phi \wedge p_i \wedge p_j$ is unsatisfiable) to avoid performing duplicate work. In the cube-and-conquer partitioning strategy, a set of N atoms is selected, and each of the 2^N possible cubes using these atoms is used as a partitioning formula, resulting in 2^N partitions. Scattering [19] is an alternative strategy which differs from cube-and-conquer in that it creates partitioning formulas that are not cubes. Instead, scattering produces a series of N partitioning formulas as follows. The first partitioning formula is some cube C_1 . The second is $\neg(C_1) \wedge C_2$ for some new cube C_2 . The next is $\neg C_1 \wedge \neg C_2 \wedge C_3$ for a new cube C_3 , and so on. The N^{th} partitioning formula is simply $\neg C_1 \wedge \dots \wedge \neg C_{N-1}$. Note that by construction, the partitioning formulas are disjoint. However, there is considerable freedom in how the cubes are chosen.

III. RELATED WORK

As mentioned, much of the existing research literature on parallel and distributed SMT solving focuses on *portfolios*. A portfolio consists of multiple solver instances running in parallel, each of which attempts to solve the same problem. The instance that finishes first produces the result and ends the portfolio run. Each instance differs from the others in some way: a different solver or configuration is used, or the problem has been perturbed in some equisatisfiable way. Some portfolio frameworks enhance this basic strategy by sharing information (e.g., learned lemmas or clauses) among the solver instances running in parallel. Z3 was one of the first solvers to support portfolio solving with sharing [14]. SMTS [20] is a parallel framework that also supports portfolio solving with sharing [21]. In fact, SMTS implements

the parallelization tree formalism [22] [23], which involves recursively combining both partitioning and portfolio solving. Our work mainly explores partitioning. We focus on finding specific effective partitioning strategies, which could then be integrated into a framework such as SMTS. Interestingly, the two approaches (portfolio solving and partitioning) *can* be effectively combined as observed in [20] and as we discuss in Section V. Note that we do not (yet) consider information sharing, as this would add another layer of complexity, and there is enough to understand without it.

There is also previous work on partitioning strategies. *Cube-and-conquer* [15] and other types of splitting [24] have been successfully applied to SAT problems, but there is no consensus on how these approaches should be lifted to the SMT context. OpenSMT2 supports two different lookahead strategies for creating cubes in a cube-and-conquer-like partitioning strategy [17]. One is based on the global number of free atoms, and the other is based on the number of unassigned atoms in the clauses. Previous evaluations using these strategies were mixed, with the strategies performing well on some benchmarks but not on others. OpenSMT2 also supports a *scattering* strategy that was originally developed for solving SAT problems [19]. In their implementation, each cube is obtained by taking atoms from the decisions made up along a particular search branch during a run of OpenSMT2. The number of literals in each cube varies according to a heuristic. Details can be found in [19], [21], [23]. When the scattering partitioning strategy was compared to portfolio solving in [21], they found that portfolio performed better on quantifier-free linear real arithmetic (QF_LRA) benchmarks, especially on unsat problems. We compare all of the OpenSMT2 strategies with our own strategies in our evaluation.

PBolector [25], a parallel SMT solver built on top of the Bolector SMT solver [26], uses a cube-and-conquer style strategy for QF_BV SMT formulas, with the goal of evaluating whether lookahead methods work well in combination with term-rewriting rules and bit-blasting techniques. On quantifier-free bitvector (QF_BV) benchmarks, PBolector saw familiar results: each configuration of their solver performed well on a subset of the benchmarks while performing poorly on others. Because of our focus on partitioning strategies rather than implementations and the similarity of its partitioning strategy (lookahead-based) to that of OpenSMT2, we do not directly compare with PBolector.

Previous work on partitioning has also been limited in terms of which SMT-LIB logics were supported: benchmarks over quantifier-free uninterpreted functions (QF_UF) were used in [23], QF_BV benchmarks in [25], and both QF_UF and QF_LRA benchmarks in [17], [20], [21]. We are the first to implement a general-purpose partitioning strategy that works for all SMT-LIB logics. We comment on which types of problems are well-suited for our partitioning algorithms in Section V.

IV. PARTITIONING

In this section, we introduce a set of partitioning strategies parameterized in four dimensions: atom source, selection heuristic, partition type, and partition timing. Pseudocode for partitioning based on these parameters is given in Algorithm 1. More details on these parameters and their relationship to Algorithm 1 are given in the following subsections, but briefly, the atom source and selection heuristic specify *what* the partitions are made of, the partition type specifies *how* the partitions are made, and the partition timing specifies *when* the partitions are made.

In all cases, partitions are made by invoking an instrumented version of an SMT solver that calls Algorithm 1 periodically during the solving process. We call this solver the *partitioning solver*. By instrumenting an existing solver, our approach takes advantage of well-tested infrastructure for parsing, preprocessing, and reasoning about SMT problems. The first step in Algorithm 1 is to check whether, according to the *partition timing* heuristic (see Section IV-C), it is the right time to make a partition. If not, nothing is done. Otherwise, a set of atoms is collected from the specified source (see Section IV-A). If an insufficient number of atoms is collected, again, nothing is done. Otherwise, *makePartitions* is called. Note that the behavior of *makePartitions* depends on the *partition type* (see Section IV-B). If *ptype* is CUBE, then *makePartitions* will emit all partitioning formulas and return true, and Algorithm 1 will not be called again. If *ptype* is SCATTER, then as long as the number of partitions generated so far is less than $N - 2$, *makePartitions* creates a single partitioning formula and returns false. In this case, the partitioning solver also blocks the part of the search space corresponding to the generated partitioning formula by adding the negation of the cube part of the partitioning formula as a lemma (called a *blocking lemma*). The partitioning solver then continues to work on solving the problem until its next call to Algorithm 1. If *ptype* is SCATTER and the number of partitions generated is $N - 2$, then *makePartitions* creates the last 2 partitions (as described in Sec. IV-B, below) and returns true.

It is possible that the partitioning solver actually solves the problem. If the solver determines that the problem is satisfiable, or if it finds that the problem is unsatisfiable before any partitions have been made, then the problem has been solved, and there is no need to continue or to solve any partitioned formulas. However, if the partitioning solver returns unsatisfiable after having emitted some partitions, then these partitions still need to be solved. This is because of the blocking lemmas that were added which prune the parts of the search space (in the partitioning solver) covered by the emitted partitions.

A. Atom Source and Selection Heuristics

There are two parameters for atom selection: *atomSource* and *atomSelHeur*. The *atomSource* parameter describes where the atoms should come from. We explore three different sources of atoms for our partitioning strategies: the SAT heap, which is a priority queue of Boolean variables in the SAT

Algorithm 1 Partitioning strategy pseudocode.

Input: $N \geq 2$, $cubeSize = \log_2(N)$
Input: $atomSource \in \{\text{HEAP}, \text{DECISION}, \text{CL}\}$
Input: $atomSelHeur \in \{\text{RAND}, \text{SPEC}\}$
Input: $ptype \in \{\text{CUBE}, \text{SCATTER}\}$
Input: $t_1, t_2 \geq 0, tHeur \in \{\text{CHECK}, \text{TIME}\}$
Output: returns true iff done partitioning

- 1: $timeToPart \leftarrow isTimeToPartition(t_1, t_2, tHeur)$
- 2: **if** $timeToPart$ **then**
- 3: $atoms \leftarrow collectAtoms(atomSource, atomSelHeur)$
- 4: **if** $atoms.size() \geq cubeSize$ **then**
- 5: $atoms.resize(cubeSize)$
- 6: **if** $makePartitions(ptype, atoms, N)$ **then**
- 7: **return** true
- 8: **end if**
- 9: **end if**
- 10: **end if**
- 11: **return** false

solver; the decision trail of the SAT solver, which contains the decisions made by the SAT solver along its current search branch; and conflict-or-lemma (CL) atoms, which are atoms contained in the lemmas and conflict clauses sent from the theory solver to the SAT solver. These sources of atoms correspond to HEAP, DECISION, and CL in Algorithm 1, respectively. The $atomSelHeur$ parameter describes how atoms should be selected from the available atoms in the source. Every source supports selecting atoms at random (RAND in Algorithm 1). The other option is to use a heuristic specific to the source (SPEC). We describe these below. To guarantee that partitions can be made quickly, each of the heuristics is lightweight and does not rely on sophisticated or computationally expensive score calculations.

As mentioned above, when $atomSource$ is HEAP, the source of atoms is an internal data structure in the SAT solver. Typically, SAT solvers make decisions based on the *activity* score of each variable. The activity of a variable is determined by how often it appears in conflicts [27], and the variable with the highest score is used when the SAT solver makes decisions. When using HEAP as its source, the SPEC heuristic simply chooses the $cubeSize$ variables with the highest activity scores. The rationale is that highly active variables may be a good choice for helping shape partitions. Note that this heuristic requires no additional computation by the partitioning algorithm because the SAT solver already orders the variables by their activity.

Variables that are good for SAT decisions may not always be ideal for SMT partitions. For example, some high-activity variables may be closely related to other high-activity variables, because they represent theory atoms that contain similar terms. Ideally, however, variables used in partitions should be as *independent* as possible, so that each partition has roughly the same difficulty. The DECISION option attempts to address this weakness. It uses the decision trail as a source of atoms. The decision trail contains all the variables that have been decided

on along a particular branch of the search tree during the solving process. The rationale is that variables in the decision trail are, roughly speaking, more likely to be independent (for example, if two variables entail each other, then deciding on one will always propagate the other, so they cannot both be in the decision trail at the same time). When selecting atoms from the decision trail, the SPEC heuristic chooses the earliest decisions in the trail. As before, this heuristic requires no additional computation by the partitioning algorithm, because the decisions are stored in the trail from least to most recent.

Finally, the CL option uses conflict clauses and lemmas coming from theory solvers as the atom source. Intuitively, atoms that appear in conflict clauses and lemmas are those that are “contributing” in some way to the solution process. While the appearance of an atom in a conflict clause is reflected in its activity score, an appearance in a lemma has no effect on the activity score. This is because when lemmas are generated, they are simply added as additional clauses. The role of a lemma is to help guide the search in a theory-specific way. Thus, we expect that atoms appearing in lemmas may be important. When selecting atoms from conflict clauses and lemmas, the SPEC heuristic selects those atoms that occur most frequently. These atoms are tracked as they are sent from the theory solver to the SAT solver, and a counter is maintained for each atom.

There are two additional issues to consider when selecting atoms: the number of atoms to use for each partition and filtering out unusable atoms. In Algorithm 1, we fix the number of atoms per partition, $cubeSize$, to be $\log_2(N)$ where N is the number of requested partitions. However, as we discuss below, for the SCATTER partition type, this is not necessary and $cubeSize$ could be set as an additional parameter. Regarding filtering, because the construction of partitions is done by appending a partitioning formula to the original formula, anything appearing in the partitioning formula must make sense in this context. In particular, if an atom contains some symbol generated internally by the solver (i.e., not appearing in the original problem), we filter that atom out. This filtering is done during the *collectAtoms* routine.

B. Partition Type

We consider two ways of creating partitioning formulas from the selected atoms. The first, the CUBE partition type, follows the *cube-and-conquer* approach. The second, the SCATTER partition type, uses a scattering strategy.

1) *Cubes*: The cube strategy requires $\log_2(N)$ atoms to be selected during the atom selection phase. Once the required number of atoms has been collected, they are immediately used to create N mutually exclusive cubes. The partitioning solver then terminates. These cubes, C_1 through C_{2^n} , correspond to each possible conjunction of atoms that can be created from the selected atoms. For example, if the two atoms selected are x_1 and x_2 , then the following partitions would be emitted: $C_1 = x_1 \wedge x_2$, $C_2 = \neg x_1 \wedge x_2$, $C_3 = x_1 \wedge \neg x_2$, and $C_4 = \neg x_1 \wedge \neg x_2$.

2) *Scattering*: Scattering is a dynamic strategy for creating partitioning formulas. When the partition type is SCATTER, Algorithm 1 produces only a single partition with each call to *makePartitions*. The partitioning formula constructed at each call takes the current cube and conjoins it with the negation of previous cubes, as described in Section II. After each generated partition, the negation of the partitioning formula is added as a lemma to the partitioning solver, to ensure that it explores a different part of the search space during the rest of its run.

Note that *cubeSize* does not have to be equal to $\log_2(N)$ when using scattering. Indeed, it can even vary from partition to partition. In [23], a particular strategy is suggested that does vary the *cubeSize* across partitions. In this paper, we simply use $\log_2(N)$, for the *cubeSize*.

Note that it takes at least $N - 1$ calls to Algorithm 1 to compute N partitions with the SCATTER partition type. The final partition is emitted immediately after the $N - 1^{st}$ partition, because the final partition is simply the negation of all previously used cubes.

C. Partition Timing

Partition timing specifies *when* partitions should be made. There is a trade-off between collecting sufficient information to create good partitions and avoiding spending unnecessary time on partitioning that could have been used for solving. Algorithm 1 supports two different kinds of partition timing. The first, when *tHeur* is CHECK, simply counts the number of times that Algorithm 1 has been called (in our implementation, this is done during the *check* phase as we explain in Section V below). The second possibility is TIME, which simply measures the amount of time that has passed. In Algorithm 1, two inputs, t_1 and t_2 , are used for timing. The t_1 parameter determines how long to wait (either number of checks or time in seconds) before the partitioning solver creates any partitions. The idea of this parameter is to allow the partitioning solver to make some progress and get into an interesting state before starting partitioning. The t_2 parameter determines how long (again in either checks or seconds) to wait *between* each pair of partitions. Note that for the CUBE partition type, t_2 is irrelevant, because all partitions are created at once.

There is another trade-off between predictability of partitioning time and predictability of partitioning formulas. When counting checks, the partitioning formulas are deterministic (as long as the execution of the SMT solver is deterministic). When using time, however, there can be variation in the current state of the solver at time t from one run to the next. Thus, it may seem like check counting is preferable. The problem is that the number of checks varies greatly from one problem to the next. One SMT formula may trigger thousands of checks in the solver in the first minute of solving, while other SMT formulas have only a handful. Thus, for predictable partitioning time (though less predictable partitions), it can be better to use time instead of check counting to help ensure that

the time to create partitions is relatively stable across many problems. We discuss this further in Section V-A, below.

V. EVALUATION OF PARTITIONING STRATEGIES

We instrumented CVC5 to be a partitioning solver by (i) implementing Algorithm 1 in CVC5; and (ii) having CVC5 call Algorithm 1 after each decision made by its internal SAT solver. Below, we report on several sets of experiments with this instrumented version of CVC5. We ran all experiments reported here on a cluster with 26 nodes running Ubuntu 20.04 LTS, each with 128 GB of RAM, and two Intel Xeon CPU E5-2620 v4 CPUs with 8 cores per CPU. Although both CVC5 and OpenSMT2 are used as partitioning solvers, all partitions, scrambles, and original formulas are solved using CVC5.

In our evaluation, we compare several configurations of our CVC5-based partitioning solver and an OpenSMT2-based partitioning solver on a set of benchmarks drawn from the cloud track of the 2022 edition of SMT-COMP [28], the SMT-LIB QF_LRA benchmarks, and the SMT-LIB QF_UF benchmarks [29]. The SMT-COMP cloud benchmarks were selected by the SMT-COMP organizers to be challenge problems for parallel solving and are thus a good target for this work. The QF_LRA and QF_UF benchmarks have been the subject of previous studies on parallel solving [17], [23], [21].

We exclude benchmarks based on several criteria. First, we exclude any benchmark with quantifiers. The challenges for quantified benchmarks are typically the result of too many possible quantifier instantiations. In contrast, partitioning targets challenges stemming from large Boolean search spaces. Second, we exclude benchmarks that are solved in less than 600 seconds by the sequential version of CVC5. This is simply to focus on problems that are challenging for sequential solvers. Third, if no partitions can be made by the partitioning solver, the benchmark is excluded. This can happen if the SAT solver makes no or almost no decisions and Algorithm 1 is not called enough times to create partitions. We consider such benchmarks poor candidates for solving via partitioning. Finally, benchmarks that are solved during partitioning by any partitioning solver are excluded, again because we consider them easy, and furthermore, our aim is to compare partitioning strategies, not partitioning solvers. One way that a problem can be solved during partitioning if it is trivial and is solved before any calls to *makePartitions* in Algorithm 1. The other way a problem can be solved during partitioning can only happen if the SCATTER partitioning strategy is used *and* the problem is satisfiable. In this case, it is possible that after some number of calls to *makePartitions*, each of which blocks some part of the search space, the partitioning solver stumbles upon a satisfying solution before the next call to *makePartitions*. After these filters are applied, we are left with 214 challenging benchmarks in 5 SMT-LIB logics: QF_LRA (139), QF_IDL (48), QF_LIA (16), QF_UF (7), and QF_RDL (4).

To measure the performance of a particular partitioning strategy on a given benchmark, we first measure the time to partition it. Then, we run each partition on the cluster (with a maximum of 16 jobs per node, i.e., one per core, and 8 GB

TABLE I
EFFECT OF WAIT TIMES ON CUBING OF 125 QF_LRA BENCHMARKS

Time	Solved	PAR-2
1s	40	223636
3s	42	219187
15s	42	219313

of memory per job) using the CVC5 SMT-COMP 2022 script (which runs a theory-dependent sequential instance of CVC5) and record the time. Then, we record the total solving time for the benchmark as the sum of the partitioning time and either the maximum time required to solve any of its partitions, if the benchmark is unsatisfiable, or the minimum runtime of the satisfiable partitions, if the benchmark is satisfiable. Note that this simulates a parallel run in which all of the partitions are run simultaneously on separate cores.

To evaluate a partitioning strategy on the set of benchmarks as a whole, we use the PAR-2 score [30], which is also used for scoring the annual SAT competition. The PAR-2 score is the sum of the runtimes for all solved instances plus *twice* the timeout value multiplied by the number of unsolved instances. The lower the PAR-2 score is, the better. We use the PAR-2 score because it provides a single metric that incorporates both runtime and number of benchmarks solved. In this work, we are primarily interested in the scalability of the different partitioning strategies, so most experiments measure the PAR-2 score for different numbers of partitions. We use a 20 minute timeout when solving partitions, which includes both the time to partition and the time to solve.

In the following subsections we explore a broad design space and, in the spirit of transparency, we chose to share the outcomes of both successful and less successful partitioning strategies. We chose to share as much as possible in an effort to both document our process and enrich future research in this area. When first developing our partitioning strategies, we tested them on 23 problems, a little more than 10% of the total problems in our set of challenge benchmarks. It was not until after extensive testing on this smaller subset that we ran the full set of 214 benchmarks with our various strategies to produce the data for the graphs that follow.

A. Partitioning Strategies

We first explore the different partitioning strategies in the parameter space of Algorithm 1, starting with the partition timing parameter.

First, we measure the impact of t_1 when using TIME for $tHeur$. In general, we find that if the value of t_1 is too small, then the partitioning solver does not have enough time to provide good atoms, but beyond a threshold there is little additional benefit. To demonstrate this, Table I shows a set of results using different values of t_1 . For these runs, we set $atomSource$ to HEAP, $pType$ to CUBE, and $atomSelHeur$ to SPEC, and run on a subset of benchmarks consisting of 125 QF_LRA benchmarks. We see that with $t_1 = 3$, we are able to solve more problems and obtain a better PAR-2 score than with

$t_1 = 1$. At the same time, increasing t_1 to 15 does not result in any additional problems being solved and has a small negative effect on the PAR-2 score. Additional experiments (not shown here) using other parameter settings and benchmarks show similar results. Based on these observations, we use $t_1 = 3$ when using TIME.

Next, we compare $tHeur = CHECK$ and $tHeur = TIME$. For these experiments, we use $pType = SCATTER$ (since otherwise t_2 plays no role) and $atomSelHeur = SPEC$. For $tHeur = TIME$, we use $t_1 = 3$ and $t_2 = 0.1$, so the first call to Algorithm 1 is after three seconds, and each subsequent call is after an additional tenth of a second. For the CHECK runs, we use $t_1 = 1$ and $t_2 = 1$, so the first call is after the first check, and each subsequent call is after the next check. Figure 1 shows the results¹ of our experiments for different atom sources and partition sizes. This figure shows that when $atomSource$ is DECISION or HEAP, TIME clearly outperforms CHECK. With CL, the results are not conclusive, with both strategies performing similarly.

One likely reason for the poor performance of CHECK in the first two cases is that with $t_1 = 1$ we start partitioning on the first call to Algorithm 1. As we mentioned above, it is usually better to wait some time before partitioning. However, there is a tremendous amount of variation in the number of calls to Algorithm 1 across different benchmarks. This makes it difficult to find values other than 1 for t_1 and t_2 that work consistently across all benchmarks. On the other hand, using TIME with $t_1 = 3$ and $t_2 = .1$ tends to perform well across different benchmarks and parameters. For these reasons, we use these parameters in the remaining experiments.

It is worth noting that the disadvantage of nondeterminism when using TIME remains. In future work, we hope to find a way to get the consistency of TIME while also having deterministic results.

We now turn our attention to the other parameters of Algorithm 1. Figure 2 shows the six possible strategies using $atomSelHeur = SPEC$ compared with a random strategy (TIME-HEAP-CUBE-SPEC) as a baseline (other random strategies are similar). The HEAP strategies consistently perform worse than the random strategy, which suggests that relying on the SAT solver’s priority queue heuristic is not particularly effective for making partitions.

The DECISION strategies, on the other hand, both perform quite well, beating the random strategy most of the time. This suggests that selecting the earliest decisions from the decision trail is a useful heuristic when selecting atoms for partitioning. The CL strategies are mixed. The CUBE variant performs favorably compared to random, while the SCATTER variant generally does not. Based on these results, we choose three configurations as our top partitioning strategies, and refer to them in the following as decision-cube (TIME-DECISION-CUBE-SPEC), decision-scatter (TIME-DECISION-SCATTER-SPEC), and cl-cube (TIME-CL-CUBE-SPEC).

¹These and subsequent experiments are on the full set of 214 benchmarks.

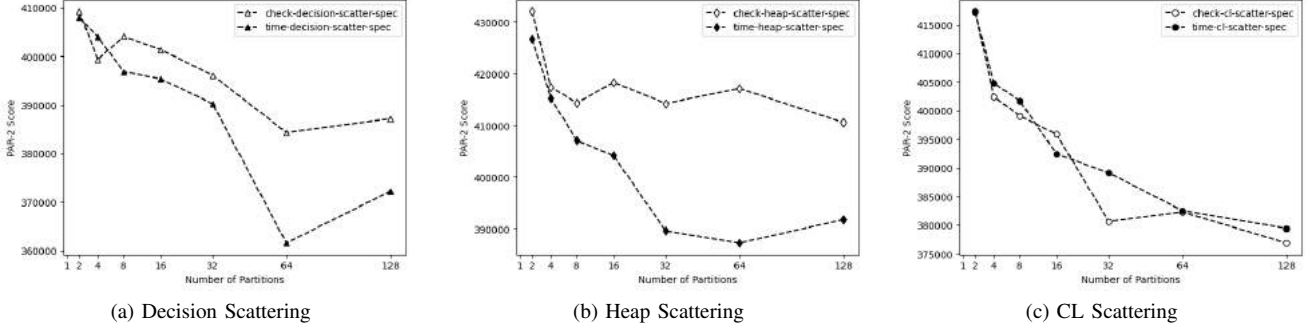


Fig. 1. Comparison of using TIME vs CHECK for *tHeur* when scattering.

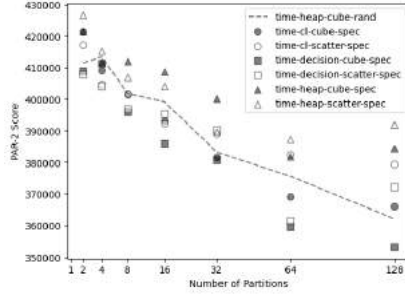


Fig. 2. Various partitioning strategies vs a random strategy.

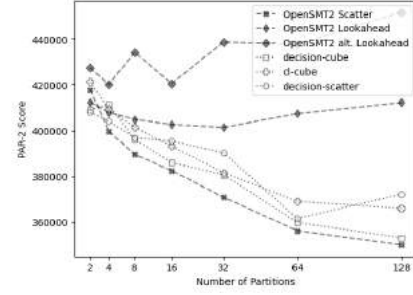


Fig. 3. CVC5 partitioning strategies vs OpenSMT2 partitioning strategies

Clearly, there are many more possible strategies and variants that could be explored. We expect this to be a fruitful direction for future work.

B. Comparison to OpenSMT2 Partitioning Strategies

The most extensive studies on partitioning strategies in previous work are from the OpenSMT2 team. We next compare our best partitioning strategies, decision-cube, cl-cube, and decision-scatter, to the three partitioning strategies available in OpenSMT2. Recall that none of the selected benchmarks are solved during partitioning, so this is a comparison only of the partitioning strategies, not the solvers. Figure 3 shows the results of this comparison. The two lookahead partitioning strategies in OpenSMT2 perform worse than our three best strategies. This is partly because they are slow and often time out during partitioning. On the other hand, the OpenSMT2 scattering strategy performs very well. In particular, their scattering strategy outperforms our individual partitioning strategies, though decision-cube is quite close. Interestingly, the OpenSMT2 scattering strategy also uses decisions as its source of atoms, making it very similar to our decision-scatter strategy. However, they have a few additional parameters that have been fine-tuned. For example, they vary the number of literals per partitioning formula, something that our strategy does not do. This suggests that our decision-scatter strategy could likely be improved in a similar way. In the spirit of “if you can’t beat them, join them,” we replace our decision-scatter strategy with the OpenSMT2 scattering strategy in our

list of top performing strategies going forward and refer to it as *osmt-scatter*.

VI. PORTFOLIOS OF PARTITIONING STRATEGIES

One consistent observation in this and previous work is that there is a lot of variation in how well strategies work across benchmarks. Every strategy fails on some benchmarks and works well on others. Given the success of portfolio solving in general, a natural question is whether a portfolio of partitioning strategies can outperform individual partitioning strategies. It is not immediately obvious whether this should be true, since, to be fair, we require that partitioning portfolios divide up their partitioning budget. For example, we would compare a partitioning portfolio using 2 strategies, each creating 16 partitions to an individual strategy that can use 32 partitions. We explore four different types of portfolios, each with a common goal of maximizing the diversity of solving strategies while minimizing the number of solver instances.

A. Types of Portfolios of Partitioning Strategies

The first type of portfolio is a *partitioning portfolio* where multiple strategies are used to create the same number of partitions as a single strategy. The goal of using this type of portfolio is to allow for different partitioning strategies to compensate for the weaknesses of the other strategies included in the partitioning portfolio. Figure 4a shows the results of using two different partitioning portfolios and compares them to the best individual strategy, *osmt-scatter*. Portfolio 1 creates $N/2$ partitions with each of decision-cube and cl-cube,

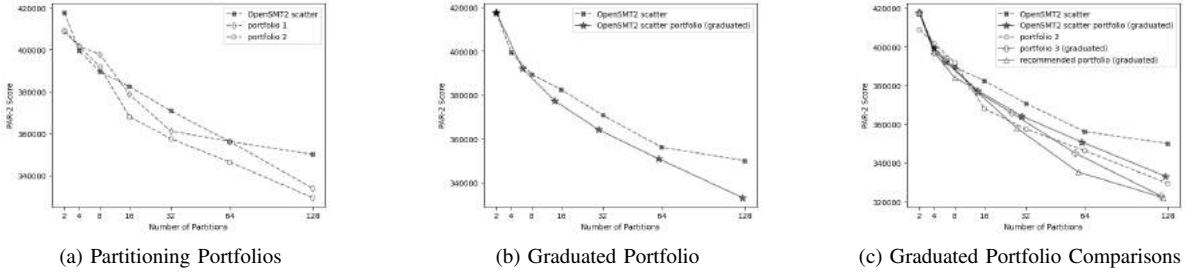


Fig. 4. Comparison of various partitioning portfolios.

and the results demonstrate that a portfolio of two (worse) strategies can do better than the best individual strategy. Portfolio 2 creates $N/2$ partitions with decision-cube, $N/4$ partitions with cl-cube, and $N/4$ partitions with osmt-scatter and demonstrates that adding more variety in the portfolio can result in even better performance. These results suggest that the orthogonality of the individual strategies is high and that adding more strategies at the expense of the number of partitions can be beneficial.

The next type of portfolio is a *graduated portfolio* which takes the above idea of maximizing diversity a bit further. Since diverse strategies are helpful, we can also diversify within a single strategy by instantiating that strategy several times, each time using a different number of partitions. To construct a graduated portfolio with a single partitioning strategy, the strategy is used to create 2 partitions, then it is used to create 4 partitions, and so on until the total desired number of partitions is nearly met (powers of 2 may not perfectly add up to the total number of desired partitions) but never exceeded. To visualize the value of graduated partitioning, Figure 4b compares the stand-alone osmt-scatter strategy with a graduated partitioning portfolio version of the same strategy. To see the difference, note that for $N = 16$, the stand-alone strategy runs osmt-scatter once to make 16 partitions, whereas the graduated portfolio runs osmt-scatter three times, making 2, 4, and 8 partitions, respectively. Notice that although the graduated portfolio uses two fewer partitions for every plotted value of N , it clearly outperforms the stand-alone strategy, especially as the number of partitions is increased. Strategies with the fewest partitions require the least resources, so in some sense, they are providing the greatest diversity at the lowest cost. Thus, we use graduated portfolios to achieve the most diversity at the lowest cost.

Notice that the above two strategies can be combined into a third type of portfolio called a *graduated partitioning portfolio*. A graduated partitioning portfolio based on m individual partitioning strategies is constructed as follows. First, take each of the m individual strategies and parameterize them by N for values of N that are powers of 2, e.g., osmt-scatter-2 is the strategy that uses osmt-scatter to make 2 partitions, and cl-cube-64 is the strategy that uses cl-cube to make 64 partitions. Now, rank all of the strategies, with smaller values of N being ranked higher. To break ties when $m > 1$, use a ranking on

individual strategies (for our top performing strategies, we rank osmt-scatter- N higher than decision-cube- N higher than cl-cube- N). Finally, to obtain the graduated portfolio strategy for N partitions, simply collect strategies from this list, in order, until it is no longer possible to add strategies without exceeding N . For example, for $N = 32$ and $m = 3$, we would choose all of the 2-partition strategies, all of the 4-partition strategies, and the osmt-scatter-8 strategy, for a total of 26 partitions. We do not attempt to use the remaining 6 partitions in our “partition budget.”

We experimented with all possible combinations of $m = 1, 2, 3$ and our top performing strategies. Figure 4c shows selected results (and also the best strategies from Figures 3, 4a and 4b for comparison). Portfolio 3 is the graduated partitioning portfolio with $m = 3$ using all three top-performing strategies: osmt-scatter, decision-cube, and cl-cube. Portfolio 3 outperforms portfolio 2 for large numbers of cores. However, the strongest portfolio uses $m = 2$ and only combines osmt-scatter and decision-cube (in this case, it appears that the diversity of cl-cube does not compensate for the lack of additional versions of the other two strategies). We refer to this as the “recommended portfolio.” Our recommended portfolio strategy has consistently better performance than all other strategies we have considered, even though it uses fewer partitions than the other strategies.

The fourth and final type of portfolio we refer to as a *hybrid portfolio*. This type of portfolio combines a traditional portfolio and a portfolio of partitioning strategies. For this approach, given N cores, we simply run a recommended portfolio of size $N/2$ and a traditional portfolio with the remaining cores. Once again, the goal is to diversify the solving approaches while keeping the number of solver instances as low as possible. Results for this approach are discussed in the following section.

B. Comparison to a Traditional Portfolio

Finally, we conclude by trying to address a very practical question. Given our results, how should one go about using N -way parallelism to solve a challenging SMT problem? In particular, how do the best partitioning approaches compare with traditional portfolio approaches. To represent the latter, we use a *scrambling* portfolio. Given a problem and a value of N , we construct a scrambling portfolio of size N by running N versions of the problem: the original problem plus $N - 1$

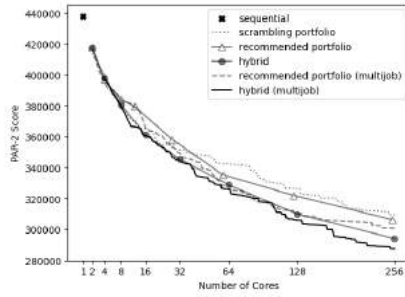


Fig. 5. Comparison of different portfolios. Hybrid and recommended portfolios are graduated.

TABLE II
NUMBER OF PROBLEMS SOLVED WITH 256 CORES

Strategy	Solved (out of 214)
sequential	54
scrambling portfolio	107
recommended portfolio	109
hybrid	111
recommended portfolio (multijob)	112
hybrid (multijob)	114

copies obtained using the SMT-COMP scrambler [31] with different random seeds. The runtime for the scrambling portfolio on the problem is then the minimum of these runtimes (we include the time required to scramble in the individual runtimes, but this is typically negligible).

Figure 5 compares, for different values of N , a scrambling portfolio with our recommended portfolio, based on a simulation where each partition or scramble is run on a separate core in parallel (it also includes the performance of a single sequential run for comparison). We see that the scrambling portfolio does indeed outperform the recommended partitioning portfolio for $N < 32$, but the recommended portfolio wins as the number of cores gets larger.

The result of using the hybrid portfolio is shown as “hybrid” in Figure 5. Recall that for this approach, half the cores are used to run a recommended portfolio while the other half run a scrambling portfolio. Remarkably, the hybrid strategy clearly outperforms the scrambling portfolio, even for small numbers of cores. We note that, to our knowledge, this is the first time any parallel strategy has been shown to be consistently superior to a traditional portfolio.

Finally, there is one more optimization that we can apply: we can use the *multijob* strategy referenced in the original cube-and-conquer paper [15]. The key idea is that because partitions can be very uneven (in runtime), using one core per partition means that many cores (the ones that get assigned easy partitions) will be idle most of the time. The multijob strategy simply consists of using *many more partitions* than cores, and then scheduling the partitions as cores become available. This automatically load-balances the partitions and results in being able to include the results from more partitions without much additional (wall-clock) runtime. This

optimization always improves the performance of partitioning approaches, but it has no effect on traditional portfolios since a traditional portfolio cannot benefit from load-balancing. Consider a problem that is unsolved by the (non-multijob) hybrid strategy. For this problem to be unsolved, all scrambling jobs must time out and at least one of the partitions for each partitioning strategy must also time out. We observe that, in practice, many of the partitions finish quickly, meaning that the resources allocated to those partitions are idle while the other jobs run to timeout. The multijob strategy simply uses these additional available resources to run more partitioning strategies.

The recommended portfolio (multijob) line shows the result of simulating a recommended portfolio that includes *all* versions of osmt-scatter and decision-cube (from 2 up to 128) for different numbers of cores. We schedule the smaller partitions first, and no core is allowed to do more than 20 minutes of work. The hybrid (multijob) shows the results of using $N/2$ cores for a scrambling portfolio and running the multijob scheduling algorithm on the other $N/2$ cores. The best strategy is hybrid (multijob) and with 256 cores, it improves the PAR-2 score by 34% (compared to a single sequential solver). Table II shows the total number of problems solved by each strategy when 256 cores are used. Note that each additional problem solved represents a significant step forward as these are very difficult problems unsolved by the previous techniques. These results show that in order to achieve robustness and consistent performance improvement, advanced portfolio techniques that incorporate multiple partitioning strategies should be preferentially used over more fragile individual partitioning strategies.

VII. CONCLUSION

We have shown that a portfolio of partitioning strategies, hybrid portfolios in particular, can outperform traditional portfolio solving for a range of SMT problems. These new strategies are a step toward better utilization of HPC and cloud systems for solving SMT problems. Additionally, we show that these new portfolio techniques perform much better than individual partitioning strategies and even manage to outperform a traditional portfolio.

There are many promising directions for future work that we are looking forward to investigating. For individual partitioning strategies, we plan to systematically explore the *cubeSize* parameter for SCATTER, improve our decision-scatter algorithm to work similar to OpenSMT2’s version, explore additional atom selection heuristics, and find a way to achieve the consistency of TIME for *tHeur* without sacrificing determinism. We also plan to implement information sharing, explore recursive partitioning, push the multijob optimization further, and expand the benchmarks for evaluation, including other quantifier-free logics and quantified benchmarks.

REFERENCES

- [1] A. Niemetz and M. Preiner, “Bitwuzla at the SMT-COMP 2020,” *CoRR*, vol. abs/2006.01621, 2020. [Online]. Available: <https://arxiv.org/abs/2006.01621>

- [2] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “cvc5: A versatile and industrial-strength SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24
- [3] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, “The Math-SAT5 SMT Solver,” in *Proceedings of TACAS*, ser. LNCS, N. Piterman and S. Smolka, Eds., vol. 7795. Springer, 2013.
- [4] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, “OpenSMT2: An SMT solver for multi-core and cloud computing,” 2016.
- [5] B. Dutertre, “Yices 2.2,” in *Computer-Aided Verification (CAV’2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, Jul 2014, p. 737–744.
- [6] L. de Moura and N. Björner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 337–340.
- [7] L. Cordeiro and B. Fischer, “Verifying multi-threaded software using SMT-based context-bounded model checking,” in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 331–340. [Online]. Available: <https://doi-org.stanford.idm.oclc.org/10.1145/1985793.1985839>
- [8] A. Komuravelli, A. Gurfinkel, and S. Chaki, “SMT-based model checking for recursive programs,” *Formal Methods in System Design*, vol. 48, pp. 175–205, 2014.
- [9] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffei, “Refinement types for secure implementations,” *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 2, feb 2011. [Online]. Available: <https://doi-org.stanford.idm.oclc.org/10.1145/1890028.1890031>
- [10] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, “Counterexample-guided quantifier instantiation for synthesis in smt,” in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 198–216.
- [11] J. Peleska, E. Vorobev, and F. Lapschies, “Automated test case generation with SMT-solving and abstract interpretation,” in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 298–312.
- [12] W. Steiner, “An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks,” in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 375–384.
- [13] R. Sebastiani and P. Trentin, “OptiMathSAT: A tool for optimization modulo theories,” *Journal of Automated Reasoning*, pp. 1–38, 2015.
- [14] C. M. Wintersteiger, Y. Hamadi, and L. de Moura, “A concurrent portfolio approach to SMT solving,” in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 715–720.
- [15] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, “Cube and conquer: Guiding CDCL SAT solvers by lookaheads,” in *Hardware and Software: Verification and Testing*, K. Eder, J. Lourenço, and O. Shehory, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 50–65.
- [16] M. J. H. Heule and H. van Maaren, “Look-ahead based sat solvers,” in *Handbook of Satisfiability*, 2021.
- [17] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, “Lookahead in partitioning SMT,” in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 271–279.
- [18] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to dpll(t),” *J. ACM*, vol. 53, no. 6, p. 937–977, nov 2006.
- [19] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, “A distribution method for solving sat in grids,” in *Theory and Applications of Satisfiability Testing - SAT 2006*, A. Biere and C. P. Gomes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–435.
- [20] M. Marescotti, A. Hyvärinen, and N. Sharygina, “SMTS: Distributed, visualized constraint solving,” in *Logic Programming and Automated Reasoning*, 2018.
- [21] M. Marescotti, A. E. J. Hyvärinen, and N. Sharygina, “Clause sharing and partitioning for cloud-based SMT solving,” in *Automated Technology for Verification and Analysis*, C. Artho, A. Legay, and D. Peled, Eds. Cham: Springer International Publishing, 2016, pp. 428–443.
- [22] A. Hyvärinen and C. M. Wintersteiger, “Parallel satisfiability modulo theories,” in *Handbook of Parallel Constraint Reasoning*, 2018.
- [23] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, “Search-space partitioning for parallelizing smt solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2015*, M. Heule and S. Weaver, Eds. Cham: Springer International Publishing, 2015, pp. 369–386.
- [24] G. Andersson, P. Bjesse, B. Cook, and Z. Hanna, “Design automation with mixtures of proof strategies for propositional logic,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 8, pp. 1042–1048, 2003.
- [25] C. Reisenberger, “Pboolector: a parallel smt solver for qf_bv by combining bit-blasting with look-ahead,” Ph.D. dissertation, Master’s thesis, Johannes Kepler Universität Linz, Linz, Austria, 2014.
- [26] A. Niemetz, M. Preiner, and A. Biere, “Boolector 2.0,” *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014. [Online]. Available: <https://doi.org/10.3233/sat190101>
- [27] A. Biere and A. Fröhlich, “Evaluating cdcl variable scoring schemes,” in *Theory and Applications of Satisfiability Testing - SAT 2015*, M. Heule and S. Weaver, Eds. Cham: Springer International Publishing, 2015, pp. 405–422.
- [28] H. Barbosa, F. Bobot, and J. Hoenicke, “SMT-COMP 2022.”
- [29] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” <http://smtlib.cs.uiowa.edu>, 2016.
- [30] N. Froleys, M. Heule, M. Iser, M. Järvisalo, and M. Suda, “SAT competition 2020,” *Artificial Intelligence*, vol. 301, p. 103572, 2021.
- [31] T. Weber, A. Niemetz, J. Hoenicke, A. Hyvärinen, H. Barbosa, and M. Schlaipfer, “SMT-COMP benchmark scrambler,” <https://github.com/SMT-COMP/scrambler>, 2023.

CRV: Automated Cyber-Resiliency Reasoning for System Design Models

Daniel Larraz^{*} , Robert Lorch^{*} , Moosa Yahyazadeh^{*} , M. Fareed Arif[†] ,
Omar Chowdhury[‡] , Cesare Tinelli^{*} 

^{*}The University of Iowa, Iowa City, USA, ✉ daniel-larraz@uiowa.edu

[†]The University of Oxford, Oxford, UK

[‡]Stony Brook University, Stony Brook, USA

Abstract—We present the design and implementation of an automated static analysis approach and corresponding diagnostic tool, called Cyber Resiliency Verifier (CRV), to check whether a system design satisfies its end-to-end guarantees when the integrity of one or more of its components cannot be guaranteed. CRV’s key insight is to reason about *effects of integrity attacks* instead of concrete attacks, enabling it to reason also about the impact of future attacks having the same captured effects. We demonstrate CRV’s effectiveness with a case study on a realistic design of an unmanned aerial delivery drone.

I. INTRODUCTION

Security vulnerabilities in critical systems can have catastrophic impacts. Even when a vulnerability is discovered, performing root cause analysis and then adding security mechanisms *a posteriori* can be expensive, challenging, or infeasible. Exploitable weaknesses in a system’s design are arguably harder to mitigate after deployment due to backward compatibility requirements, operational cost, and QoS constraints. This paper focuses on enabling system architects to identify and mitigate such design weaknesses *at the system design stage*.

A major reason systems have exploitable design weaknesses is that during the design phase, security considerations often take a secondary role to other requirements such as time to market. In addition, current design analysis tools and methodologies often pay scant attention to security considerations. The lack of sophisticated capabilities for the identification of security vulnerabilities at an abstract design level is an impediment for the model-based system design paradigm to reach its full potential. Although it is impossible to fully avoid vulnerabilities in the implementation, model-based analysis tools can nevertheless help designers design systems with *cyber-resiliency* in mind, that is, design systems whose functionality and integrity guarantees degrade gracefully under attack. We broadly define a system’s *integrity properties/guarantees* as functional properties that must be satisfied for achieving its desired functionality. We propose a general approach and a highly automated tool, CRV, whose rich diagnostic information allows a system architect to assess under different threat models the resiliency of a system design with respect to desired integrity properties. A static analysis tool like CRV allows a system architect to account for security considerations already in the design phase. In particular, it enables *what-if*-type analyses exploring the effect that violations of integrity

properties in a sub-system or software component may have on the overall system guarantees, avoiding surprises like a recent supply chain attack [29].

A typical workflow prescribed by CRV starts with the system architect developing a system model in a suitable modeling language and identifying critical functional properties that the system must maintain even when subject to attacks that compromise one or more system components. A system’s design model (or, *system design*) contains *system architecture* information as well as *behavioral information* on one or more components. For cyber-resiliency analysis, the designer additionally selects a subset of the model’s components and/or connections whose integrity cannot be guaranteed. CRV then automatically instruments the design to account for the integrity issues. This instrumentation reduces the problem of assessing the cyber-resiliency of the design to a model checking problem: the satisfaction of the original functional properties also in the instrumented model. A violation of one of these properties implies that the original design (before instrumentation) is not resilient in the presence of attacks captured by the chosen threat model. When CRV discovers a violation, it provides as evidence an execution trace of the instrumented system for each violated property. These traces demonstrate the viability of attacks from the chosen threat model, as well as their effects on system properties. Additionally, for each violation, CRV provides a list of system components whose misbehavior may have contributed to the violation. For each property that remains satisfied even under attack, CRV provides a list of critical sub-components that may have contributed to satisfaction of the property. We demonstrate the effectiveness of CRV by evaluating it with respect to a case study of an UDD.

Contributions. In summary, this work makes the following technical contributions: (i) a *general framework and tool* for analyzing the resiliency of a system design with respect to desired functional properties against one or more threat models, including *replay attacks*; (ii) a novel notion of *attack/threat effects* that allows for the automatic instrumentation of design models and takes into account both known and unknown integrity attacks by reducing resiliency analysis to model checking; (iii) a formalization of attack effects in terms of the Dolev-Yao model, a formal model common in cryptographic protocol verification; (iv) a new automated process, *blame assignment*, to identify compromised components of a system

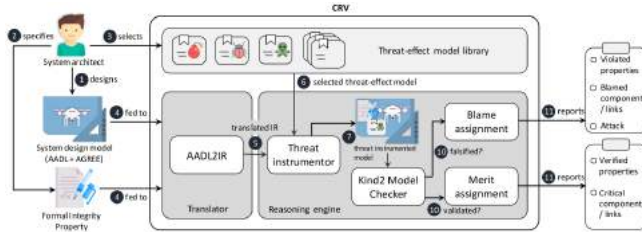


Fig. 1: CRV Architecture and Workflow Diagram.

design whose misbehavior can contribute to the violation of one or more functional properties of the system; (*v*) an automated process, *merit assignment*, that identifies components of a system design that positively impact the satisfaction of desired properties; and (*vi*) a *case study* of CRV against a model of an unmanned delivery drone.¹

II. DESIGN OVERVIEW OF CRV

We start this section with the problem definition. We then present CRV’s architecture, describe its interactions with the system designer, and discuss the underlying challenges.

A. Problem Definition and Scope

CRV automatically checks if a system design provides the required functional guarantees when the integrity of some of its sub-component cannot be ensured. CRV reasons about a hierarchical system design \mathcal{D} specifying the system architecture, the function behavior of each component, one or more functional guarantees Φ_i , and a threat model \mathcal{T}_m indicating which components and connections are vulnerable to attacks. CRV attempts to prove whether \mathcal{D} can maintain Φ_i even when, based on \mathcal{T}_m , the integrity of one or more components or connections is compromised.

We consider only systems whose designs can be expressed as *synchronous* (finite- or infinite-state) transition systems. For such systems, CRV focuses on *integrity* properties, functional properties that can be violated by compromising the integrity of their components or their interconnections. Technically, CRV currently analyzes only system properties that can be expressed as *temporal safety properties* of the form $\Box \bigwedge_i a_i \rightarrow \Box \bigwedge_j g_j$ where a_i and g_j are quantifier-free, first-order past-time LTL formulas and \Box is the *always/globally* operator.

A large set of desired system-level integrity properties can be modeled as temporal safety properties of the form supported by CRV. Examples of such properties for the model in our case study (see Section VI) would be the requirement that the delivery drone never deliver a package to an off-limits location or that it deliver a package to a given drop location only if the location is clear. Many other examples exist in practice (see, e.g., [19], [20], [15]).

Currently outside the scope of CRV are *confidentiality*, *authentication*, and *availability* properties. Traditional analyses of confidentiality properties such as non-interference only

consider attacks on system-level inputs, as opposed to attacks on individual components of the system under analysis. Supporting that traditional analysis in CRV would not be difficult. Both the approach and the tool could be easily extended to compose the input system design with itself (i.e., have two copies of the system running in parallel) while asserting the top-level safety property that the public outputs are equal whenever the public inputs are equal. In contrast, in our case study, where we also look at attacks on system components, considering non-interference would require a new class of properties whose violation may include *both* confidentiality and integrity guarantee violations at the component level.

Authentication properties are relevant only to protocols that use cryptographic constructs, as they are easily violated without such constructs. For those cases, a cryptographic protocol verifier could be incorporated in principle into CRV’s workflow to enable reasoning about cryptographic constructs. Investigating this integration is left to future work.

Although CRV could potentially reason about availability properties (e.g., under the right conditions, the delivery drone will make a successful delivery), adding this capability would be more challenging in general. The reason is that such properties translate to *liveness* properties, and current model checking technology is not advanced enough to prove (unbounded) liveness for most realistic models of infinite-state systems. However, we intend to provide support for some restricted classes of availability properties in the future.

B. CRV Architecture and Workflow

The high-level architecture of CRV and its interaction with the system designer is shown in Figure 1. The designer starts by developing, in a suitable modeling language, the design of the system to be analyzed. Currently, CRV is available [32] through a plug-in of the OSATE IDE for the Architecture Analysis and Design Language (AADL) [21] extended with AGREE [11] contract language. The design describes the system’s architecture in terms of system components, component interfaces and interconnections, and a list of components and connections considered vulnerable to attack. In addition, the design also contains behavioral information for some components, expressed in the form of *assume-guarantee* contracts, capturing how input and internal state values are converted to outputs and state updates. Finally, the architect adds system-level properties that the model should satisfy, expressed as guarantees of the top-level component in the design.

When invoked, CRV’s *front-end* translates the design and the system level properties into an intermediate representation (IR). CRV can support other modeling languages with the addition of the corresponding IR translators. The IR and one or more user-selected threat models are then fed into the *threat instrumentor module* which modifies the model’s IR to include adversarial influences according to the list of vulnerable components and connections. Finally, the threat-instrumented model is fed to the Kind 2 model checker [9],

¹A VM image containing tool, models, and related instructions is available to the reviewers at: <http://clc.cs.uiowa.edu/fmcd23/>.

[25] in order to prove or disprove that the model satisfies the desired properties.

For each satisfied property, CRV’s *merit assignment module* identifies system components and connections in the design which are critical for satisfying the property. Dually, for each violated property, the *blame assignment module* identifies the vulnerable components/connections that contributed to the violation. An *attack trace*, describing the attacker’s behavior and the system’s response, is also presented as evidence.

C. Challenges

CRV addresses the following three analysis challenges.

Scalability. The complexity of model checking problems CRV solves ranges from NP-hard to undecidable. CRV addresses this scalability challenge by leveraging Kind 2’s reasoning support for complex, hierarchical systems in the form of *compositional* verification where verification results for sub-systems/components are used to discharge verification conditions of higher-level components.

Behavioral modeling. Capturing components’ behavior in a design model at the right abstraction level for a successful analysis is a challenging task. More abstract behaviors simplify the automated analysis but can lead to an increased number of *false positives*: execution traces that falsify the property but are not actual executions of the modeled system. In contrast, more detailed behavior decreases or eliminates false positives but can increase the analysis complexity to the point of overwhelming the model checker. We prescribe capturing abstract behavioral information in sufficient detail in the form of *assume-guarantee contracts* for selected components. Such contracts state that as long as a component’s environment satisfies the contract’s assumptions, the component’s behavior will satisfy the contract’s guarantees.

Threat instrumentation. The final challenge is how to incorporate the adversarial influence in the design. One possible approach is to explicitly consider concrete attacks (such as buffer overflow, malware attacks, and so on). Unfortunately, this approach has serious shortcomings: (i) listing all possible integrity attacks can be cumbersome and time-consuming; (ii) the attacker model becomes outdated with the discovery of newer attacks; (iii) the design may not contain implementation-level details for the sake of scalability, making it difficult to describe specific attacks (e.g., SQL injection); (iv) no guarantees can be provided against *zero-day* attacks. To address these challenges we consider attack *effects* instead of concrete attacks, as explained in the next section.

III. DESIGN MODEL INSTRUMENTATION

In this section, we explain the notion of attack effects and then discuss the automatic instrumentation of the model.

The Problem. The threat instrumentation process in CRV solves the following main problem: *Given a formal description of a system component in terms of its input/output interface and behavior, how do we capture all possible integrity attacks that can impact the component’s behavior?* The problem can be further decomposed into two technical questions:

(Q_1) how to consider all possible integrity attacks efficiently; (Q_2) how to incorporate the different attacks, some of which can be implementation-specific, into a design containing only abstract information.

A. Attack Effects

Our main insight for addressing question Q_1 above is to switch attention from attacks to their *effects*. More precisely, we argue that, in the context of resiliency analysis, it is sufficient to consider the *effects of integrity attacks on a system component’s behavior* instead of the concrete attacks themselves. Capturing the effects frees us from having to worry about the details of how each attack is achieved in concrete. Effects of integrity attacks can be viewed as *attack abstractions* which group together integrity attacks according to their consequences on a system. In addition to simplifying reasoning, this sort of abstraction allows us to reason at once about *all* integrity attacks having a certain set of effects.

In general, an integrity attack can have the following three effect types: (E_1), or *standard attacker*, lets the adversary modify the behavior of the attacked component at will—as done, for instance, in buffer overflow attacks; (E_2), or *replay attacker*, lets the adversary replay previous values of data sent across vulnerable network connections; and (E_3) which can render the component unresponsive—for instance, by making it crash. Note that (E_1) is strictly stronger than (E_2)—the user chooses between (E_1) and (E_2) based on the system and adversary under consideration. Both cases include (E_3). In our context, these effect classes are sufficient to account for any kind of integrity attack in a system design.

Capturing Attack Effects In view of our classification of attack effects, question (Q_2) becomes how to incorporate the three types of effects (E_1 – E_3) into the design *automatically*. When instrumenting a component or connection with effects of type (E_1), CRV does not constrain in any way the behavior an adversary would choose. This allows it to reason about *all relevant* attack strategies that violate the desired functional properties. When instrumenting a connection with effects of type (E_2), CRV constrains the adversary’s behavior to only inject values that were previously sent along the connection.

Technically, we model adversarial actions by adding *non-determinism* in selected places in the model, accounting for unconstrained adversary behavior, and letting the model checker find a concrete behavior (*i.e.*, an execution trace) that results in a property violation. In essence, we let the model checker play the adversary by allowing it to replace the output of a compromised component by any value (of the correct type) of its choosing or, in the case of a replay attacker, by any previous output value.

In terms of the Dolev-Yao model [13], a formal model common in cryptographic protocol verification, CRV effectively places a *restricted* Dolev-Yao-style adversary after each output of a vulnerable component, allowing the adversary to: arbitrarily change an output, mimicking (E_1) attacks; replay a previous value, mimicking (E_2) attacks; or drop an output, mimicking (E_3) attacks. Note that the standard Dolev-Yao

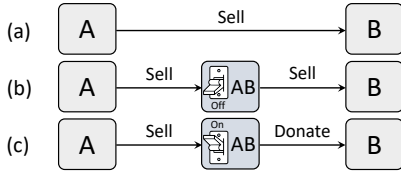


Fig. 2: (a) Non-instrumented channel between A and B; (b) instrumented but not enabled channel between A and B; and (c) enabled instrumented channel between A and B.

model considers a *network adversary* to be placed only on *public channels* where it has the following capabilities: (a) it can sniff, (b) drop, and (c) modify any message passing through the channels; (d) it can send messages impersonating the protocol participants; moreover, (e) it can exercise capabilities (a)–(d) while conforming to cryptographic assumptions. Our restriction of the model does not include capabilities (a) and (e) as CRV does not currently consider confidentiality properties nor reasons about cryptographic constructs.² Our instrumentation further deviates from the Dolev-Yao model by placing an adversary in a *non-public* channel (i.e., encrypted and integrity protected) when the sender in the channel is marked by the designer as vulnerable to integrity attacks.

As an example, consider the design fragment shown in Figure 2(a) where component A feeds its single output, of enumeration type {Sell, Donate}, only to component B. If A is vulnerable to integrity attacks (e.g., buffer overflow) *impacting its integrity*, then we instrument the model by placing a new component AB between A and B, effectively simulating a vulnerable public communication channel as in the Dolev-Yao model. The new component has two inputs, namely the output of A and a Boolean input corresponding to an enabling switch (analogous to an activation variable in fault analysis), and a single output sent as input to B. When the switch is off (see Figure 2(b)), the AB component behaves as a benign lossless channel, faithfully forwarding the output of A to the output of B. When the switch is on, the AB component can behave maliciously by replacing the output of A with a different value chosen non-deterministically (see Figure 2(c)). We use the switch inputs because they are essential for generating diagnostic information (i.e., blame assignment), where each switch is treated symbolically as a model parameter.

B. Utility of CRV’s What-if Analyses and Threat Models

CRV comes with a set of user-selectable built-in threat models, which allow it to determine automatically vulnerable components and connections. To use these, the designer annotates model components and connections with a few security-related meta-level attributes, for example, the pedigree of a component expressed by an enumerated type like {COTS, Sourced, inHouse}. (See [14] for a detailed list of such meta-level attributes.) Then, CRV identifies components and connections that should be considered vulnerable to attack

²Capability (e) can be simulated by incorporating a cryptographic protocol verifier into CRV’s workflow.

in the selected threat model(s). The details of the built-in threat models in CRV are presented in Section III-C.

Based on our presentation so far, a system designer can pose *what-if* queries of the sort: *What happens if a set X of vulnerable components and connections is subject to integrity attacks? Does the system design still ensure the satisfaction of the desired properties?* To illustrate the utility of the underlying *what-if* analyses of CRV, we describe how a system architect (i) could model *supply chain attacks* and (ii) design *zero trust networks*.

To model a *supply chain attack* [29], the system designer annotates each model component corresponding to an outsourced subsystem as vulnerable to attack. Given this information, CRV determines if violations of the outsourced component’s contract (in this case, due to supply chain vulnerabilities causing the component to misbehave) lead to violations of system-level integrity properties. If so, the designer should consider mitigating actions such as producing the subsystem component in-house or pursuing additional supply chain protections. If not, the model is resilient to supply chain attacks.

Zero trust networks are networks where every component performs input validation, rather than assuming that internal network connections are safe from attack [28]. As input validation can be computationally expensive, a system designer might want to verify if validation can be safely skipped for some input channels. To model this situation, the designer marks the corresponding model connections as vulnerable, and CRV will report if attacks on such connections affect system-level properties. If no system-level properties are violated under the chosen threat model(s), the system is cyber-resilient enough for the designer to consider forgoing validations steps on inputs coming through the marked connections.

C. Built-in Threat Models

CRV can automatically identify the vulnerable model components and connections according to some built-in threat models. A *threat (effect) model*, in this context, conservatively describes the criteria under which certain components and channels can be considered vulnerable to attacks (such as, logic bomb, remote code injection, and malware) that impact the control-flow integrity of the components/channels matching the criteria.

More operationally, we express threat models as queries on the design model data that specify the criteria for classifying components or channels as vulnerable to certain integrity attacks. Concretely, a specific threat model can be expressed as a query whose result is a set of components/connections that satisfy specific constraints on the meta-level attributes.

As an example, a component is automatically classified as susceptible to logic bomb or software Trojan attacks if (i) the component’s type is software or software hybrid; (ii) its pedigree is either COTS, or Sourced without supply chain protection or tamper protection; and (iii) the component has not gone through static analysis or adversarial testing for logic bombs. A list of threat model descriptions currently used by CRV is given in [14]. Note that it is easy to expand such a list

since new meta-level attributes and threat model descriptions can be added modularly.

IV. MODEL CHECKING AND DIAGNOSTICS

We now discuss how CRV takes advantage of automated reasoning to perform resiliency analysis and generate meaningful traceability and diagnostic information from it.

Model Checking for Resiliency Analysis. We reduce the problem of checking the resiliency of a system design to integrity attacks to a model checking problem for the threat-instrumented version of the design. In CRV’s workflow, the threat instrumented model and the desired functional properties are fed to the Kind 2 model checker [9]. Using induction-based techniques, Kind 2 tries to prove that each property is satisfied for any possible execution, including those containing the attacks contemplated by the instrumentation. In parallel with that, Kind 2 uses bounded model checking techniques to exhaustively search for execution traces in the instrumented model that violate one or more of the given properties. For each property, Kind 2 can output three possible verification results: **SAFE**, meaning that the instrumented design satisfies the property; **UNSAFE**, meaning that the design allows executions violating the property; and **UNKNOWN**, returned for instance when the model checker times out. For each definite answer (**SAFE** or **UNSAFE**), Kind 2 provides additional diagnostic information, as discussed below. The **UNKNOWN** case is due to the undecidability of the model checking problem for infinite-state systems in general and to the high computational complexity of the problem in decidable subcases. We emphasize that Kind 2 employs *sound* proving techniques which are, however, necessarily *incomplete* in the case of infinite-state systems.

End-to-End Security Properties. A designer may wonder if they could simply model each component in isolation, rather than dealing with a complex, hierarchical model spanning the entire system. Thanks to CRV the latter strategy is preferable as it enables the designer to reason formally about end-to-end properties that reference output from *multiple components*. To start, end-to-end reasoning enables the user to prove that system-level properties hold in the benign case, which is an important initial sanity check. More important, CRV’s analysis may show that property violations for one or more individual sub-components do not actually lead to system-level attacks — which is the case when CRV proves that system-level properties are still preserved. In contrast, without a tool like CRV, the designer has to *manually* reason about whether attacks on individual components can compose to a violation of a system-level property.

A. Attack Traces

For each property that it proves **UNSAFE**, the model checker returns as evidence to CRV an input/output counterexample trace, in effect an attack trace. The trace contains detailed information of the attacker’s actions (*i.e.*, the non-deterministic choices made in the instrumented channels) as well as the reactions of the other components to those actions.

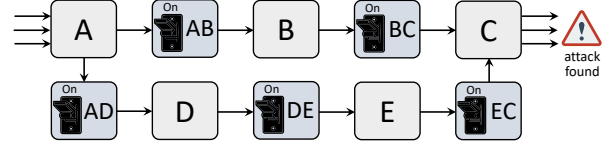


Fig. 3: An example instrumented design with all components being vulnerable.

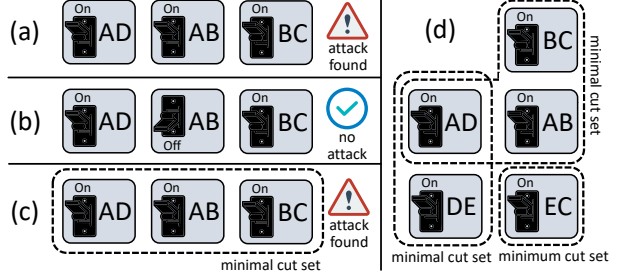


Fig. 4: An example of minimal and minimum cut-set.

B. Blame Assignment

For each property determined to be **UNSAFE**, in addition to the attack trace, CRV can also generate information regarding misbehaving components that may have contributed to the violation. We call this functionality *blame assignment*. CRV supports a *locally optimized* [24] and a *globally optimized* [23] form of blame assignment, both achieved by posing a series of queries to the backend model checker.

To understand blame assignment, consider as an example the threat-instrumented system design sketched in Figure 3 where all components are vulnerable according to the threat model specified by the user. This is reflected by the presence of an adversarial component (*e.g.*, AB) between each pair of connected components (*e.g.*, A and B). Each adversarial component is switched on, that is, it is enabled to perturb the communication between the components it links. Once the model checker finds an attack trace that demonstrates the violation of a property, the blame assignment module of CRV will try to minimize the number of enabling switches that must be turned on to cause a violation of the property. Technically, this is analogous to finding a *minimal cut set* [1], [23] for these switches. Suppose it is enough to turn on just components AD, AB, and BC in Figure 3 for the model checker to come back with an **UNSAFE** verdict for the property (see Figure 4(a)). Then these three switches form a minimal cut set only if turning off any of them (say, switch AB) changes the verification verdict to **SAFE** (see Figures 4(b)). Since there may be many different minimal cut sets for the complete set of switches, CRV tries to find the one with the smallest cardinality (*e.g.*, {EC} in Figure 4(d)).

C. Merit Assignment

If, after its resiliency analysis, the model checker returns a **SAFE** verdict for a given desired property, CRV also provides a list of components whose behavior *might be* critical for the satisfaction of the property. The conditional is necessary

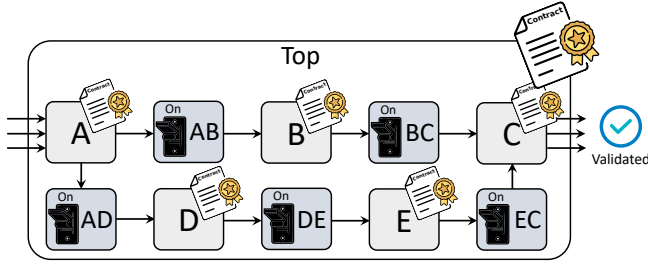


Fig. 5: The top component's contract is respected given all internal components' contract are validated.

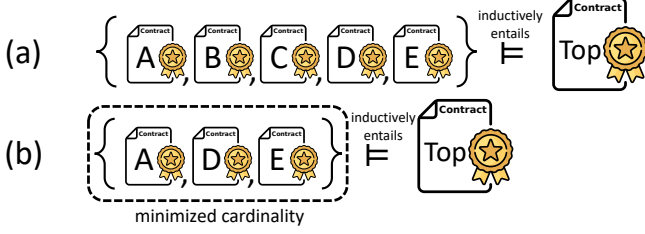


Fig. 6: Merit assignment problem as finding a minimal subset of the set in (a) that still inductively entails a given guarantee in the system's contract.

because, for efficiency reasons, the user has the option of requesting an over-approximation of the critical list. This functionality called *merit assignment* is useful for two reasons: (i) it provides better traceability in the model by confirming whether the defenses put in place by the designer in response to a (previous) violation of the property indeed play a role in maintaining the property; (ii) it provides additional information for system developers, who then know that behavioral changes to the components outside the merit assignment set will not affect the property and that, instead, extra precautions should be taken when implementing components in that set.

Merit assignment in CRV relies on the computation of a *minimal inductive validity core* (MIVC) [17], another functionality provided by Kind 2. As in other symbolic model checkers, Kind 2 represents an input model internally as a transition system consisting of an initial state predicate I characterizing the system's initial state(s) and a two-state transition relation T describing the system's behavior. The relation T can be expressed as conjunction or, equivalently, a set of constraints over states and their successors. A MIVC for a particular property P satisfied by a transition system (I, T) is a minimal subset T' of T such that (I, T') also satisfies P . The name MIVC comes from the fact that the model checker proves that a transition system satisfies P , i.e. that the property is *valid* in every execution of the system, by using *inductive* arguments. In that case, one can say that the system *inductively entails* the property.

To illustrate the concept, let us consider the example in Figure 5. Suppose the model checker is able to prove that the composition of the top component's behavior and that of each of its subcomponents, expressed by its contract, inductively entails a desired system-level property, expressed as a guarantee in the top-level contract (see Figure 6(a)).

Merit assignment provides a minimal subset of subcomponents whose contracts suffice for the proof. Concretely, if the contracts of the subcomponents A, D, and E are both sufficient and necessary to construct a proof of the guarantee (see Figure 6(b)), then A, D, and E, and only those, will be included in the merit assignment. Furthermore, for each included subcomponent, CRV will single out the individual guarantees in the subcomponent's contract that are enough to prove the desired top-level property.

In our case, due to inherent runtime complexity reasons, we resort to identifying *approximate MIVC*, i.e. (not-necessarily minimal) supersets of a MIVC, another functionality provided by Kind 2. Experimental results by Larraz *et al.* [23] show that approximate MIVC typically approximate true MIVC very closely while requiring significantly less time to compute.

V. IMPLEMENTATION OF CRV

Although designed to be incorporated in various modeling environments, CRV is currently available as a functionality of VERDICT, a larger cyber-resilience analysis tool developed with partners at GE Research [27]. In this section, we provide some details specific to CRV's implementation.

Modeling Language. We instantiated CRV for the AADL modeling framework [16]. An AADL model can capture the architecture of a synchronous reactive system in terms of components and their interconnections. *Components* of an AADL model are *systems*, which group together other components (or subsystems), and *data*, which define the data types used by the various systems. To model interactions among (sub)systems one defines an interface for each of them consisting of *data ports*, *event ports*, and *connections*. AADL can be extended by users in two ways. The first is the addition of user-defined attributes, called *properties* in AADL. This allows us to capture whether components or their connections should be considered vulnerable to attack. AADL also has an extension mechanism based on *language annexes* with which one can embed a domain-specific language (DSL) in AADL and use it to enrich the design description. One such annex contains the AGREE DSL [11] which allows one to express behavioral information for synchronous components formally and declaratively. Component behavior is specified in AGREE either as an assume-guarantee contract or as an *implementation* consisting of equational constraints on ports and internal state variables.

Front-end Translator. We developed a translator for CRV, written in Java, that takes as input an AADL design model enriched with security-related AADL properties and component-level behavioral specs in AGREE, and translates it to an intermediate textual representation (IR). Our IR is general enough to accommodate a wide variety of modeling languages supporting the synchronous model of computation (e.g., Simulink+Stateflow). The IR is close in structure to the synchronous dataflow language Lustre [18].

Threat Instrumentor. The threat instrumentor module of CRV is written in Java. It takes the IR version of the design model and the selected list of vulnerable components and

connections, and returns a threat-instrumented design, also written in the IR language.

Standard Attacker. When CRV generates the instrumented model, a component is automatically generated to specify the adversary’s behavior. A standard attacker, such as in Figure 2(c), is modeled as an intermediate component which takes messages from component *A* as input and produces adversarially-instrumented messages to pass to component *B*.

Bounded Replay Attacker. A replay attacker component is obtained by adding a contract to a standard attacker that restricts its output messages to be equal to previous (legitimate) messages from the last *n* time steps. This models adversaries that can only replay past outputs and have a bounded memory. The number *n* is configurable by the user in CRV’s front end.

Unbounded Replay attacker. We also include support for a replay attacker with unbounded memory, i.e., the ability to replay messages that were sent arbitrarily far in the past. This is achieved by representing the sequence of past outputs in the model as an uninterpreted function from time steps to output values that is progressively constrained at each step with the current output value, and by allowing the model checker to query the function at any step up to the current one.

Model Checker. As already discussed, CRV uses the Kind 2 model checker in the backend for its analysis. An internal translator in CRV, not shown in the architecture in Figure 1, translates the instrumented IR to a system model written in Kind 2’s input language, an extension of Lustre with support for assume-guarantee contracts [8]. CRV then asks Kind 2 to prove the correctness of the top-level component in the model, standing for the entire system, with respect to its contract. Kind 2 returns its results to CRV incrementally as it proves or disproves each top-level (i.e. system-level) guarantee. In turn, CRV converts those results in terms of diagnostic information on the input AADL model and provides it to the user.

Diagnostic Information. The basic level of diagnostic information tells the user if each system-level guarantee is satisfied, violated, or undetermined (because of a timeout). It also provides a counterexample trace for each violated guarantee. The next level includes blame assignment for the violated guarantees and merit assignment for the satisfied ones. We implemented the bulk of the blame and merit assignment functionality directly as an extension of Kind 2, which is written in OCaml. The two features collectively span over 2K lines of OCaml code. For locally optimized blame assignment, we rely on the MaxSMT functionality provided by the Z3 SMT solver [12].

OSATE Plugin. We provide CRV’s functionalities through a plugin developed with our industrial collaborators [32] for OSATE [31], a development and analysis environment for AADL models.

VI. A CASE STUDY ON UNMANNED DELIVERY DRONE

We now discuss a case study on analyzing a realistic model of a hypothetical unmanned delivery drone (UDD).

Goal. The case study had the following objectives: (i) provide evidence that CRV can analyze complex designs; (ii)

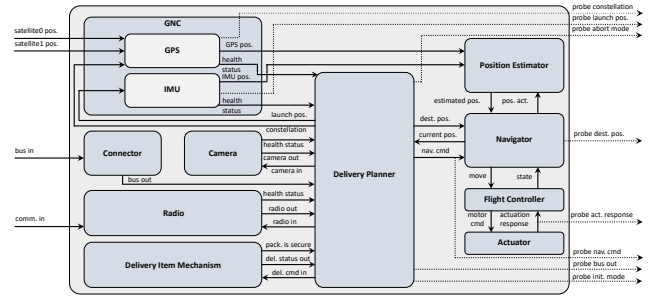


Fig. 7: System architecture of unmanned delivery drone (UDD)

concretely illustrate the responsibility of the human designer in the CRV workflow (i.e., the manual steps); (iii) show the interaction of the system designer with CRV; (iv) assess the value of CRV’s blame and merit assignment features for debugging the design; (v) evaluate CRV’s runtime performance.

High-level UDD system description. The drone is a part of a last-mile delivery unit consisting of a van with packages to be delivered to suburban locations, and one or more delivery drones, also stored in the van. Once the van arrives at a location close to multiple delivery sites, each delivery drone is initialized with its current position and delivery location, and is loaded with the package for that delivery location. Once the drone takes off with the package, it uses inputs from a GPS receiver and an Inertial Measurement Unit (IMU) to navigate to the delivery location. When it reaches the delivery location, the drone uses an on-board camera to capture an image of the delivery site to confirm that the landing location is clear and so it is safe to drop the package. For high-value packages, the delivery drone uses radio communication to get confirmation from the operator in the van. If there are no obstacles in the delivery location and a confirmation (if needed) is received from the operator, the drone’s controller activates a delivery mechanism to drop off the package. The drone then returns to the van for another delivery or storage.

Scenarios. For our case study, we consider two scenarios and 7 functional properties to demonstrate CRV’s effectiveness. As a result, we identified one design weakness/attack for each property. In Sections VI-A through VI-C we focus on one scenario consisting of a single property. The remaining scenarios and properties are presented in Section VI-D.

A. System Architecture of UDD

The system designer first develops the architecture of the UDD, which can be visualized graphically with a diagram like the one in Figure 7. In AADL, this is done by defining the top-level component and each of its subcomponents as individual AADL systems, and then specifying their interface and connections. As an example, here is a specification of the interface of the `DeliveryItemMechanism` system in our model:

```

system DeliveryItemMechanism
features
  delivery_cmd_in: in data port PackageDeliveryCmd;
  delivery_status_out: out data port DeliveryStatus;
  package_is_secure: out data port Boolean;
end DeliveryItemMechanism;

```


For each component, the designer identifies a set of input and output ports used by the component to communicate with its environment, and specifies the type of data exchanged in each port. In addition to basic types, such as `Boolean` and `Integer`, AADL allows the use of user-defined types for ports.

After that, the architect can describe the internal structure of each composite system, by adding a *system implementation* (not shown here) that lists the system's subcomponents and specifies how they are connected together.

B. Design Model of UDD

Next, the designer specifies the behavior of each *leaf* component of the architecture, *i.e.*, a component with no sub-components. This is achieved by associating to it an assume-guarantee contract written in AGREE. Intuitively, assumptions describe the expectations the component has on its inputs and on the global values it has access to, while guarantees describe restrictions on the output values it produces. Behaviorally, each component is a reactive system, instantaneously producing output based on its current input and internal state. Assume-guarantee contracts in AGREE are essentially statements in a linear temporal logic rich enough to precisely describe a component's reactive behavior from the point of view of an observer that, at all times, has access to all the input and output values generated until then. Atomic formulas in this logic are first-order predicates that relate the values of the various ports and intermediate variables. Contracts provide a mechanism for capturing the information needed to specify and reason about component-level safety properties at the desired level of abstraction. Now, let us assume that for the `DeliveryItemMechanism` component the following is known:

- 1) Initially the delivery has *not started*.
- 2) If a delivery command is issued, the delivery status must become different from *not started*.
- 3) If no command is issued or an *abort* command is received, then the delivery status resets to *not started*.

This is a minimal amount of information about the expected behavior of the `DeliveryItemMechanism` component. We explain how to formalize it in AGREE below, using abstract syntax for conciseness.

To formalize (1), we add the following guarantee stating that the delivery status `s` (abbreviating `delivery_status_out`) is equal to `not_started` initially:

$$G_1: s = (\text{not_started} \rightarrow \text{true})$$

The infix initialization operator \rightarrow is an AGREE operator. An expression of the form $e_1 \rightarrow e_2$ evaluates to the value of expression e_1 initially and to the value of e_2 at all later steps of the system's execution. To formalize aspect (2), we add the following guarantee where `c` abbreviates `delivery_cmd_in`:

$$G_2: \text{true} \rightarrow (c = \text{release} \Rightarrow s \neq \text{not_started})$$

Similarly, we capture aspect (3) with this guarantee:

$$G_3: \text{true} \rightarrow (c = \text{no_op} \vee c = \text{abort} \Rightarrow s = \text{not_started})$$

So far, we have only added constraints about the output port `delivery_status_out`. Any system execution that satisfies

those constraints will be considered valid during the analysis performed by CRV.

Desired Functional Requirements. The next step is to review the list of functional (or *safety*) requirements for the system that may affect its integrity, and formalize them as cyber-resiliency properties in AGREE. For instance, suppose we have the following cyber-requirement for `DeliveryDroneSystem` which forbids package delivery to certain locations (while allowing the UDD to still fly-over them):

P7. *The drone will never initiate a packet release in an off-limits location.*

To formalize P7, we have to first identify the components and ports of the system that are relevant to this property. In our example, `DeliveryPlanner` is the component that issues the command to release a package by setting the output port `delivery_cmd` to `release`, while `DeliveryItemMechanism` is the component that receives the command and proceeds with the delivery. Moreover, to know where the drone should release the package, the `DeliveryPlanner` reads the delivery location from the input port `bus_in` through the `Connector` component when the drone is in the van, and then it passes this value to the `Navigation` component. In addition, we also need to know when a location is off-limits. For that, we can define a new predicate `InRA` over locations that evaluates to true if and only if its input location is within a restricted area.

Then we can express the cyber-property with the following top-level guarantee where `dl` is the delivery location:

$$P_7: \text{InRA}(\text{dl}) \Rightarrow s = \text{not_started}$$

Vulnerable Components and Connections. To make the design amenable to CRV's analysis, designers also need to annotate components and links that are vulnerable to attack. For instance, suppose that `DeliveryPlanner` is considered vulnerable to attack (*e.g.*, it is an outsourced software component that has no supply chain security and no tamper protection, and has not been statically analyzed).

C. CRV Analysis

Analysis in the Benign Case. The designer must check that the system design model satisfies its guarantees in the *benign scenario* where no threat models are enabled. For our example, CRV finds an execution that violates P7 because the delivery location information provided as input during initialization is actually an off-limits location. In this case, there are two possibilities: (S_1) the designer decides to prohibit that initially provided delivery locations be off-limits; (S_2) the designer decides to treat initialization with an off-limit location as a realistic possibility, perhaps as a consequence of malicious code in the (external) software (in the van) that provides initialization values for the UDD system. Assume (S_2) cannot happen in the benign scenario. Then the designer adds the following assumption to the contract of `DeliveryDroneSystem` to capture (S_1) where `tl` is the target location provided through the system's input bus:

$$A_1: \neg \text{InRA}(\text{tl})$$

In this case, with no threats enabled, CRV is able to prove property P7 valid.

Analysis under Threat Effects. After verifying with CRV that P7 holds in the benign case, the system designer can run an adversarial analysis. In this case, CRV will find a violation of property P7 that involves an attack to the UDD's delivery planner. From the blame assignment diagnostics, we observe that the possible violation may result from an attack on the `DeliveryPlanner` component that causes it to violate its contract. The blame assignment analysis additionally identifies a minimal set of ports, `dest_location` and `delivery_cmd`, that is enough to compromise to carry out the attack successfully. One can also examine the counterexample trace leading to the violation of the property and observe that it occurs when the vulnerable `DeliveryPlanner` maliciously instructs the `DeliveryItemMechanism` component to initiate the release of the package while the drone is flying over an off-limits location.

Analysis after Mitigation. After CRV presents a new attack, the designer can see how to address its root cause by considering the vulnerable components and ports relevant to the attack. Then, they can run another *what-if* analysis by considering a situation where additional security measures have been introduced to make some of the vulnerable components more cyber-resilient. For example, suppose the designer sees the `DeliveryPlanner` component as a candidate for enhanced cyber-resiliency measures. When the component is labeled as invulnerable to attack, a new analysis of the system confirms that this fix is sufficient to rule out any attacks compromising property P7. Moreover, the merit assignment post-analysis reassures the user that the change indeed plays a role in the satisfaction of P7 in an adversarial environment.

Performance. The AADL+AGREE model for the UDD system in this case study consists of around 1800 lines of specs, and includes 7 functional properties as top-level guarantees. On average, each call to the tool, which triggers the threat instrumentation, the verification of the properties, and the merit/blame assignment analysis, takes 30s on a 1.10GHz Intel(R) Core i7-10710U CPU machine with 16GB of RAM.

D. Details on Case Study Experiments

We used CRV to analyze the cyber-resiliency of the UDD system with respect to two sets of system-level safety properties. The first set consists in the following five properties:

```

guarantee "P1: When the drone is switched on,
the GPS component uses the constellation
received most recently":
  isOn =>
    most_recent_constellation = probe_constellation;

guarantee "P2: Launch location for IMU is
initialized properly":
  isOn =>
    most_recent_launch_loc = probe_launch_loc;

guarantee "P3: Delivery location for navigation
is initialized properly":
  isOn =>
    most_recent_delivery_loc = probe_delivery_loc;

```

```

guarantee "P4: A command to release a valuable
package is issued only if drone has received
confirmation from base":
  release_cmd and valuable_package =>
    target_confirmed;

```

```

guarantee "P5: The drone will always request
confirmation from base before starting delivery
of a valuable package":
  delivery_started and valuable_package =>
    confirmation_requested;

```

First, we checked the system model satisfies the five properties without considering the effects of any threat model. CRV was able to prove that in 9s. Then, we analyzed the cyber-resiliency of the system against all threats in the CRV library. As a result, CRV was able to find in less than 5s four network injection attacks on `bus1` leading to the violation of properties P1, P2, P3, and P4, and one logic bomb attack on the `DeliveryPlanner` causing the violation of property P5. Note this is just one possible blame assignment result, the model admits other minimal results. After reviewing the results, we considered a scenario where the four network injection attacks were not possible because the `bus1` connection was a *trusted* connection, and the logic bomb attack was not feasible anymore after the decision to develop the `DeliveryPlanner` internally instead of outsourcing it. To reflect the new scenario we changed the meta-level attribute `connectionType` of the `bus1` connection from `Untrusted` to `Trusted`, and the meta-level attribute `pedigree` of the `DeliveryPlanner` component instance from `Sourced` to `InternallyDeveloped`. After the change, we could check that no more new attacks were possible by analyzing again the modified model against all threats in the CRV library. This time CRV proved all five properties valid in 20s.

The second set consists in the following two properties:

```

guarantee "P6: The drone issues a command to
release a package only if the delivery location
is the most recent delivery location provided":
  release_cmd =>
    probe_delivery_loc = most_recent_delivery_loc;

guarantee "P7: The drone never initiates a package
release to an off-limits location":
  delivery_status <> NOT_STARTED =>
    not InRestrictedArea(probe_delivery_loc);

```

In this new scenario, we considered the values of the meta-level attributes for the `bus1` connection and the `DeliveryPlanner` component instance to be the original ones. Again, we started checking that properties P6 and P7 were satisfied by the system, which CRV confirmed after 8s. Then, we focused on analyzing the cyber-resiliency of the system with respect to property P7. Similarly to the first scenario, CRV was able to detect in 3s a logic bomb attack on the `DeliveryPlanner` that leads to the violation of P7. We chose to neutralize the attack by implementing a runtime monitor, as explained in Section VI, instead of enforcing the internal development of the component. After integrating the behavioral defense in the model, CRV could prove the satisfaction of property P7 in 3s. The next step was to analyze

the cyber-resiliency of the system with respect to property P6. In 3s, CRV confirmed that a logic bomb attack on the `DeliveryPlanner` would still be able to falsify property P6. In this case, we decided to apply a hybrid solution. First, we forced the `DeliveryPlanner` component to be internally developed to prevent logic bomb attacks. Then, we changed the model design to incorporate a MAC protection, to make the system cyber-resilient to network injection attacks on the `bus1` connection. After those changes, CRV could prove the satisfaction of property P6 and P7 in 168s.

VII. RELATED WORK

There are at least four relevant lines of work that analyze system designs for security threats. They are based respectively on: general model checkers, cryptographic protocol verifiers, fault analysis tools, and specialized analyzers. We highlight the main differences between CRV and each of these classes of tools.

A. Model Checkers

Model checkers used for cyber-security generally perform analysis at one of two granularities: system-level and component-level. The former considers only system-level inputs to be adversarially-controlled, overlooking cases where the integrity of individual components is violated. The latter considers inputs of an individual component to be adversarially-controlled, letting one mimic the integrity violation of that component. To lift the analysis from individual components to the whole system, manual efforts are needed to recompose component-level guarantees into global/system-level properties. Correspondingly, component-level counterexamples have to be lifted to the system-level in order to construct a system-wide attack. In contrast, the results obtained from the analysis of the threat-instrumented model in CRV can be automatically interpreted at the system level.

Instrumented versions of a system model are used in traditional fault analysis to study the system’s behavior in the presence of faults. The Safety Annex for AADL [30] allows one to specify the behavior of systems and components in the presence of faults. The tool supports the computation of *all* minimal cut sets, but not the direct computation of an individual solution. Similarly, the xSAP [4] platform offers library-based specification of faults, automatic model-extension with fault specifications, and the generation of minimal cut sets. However, its primary emphasis also lies in fault analysis rather than security. Finally, all these techniques focus on the globally optimized setting. To our knowledge, the use of the locally optimized setting [24] is novel.

B. Cryptographic Protocol Verifiers

Cryptographic protocol verifiers (CPVs) such as Tamarin [26] and ProVerif [5], [22], [2] can reason about cryptographic constructs and support confidentiality properties (e.g., observational equivalence), neither of which are currently supported by CRV. In contrast, CRV has the following advantages: (i) it can support rich system descriptions with linear (integer and

real) arithmetic constraints and temporal constraints, which are not supported by current CPVs; (ii) it can provide diagnostic information in the form of blame and merit assignment at the end of the analysis, which is unavailable in CPVs; (iii) it can analyze rich stateful systems more scalably than current CPVs; (iv) it supports automated analysis of rich safety properties beyond what is supported by tools such as ProVerif [5] or their extensions [22], [2]; (v) thanks to compositional verification, it can support the analysis of large systems that are not amenable to analysis by CPVs. In a sense, CRV and CPVs are complementary. One way to support confidentiality properties and cryptographic constructs in CRV’s workflow would be by integrating a CPV in it.

C. Fault analysis tools

Fault analysis tools, especially the ones that consider Byzantine faults, are the closest in spirit to the CRV work. However, they *generally* are neither amenable to seamless integration with CPV (due to the lack of support for replay attackers), which is needed to analyze rich properties of systems containing cryptographic constructs, nor do they support meta-level diagnostic analyses.

D. Specialized analyzers

These analyzers focus on analyzing specific protocols in a particular domain (e.g., cellular networks [19], [20], [3], TCP/IP [6], WiFi [33]), a very limited set of security properties (e.g., non-interference, cache side channel, transient execution vulnerabilities), or particular systems (e.g., IoT [7]). Among these efforts, the closest to our approach are LTEInspector [19] and 5GReasoner [20] where the Dolev-Yao adversary model is used to perturb the public communication between two components when model checking the protocol under analysis. Other tools such as ThreatGet [10] only analyze systems at the architectural level with a pre-defined set of threats. Besides the restriction to specific domains, none of the prior work is capable of statically analyzing reactive system designs with respect to integrity properties — the focus of CRV’s analysis.

VIII. CONCLUSION

We have presented CRV, a general approach and tool to statically check the cyber-resiliency of a design against current and *future* integrity attacks. A case study with an unmanned delivery drone system demonstrates that CRV can analyze effectively the cyber-resiliency of complex designs with respect different integrity properties.

Possible future directions of research include enhancing CRV’s capability to support a limited form of *availability properties*, which can be formalized as liveness properties, and *confidentiality properties*, formalizable as non-interference properties. This would enable CRV to support the analysis of functional properties whose violation require a combination of integrity, availability, and confidentiality attacks. Additionally, CRV could be extended to integrate a CPV and hence consider cryptographic constructs.

ACKNOWLEDGMENTS

This work was partially supported by DARPA grant #N66001-18-C-4006 and by the US Air Force Research Lab.

REFERENCES

- [1] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund. Designing safe, reliable systems using scade. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, First International Symposium, ISOla 2004, Paphos, Cyprus, October 30 - November 2, 2004, Revised Selected Papers*, volume 4313 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2004.
- [2] Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark D. Ryan. Statverif: Verification of stateful processes. *Journal of Computer Security*, 22(5):743–821, July 2014.
- [3] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, page 1383–1396, New York, NY, USA, 2018. Association for Computing Machinery.
- [4] Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xsap safety analysis platform. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 533–539. Springer, 2016.
- [5] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Found. Trends Priv. Secur.*, 1(1–2):1–135, October 2016.
- [6] Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, and Paul Yu. Principled unearthing of tcp side channel vulnerabilities. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, page 211–224, New York, NY, USA, 2019. Association for Computing Machinery.
- [7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 147–158, 2018.
- [8] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. CoCoSpec: A mode-aware contract language for reactive systems. In Rocco De Nicola and Eva Kühn, editors, *Proceedings of the 8th International Conference on Software Engineering and Formal Methods, Vienna, Austria*, volume 9763 of *Lecture Notes in Computer Science*, pages 347–366. Springer, 2016.
- [9] Adrien Champion, Alain Mebsout, Christoph Stickels, and Cesare Tinelli. The Kind 2 model checker. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 510–517. Springer, 2016.
- [10] Sebastian Chlup, Korbinian Christl, Christoph Schmittner, Abdelkader Magdy Shaaban, Stefan Schauer, and Martin Latzenhofer. THREAT-GET: towards automated attack tree analysis for automotive cybersecurity. *Inf.*, 14(1):14, 2023.
- [11] Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, pages 126–140, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [12] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [13] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
- [14] Michael Durling, Heber Herencia-zapana, John Interrante, Baoluo Meng, Abha Moitra, Kit Siu, Vidhya Tekken Valapil, Daniel Prince, Cesare Tinelli, Omar Chowdhury, Daniel Larraz, Moosa Yahyazadeh, and Fareed Arif. DARPA: Cyber Assured Systems Engineering (CASE) — VERDICT Project, 2020. Available at <https://github.com/ge-high-assurance/VERDICT/wiki>.
- [15] Mitziu Echeverria, Zeeshan Ahmed, Bincheng Wang, M. Fareed Arif, Syed Rafiul Hussain, and Omar Chowdhury. PHOENIX: device-centric cellular network protocol monitoring using runtime verification. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.
- [16] P. H. Feiler, B. A. Lewis, and S. Vestal. The sae architecture analysis design language (aadl) a standard for engineering performance critical systems. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1206–1211, Oct 2006.
- [17] Elaheh Ghassabani, Andrew Gacek, and Michael W Whalen. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 314–325, 2016.
- [18] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
- [19] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society. `tex.ids:hussainLTEInspectorSystematicApproach2018a`.
- [20] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, page 669–684, New York, NY, USA, 2019. Association for Computing Machinery.
- [21] Software Engineering Institute. AADL – Architecture Analysis and Design Language. <http://aadl.info>. Accessed: May 20, 2023.
- [22] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. *Verifpal: Cryptographic Protocol Analysis for the Real World*, page 159. Association for Computing Machinery, New York, NY, USA, 2020.
- [23] Daniel Larraz, Mickaël Laurent, and Cesare Tinelli. Merit and blame assignment with Kind 2. In Alberto Lluch-Lafuente and Anastasia Mavridou, editors, *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings*, volume 12863 of *Lecture Notes in Computer Science*, pages 212–220. Springer, 2021.
- [24] Daniel Larraz and Cesare Tinelli. Finding locally smallest cut sets using max-smt. *ACM SIGAda Ada Letters*, 42(2):32–39, Apr 2023.
- [25] Daniel Larraz, Arjun Viswanathan, Cesare Tinelli, and Mickaël Laurent. Beyond model checking of idealized Lustre in the Kind 2. *ACM SIGAda Ada Letters*, 42(2):40–44, Apr 2023.
- [26] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [27] Baoluo Meng, Daniel Larraz, Kit Siu, Abha Moitra, John Interrante, William Smith, Saswata Paul, Daniel Prince, Heber Herencia-Zapana, M. Fareed Arif, Moosa Yahyazadeh, Vidhya Tekken Valapil, Michael Durling, Cesare Tinelli, and Omar Chowdhury. VERDICT: A language and framework for engineering cyber resilient and safe systems. *Systems*, 9(1), 2021.
- [28] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. Technical report, National Institute of Standards and Technology, 2020.
- [29] SolarWinds. Solarwinds security advisory. Available at <https://www.solarwinds.com/securityadvisory>.
- [30] Danielle Stewart, Jing Janet Liu, Michael W Whalen, Darren Cofer, and Michael Peterson. Safety annex for the architecture analysis and design language. In *10th European Conference Embedded Real Time Systems ERTS*, 2020.
- [31] OSATE team. OSATE – Open Source AADL Tool Environment. <https://osate.org>. Accessed: May 20, 2023.

- [32] VERDICT team. VERDICT – Verification Evidence and Resilient Design in Anticipation of Cybersecurity Threats. <https://github.com/ge-high-assurance/VERDICT>. Accessed: May 20, 2023.
- [33] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

Towards a Correct-by-Construction Design of Integrated Modular Avionics

Baoluo Meng^{}, Joyanta Debnath^{}, Sarat Chandra Varanasi^{},
Emmanuel Manolios, Michael Durling, Saswata Paul^{}

General Electric Research
Niskayuna, NY 12309, USA

Email: {baoluo.meng, joyanta.debnath, saratchandra.varanasi, emmanuel.manolios, durling, saswata.paul}@ge.com

Daniel Prince, Saif Alsabbagh, Richard Haadsma, Craig McMillan
GE Aviation Systems
Grand Rapids, MI 49512, USA

Email: {daniel.prince, saif.alsabbagh, richard.haadsma, craig.mcmillan}@ge.com

Chi Zhang, Tim Oates
University of Maryland, Baltimore County
Baltimore, MD 21250, USA

Email: {chzhang1, oates}@umbc.edu

Abstract—This paper presents a formal language and framework, OYSTER, to develop correct-by-construction design of Integrated Modular Avionics (IMA). The OYSTER language is created as an annex to the Architecture Analysis and Design Language (AADL) for encoding constraints for aspects of IMA. The OYSTER constraints involve determining the correct locations of hosted applications within an IMA system, validity of the port connections involved in the design, and the conformance of virtual links allocated with bandwidth and jitter requirements. OYSTER also allows synthesis of communication paths for the allocated virtual links. The OYSTER prototype tool is developed as a plugin to the Open Source AADL Tool Environment (OSATE), and invokes Satisfiability Modulo Theories (SMT) solvers to synthesize correct-by-construction architecture designs for IMA. In addition, behaviors of applications running on IMA components and their safety properties can be modeled in the Assume Guarantee REasoning Environment (AGREE) annex and checked by the Kind2 model checker. The verification results are guaranteed to be correct by the independently verifiable proof certificates produced by Kind2. Finally, the paper evaluates OYSTER on a GE Aviation use case – a fuel control system IMA, and discusses the lessons learned.

I. INTRODUCTION

Integrated Modular Avionics (IMA) [36] are hybrid platforms that provide computing, communication, and I/O services for modern military and commercial aircraft. The implemented real-time embedded systems are architected and overlaid on the partitioned platform resources to form a highly-integrated system with full isolation and independence of each individual system. The platform elements are architected to maintain a high-integrity, fault-tolerant environment necessary for hosting critical system functionality. IMAs are able to simultaneously support critical and non-critical applications (at both high and low integrity levels) due to partitioned boundary layers between the applications.

Since the failure of IMA systems can have catastrophic consequences, the development of IMA platforms for modern commercial and military aircraft involves rigorous processes and tools along with tedious manual work to ensure that no

errors are introduced. Therefore, IMA solutions are oftentimes produced by commercially available and/or internally developed proprietary tools, many of which have been certified for use by regulatory authorities such as the Federal Aviation Administration (FAA) and European Aviation Safety Agency (EASA). However, this makes IMA architecture solutions expensive to implement and changes to an IMA design (e.g., requirements changes) can be labor-intensive.

During a typical IMA development cycle today, a systems integrator will spend thousands of person-hours collecting, integrating, and fine-tuning an overall IMA system for qualification and fielding on real aircraft. One of the main contributors to the cost of integration of these network-based systems is the management of their changes and impacts of the changes on the rest of the IMA. Doing so requires systems expertise, knowledge about the implementation details of the IMA construction, and operational details of the qualified verification and configuration tools that must be used. The frequency of changes during IMA development can be very high and can even get to a point where it may become impractical/unsustainable for a human integrator to be able to absorb and understand all of the changes and their potential impacts. Additionally, the financial and scheduling constraints usually do not allow for a full “stop work” to assess the changes every time. Therefore, the access to tools that can aid in decision-making and can recommend appropriate changes for converging on a qualifiable and fieldable solution is crucial when dealing with systems of this scale and complexity.

Formal methods are mathematically-rigorous means for the specification, development, and verification of software and hardware systems, that can be harnessed for ensuring error-free system integration. Techniques such as model checking [17] and Satisfiability Modulo Theories (SMT) [9] can be used for automatically detecting if a given architecture violates a given property and for synthesizing potential changes in an architecture that might be needed in order to satisfy a given property. Therefore, such formal methods techniques

are well-suited for the development of IMA solutions. In this paper, we introduce the OYSTER framework, which uses model checking and SMT-based techniques for the synthesis of *correct-by-construction* IMA architectures. We have evaluated the feasibility of using OYSTER on real-life industrial applications by applying it on an IMA use case provided by our industry partners at GE Aviation.

This paper makes the following contributions to aid in the development of IMA solutions:

- Development of a formal language namely, OYSTER, to encode IMA architecture constraints and a translation scheme to SMT.
- An end-to-end tool prototype to synthesize a correct-by-construction IMA architecture solution using SMT solver and model checkers.
- A framework prototype for generating *independently verifiable* solutions for behavior models towards certification.
- Evaluation of the OYSTER framework on a fuel control system IMA use case provided by GE Aviation to show its practical feasibility.

The rest of the paper is structured as follows: Section II provides an overview of the overall architecture of the OYSTER framework; Section III details the OYSTER language for encoding IMA requirements; Section IV describes the architecture synthesis and proof generation capabilities of OYSTER; Section V presents an application of OYSTER on our use case followed by a discussion about lessons learned during the process; Section VI summarizes related work on system architectures synthesis; and finally, Section VII concludes the paper with a discussion on possible directions of future work.

II. OYSTER OVERALL ARCHITECTURE

The OYSTER framework is depicted in Figure 1. The goal of this framework is to leverage formal methods tools to auto-synthesize a correct-by-construction IMA platform architecture, and prove formal properties about IMA behaviors using model checkers. The framework starts with modeling IMA software and hardware components in Architecture Analysis and Design Language (AADL) [23] that will be used to construct an IMA platform (corresponding to ①). The IMA architectural requirements are collected and encoded in OYSTER annex (②). The two pieces of information are translated to input to Satisfiability Modulo Theories (SMT) solvers for architecture synthesis (③). A satisfiable solution from the solver is converted back to an instantiated AADL model (④) that meets all the constraints specified in OYSTER. In addition, the model behavior and safety properties (⑤) can be manually added to the synthesized model to be further checked by a model checker – Kind2 [16] (⑥). In case the formal properties are proved valid, Kind2 (⑦) will produce three kinds of proof certificates in SMT-LIB [8] and Logic Framework with Side Condition (LFSC) [39] formats (⑧). In case the formal properties are disproved, we will leverage the counter-example and the blame assignment feature [28] of Kind2 to help system engineers to localize the issues.

The first one certifies the front-end translation faithfulness, which is in SMT-LIB format. It requires an independently developed model checker for Lustre, e.g., JKind by Collins Aerospace [25], to be part of the certificate generation process. The second one encodes the k -inductive proof steps in SMT-LIB format for the validity of formal properties and can be independently verified by third-party SMT solvers, e.g., Z3 [33], cvc5 [7], Yices2 [22] etc. Finally, the LFSC proof certificate is a formal proof that the safety properties are invariants in the system, and can be independently verified by LFSC proof checkers (⑨).

The framework enables a system architect/developer to use SMT techniques to auto-generate system architecture models and utilize a back-end model checker (Kind2) to produce proof certificates from verification of safety properties of system behavior models.

III. OVERVIEW OF IMA REQUIREMENTS & OYSTER ANNEX

IMA architectural requirements state several constraints about the location of components on the IMA cabinets along with network connectivity and bandwidth constraints between several several components situated within the cabinets. Moreover, Virtual Link flows specific information flows from various sensors to actuators. Initially, all the components present in the cabinet are stated along with their port connections and virtual link flows as the OYSTER AADL annex [3].

Essentially, annexes enable descriptions of Domain Specific Languages extending the basic AADL definitions. In our case, the OYSTER annex captures the IMA requirements. OYSTER is a front-end AADL annex language to capture these architectural constraints. Then the actual synthesis is performed by translating the OYSTER annex into SMT and invoking an SMT-solver. The solution returned by the SMT-solver is translated back to AADL which represents the synthesized IMA Architecture. If the set of OYSTER annex constraints are unsatisfiable, we report the UNSAT core from Z3 back to the user. Although UNSAT cores from SMT solvers are not necessarily unique or minimal, reporting it back to the user serves as an initial step towards providing actionable feedback for the user while specifying IMA requirements. The synthesized AADL architecture is subject to further behavior analyses as mentioned in the OYSTER toolchain workflow.

The OYSTER language supports modeling the following constraints: fixed-location constraints (FLCs), co-location constraints (CLCs), resource utilization constraints (UCs), separation constraints (SCs), virtual link constraints (VLCs), and port connection constraints (PCCs). The OYSTER language is also equipped with syntax highlighting and type checking for usability. The OYSTER language and its informal semantics is presented next.

Fixed-location Constraints. FLCs constrain a component X to map to another component Y within the IMA architecture. For example, a General Processing Module (GPM) `GPM_L1` is mapped to a Common Computing Resource (CCR) `CCR_L1` as:

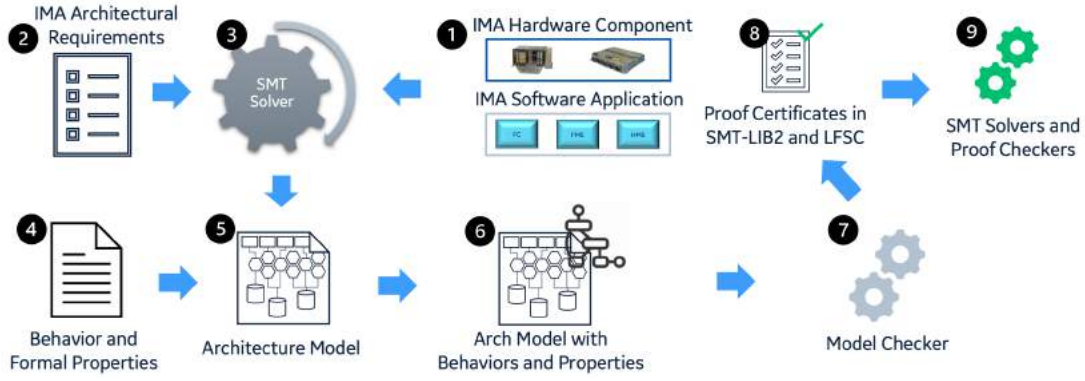


Fig. 1. High-level overview of OYSTER framework

```
{Fixed-Location-Constraint FLC1: (GPM_L1 -> CCR_L1);
```

Co-location Constraints. CLCs co-locate components $\{C_1, \dots, C_k\}$ within a same target component C_τ . For example, a GPMApp and ACSApp can be co-located to CCR_L1 as:

```
Co-Location-Constraint CLC1 :
(GPMApp APP_FIDO) and (ACSApp SwitchApp_L1) -> CCR_L1;
```

Utilization Constraints. UCs state that the resources allocated to the various hosted applications (CPU, RAM, ROM) should not exceed the resources available on a computing resource. For example, in OYSTER, one could state that the sum of CPU allocated to applications named APP_FIDO and APP_FILE_SYSTEM should not exceed the CPU provided for the computing resource CCR_L1, as shown below.

```
Utilization-Constraint UC1 [CPU]:
(CCR CCR_L1: cpuProvided) > (GPMApp APP_FIDO: cpuUsed) +
-> (GPMApp APP_FILE_SYSTEM: cpuUsed);
```

Separation Constraints. Separation constraints specify that a given set of components shall not be mapped to same components. The following separation constraint states that the applications APP_FUEL_SYSTEM_CONTROL, APP_FIDO and APP_FILE_SYSTEM should be hosted on different GPM component.

```
Separation-Constraint SC1:
(GPMApp APP_FUEL_SYSTEM_CONTROL, APP_FIDO,
-> APP_FILE_SYSTEM) -> distinct GPM;
```

Virtual Link Constraints. A virtual link constraint (VLC) defines both unicast and multicast flows of a virtual link. All the flows in a virtual link can have only one source publisher, but may have one or more destination subscribers. In addition, a VLC constraint allows users to specify a set of messages for each flow in the virtual link. Each of these sets are separated by a “#”. A message in a VLC is represented as MessageSize@RefreshPeriod.

```
-- Message size unit = byte, Refresh period unit = msec
Virtual-Link-Constraint VL1: (App1 ~> App2, App3)
-> {12@1000} # {12@1000, 12@1000};
Virtual-Link-Constraint VL2: (App4 ~> App5) {20@80};
```

Port Connection Constraints. PCCs specify physical bidirectional connections between two components.

```
-- GPM <=> ACS connections bandwidth unit = byte
Port-Connection-Constraint PCC1: (GPM_L1.portA <->
-> ACS_L1.port1) 1000000000; -- 1 Gigabyte
```

IV. IMA SYNTHESIS & PROOF CERTIFICATES GENERATION

The goal of IMA synthesis is to automatically synthesize an IMA architecture that satisfies all the constraints encoded in the OYSTER annex. The inputs to the synthesis task comprise of an AADL model annotated with AADL properties along with OYSTER constraints. The inputs are then translated to SMT-LIB for constraint solving. In our case, OYSTER uses the Z3 SMT solver. A satisfiable solution from Z3 is then automatically translated to an AADL model containing detailed AADL implementations respecting component locations, their port connections, and virtual link flows satisfying required OYSTER constraints. The OYSTER toolchain provides a plethora of options and toggles for the user to either check or uncheck for Virtual Link Synthesis (feasible or optimal solution), check the Virtual Links’ Network Bandwidth Utilization and also the capability to schedule GPM Applications hosted on a designated GPM Processor.

A. From OYSTER Annex to SMT

The components are categorized by their avionics types and declared as SMT enumerated types. For instance, we declare enumerate types *ACS* and *GPM* in SMT for the Avionics Cabinet Switch (ACS) and General Processing Module (GPM) respectively.

Fixed-Location Constraints. FLCs are translated to uninterpreted functions. For an FLC, $GPM_L1 \rightarrow CCR_L1$, An uninterpreted function $gpm_to_ccr : GPM \rightarrow CCR$ is declared and an assertion will be declared: $gpm_to_ccr(gpm_l1) = ccr_l1$.

Co-Location Constraints. For each component, C_k that is mapped to a target component C_τ , a function $f_{(C_k, C_\tau)}$ is declared and its type is $(Type_{C_k} \rightarrow Type_{C_\tau})$. The co-location of two components C_i, C_j itself to C_τ is declared as a constraint asserting the equality of $f_{(C_i, C_\tau)}$ and $f_{(C_j, C_\tau)}$.

Separation Constraints. Separation Constraint is the dual of Co-location constraint. For components, C_i, C_j to be separated with respect to a target component C_τ . The entire process is the same as that of Co-location constraint, but for the last step where we state $f_{(C_i, C_\tau)} \neq f_{(C_j, C_\tau)}$.

Utilization Constraints. The utilization constraints state that the computing resources provided to the CCR are sufficient to the usage needs of hosted applications. We encode an uninterpreted function of type $(Type_C \times Type_R \rightarrow Int)$ for each component – resource pair, and assert that the sum of the resources used meets the resources provided.

Port Connection Constraints. A port connection constraint c , from $portA$ to $portB$ is represented by declaring a function of type $(Port \times Connection) \rightarrow Port$, where the sort *Connection* is used to capture the name of the connection itself, from $portA$ to $portB$. Then, the definition is instantiated as an assertion: $port_connection_port(portA) = portB$.

Virtual Link Constraints. There is a Virtual Link Constraint between a source s and a destination t , with a refresh rate of r and message size m . Then, Virtual Link Constraints induce multiple sets of SMT constraints, with each set constraining the desired Virtual Link specification in the IMA System. Each such criteria and their associated constraints are described:

- **Path Constraints.** The port connections involved in the Virtual Link from s^i and t^i need to be synthesized for a virtual link i . This synthesis can be formulated as a connection selection problem. The set of connections selected constitute the path between s^i and t^i . Each connection c_{xy} represents a port connection between component x and component y . If a connection c_{xy} is selected, then it should be assigned a weight of 1, otherwise a weight of 0. Furthermore, the sum of all outgoing connections c_{sk} from the source s^i to k must be 1, to enforce only one outgoing flow. The sum of all connections involving an intermediate component not in s, t must equal 2, to enforce one incoming flow and one outgoing flow. Finally, the sum of all incoming connections to c_{kt} must equal 1, to enforce only one incoming flow. Formally,

$$\sum_{l=1}^{outdeg(s)} c_{sl}^i = \sum_{l=1}^{indeg(t)} c_{lt}^i = 1$$

$$\sum_{l=1}^{indeg(x)} c_{lx}^i + \sum_{l=1}^{outdeg(x)} c_{xl}^i = 2 \text{ for } x \neq s, x \neq t$$

The shortest path can be selected by optimizing over the sum of weights of all connections in a path from s^i to t^i while satisfying the above constraints. The optimization is performed using MaxSMT solving capabilities of Z3 [13].

- **Bandwidth Constraints.** The virtual links specified in the OYSTER annex must also adhere to the bandwidth constraints on the Aircraft Data Network. The constraints and concepts are defined the ARINC 664 specification

[4]. And two important parameters need to be synthesized for each virtual link: Bandwidth Allocation Gap (BAG), Maximum Transmission Unit (MTU). The BAG represents the minimum interval between frames on the virtual link. The MTU represents the largest size of data packet in a single frame that can be transmitted over a network connection. They should also satisfy BAG, MTU, and the jitter constraints. The complete set of ARINC 664 constraints can be found elsewhere [4]. To ensure the paper remains self-contained, we incorporate the summarized formulas from the literature [6]. All the constraints involved are linear constraints over integers and can be straightforwardly encoded in SMT-LIB. For virtual link i , BAG^i denotes its BAG, n^i represents total number of messages in i (indexed from 1 to n^i), s_j^i the message size of j^{th} message, p_j^i the refresh period for the j^{th} message and MTU^i its MTU. Let B denote the bandwidth of the entire network. Then, the following constraints need to satisfied:

Real-time Constraints on Messages:

$$\sum_{j=1}^{n^i} \frac{\lceil s_j^i / MTU^i \rceil}{p_j^i} \leq 1 / BAG^i$$

Bandwidth Constraints:

$$8 * \sum_{i=1}^n \frac{(MTU^i + 67)}{BAG^i} * 10^3 \leq B$$

Jitter Constraints:

$$40 + 8 * \sum_{i=1}^n \frac{MTU^i + 67}{B} \leq 500$$

General BAG and MTU Constraints:

$$BAG^i \in \{2^k | k \in \mathbb{N} \wedge 1 \leq k \leq 7\}$$

$$MTU^i \in \mathbb{N} \wedge 1 \leq MTU^i \leq 1471$$

GPM Applications Scheduling. The OYSTER toolchain can also generate static schedules for applications hosted on GPM using SMT solvers. Four important characteristics are associated with applications: start time, duration, period, and priority. The start time denotes when to execute an application. Duration defines the time taken to execute an application. Period refers to the frequency of execution of the application. Priority indicates the order to execute an application relative to the other applications. The input to the scheduling problem is the priority, duration and period of applications. The output is the start time for each application so that the priorities are respected and no duration overlaps. The inputs for GPM application scheduling are captured in AADL as an OYSTER property and annotated against GPM applications specified within the IMA system.

We first define the schedulability condition for a pair of applications i, j , followed by constraints involved in the

scheduling problem, where $GCD(x, y)$ denotes the greatest common divisor of integers x, y .

$$sched(i, j) \triangleq start_j > start_i \wedge duration_i \leq start_j - start_i \leq GCD(period_i, period_j) - duration_j$$

- No scheduling conflicts:

$$\forall i, j \ start_i \geq 0 \wedge start_j \geq 0 \implies sched(i, j)$$

- High priority apps start early:

$$\forall i, j \ priority_i < priority_j \implies start_i < start_j$$

- Applications start times are all distinct:

$$\forall i, j \ i \neq j \implies start_i \neq start_j$$

- Every application must be scheduled: $\forall i \ start_i \geq 0$

UNSAT Cores and Feedback When the constraints specified in the OYSTER annex are unsatisfiable, OYSTER toolchain computes the UNSAT core using Z3 and reports the unsatisfiable constraints at a high level. If the location constraints are unsatisfiable, then OYSTER recommends the developer to check all fixed-location, separation and co-location constraints that may be inconsistent with each other. UNSAT cores concerning utilization constraints are straightforwardly reported recommending the developer to check the resources allocated (CPU, Memory) against the resources being used. UNSAT cores for virtual links often involving exceeding the maximum bandwidth allocation. In such a case, OYSTER recommends to the developer to either increase the maximum allocated bandwidth or to reduce the number of virtual links allocated. For GPM Application the conflicts in schedulability between pairs of GPM apps are reported back to the developer.

B. Proof Certificates Generation by Model Checking

Another feature of OYSTER is to enhance the synthesized architecture with behaviors and safety properties in the Assume Guarantee REasoning Environment (AGREE) [18] annex of AADL. It utilizes the Kind2 model checker to prove whether the model satisfies the safety properties. In case the properties are proved valid, accompanying proof certificates will be generated by Kind2 and can be independently verified by third party tools to ensure the correctness of the results. Different behavior aspects of IMA such as application execution schedule and latency analysis can be encoded in this framework. In this work, we consider the execution schedule of applications on a GPM. The schedule defines the start time, priority and duration of each task. It is essential for ensuring that the system operates efficiently and effectively, as it helps to resolve conflicts between different applications and functions, prevent overloading of resources, and optimize the use of processing power and memory. We have modeled the application execution schedule as a behavior model in AGREE annex [2]. The formal properties of interests are that pair-wise applications shall not have any conflicts. To further increase the stakeholders' confidence in the correctness of the IMA solutions, we introduce additional model checking layer to ensure the correctness of schedules by the proof certificates generated by the model checker.

V. EVALUATION

To demonstrate the capabilities of the OYSTER tool¹, GE Aviation developed a smaller-scale (yet fully-defined) IMA Architecture for a rotorcraft air vehicle. One of the major avionics systems of this architecture is the Fuel Control System, which we have chosen as the basis of the OYSTER use case. More specifically, the focus is on the IMA aspects of the architecture that host this Fuel Control System ARINC 653 Application and gateway its data throughout the Aircraft Data Network (ADN). The Fuel Control Application is responsible for managing various aspects of the Fuel Control System of an aircraft. These functions include pumping of fuel, delivering fuel to the engines, monitoring fuel flow, etc. This application allows the flight crew to control fuel tank selection, shutoff valve functions, and the main and standby pumps. It also provides monitoring and reporting of fuel system characteristics such as fuel quantity, temperature, and pressure. A notional diagram of the IMA fuel system control application use case can be found elsewhere [1].

The OYSTER toolchain was developed as a plugin for the Open Source AADL Tool Environment (OSATE) [15]. The IMA use case involves 43 IMA components that also involves 5 virtual links, and 6 applications to be scheduled on a particular GPM (on GPM_R2). The total number of OYSTER constraints are in the order of hundred constraints. Configuring them manually can be a challenging task. OYSTER makes it easier to specify these constraints and uses formal methods tools to synthesize correct-by-construction solutions. The IMA architecture synthesis takes 7.751s. The schedule synthesized for our use case was encoded in AGREE annex to simulate the execution of schedules. We consider 15 formal properties for 6 applications, and the entire process for proving properties takes 6.441s. However, with proof certificates generation, the process takes 33.196s, which is expected because proofs generation is expensive. We have also successfully validated the correctness of the proof certificates by running Z3, cvc5, and LFSC checker. The performance evaluation indicates that OYSTER is usable practically. We leveraged a model checker to generate proof certificates to ensure formal properties of the behavior model are indeed valid. For our example use case (application schedules), the model checking scales well (proved all properties in under 3 mins) to a typical number of applications (10 apps) that one would expect in practical IMA systems. All experiments were conducted by running OYSTER on an Intel(R) Core(TM) i7 CPU @ 2.9GHz Processor with 4 cores and 16 GB RAM running macOS Ventura (Version 13.1) .

A. Lessons Learned

For any aircraft development program, the airworthiness of the entire aircraft must be established through a rigorous certification process. This extends not just to the airframe itself, but also to the computing systems installed on the aircraft (i.e. the IMA). One key aspect of an IMA is that a

¹OYSTER GitHub: <https://github.com/ge-high-assurance/OYSTER>

majority of system functions (both safety critical and non-safety critical) are now implemented as software applications running on the IMA platform. As such, this airborne software is also required to adhere to the same rigorous certification process. Guidance for certifying airborne software is provided in DO-178C. In order to make OYSTER part of the GE Aviation IMA development pipeline, there are several key challenges to overcome.

- Since the goal of OYSTER is to automate and replace some of the steps currently performed by the existing qualified toolchain, it will have to meet all of the objectives of DO-178C Software Considerations in Airborne Systems and Equipment Certification or DO-330 tool qualification standard.
- There are several steps in our process where translation of data has to occur to go from one tool to the next. Each of these points will have to have a qualified verifier developed to prove that nothing was corrupted/changed/lost during transformation.
- One major concern is the format of the formal proof certificates and other verification evidence that is being produced. It is not easily human-readable, and even if the format was changed to something easier to read, an FAA certification authority with no knowledge of formal methods would be unable to understand the proofs. We cannot assume that aviation/avionics experts will have this expertise.
- We expect the learning curve for a systems integrator on a real aircraft development program to be able to use these tools without help will be steep. Training for system developers will be needed. We also need to understand the level of expertise with formal methods tools such as SMT, Kind2 model checker, and AADL modeling that they are expected to have.

VI. RELATED WORK

Several works exist in the literature on the generation of schedules and architectural models. SMT-based system scheduling synthesis for applications have been proposed for time-triggered platforms [12], [11], [10] and TTEthernet networks [19]. SMT-based techniques have also been proposed for the synthesis [24], [35] and refinement [21], [20], [26] of system architectures. SAT-solving techniques have been proposed for generating architectural models [31], [37]. Correct-by-construction techniques for developing architectural models include approaches that use the “B” method [29], linear temporal logic [34], [41], mixed integer programming [42], AADL-based tools and techniques have been developed for synthesis [27], reconfiguration [43], and verification [32], [38], [40], integrated design modeling [14], and domain-specific languages [5]. The CoBaSa framework has been applied to industrial-scale IMA architecture synthesis problems [31] and is closely related to our work. The difference lies in the use of solvers and solver theories. CoBaSa uses Pseudo-boolean (PBSAT) and Integer Linear Programming (ILP) solvers to perform IMA architecture synthesis [31], [30],

whereas OYSTER uses modern SMT solvers with combination of Quantifier-free Equality under Uninterpreted Functions (QF_EUF), Quantifier-Free Linear Integer Arithmetic (QF_LIA), Boolean and Algebraic Datatype theories. OYSTER encodes IMA constraints in SMT allowing for reporting of UNSAT cores, which could easily localize issues and provide useful feedback to journeyman developers about the constraints being violated; whereas CoBaSa does not.

VII. CONCLUSION AND FUTURE WORK

We have presented a formal language and end-to-end framework called OYSTER to automatically synthesize aspects of industrial IMA platforms. The language enables users to encode IMA architecture constraints. The toolchain takes in the AADL models annotated with OYSTER constraints as input and auto-synthesizes a correct-by-construction IMA architecture instantiated with implementation details. The synthesized architecture can be annotated with behavior models and safety properties. The safety properties can then be discharged by the Kind2 model checker, which is guaranteed to be correct by the formal proofs generated by the model checker. Users may independently verify the correctness of the proofs by using third-party SMT solvers and proof checkers. The framework was applied on a use case provided by GE Aviation, and the evaluation showed promising results along with lessons learned. One potential direction of future work would be to support multi-core scheduling (e.g., schedule GPM applications in multiple GPM processors) and integrate OYSTER with GE Aviation’s existing development pipeline. Another research direction is to extend OYSTER to support more aspects of IMA and seek certification for OYSTER solutions.

Acknowledgement & Disclaimer: Distribution Statement “A” (Approved for Public Release, Distribution Unlimited). This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

- [1] OYSTER Integrated Modular Avionics Use Case Architecture. https://github.com/ge-high-assurance/OYSTER/blob/main/models/notional_architecture/IMA.PNG (2022), [Online; Accessed 2023-08-14]
- [2] OYSTER Integrated Modular Avionics Use Case Behavior Model. https://github.com/ge-high-assurance/OYSTER/blob/main/models/FuelControlSystem_Behavior/FuelControlSystem_Behavior.aadl (2022), [Online; Accessed 2023-08-14]
- [3] OYSTER Language Grammar. <https://github.com/ge-high-assurance/OYSTER/blob/main/tools/plugin/verdict/com.ge.research.osate.oyster.dsl/src/com/ge/research/osate/oyster/dsl/Oyster.xtext> (2022), [Online; Accessed 2023-08-14]
- [4] Airlines Electronic Engineering Committee: Aircraft Data Network Part 7, AFDX NETWORK, ARINC Specification 664 (2002)
- [5] Alves, R., Amaral, V., Cintra, J., Tavares, B.: A Family of Domain-Specific Languages for Integrated Modular Avionics. In: International Conference on the Quality of Information and Communications Technology, pp. 239–254. Springer (2019)
- [6] An, D., Kim, K.H., Kim, K.I., et al.: Optimal Configuration of Virtual Links for Avionics Network Systems. International journal of aerospace engineering **2015** (2015)

- [7] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
- [8] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)
- [9] Barrett, C., Tinelli, C.: Satisfiability Modulo Theories. Springer (2018)
- [10] Beji, S., Hamadou, S., Gherbi, A., Mullins, J.: SMT-based cost optimization approach for the integration of avionic functions in IMA and TTEthernet architectures. In: 2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications. pp. 165–174. IEEE (2014)
- [11] Biewer, A., Andres, B., Gladigau, J., Schaub, T., Haubelt, C.: A symbolic system synthesis approach for hard real-time systems based on coordinated smt-solving. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 357–362. IEEE (2015)
- [12] Biewer, A., Gladigau, J., Haubelt, C.: Towards tight interaction of asp and SMT solving for system-level decision making. In: ARCS 2014; 2014 Workshop Proceedings on Architecture of Computing Systems. pp. 1–7. VDE (2014)
- [13] Bjørner, N.S., Phan, A.: νZ - Maximal Satisfaction with Z3. In: Kutsia, T., Voronkov, A. (eds.) 6th International Symposium on Symbolic Computation in Software Science, SCSS 2014, Gammarth, La Marsa, Tunisia, December 7–8, 2014. EPiC Series in Computing, vol. 30, pp. 1–9. EasyChair (2014). <https://doi.org/10.29007/jmxj>, <https://doi.org/10.29007/jmxj>
- [14] Bonev, M., Hvam, L., Clarkson, J., Maier, A.: Formal computer-aided product family architecture design for mass customization. Computers in industry **74**, 58–70 (2015)
- [15] Carnegie Mellon University: Open Source AADL Tool Environment (OSATE). <https://osate.org/index.html> (2016), [Online; Accessed 2023-08-14]
- [16] Champion, A., Mebsout, A., Stickse, C., Tinelli, C.: The Kind 2 model checker. In: International Conference on Computer Aided Verification. pp. 510–517. Springer (2016)
- [17] Clarke, E.M.: Model checking. In: Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17. pp. 54–56. Springer (1997)
- [18] Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional Verification of Architectural Models. In: NASA Formal Methods Symposium. pp. 126–140. Springer (2012)
- [19] Craciunas, S.S., Oliver, R.S.: SMT-based Task-and Network-Level Static Schedule Generation for Time-Triggered Networked Systems. In: Proceedings of the 22nd international conference on real-time networks and systems. pp. 45–54 (2014)
- [20] Delmas, K., Delmas, R., Pagetti, C.: SMT-based Architecture Modelling for Safety Assessment. In: 2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES). pp. 1–8. IEEE (2017)
- [21] Delmas, K., Delmas, R., Pagetti, C.: SMT-based Synthesis of Fault-Tolerant Architectures. In: International Conference on Computer Safety, Reliability, and Security. pp. 287–302. Springer (2017)
- [22] Dutertre, B.: Yices 2.2. In: International Conference on Computer Aided Verification. pp. 737–744. Springer (2014)
- [23] Feiler, P.H., Gluch, D.P.: Model-based engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2012)
- [24] Finkbeiner, B., Schewe, S.: SMT-based Synthesis of Distributed Systems. In: Proceedings of the second workshop on Automated formal methods. pp. 69–76 (2007)
- [25] Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind Model Checker. In: International Conference on Computer Aided Verification. pp. 20–27. Springer (2018)
- [26] Goldman, R.P., Bryce, D., Pelican, M.J., Musliner, D.J., Bae, K.: A Hybrid Architecture for Correct-by-Construction Hybrid Planning and Control. In: NASA Formal Methods Symposium. pp. 388–394. Springer (2016)
- [27] Hardin, D.S., Slind, K.L.: Formal Synthesis of Filter Components for Use in Security-Enhancing Architectural Transformations. In: 2021 IEEE Security and Privacy Workshops (SPW). pp. 111–120. IEEE (2021)
- [28] Larraz, D., Laurent, M., Tinelli, C.: Merit and Blame Assignment with Kind 2. In: Formal Methods for Industrial Critical Systems: 26th International Conference, FMICS 2021, Paris, France, August 24–26, 2021, Proceedings 26. pp. 212–220. Springer (2021)
- [29] Lecomte, T., Servat, T., Pouzancre, G., et al.: Formal Methods in Safety-Critical Railway Systems. In: 10th Brazilian symposium on formal methods. pp. 29–31 (2007)
- [30] Manolios, P., Papavasileiou, V.: ILP Modulo Theories. In: Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings 25. pp. 662–677. Springer (2013)
- [31] Manolios, P., Vroon, D., Subramanian, G.: Automating Component-Based System Assembly. In: Proceedings of the 2007 international symposium on Software testing and analysis. pp. 61–72 (2007)
- [32] Meng, B., Larraz, D., Siu, K., Moitra, A., Interrante, J., Smith, W., Paul, S., Prince, D., Herencia-Zapana, H., Arif, M.F., et al.: VERDICT: A Language and Framework for Engineering Cyber Resilient and Safe System. Systems **9**(1), 18 (2021)
- [33] Moura, L.d., Bjørner, N.: Z3: An Efficient SMT Solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
- [34] Nilsson, P., Hussien, O., Balkan, A., Chen, Y., Ames, A.D., Grizzle, J.W., Ozay, N., Peng, H., Tabuada, P.: Correct-by-construction adaptive cruise control: Two approaches. IEEE Transactions on Control Systems Technology **24**(4), 1294–1307 (2015)
- [35] Peter, S., Givargis, T.: Component-based synthesis of embedded systems using satisfiability modulo theories. ACM Transactions on Design Automation of Electronic Systems (TODAES) **20**(4), 1–27 (2015)
- [36] Prisaznuk, P.J.: Integrated Modular Avionics. In: Proceedings of the IEEE 1992 National Aerospace and Electronics Conference@ m_NAECON 1992. pp. 39–45. IEEE (1992)
- [37] Reimann, F., Lukasiewicz, M., Glass, M., Haubelt, C., Teich, J.: Symbolic system synthesis in the presence of stringent real-time constraints. In: Proceedings of the 48th Design Automation Conference. pp. 393–398 (2011)
- [38] Siu, K., Moitra, A., Li, M., Durling, M., Herencia-Zapana, H., Interrante, J., Meng, B., Tinelli, C., Chowdhury, O., Larraz, D., et al.: Architectural and Behavioral Analysis for Cyber Security. In: 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC). pp. 1–10. IEEE (2019)
- [39] Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods in System Design **42**(1), 91–118 (2013)
- [40] Suo, D., An, J., Zhu, J.: AADL-based modeling and TPN-based Verification of Reconfiguration in Integrated Modular Avionics. In: 2011 18th Asia-Pacific Software Engineering Conference. pp. 266–273. IEEE (2011)
- [41] Wongpiromsarn, T., Topcu, U., Murray, R.: Formal Synthesis of Embedded Control Software: Application to Vehicle Management Systems. In: Infotech@Aerospace 2011. p. 1506 (2011)
- [42] Xuan, Z., Xiong, H., Feng, H.: Hybrid partition-and network-level scheduling design for distributed integrated modular avionics systems. Chinese Journal of Aeronautics **33**(1), 308–323 (2020)
- [43] Zhang, Q., Wang, S., Liu, B.: Approach for Integrated Modular Avionics Reconfiguration Modelling and Reliability Analysis based on AADL. Iet Software **10**(1), 18–25 (2016)

Fortis: A Tool for Analysis and Repair of Robust Software Systems

Changjian Zhang
Carnegie Mellon University
Pittsburgh, PA USA
changjiz@andrew.cmu.edu

Ian Dardik
Carnegie Mellon University
Pittsburgh, PA USA
idardik@andrew.cmu.edu

Rômulo Meira-Góes
The Pennsylvania State University
State College, PA USA
romulo@psu.edu

David Garlan
Carnegie Mellon University
Pittsburgh, PA USA
dg4d@andrew.cmu.edu

Eunsuk Kang
Carnegie Mellon University
Pittsburgh, PA USA
eunsukk@andrew.cmu.edu

Abstract—This paper presents Fortis, a tool for automated, formal analysis and repair of robust discrete systems. Given a system model, an environment model, and a safety property, the tool can be used to automatically compute *robustness* as the amount of *deviations* in the environment under which the system can continue to guarantee the property. In addition, Fortis enables automated repair of a given system to improve its robustness against a set of intolerable deviations through a process called *robustification*. With these techniques, Fortis enables a new process for developing *robust-by-design* systems. The paper presents the overall design of Fortis as well as the key details behind the robustness analysis and robustification techniques. The applicability and performance of Fortis are illustrated through experimental results over a set of case study systems, including a radiation therapy system, an electronic voting machine, network protocols, and a transportation fare system.

I. INTRODUCTION

Typical verification tasks involve the following question: Given a model of a system (M) and an environment (E), does the system satisfy a desired property (P) under the environment (i.e., $M \parallel E \models P$)? The model E here captures various assumptions that the system makes about its environment to establish P . For example, such assumptions may state that a human operator in a safety-critical system (e.g., a medical device) performs a set of actions in an expected order, or that the underlying network in a distributed system is reliable and delivers messages correctly from one node to another.

In practice, once the system is deployed, the actual environment may *deviate* from this model, either due to modeling errors, faults, or natural changes in the environment. For example, the operator may inadvertently commit errors from time to time (e.g., omitting or repeating an action); the network might experience an unexpected disruption and fail to guarantee reliable delivery (e.g., losing or duplicating messages). Ideally, a system that is *robust* would continue to ensure its most critical properties even under possible deviations in the environment.

In this paper, we present Fortis¹, a tool for formal analysis and repair of robust software systems. Our tool is based on a formal definition of robustness for discrete systems introduced in our prior work [1]: Given system M , environment E (both specified as a labeled transition system (LTS)) and safety property P , the *robustness* of the system, denoted Δ , is defined as the set of all possible deviations in E under which M continues to satisfy P . More specifically, Δ consists of traces that do not belong to the trace set of E , capturing additional environmental behaviors beyond the normative environment. For example, if M describes the design of a medical system (e.g., radiation therapy system), E the expected behavior of the human operator, and P a safety requirement (e.g., “Patients should be protected from radiation overdose”), Δ would represent the set of possible operator errors under which the system can still ensure safety. Conceptually, Δ represents the safe operating envelope of the system: As long as the environmental deviations remain within this envelope, the system can guarantee P .

Building on this definition, Fortis provides various types of analysis tasks to support rigorous design and analysis of robust systems. First, given M , E , and P , Fortis can be used to automatically compute Δ as a qualitative measure of the system robustness. Our tool can also be used to compute deviations that lie outside of Δ (which we call *intolerable deviations*), showing how P may be violated when the environment moves outside of the operating envelope. In addition, given a pair of alternative system designs, M_1 and M_2 , Fortis can also be used to formally compare the two with respect to their robustness (i.e., compute a set of deviations that one design can tolerate but the other cannot).

Once the above analysis reveals that the given system is not robust against certain deviations, the developer may wish to modify M to further improve its robustness. To support this task, Fortis also provides a type of system repair called *robustification* [2]: Given M , P , E , and a set of intolerable

¹<https://github.com/cmu-soda/Fortis>

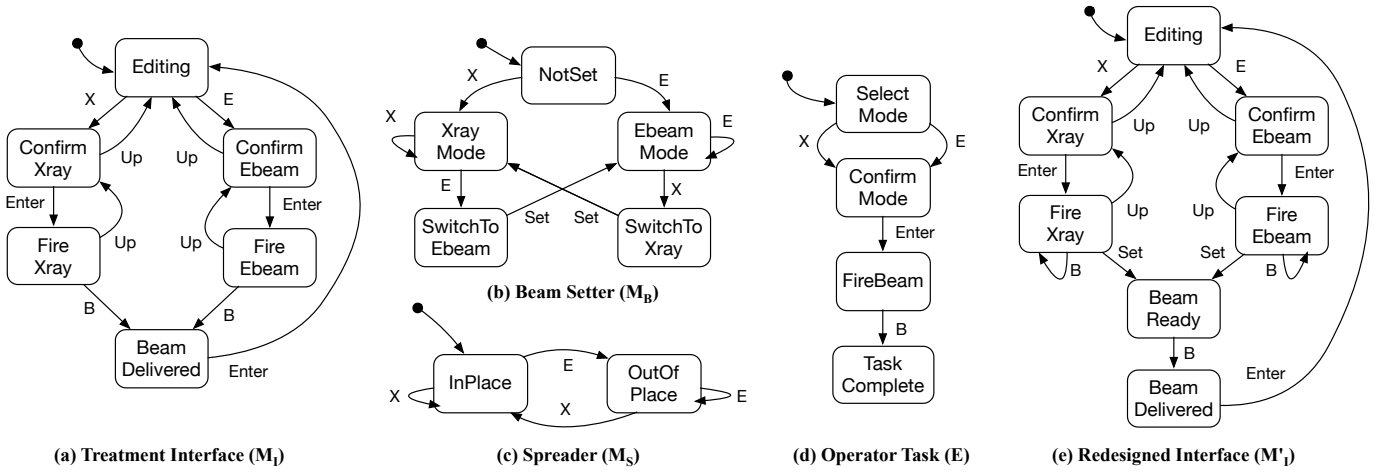


Fig. 1. Labeled transition systems for a radiation therapy system ($M = M_I || M_B || M_S$).

deviations, $\bar{\delta}$ such that $M || E' \not\models P$ (where $E' = E \oplus \bar{\delta}$ is the deviated environment), the goal is to synthesize a more robust system, M' , such that $M' || E' \models P$.

As far as we are aware, Fortis is the first tool that is capable of providing the types of robustness analysis and repair described above. Compared to the prototypes presented in our prior publications, additional engineering effort has been taken to integrate them into a uniform framework. Together with these techniques, we believe Fortis enables a new methodology for developing *robust-by-design* systems. Developers can start with an initial design that guarantees its desired properties under the normative environment. Then, they can use Fortis to understand the robustness of the initial design and generate the deviations that it cannot tolerate. Finally, developers can decide which of these deviations the system should be able to tolerate, and use Fortis to automatically generate a more robust design. Developers may iterate this process for multiple times until a satisfactory design is met.

To evaluate the tool, it has been applied to a wide range of case study systems, including a radiation therapy system (similar to the well-known Therac-25 system [3]), an electronic voting system, network protocols, an infusion pump, and a fare collection protocol used in a public transportation system. Our experiments show that Fortis can automatically compute robustness for complex system models under several seconds, and also synthesize repairs for most of the case studies under a set timeout.

The rest of the paper is structured as follows. We first demonstrate use cases of Fortis using an example involving a radiation therapy system (Section II). We then present an overview of the tool architecture (Section III) and describe the key details of the analysis and robustification techniques (Section IV). Next, we illustrate the applicability and performance of Fortis over the case studies (Section V). We conclude with the related work (Section VI) and a discussion of limitations and possible extensions (Section VII).

II. MOTIVATING EXAMPLE

Consider a radiation therapy machine similar to the well-known Therac-25 machine [3]. Figure 1 shows the labeled transition systems of the main system components, including (a) *Treatment Interface* (M_I), which allows the operator to choose the radiation mode and fire the beam, (b) *Beam Setter* (M_B), which switches between the two radiation modes (Electron and X-ray), and (c) *Spreader* (M_S), which is inserted during the X-ray mode to attenuate the effect of the high-power X-ray beam and limit possible overdose (X-ray delivers roughly 100 times higher level of current than the Electron beam). The overall system is the composition of the three components, i.e., $M = M_I || M_B || M_S$.

An important safety requirement for the system is that the spreader must be in place when the beam is delivered in the X-ray mode. This requirement can be formally defined in linear temporal logic [4] as: $\mathbf{G}(BeamDelivered \wedge XrayMode \Rightarrow InPlace)$. Furthermore, the task to be carried out by the operator is specified as an environment model (E) in Figure 1(d): In the normal treatment process, the therapist selects the correct mode for a given patient by pressing either X or E , confirms the mode by pressing $Enter$, and finally initiates the therapy by pressing B .

Applying a verification technique such as model checking [5] would show that the above system satisfies the safety property under the normative operator behavior, i.e., $M || E \models P$. Beyond this standard verification task, Fortis offers the following additional tasks:

a) *Computing robustness*: In addition to stating that M satisfies P under E , Fortis can be used to generate the set of all deviations (i.e., environmental traces that do not belong to the behavior of E) under which the system can still guarantee P . This set captures the overall *robustness* of the system, and can aid the developer in understanding the system's ability in handling deviations in the environment. In the radiation therapy system example, one of the deviations that Fortis generates is $\langle X, B \rangle$, which depicts the operator

omitting to confirm the radiation mode before firing; the system guarantees P even under a new environment E' where this trace has been added as an additional behavior.

b) Generating intolerable deviations: Complementary to the previous analysis, Fortis can also be used to generate deviations for which the system is not able to guarantee P . In the radiation therapy example, one such deviation is $\langle X, Up, E, Enter, B \rangle$, depicting a scenario where the operator accidentally selects the X-ray mode and corrects the mistake by pressing Up and then E . When the operator presses B to fire the beam, the beam setter might still be in transition from X-ray to the Electron mode (in state `SwitchToEbeam` in M_B , Figure 1(b)) while the spreader is out of place, causing P to be violated. The output from this analysis can help identify parts of the system design that can be made more robust.

c) Comparing designs: Given two different versions of a system (e.g., Therac-25 and its predecessor, Therac-20, which was known to be safer thanks to an additional hardware interlock that was subsequently removed [3]), Fortis can also be used to formally compare the two with respect to their robustness. For example, Fortis would show that Therac-20 is strictly robust than Therac-25 by generating a deviation (e.g., $\langle X, Up, E, Enter, B \rangle$) for which the former can guarantee P while the latter cannot.

d) Robustifying the system: Fortis can be used to automatically improve an existing design through a process called *robustification*: If M is not robust against some given set of deviations ($\bar{\delta}$), generate a more robust design, M' , that can satisfy P under $\bar{\delta}$ (i.e., M' is strictly more robust than M). For example, given deviation $\langle X, Up, E, Enter, B \rangle$, Fortis automatically synthesizes $M' = M'_I \| M_B \| M_S$; this new design, M' , guarantees the safety property by preventing the operator from firing the beam until the mode switch has completed (i.e., state `BeamReady`), as shown in Figure 1(e).

III. OVERVIEW OF FORTIS

Figure 2 shows the overall architecture of Fortis. Given LTS-based specifications of machine M , its normative environment E , and safety property P as input, and the *Model Parser* compiles them into our internal data structure for LTS. Depending on the type of task that the user wishes to perform, the input models are then passed onto *Robustness Analysis* or *Robustification*.

a) Robustness Analysis: To compute robustness, we first generate the *weakest assumption* of M with respect to environment E and property P . In assume-guarantee style of reasoning [6], the weakest assumption captures the largest possible environmental behavior under which the machine satisfies a given property. Then, robustness, denoted Δ , is computed as the set of traces that are in the weakest assumption but not in the expected environment E . In general, Δ may be infinite, and not in a form that is easily comprehensible by the user. Thus, we partition Δ into a finite set of *equivalence classes*, each of which contains traces that describe the same type of deviation (e.g., the type of user error where one omits an action), and sample *representative traces* from those classes.

Finally, if a *deviation model* is provided, we use it to generate *explanations* that describe how the environment may deviate from its expected behavior in a particular way. The final output is a set of pairs of a representative trace and its corresponding explanation. Section IV-A describes some of these steps in more detail.

b) Robustification: To robustify a machine, the user specifies a set of intolerable deviations ($\bar{\delta}$), which are then used to transform the normative environment (E) into a deviated environment (E'). Note that the robustness analysis module can be used to generate *all* the intolerable deviations $\bar{\Delta}$, which can help designers identify the undesirable deviations of interest. Optionally, the user can also specify the *preferred behaviors* (i.e., execution traces) expected to be retained in the new design and the *costs* to control and observe events, to generate repairs that are optimal with respect to these two metrics.

Internally, Fortis leverages supervisory control theory [7] to synthesize new designs; in particular, it currently uses the state-of-the-art controller synthesizer called Supremica [8]. To find optimal repairs, the *Design Optimizer* repeatedly invokes the synthesizer for different combinations of the preferred behaviors and the event costs, exploring the multi-objective space to generate *Pareto-optimal* solutions [9] as the final output of the tool (more details in Section IV-B).

c) Bridging Robustness and Robustification: In addition to the implementation of the two techniques proposed in our prior work, Fortis also provides an integration of them that bridges the gap to close the loop for robust-by-design development process, represented as the dashed line connecting the two modules in Figure 2. Specifically, it enables the user to first compute the robust deviations (Δ) and intolerable deviations ($\bar{\Delta}$); and after the user has decided on the deviations that they want the system to be robust against, it can then generate the corresponding deviated environment model (E'). Finally, this deviated model can be used as the input to robustify the system design.

IV. ANALYSIS AND ROBUSTIFICATION METHODS

In this section, we provide key implementation details behind the robustness analysis, robustification techniques, and their integration in Fortis.

A. Robustness Analysis

a) Robustness Computation: In our definition [1], the *robustness* of machine M with respect to environment E and property P (denoted $\Delta(M, E, P)$) is defined as the *maximal* set of traces that do not cause a property violation and do not belong to the trace set of the normative environment. This set is computed by calculating the difference between (1) the *weakest assumption* of M w.r.t. E and P , and (2) the environment E , i.e., $\Delta(M, E, P) = \text{beh}(W_{M,E,P}) \setminus \text{beh}(E)$, where $\text{beh}(\cdot)$ is the set of all traces of a given LTS and $W_{M,E,P}$ is the weakest assumption. Specifically, Fortis uses the approach developed by Giannakopoulou et al. [10] to compute the weakest assumption as an LTS.

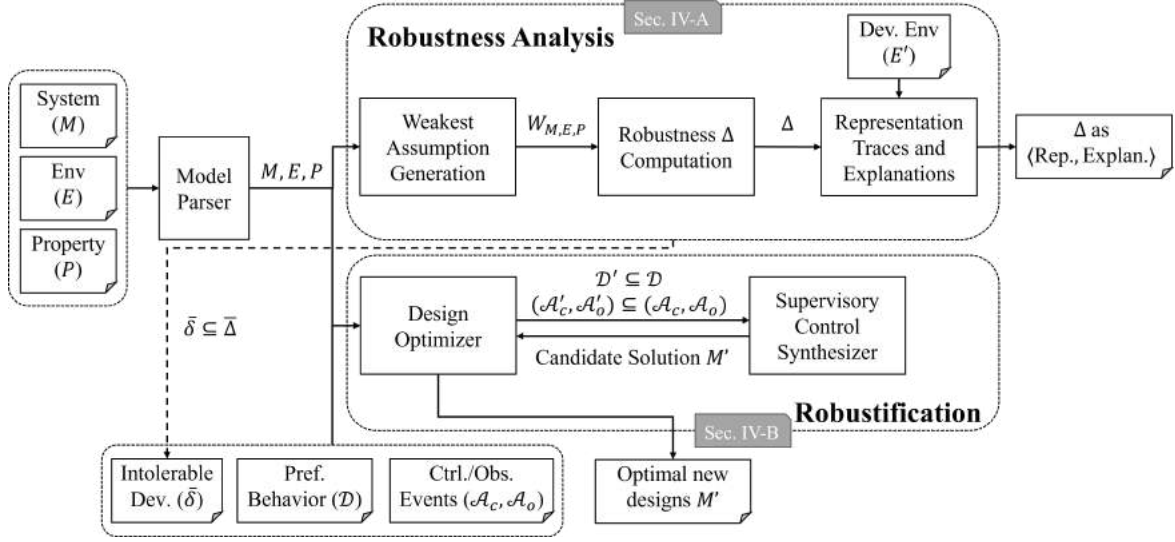


Fig. 2. The architecture of Fortis.

b) Robustness Comparison: Similar to robustness computation, given two designs M_1 and M_2 , Fortis leverages their weakest assumptions to compare their robustness. In particular, given machine M_1 , M_2 , and the same environment E and property P , the robustness comparison is achieved by:

$$\Delta(M_1) - \Delta(M_2) = \text{beh}(W_{M_1,E,P}) \setminus \text{beh}(W_{M_2,E,P})$$

where $W_{M_1,E,P}$ and $W_{M_2,E,P}$ are the weakest assumptions for M_1 and M_2 with respect to E and P , respectively.

c) Robustness Representation: In general, the set of traces that represent Δ may be infinite, and an LTS-based representation may not be readily comprehensible by the user, even for relatively simple models like the radiation therapy machine. To address this, Fortis generates a succinct and finite representation of Δ . It groups the traces in Δ into a finite set of *equivalence classes* $\Pi_{s,a}$, where s is a state that directly leads to a violation of E by taking the transition a (note that Δ contain traces that belong to $W_{M,E,P}$ but not in E). Therefore, $\Pi_{s,a}$ describes a class of robust traces that share the same normative behaviors in E that all end in state s and deviate from E by the same event a .

Finally, from each equivalence class, we sample a single *representative* trace that represents this class. In particular, the representative trace for $\Pi_{s,a}$ is generated by finding the shortest trace in $W_{M,E,P}$ from the initial state to state s and then appending event a to it. For example, in the radiation therapy machine, $\langle X, B \rangle$ represents the equivalence class of behaviors that deviate in action B from state s_X , where s_X is the state reached by the normative behavior $\langle X \rangle$ and B is the first deviated action leading to a trace not defined in E . The final output from this step is a finite set of representative traces that describe different types of environmental deviations.

d) Deviation Explanation: Representative traces describe how the environment deviates from the expected behavior as *observed* by machine M . However, they do not describe

the internal *faults* within the environment that cause these deviations in the first place. For example, it is unclear what kind of faults cause the deviation of $\langle X, B \rangle$. If a *deviation model* that contains these internal events is provided by the user, Fortis uses it to generate an *explanation* of how a particular deviation arises due to an internal behavior of the environment. In particular, given deviation model D and representative trace σ , an explanation is generated by finding trace σ_{exp} in D that is equivalent to σ when projected over the observable events in M but contains additional faulty events.

A deviation model is created by augmenting the normative environmental model E with additional transitions on faulty events. For example, a deviation model for the radiation therapy machine may specify that from state **ConfirmMode** in $E (Figure 1(d)), the operator might commit a type of error called *omission error* [11], i.e., omitting *Enter* and pressing *B*; this would be specified as an additional transition from **ConfirmMode** to **FireBeam** on an internal faulty event *Omission*. Then, Fortis would generate an explanation for the representative trace $\langle X, B \rangle$ as $\langle X, \text{Omission}, B \rangle$. In [1], it is further described how a deviation model can be automatically generated by applying domain-specific patterns of deviations (e.g., patterns of common human errors [11]) to the normative environment E .$

B. Robustification

Fortis finds not just any solution to the robustification problem, but *optimal* repairs of M . In particular, it attempts to optimize two different quality metrics: (1) the amount of behaviors retained from M to new design M' and (2) the cost of modifications needed to achieve M' . Specifically, for (1), we introduce the notion of *preferred behaviors* \mathcal{D} , which are specified as traces and represent operational scenarios that the user wishes the new design to retain. For (2), we introduce the notion of *controllable* \mathcal{A}_c and *observable* \mathcal{A}_o events that indicate whether M' can control and observe additional events,

respectively, for the purpose of robustification. In addition, the user can optionally assign a cost to these events, to distinguish events that are more costly to control or observe.

These two metrics lead to conflicting objectives: With additional controllable and observable events, the new design can preserve more behaviors but the modification cost also increases. Thus, finding optimal repairs is a multi-objective optimization problem, where the goal is to generate a set of Pareto-optimal solutions [9]. Fortis implements a novel algorithm to generate such solutions, using controller synthesis as a primitive operation. At high-level, the Design Optimizer first attempts to synthesize a controller that preserves the maximum number of preferred behaviors ($\mathcal{D}_{max} \subseteq \mathcal{D}$) with all controllable and observable events (\mathcal{A}_c and \mathcal{A}_o). Then, the optimizer incrementally removes elements from \mathcal{D}_{max} to find solutions with a lower modification cost. In particular, given a particular subset of preferred behaviors $\mathcal{D}' \subseteq \mathcal{D}_{max}$, it searches for a controller that uses a minimal subset of the controllable and observable events.

For the radiation therapy system, one possible repair that Fortis generates results in the machine simply disabling action *Up* in state *ConfirmXray*, to prevent the operator error from occurring in the first place. While this solution technically achieves the safety property, it is arguably undesirable, since it prevents the operator from switching the radiation mode. To rule out such solutions, the Fortis user may specify preferred behaviors as traces $\langle X, Up \rangle$, $\langle E, Up \rangle$, stipulating that the operator should be able to select *Up* after *X* or *E* to change the mode. In addition, the user may assign a cost to event *Set* to reflect the cost of making it controllable or observable by the interface (M_I) or the spreader component (M_S).

Given the additional inputs on preferred behaviors and costs, as an alternative solution, Fortis generates a repair like the one in Figure 1(e); this solution retains the ability for mode switching, while being more costly since it involves the system observing an additional event, *Set*, to synchronize on the completion of mode switching. The user of Fortis can examine these alternative solutions and select the final one depending on the trade-offs between the two metrics that they are willing to make.

C. Integration

Compared to the prototype implementations in our prior work, Fortis integrates the two techniques into a uniform framework, i.e., they accept the same formats of input model files and can produce results in a uniform manner. Also, we leverage Automatalib [12], a well-maintained open source library for transition systems, to provide a common interface for our internal representations for LTS, which implements the CompactFSM data structure for efficient model manipulations. Moreover, we replace some algorithm implementations with more efficient data structures like BitSet (for NFA to DFA conversion) to further improve the performance of the computation process.

Fortis also bridges the gap between the robustness computation and robustification. In particular, after generating the

representative traces for robustness and intolerable deviations, and their explanations given a deviation model D , the user can select the deviations δ' that the new design should be robust against. For example, $\langle X, Commission, Up, E, Enter, B \rangle$ is the explanation of an intolerable deviation for the radiation therapy machine, which could be included in δ' as the user wishes the new design to be robust against it.

To robustify the system against δ' , Fortis can automatically generate a new deviation model D' such that it only includes the deviations in δ' (as the original D may include more than one kinds of human errors, e.g., *commission error*, *omission error*, or *repetition error*). Then, the user can use D' as the input to the robustification process to synthesize the new design M' .

V. EXPERIMENTS

In this section, we illustrate the applicability and performance of Fortis through experiments over a set of case studies.

A. Implementation and Usage

Fortis leverages Automatalib [12] and LTSA [13] for model specification and manipulation, and Supremica [8] for supervisory control synthesis. It supports commonly used specification languages such as AUT and FSM (through Automatalib) and FSP (language used by LTSA) to specify and output system models.

Currently, Fortis implements a command-line interface. Users provide system and property specifications through command-line arguments or a JSON configuration file, and the tool produces computation results into command-line outputs. For example, Figure 3 shows the input and output for robustness computation and robustification of the radiation therapy example.

B. Case Studies

a) *Voting Machine*: We consider a simplified design of an electronic voting system (called ES&S iVotronic, described in more detail in [14]) that was used in several state-wide elections in the US. This system is particularly interesting to study from the perspective of robustness, as it was susceptible to an election fraud involving malicious election officials [15]. In this machine, for the last step of a voting process, the voters were asked to confirm their vote by pressing the *confirm* button. However, some voters would inadvertently forget to do so before exiting the voting booth (committing what is generally called *post-completion error* [16]). This would then allow a malicious official to enter the booth, press *back*, and then modify the vote to their liking. In our model, the property to guarantee is that the machine should record each voter's selection exactly as made by that voter, and the intolerable deviation of interest is voters omitting the confirmation step. We successfully used Fortis to generate alternative designs that would prevent malicious officials from modifying the vote (e.g., keeping track of who enters the booth and disabling *confirm* if an official is in the booth).

```

# Run robustness computation
java -jar fortis.jar robustness -s sys.lts -e env0.lts -p p.lts -d env.lts
[INFO] BaseCalculator - Generating robust behavior representation traces by equivalence classes...
[INFO] BaseCalculator - Generating the weakest assumption...
[INFO] SubsetConstructionGenerator - Compose System and Property...
[INFO] SubsetConstructionGenerator - System: #states = 22, #transitions: 44
[INFO] SubsetConstructionGenerator - S|P: #states = 20, #transitions: 40
[INFO] SubsetConstructionGenerator - Pruning and determinising the model...
[INFO] Robustness - Total time: 00:00:00:021
[INFO] Robustness - Equivalence class 'EquivClass(s=1, a=up)':
[INFO] Robustness - RepTrace(word=x,up, deadlock=false) => x,up
[INFO] Robustness - Equivalence class 'EquivClass(s=8, a=up)':
[INFO] Robustness - RepTrace(word=e,up, deadlock=false) => e,up
...

# Run robustification
java -jar fortis.jar robustify config-fast.json
...
[INFO] SolutionIterator - Start iteration 1...
[INFO] SolutionIterator - Try to weaken the preferred behavior by one of the 0 behavior sets:
[INFO] SolutionIterator - This iteration completes, time: 00:00:00:093
[INFO] SolutionIterator - Number of controller synthesis process invoked: 5
[INFO] SolutionIterator - New solution found:
[INFO] SolutionIterator - Size of the controller: 63 states and 130 transitions
[INFO] SolutionIterator - Number of controllable events: 4
[INFO] SolutionIterator - Controllable: [enter, fire_ebeam, fire_xray, setMode]
[INFO] SolutionIterator - Number of observable events: 8
[INFO] SolutionIterator - Observable: [b, e, enter, fire_ebeam, fire_xray, setMode, up, x]
[INFO] SolutionIterator - Number of preferred behavior: 4
[INFO] SolutionIterator - Preferred Behavior:
[INFO] SolutionIterator - x,up,e,enter,b
[INFO] SolutionIterator - e,up,x,enter,b
[INFO] SolutionIterator - x,enter,up,up,e,enter,b
[INFO] SolutionIterator - e,enter,up,up,x,enter,b
[INFO] SolutionIterator - Utility Preferred Behavior: 48
[INFO] SolutionIterator - Utility Cost: -1
...

```

Fig. 3. Fortis' input and output for robustness computation and robustification of the radiation therapy system. For robustness computation, the log indicates two of the representative traces found by Fortis, i.e., $\langle x, up \rangle$ and $\langle e, up \rangle$. For robustification, the log indicates one redesign found by Fortis where all preferred behaviors are satisfied under the events given in the solution to be controlled and observed. The concrete redesign model is written to a model specification file in the AUT format.

b) Network Protocols: Consider the problem of transmitting a sequence of messages between a pair of nodes (*sender* and *receiver*) in a specific order. We consider two protocols for network communication: (1) A naive protocol where the sender assumes a perfectly reliable communication channel, and (2) the Alternate Bit Protocol (ABP) [17], which is designed to guarantee integrity of messages over unreliable channels (e.g., message loss or duplication). The normative environment (E) here is the reliable channel, which relays (1) a message from the sender to the receiver and (2) then an acknowledgement from the receiver back to the sender. Our notion of deviations can be used to capture different ways in which an unreliable channel might behave, such as reordering (e.g., from expected trace $t = \langle msg_1, msg_2 \rangle$ to deviation $t' = \langle msg_2, msg_1 \rangle$), losing ($t' = \langle msg_2 \rangle$), or duplicating messages ($t' = \langle msg_1, msg_1, msg_2 \rangle$). In particular, we used Fortis to formally compare the robustness of the two protocols, to compute faults in the channel that ABP can handle but the naive one cannot.

c) Oyster: We consider the *Oyster* card fare collection protocol used in public transportation in London, UK (described in [18]). In this system, the user taps their card on the entry gate at the beginning of their journey and on the exit gate at the end. The protocol also allows the user to pay their fare through other means such as credit cards and mobile payments.

In the normative case, the user chooses the appropriate method of payment, and taps in and out with the same method. The property of interest here is avoiding *card collision*, where two different methods of payment are used in the same journey. For instance, the user may tap the Oyster card at the entry gate but then (by mistake) use their mobile phone at the exit, possibly being charged a higher fare than required.

d) Infusion Pump: We model an infusion pump machine for dispensing medication to patients through tube lines [19]. The machine also includes a built-in battery that activates when the power cable is unplugged, and an alarm that goes off when the battery is low. Normally, the operator plugs in the device, configures the medication dose and starts the dispensation. Deviations here correspond to operator errors or power loss. In particular, if the cable is accidentally unplugged and battery runs out during dispensation continues, this might cause serious medical accidents, such as overdose. Thus, the property to be guaranteed here is that *if the machine loses power, it should immediately stop any on-going dispensation*. This case study is the most complex out of the ones that we have done so far and is intended to demonstrate the scalability of Fortis.

C. Experimental Results

For each of the above case studies, we used Fortis to (1) compute the robustness of the system and (2) synthesize

TABLE I
EVALUATION RESULTS. ALL PROBLEMS WERE RUN ON A LINUX MACHINE WITH A 3.6GHz CPU AND 24GB
MEMORY UNDER A 30 MINUTE TIMEOUT.

	Robustness Computation*			Robustification†					
	$ M P $	Time (s)	$ \mathcal{D} $	$ \mathcal{A} $	$ M E' $	Naive		With heuristics	
						#Syn.	Time (s)	#Syn.	Time (s)
Therapy	20	0.025	4	5	21	32	0.812	6	0.469
Naive	41	0.029	2	8	14	1	0.226	1	0.242
ABP‡	23	0.033	-	-	-	-	-	-	-
Voting-1**	53	0.033	1	13	12	2,576	24.100	9	0.507
Voting-2	277	0.066	1	23	31	-	T/O	16	1.908
Voting-3	821	0.106	1	32	44	-	T/O	21	20.172
Voting-4	1,829	0.188	1	41	57	-	T/O	-	T/O
Pump-1	163	0.036	2	12	104	2,304	59.584	13	1.129
Pump-2	1,679	0.149	4	16	760	-	T/O	17	10.817
Pump-3	19,435	1.227	6	20	6,248	-	T/O	21	457.839
Oyster	1,729	0.280	2	4	900	16	1.799	1	0.686

* $|M||P|$ is the number of states of the composition of machine M and property P , and the worst-case complexity of the computation is $O(2^{|M||P|})$.

** In Voting- n , n represents the number of voters and officials in the system; similarly, n in Pump- n is the number of the dispensation lines connected to the pump.

† $|\mathcal{D}|$ is the number of preferred behaviors, $|\mathcal{A}|$ the number of controllable and observable events with cost, $|M||E'|$ the number of states of machine M composed with deviated environment E' . The size of the search space is approximately $O(2^{|\mathcal{D}|+|\mathcal{A}|+|M||E'|})$. #Syn. is the number of calls to the controller synthesizer.

‡ Robustification is not applicable to ABP as it already satisfies P under the given deviations.

robustified designs against a given set of intolerable deviations. Table I summarizes the results from the experiments. For robustness computation, although the worst-case complexity is exponential to the size of the machine M and property P , i.e., $O(2^{|M||P|})$, Fortis can efficiently compute robustness even for a very large model like *Pump-3* with 19,435 states in 1.227 seconds.

On the other hand, robustification is a much more complex problem to solve with a much larger search space, where the worst case complexity is exponential to the number of states of machine M and deviated environment E' , plus the number of preferred behaviors \mathcal{D} and controllable/observable events \mathcal{A} , i.e., $O(2^{|\mathcal{D}|+|\mathcal{A}|+|M||E'|})$. For example, the worst-case complexity for robustifying Pump-2 is about 6×10^{234} .

In addition, we found that controller synthesis is often the bottleneck. The time to solve one synthesis instance grows quickly with the increasing size of the system. Moreover, for the same problem, the synthesis becomes harder to solve when fewer controllable and observable events are provided (while minimizing the cost). Compared to naively searching solutions with brute-force (shown under the Naive columns), Fortis tackles the performance issue by introducing several search heuristics for pruning the search space and reducing the number of synthesis calls, which are described in more detail in [2]. While we believe that Fortis performs reasonably well on robustification of complex models, we plan to explore alternative synthesis techniques (such GR(1) synthesis [20], [21] or constraint-based methods [22]) to further improve its performance.

VI. RELATED WORK

Techniques for reasoning about system robustness have been investigated in prior works [23], [24], [25], [26], [27]. Most of these works adopt a *quantitative* notion of robustness (e.g., given bounded perturbations in its input, a robust system should ensure bounded changes in the output), while we use a definition that is *qualitative* (i.e., additional behaviors that a deviating environment may exhibit). We believe that the two types of definitions are complementary: A quantitative notion is well-suited for capturing numerical deviations in physical or hybrid systems (e.g., a sensor noise), while our approach is suitable for capturing deviations that occur in discrete systems (e.g., human operator errors). In addition, our notion of deviations generalizes the one in [28], [29], where deviations are defined as additional transitions that may be introduced into the environment.

Evrostos [30] is a tool for model checking systems against rLTL [31], a variant of LTL that captures robustness. rLTL enables specifications stipulating that “small” violations in the environmental assumption should result in proportionally “small” violations in the system guarantee. In particular, rLTL leverages a multi-valued semantics to capture different levels of property violation (e.g., given an expected property of form $\mathbf{G}\psi$, one possible weaker variant is $\mathbf{F}(\mathbf{G}\psi)$). Besides the difference in the definitions of robustness used, Evrostos and Fortis differ in their goals: The former is used to specify and verify robustness as a specification, while Fortis is used to extract robustness as a property of the system. However, rLTL could potentially be used to characterize certain types of environmental deviations that are temporal in nature.

Robustification in Fortis also shares similarities with *model repair* techniques [32], [33], [34], [35], [14]. Among these, the closest work to our approach is OASIS [14], which also leverages controller synthesis to revise a machine to satisfy a security property in a deviated environment. One major difference is that Fortis uses semantic-based metrics (i.e., preserved behaviors and costs of events) to qualify solutions, whereas these works do not take into account the cost of changes (e.g., OASIS), or consider only syntactic changes to the model (e.g., the number of modified transitions and states).

VII. CONCLUSIONS AND FUTURE EXTENSIONS

We have presented Fortis, an automated tool for formal analysis and repair of robust discrete systems. The tool supports a rigorous methodology for designing robust systems, where the developer starts with an initial design and a normative environment, and then iteratively improve its robustness by identifying undesirable environmental deviations and robustifying the system against them. As far as we know, Fortis is the first tool that provides these types of robustness analyses and repair by offering a seamless workflow packaged into a single tool.

There are a number of further tool extensions that we plan to explore. Currently, Fortis supports safety properties only. Adding liveness support will involve new theoretical extensions; in particular, during robustification, new behaviors may need to be added to the system (instead of restricting its behavior, as currently done for safety), possibly leading to a much larger search space. In addition, we also plan to add a capability for *synthesizing* a robust system (M) from scratch instead of modifying an existing one (M to M'). Finally, as discussed in Section V, we will explore alternative methods for controller synthesis to further improve its scalability.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation award CCF-2144860, the NSA under Award No. H9823018D0008, and the Office of Naval Research under Award N00014172899. It was also supported by the CAMELOT project (reference POCI-01-0247-FEDER-045915) which is co-financed by the European Regional Development Fund and the Portuguese Foundation for Science and Technology under CMU Portugal. Any views, opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the organizations.

REFERENCES

- [1] C. Zhang, D. Garlan, and E. Kang, "A behavioral notion of robustness for software systems," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, p. 1–12.
- [2] C. Zhang, T. Saluja, R. Meira-Góes, M. Bolton, D. Garlan, and E. Kang, "Robustification of behavioral designs against environmental deviations," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, to appear.
- [3] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.
- [4] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.
- [5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 2001.
- [6] D. Giannakopoulou, K. S. Namjoshi, and C. S. Păsăreanu, *Compositional Reasoning*. Springer International Publishing, 2018, pp. 345–383.
- [7] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 3rd ed. Springer, 2021.
- [8] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, "Supremica—an efficient tool for large-scale discrete event systems," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5794–5799, 2017, 20th IFAC World Congress.
- [9] Y. Collette and P. Siarry, *Multiobjective Optimization: Principles and Case Studies*, ser. Decision Engineering. Springer Berlin Heidelberg, 2013.
- [10] D. Giannakopoulou, C. Pasareanu, and H. Barringer, "Assumption generation for software component verification," in *Proceedings 17th IEEE International Conference on Automated Software Engineering*, 2002, pp. 3–12.
- [11] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Generating phenotypical erroneous human behavior to evaluate human–automation interaction using model checking," *International Journal of Human-Computer Studies*, vol. 70, no. 11, pp. 888–906, 2012.
- [12] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Springer International Publishing, 2015, pp. 487–495.
- [13] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs, 2nd Edition*. London: Wiley, 2006.
- [14] T. T. Tun, A. Bennaceur, and B. Nuseibeh, "OASIS: Weakening user obligations for security-critical systems," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020, pp. 113–124.
- [15] U.S. Attorney's Office Eastern District of Kentucky, "Clay county officials and residents convicted on racketeering and voter fraud charges," Mar 2010. [Online]. Available: <https://archives.fbi.gov/archives/louisville/press-releases/2010/lo032510.htm>
- [16] J. Reason, *Human Error*. New York: Cambridge University Press, 1990.
- [17] G. Tel, *Introduction to Distributed Algorithms*, 2nd ed. Cambridge University Press, 2000.
- [18] D. Sempereboni and L. Viganò, "X-men: A mutation-based approach for the formal analysis of security ceremonies," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020, pp. 87–104.
- [19] M. L. Bolton and E. J. Bass, "Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking," in *2011 IEEE International Conference on Systems, Man, and Cybernetics*, 2011, pp. 1788–1794.
- [20] S. Maoz and J. O. Ringert, "GR(1) synthesis for LTL specification patterns," in *Proceedings of Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 96–106.
- [21] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911–938, 2012, in Commemoration of Amir Pnueli.
- [22] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, pp. 1–8.
- [23] T. A. Henzinger, J. Otop, and R. Samanta, "Lipschitz robustness of finite-state transducers," in *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, 2014, pp. 431–443.
- [24] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann, "Specification-centered robustness," in *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on, SIES 2011. Vasteras, Sweden, June 15-17, 2011*, 2011, pp. 176–185.
- [25] P. Tabuada, A. Balkan, S. Y. Caliskan, Y. Shoukry, and R. Majumdar, "Input-output robustness for discrete systems," in *International Conference on Embedded Software (EMSOFT)*. ACM, 2012, pp. 217–226.
- [26] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann, "Robustness in the presence of liveness," in *Computer Aided Verification (CAV)*, vol. 6174. Springer, 2010, pp. 410–424.
- [27] T. Kobayashi, R. Salay, I. Hasuo, K. Czarnecki, F. Ishikawa, and S. Katsumata, "Robustifying controller specifications of cyber-physical systems against perceptual uncertainty," in *International Symposium on NASA Formal Methods (NFM)*, 2021, pp. 198–213.

- [28] U. Topcu, N. Ozay, J. Liu, and R. M. Murray, “On synthesizing robust discrete controllers under modeling uncertainty,” in *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '12. Association for Computing Machinery, 2012, p. 85–94.
- [29] R. Meira-Góes, E. Kang, S. Lafortune, and S. Tripakis, “On tolerance of discrete systems with respect to transition perturbations,” *arXiv:2110.04200 [eess.SY]*, 2021.
- [30] T. Anevlavis, D. Neider, M. Philippe, and P. Tabuada, “Evrostos: The RLTL verifier,” in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. Association for Computing Machinery, 2019, p. 218–223.
- [31] P. Tabuada and D. Neider, “Robust Linear Temporal Logic,” in *25th EACSL Annual Conference on Computer Science Logic (CSL)*, vol. 62, 2016, pp. 10:1–10:21.
- [32] F. Bucchiarri, T. Eiter, G. Gottlob, and N. Leone, “Enhancing model checking in verification by ai techniques,” *Artificial Intelligence*, vol. 112, no. 1, pp. 57–104, 1999.
- [33] M. V. de Menezes, S. do Lago Pereira, and L. N. de Barros, “System design modification with actions,” in *Advances in Artificial Intelligence – SBIA 2010*, A. C. da Rocha Costa, R. M. Vicari, and F. Tonidandel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 31–40.
- [34] G. Chatzieftheriou, B. Bonakdarpour, S. A. Smolka, and P. Katsaros, “Abstract model repair,” in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 341–355.
- [35] Y. Ding and Y. Zhang, “A logic approach for LTL system modification,” in *Foundations of Intelligent Systems*, M.-S. Hacid, N. V. Murray, Z. W. Raś, and S. Tsumoto, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 435–444.

A Provably Correct Floating-Point Implementation of Well Clear Avionics Concepts

Nikson Bernardes Fernandes Ferreira^{id*}, Mariano M. Moscato^{id†}, Laura Titolo^{id†}, and Mauricio Ayala-Rincón^{id*‡}

^{*}Department of Computer Science

[‡]Department of Mathematics

University of Brasilia, Brasilia, Brazil

Email: niksonber@gmail.com, ayala@unb.br

[†]Analytical Mechanics Associates, Hampton, USA

Email: {mariano.moscato,laura.titulo}@ama-inc.com

Abstract—The NASA DAIDALUS library provides formal definitions for Detect-and-Avoid avionics concepts such as when an aircraft is well-clear with respect to the surrounding air traffic, i.e., it does not operate in such proximity to create a collision hazard. While several properties are proven correct for DAIDALUS assuming ideal real number arithmetic, an actual implementation that uses floating-point numbers may behave unexpectedly because of round-off errors and run-time exceptions. This paper presents an experience report on the application of a formal methods toolchain to extract and verify floating-point C code from a real-valued specification of the well-clear module of DAIDALUS. This toolchain comprises the PVS theorem prover, the PRECiSA floating-point analyzer and code generator, and the Frama-C analysis suite. The generated code is automatically instrumented to detect when the control flow of the floating-point program may diverge from the ideal real number specification, and it is annotated with contracts that state the maximum accumulated round-off error. The absence of overflows is also formally verified for the generated code. In order to apply the toolchain to an industrial case study such as DAIDALUS, a formally verified pre-processing of the input specification is performed, which includes a program slicing and several semantic-preserving simplifications.

Index Terms—Program Verification, Floating-Point, PVS, Detect-and-Avoid

I. INTRODUCTION

Midair conflicts are one of the most dangerous situations that may occur in the airspace domain. The USA Federal Aviation Administration (FAA) reported that over forty midair collisions occurred from January 2009 through December 2013 [1]. The primary mitigation to such situations is the longstanding principle of *See and Avoid*. In short, it states that a person operating an aircraft has the responsibility to remain vigilant to see and avoid nearby traffic [2]. The advent of Unmanned Aerial Systems (UAS) and their incorporation into the airspace introduced the need to restate this concept in terms suitable for aircraft with no crew onboard. The *Detect and Avoid* (DAA) concept emerged then as an effort to support the integration of UAVs into civil airspace. A DAA system is required to provide alerting and guidance to avoid potential conflicts.

Diverse industrial and governmental actors proposed algorithmic DAA solutions. Among them, NASA developed

the Detect and Avoid Alerting Logic for Unmanned Systems library (DAIDALUS¹) [3]. DAIDALUS provides prototypical open-source implementations in Java and C++, which were included as reference implementations of the DAA functional requirements described in RTCA's Minimum Operational Performance Standards (MOPS) DO-365 [4]. One distinguishing characteristic of DAIDALUS is that it also provides formal specifications of the algorithms along with proofs for correctness and safety properties on them, mechanically checked within the Prototype Verification System (PVS) [5]. These proofs assume ideal real number arithmetic. However, when implemented using floating-point numbers, the properties may no longer hold because of round-off errors and runtime exceptions. The adherence of the implementations to the behavior modeled by the formal specifications was checked using a testing-based approach [6]. While such an approach is usually enough for non-critical applications, the correctness of DAA implementations calls for a higher level of assurance. Given the numerical nature of several functions in DAIDALUS, it is important to provide formal guarantees on the finite-precision implementation concerning the expected behavior specified using real-numbers arithmetic.

In the past, an integrated toolchain has been proposed to automatically extract and verify floating-point C code from real-valued specifications [7]. This toolchain consists of the PVS theorem prover, the PRECiSA floating-point analyzer and code generator ([8], [9]), and the Frama-C tool suite [10]. In a nutshell, PRECiSA automatically generates a floating-point C implementation from a PVS real number specification. The extracted C code contains program contracts that relate the floating-point computations with their ideal counterpart by the maximum round-off error that may occur. These contracts enable the use of the Frama-C analysis suite which automatically generates a set of verification conditions that can be proven correct with the help of diverse backends. The toolchain proposed in [7] included a customization on Frama-C that allowed it to generate the verification conditions in the language of PVS and connect them with the NASA PVS

¹DAIDALUS is available at <https://github.com/nasa/daidalus>.

library (NASALib).

In [7], this technique was applied to one of the core functions of DAIDALUS. This paper describes the application and adaptation of this technique to one of the main modules in DAIDALUS which is devoted to the definition of *well-clear* concepts. Two aircraft are considered to be well clear of each other if appropriate distance and time variables determined by the relative aircraft states remain outside a set of predefined threshold values. Remaining outside of these threshold values guarantees they have adequate separation in relation to the surrounding traffic; therefore, midair collisions are not expected.

The toolchain presented in [7] could not be applied directly to the DAIDALUS specification because the code generation capability of PRECiSA, at its current stage, does not support some of the features of the PVS language used to formally define Well-Clear, such as abstract data types and higher-order functions. In addition, the complexity of the target module, given by the number and nature of the interactions between the functions composing it and the wide ramification of the control flow graph of the whole library, impacts the efficiency of the analysis performed by PRECiSA and the legibility of the results of this analysis. In order to make the DAIDALUS specification manageable by the toolchain, this paper proposes to apply a semantic-preserving program slicing on a simplification from higher-order to first-order declarations. This program rewriting improved the performance of the generation and verification of the C code significantly. The obtained program is formally proven equivalent to the original specification within the PVS theorem prover. In addition, a new PVS floating-point formalization is used. This formalization extends the one used in [7] with explicit handling for special values such as NaNs and infinities. This change positively impacted the analysis by enabling the verification of the absence of these values. It also significantly improved the performance of the type-checking in PVS. However, it resulted in many of the proof strategies developed in the past being unusable. Part of the work presented in this paper focuses on fixing and adapting the proofs generated by PRECiSA to this new formalization. More information about the PRECiSA project and the files related to the work presented in this paper can be found at <https://shemesh.larc.nasa.gov/fm/PRECiSA/>.

The paper is organized as follows. Section II describes DAIDALUS and explains the well-clear concept. An overview of the verification approach is presented in Section III. The application of the slicing technique to the original specification is detailed in Section IV. Then, Section V explains the code extraction and the program instrumentation and verification. Finally, Section VI provides a brief discussion of the most relevant outcomes of this work, Section VII discusses the related work, and Section VIII concludes the paper.

II. THE DAIDALUS LIBRARY

DAIDALUS is a software library developed at NASA that implements a Detect-and-Avoid alerting logic for unmanned systems. In DAIDALUS, the condition of Well-Clear is defined in the context of an encounter between two aircraft, usually

called the *ownership* and the *intruder*. These conditions are stated in an *intruder-centric* manner, meaning that the information describing the encounter is expressed relative to the state of the intruder. In particular, DAIDALUS includes definitions determining when the aircraft are in a situation of violation of well-clear. This violation occurs when (a) the two aircraft are already close enough, or (b) they will be close enough if they keep the same orientation and velocity. This notion is expressed in terms of horizontal (1) and vertical (2) well-clear violation.

This section presents a high-level description of the well-clear concepts defined in [11]. For details and further explanation, the reader is referred to that work. In the following, \mathbf{s} and \mathbf{v} denote vectors of dimension 3 and the subindices x , y , and z are used to indicate their first, second, and third component respectively. Two-dimensional vectors are used to describe the horizontal position ($\mathbf{s}_h \stackrel{\text{def}}{=} (s_x, s_y)$) and velocity ($\mathbf{v}_h \stackrel{\text{def}}{=} (v_x, v_y)$) of the ownership with respect to the intruder. Additionally, $\|\cdot\|$ denotes the Euclidean norm.

$$\text{WCV}_H(\mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \|\mathbf{s}_h\| \leq \delta_d \vee (0 \leq \tau_{mod}(\mathbf{s}_h, \mathbf{v}_h) \leq \delta_t \wedge d_{cpa}(\mathbf{s}_h, \mathbf{v}_h) \leq \delta_{hmd}) \quad (1)$$

The values δ_d , δ_t , and δ_{hmd} are parameters of the model, used as thresholds for distance and time. The function τ_{mod} , defined below, is an approximation for the time of closest point of approach, i.e., the instant in which both aircraft would be closer to each other than in any other moment. Below, and in the rest of this paper, the dot product between two vectors (for example, \mathbf{a} and \mathbf{b}) is denoted by their juxtaposition (\mathbf{ab}).

$$\tau_{mod}(\mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \begin{cases} \frac{\delta_d^2 - \mathbf{s}_h^2}{\mathbf{s}_h \mathbf{v}_h} & \text{if } \mathbf{s}_h \mathbf{v}_h < 0 \\ -1 & \text{otherwise} \end{cases} \quad (2)$$

The function d_{cpa} calculates the projected horizontal distance between the aircraft at their closest point of approach, assuming the velocity and orientation remain constant. The definition of d_{cpa} relies on the actual calculation of the *time of closest point of approach* (t_{cpa}). Both notions are formally stated below.

$$d_{cpa}(\mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \|\mathbf{s}_h + t_{cpa}(\mathbf{s}_h, \mathbf{v}_h) \mathbf{v}_h\| \quad (3)$$

$$t_{cpa}(\mathbf{s}_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \begin{cases} \frac{\mathbf{s}_h \mathbf{v}_h}{v_h^2} & \text{if } \|\mathbf{v}_h\| > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

The violation of vertical well clear is defined analogously to its horizontal counterpart; using the scalars vertical position s_z and velocity v_z , and the time to co-altitude (t_{coa}) instead of the time to the closest point of approach.

$$\text{WCV}_V(s_z, v_z) \stackrel{\text{def}}{=} |s_z| \leq \delta_z \vee 0 \leq t_{coa}(s_z, v_z) \leq \delta_{tcoa} \quad (5)$$

$$t_{coa}(s_z, v_z) \stackrel{\text{def}}{=} \begin{cases} \frac{-s_z}{v_z} & \text{if } s_z v_z < 0 \\ -1 & \text{otherwise} \end{cases} \quad (6)$$

Given their relative position and velocity, two aircraft are considered to be in well-clear violation when both horizontal and vertical violations occur.

$$\text{WCV}(\mathbf{s}, \mathbf{v}) \iff \text{WCV}_V(s_z, v_z) \wedge \text{WCV}_H(\mathbf{s}_h, \mathbf{v}_h) \quad (7)$$

The DAIDALUS library also provides conflict detection algorithms whose purpose is to check whether the well-clear condition is predicted to be violated within a given timeframe. The function WCVint_V computes a time interval, included in a given lookahead interval $\mathbf{t} = [b, t] \subset \mathbb{R}$, in which vertical well-clear is violated at every moment. If no such interval exists, the empty set is returned.

$$\text{WCVint}_V(\mathbf{t}, s_z, v_z) \stackrel{\text{def}}{=} \begin{cases} \mathbf{t} & \text{if } v_z = 0 \wedge |s_z| \leq \delta_z \\ \emptyset & \text{if } v_z = 0 \wedge |s_z| > \delta_z \\ [\max(b, c_0), \min(t, c_F)] & \text{if } v_z \neq 0 \wedge b \leq c_0, c_F \leq t \\ \emptyset & \text{otherwise} \end{cases} \quad (8)$$

where $c_0 \stackrel{\text{def}}{=} \frac{-\text{sign}(v_z) \max(\delta_z, \delta_{tcoa}|v_z|) - s_z}{v_z}$ and $c_F \stackrel{\text{def}}{=} \frac{-\text{sign}(v_z) \delta_z - s_z}{v_z}$. Definitions such as c_0 and c_F , for which no equation numbers are provided, should be understood as syntactic abbreviations used to improve the presentation.

Similarly, the function WCVint_H returns a time interval included in $[0, t]$ in which the condition of horizontal well-clear is violated at every moment, if such interval exists.

$$\text{WCVint}_H(t, s_h, \mathbf{v}_h) \stackrel{\text{def}}{=} \begin{cases} [0, t] & \text{if } a = 0 \wedge s_h^2 \leq \delta_D^2 \\ [0, \min(t, \Theta_{s_h, \mathbf{v}_h}^+)] & \text{if } a \neq 0 \wedge s_h^2 \leq \delta_D^2 \\ \emptyset & \text{if } s_h^2 > \delta_D^2 \wedge (s_h \mathbf{v}_h \geq 0 \vee \Delta_{a,b,c} < 0) \\ [\max(0, r_{a,b,c}^-), \min(t, \Theta_{s_h, \mathbf{v}_h}^+)] & \text{if } s_h^2 > \delta_D^2 \wedge s_h \mathbf{v}_h < 0 \wedge \Delta_{a,b,c} \geq 0 \wedge \\ & \Delta_{s_h, \mathbf{v}_h}^{\mathbb{R} \times \mathbb{R}} \geq 0 \wedge r_{a,b,c}^- \leq t \\ \emptyset & \text{otherwise} \end{cases} \quad (9)$$

where

- $a \stackrel{\text{def}}{=} \mathbf{v}_h^2$,
- $b \stackrel{\text{def}}{=} 2 s_h \mathbf{v}_h + \delta_t \mathbf{v}_h^2$,
- $c \stackrel{\text{def}}{=} s_h^2 + \delta_t s_h \mathbf{v}_h - \delta_D^2$,
- $\Delta_{a,b,c} \stackrel{\text{def}}{=} b^2 - 4ac$,
- $r_{a,b,c}^- \stackrel{\text{def}}{=} \frac{1}{2a} (-b - \sqrt{b^2 - 4ac})$,
- $\Theta_{s_h, \mathbf{v}_h}^+ \stackrel{\text{def}}{=} \frac{1}{\mathbf{v}_h^2} (-s_h \mathbf{v}_h + \sqrt{(s_h \mathbf{v}_h)^2 - 4 \mathbf{v}_h^2 (s_h^2 - \delta_D^2)})$, and
- $\Delta_{s_h, \mathbf{v}_h}^{\mathbb{R} \times \mathbb{R}} \stackrel{\text{def}}{=} \delta_D^2 \mathbf{v}_h^2 - (s_h \mathbf{v}_h^\perp)^2$.

The two functions defined above can be used to calculate a time interval of well-clear violation. In the following, $\mathbf{V} \stackrel{\text{def}}{=} \text{WCVint}_V(\mathbf{t}, s_z, v_z)$, and $\mathbf{H} \stackrel{\text{def}}{=} \text{WCVint}_H(t_{\text{end}} - t_{\text{begin}}, s_h + lb(\mathbf{V}) \mathbf{v}_h, \mathbf{v}_h)$ where $[t_{\text{begin}}, t_{\text{end}}] = \mathbf{t}$.

$$\text{WCVint}(\mathbf{t}, \mathbf{s}, \mathbf{v}) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \mathbf{V} = \emptyset \\ \mathbf{V} & \text{if } lb(\mathbf{V}) = ub(\mathbf{V}) \wedge \text{WCV}_H(s_h + lb(\mathbf{V}) \mathbf{v}_h, \mathbf{v}_h) \\ \emptyset & \text{if } lb(\mathbf{V}) = ub(\mathbf{V}) \wedge \neg \text{WCV}_H(s_h + lb(\mathbf{V}) \mathbf{v}_h, \mathbf{v}_h) \\ \emptyset & \text{if } ub(\mathbf{V}) - lb(\mathbf{V}) > 0 \wedge \mathbf{H} = \emptyset \\ [lb(\mathbf{H}) + lb(\mathbf{V}), ub(\mathbf{H}) + lb(\mathbf{V})] & \text{otherwise} \end{cases} \quad (10)$$

where lb and ub return the lower and upper end-point of a given non-empty closed interval, respectively.

The predicate WCV? determines if there is a subinterval of \mathbf{t} where a violation of well-clear occurs.

$$\text{WCV?}(\mathbf{t}, \mathbf{s}, \mathbf{v}) \iff \text{WCVint}(\mathbf{t}, \mathbf{s}, \mathbf{v}) \neq \emptyset \quad (11)$$

The equations in this Section are a simplified version of the definitions originally presented in [12] where properties and additional definitions can be found.

III. VERIFICATION APPROACH

The verification approach used in this paper relies on the integrated toolchain presented in [7] which is composed of several formal methods tools:

- PRECiSA [8], [9], a static analyzer for floating-point programs,²
- the global optimizer Kodiak [13],³
- Frama-C [10], a collaborative tool suite for the analysis of C code, and
- the Prototype Verification System (PVS) [14], a verification environment consisting of a specification language, a large number of predefined theories, and an interactive theorem prover.

PRECiSA is a static analyzer for floating-point programs that computes sound and accurate round-off error estimations and provides support for a large variety of mathematical operators and programming language constructs. Given a floating-point program, PRECiSA generates a symbolic error expression modeling an over-approximation of the round-off error that may occur in the program. This error expression is a function of the input variables of the program and their associated rounding error. Given input ranges for these variables, PRECiSA uses the Kodiak global optimizer to maximize the round-off error expressions. Additionally, PRECiSA generates formal certificates ensuring that these bounds are correct with respect to the IEEE Standard for Floating-Point Arithmetic (IEEE 754). These certificates are output in the language of PVS, which can be used to mechanically check their validity. Even though proofs in PVS are expected to be carried out following user guidance in general, this process is automatic thanks to an available collection of proof strategies targeted to this particular application.

One of the more recent extensions of PRECiSA [7], includes the addition of a code-extraction capability that automatically generates a floating-point C implementation from a real-number function expressed in the language of PVS. The generated C code is instrumented to detect whether the floating-point computational flow diverges from its ideal real number counterpart, and it is automatically annotated with *program contracts* stating the formal relationship between real and floating-point computations. These contracts are written in the ANSI/ISO C Specification Language (ACSL) which can be processed by Frama-C. Frama-C is a collaborative modular platform for the analysis of C programs. In this work, the Frama-C weakest precondition (WP) plug-in is used to

²PRECiSA is available at <https://github.com/nasa/PRECiSA>.

³Kodiak is available at <https://shemesh.larc.nasa.gov/fm/Kodiak/>.

generate verification conditions in the language of PVS and it is customized to integrate the PVS certificates generated by PRECiSA into the proof of such verification conditions.

An overview of the verification approach applied to the well-clear calculations in DAIDALUS is depicted in Fig. 1. First, the PVS higher-order specification of DAIDALUS is manually rewritten using only first-order constructs. This is achieved by replacing each higher-order argument with a specific function instantiation. For instance, the original definition for the violation of vertical well-clear is parametric on the technique used to approximate the time of closest point of approach. Such parameter was replaced by a particular concrete calculation of this time, resulting in the definition shown in (5). While this kind of simplification cannot be performed in general by preserving the semantics of the program, the nature of the higher-order parameters used in DAIDALUS allowed to simplify the definitions even though this change resulted in a less general specification. The first-order specification is mechanically proved equivalent to the higher-order one within PVS. The higher-order specification is instantiated with the specific functions used in the first-order one as parameters. This simplification was necessary since PRECiSA does not yet provide support for the use of higher-order arguments in the input program.

Then, a program slicing technique is applied to the first-order specification to obtain a set of simpler descriptions. This program slicing is proved to be semantically equivalent to the original specification. The next section provides more details on the slicing process and the resulting fragmentation of the specification.

Each specification slice is input to PRECiSA which automatically extracts the corresponding annotated floating-point C code and generates the corresponding PVS proof certificates ensuring the correctness of the round-off error estimations used in the code extraction and instrumentation. Since the extracted C code implements each of the slices of the original specification, it is necessary to develop a top-level module in C providing the same functionality as the involved functions in DAIDALUS. Basically, this top-level function selects the proper slice given an unrestricted input and calls the corresponding C function. The top-level function was manually developed and annotated with specific program contracts to ensure its compliance with the original specification. The details about this function are explained in Sect. V.

Frama-C was used to analyze both the automatically generated C functions from each slice and the top-level function. Finally, the verification conditions output by Frama-C were proved in the PVS theorem prover. While these proofs were made interactively for this particular application of the technique, they can be automated since they rely heavily on the structure of the program. The automation of the proofs is left as future work.

IV. SPECIFICATION SLICING

Program Slicing [15], [16] is a technique generally applied on source code to analyze particular behaviors of software.

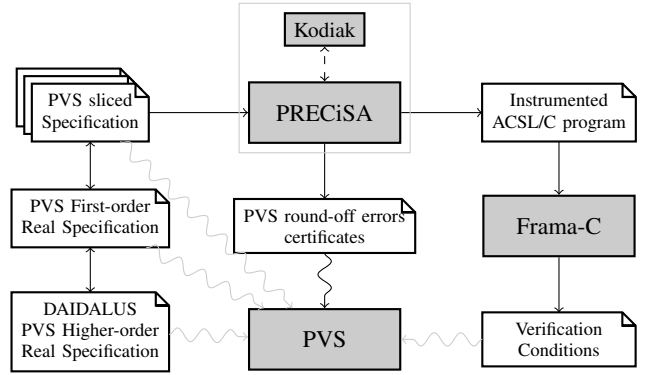


Fig. 1. Workflow of the verification approach.

TABLE I
NAME OF THE MAIN PREDICATE ON EACH SLICE.

horizontal separation	vertical separation	decrease $v_z < 0$	maintain $v_z = 0$	increase $v_z > 0$
alter	$v_x \neq 0 \vee v_y \neq 0$	$WCV?_{\downarrow}^{\leftarrow}$	$WCV?_{\rightarrow}^{\leftarrow}$	$WCV?_{\rightarrow}^{\leftarrow}$
maintain	$v_x = 0 \wedge v_y = 0$	$WCV?_{\downarrow}$	$WCV?_{\rightarrow}$	$WCV?_{\uparrow}$

However, in this work, it was applied to the specification of the definitions presented in Section II as a way to address scalability issues in the PRECiSA code extraction. The slicing approach used in this work was first introduced by Canfora et al. [17] and Ning et al. [18] and it is known as *Conditioned Slicing* [19]. Essentially, it proposes the decomposition of a program into independent simpler parts, called *slices*, according to its control flow graph as defined by the guards in the branching instructions appearing in the program. Each slice runs under the assumption of specific restrictions on the inputs, determining the execution of a particular path in the control flow graph of the original program.

For this case study, the criterion used to select the restriction on the inputs producing the slices was focused on the different cases determined by the possible relative velocities of the aircraft, as given in the branches of the source code. Three possible situations regarding relative vertical velocity were considered: maintaining separation (null relative vertical velocity), increasing separation (positive relative vertical velocity), and decreasing separation (negative relative vertical velocity). In terms of horizontal relative velocity, only the cases *altering separation* or *maintaining separation* were considered. Hence, a total of six slices were defined by applying this criterion on the predicate presented in (11) which is the topmost declaration in the Well-Clear module. Table I shows the name of the topmost predicate in each slice.

To exemplify how the slices are actually defined, Equation (12) shows the entry point for the slice describing a situation of maintaining vertical separation and altering horizontal separation, given by the conditions $v_z = 0$ and $v_x \neq 0 \vee v_y \neq 0$.

$$WCV?_{\rightarrow}^{\leftarrow}(t, s, v) \iff (|s_z| \leq \delta_z \wedge WCV_H?_{\rightarrow}^{\leftarrow}(t_F - t_0, s_x + t_0 v_x, s_y + t_0 v_y, v_x, v_y)) \quad (12)$$

where $WCV_H?_{\rightarrow}^{\leftarrow}$ is a predicate checking whether the WCV_{int_H}

function from (9) returns a non-empty interval when the conditions defining the slice hold.

$$\text{WCV}_H?^{\leftarrow}(t, \mathbf{s}_h, \mathbf{v}_h) \iff \begin{cases} r_{a, 2\mathbf{s}_h \mathbf{v}_h, \mathbf{s}_h^2 - \delta_D^2}^+ \geq 0 & \text{if } \mathbf{s}_h^2 \leq \delta_D^2 \\ \left(0 \leq r_{a, b, c}^- \leq t \wedge r_{a, b, c}^- \leq r_{a, 2\mathbf{s}_h \mathbf{v}_h, \mathbf{s}_h^2 - \delta_D^2}^+ \right) \vee \\ \left(r_{a, b, c}^- < 0 \wedge r_{a, 2\mathbf{s}_h \mathbf{v}_h, \mathbf{s}_h^2 - \delta_D^2}^+ \geq 0 \right) & \text{if } \mathbf{s}_h^2 > \delta_D^2 \wedge \mathbf{s}_h \mathbf{v}_h < 0 \wedge \Delta_{a, b, c} \geq 0 \wedge \\ \Delta_{\mathbf{s}_h, \mathbf{v}_h}^{\mathbb{R} \times \mathbb{R}} \geq 0 \wedge r_{a, b, c}^- \leq t & \\ \text{false} & \text{otherwise} \end{cases} \quad (13)$$

Above, $a, b, c, \Delta_{a, b, c}, r_{a, b, c}^-$, and $\Delta_{\mathbf{s}_h, \mathbf{v}_h}^{\mathbb{R} \times \mathbb{R}}$ are as in (9), and $r_{a, b, c}^+ \stackrel{\text{def}}{=} \frac{-b + \sqrt{\Delta_{a, b, c}}}{2a}$. The following theorem validates that the decomposition proposed by the slicing process correctly captures the semantics of the original specification.

Theorem 1: [Slicing Correctness] For all time interval $t \in \mathbb{R}$ and all pair of three-dimensional vectors $\mathbf{s}, \mathbf{v} \in \mathbb{R}^3$, $\text{WCV}?(t, \mathbf{s}, \mathbf{v})$ holds if and only if

$$\begin{cases} \text{WCV}?\uparrow(t, \mathbf{s}, \mathbf{v}) & \text{when } \mathbf{v}_z < 0 \wedge (\mathbf{v}_x \neq 0 \vee \mathbf{v}_y \neq 0) \\ \text{WCV}?\downarrow(t, \mathbf{s}, \mathbf{v}) & \text{when } \mathbf{v}_z < 0 \wedge (\mathbf{v}_x = 0 \wedge \mathbf{v}_y = 0) \\ \text{WCV}?\uparrow(t, \mathbf{s}, \mathbf{v}) & \text{when } \mathbf{v}_z > 0 \wedge (\mathbf{v}_x \neq 0 \vee \mathbf{v}_y \neq 0) \\ \text{WCV}?\downarrow(t, \mathbf{s}, \mathbf{v}) & \text{when } \mathbf{v}_z > 0 \wedge (\mathbf{v}_x = 0 \wedge \mathbf{v}_y = 0) \\ \text{WCV}?\uparrow(t, \mathbf{s}, \mathbf{v}) & \text{when } \mathbf{v}_z = 0 \wedge (\mathbf{v}_x \neq 0 \vee \mathbf{v}_y \neq 0) \\ \text{WCV}?\downarrow(t, \mathbf{s}, \mathbf{v}) & \text{when } \mathbf{v}_z = 0 \wedge (\mathbf{v}_x = 0 \wedge \mathbf{v}_y = 0) \end{cases}$$

As one of the contributions of the present work, the definition of the predicates in Table I and the theorem above, along with all the ad-hoc lemmas needed in its proof, were mechanically checked using the PVS theorem prover.

V. CODE EXTRACTION AND VERIFICATION

The round-off error occurring in the guards of conditionals can provoke the floating-point control flow to diverge with respect to its ideal real-numbers counterpart. The guards in a program where such a phenomenon can occur are called *unstable conditions*. As another of its features, the code extracted by PRECiSA is instrumented to emit a warning when such conditions may occur. This instrumentation is based on the program transformation presented in [20]. In the rest of this section, the code extraction procedure is outlined. As part of the verification presented in this paper, this procedure was applied to each of the slices of the specification described in the previous section.

A. Processing the slices

Given the specification of a real-valued program, understood as a collection of functions collaborating to compute a determined result, and the desired floating-point format (single or double precision), PRECiSA replaces each real arithmetic operator with its floating-point counterpart. Then, it modifies all the conditional statements. Each guard is replaced by a more restrictive one that takes into account the round-off error that may occur. This round-off error is computed with PRECiSA.

Additionally, a warning is emitted when the original guard may be evaluated differently in real and floating-point arithmetic. This warning is denoted by a distinguished value disjoint from the floating-point domain. Getting such a warning as the result of a computation implies that, for the inputs provided, it cannot be guaranteed that the floating-point execution follows the same control flow as its real-valued counterpart. This divergence could provoke a much bigger error in the numerical final result than the accumulated round-off error that occurred in the evaluation of an arithmetic expression. It is worth noting that, since the round-off error estimation computed by PRECiSA is a sound over-approximation of the error that may occur, false warnings may arise. However, it is guaranteed that all the instabilities are detected.

For instance, the floating-point function depicted below is the result of applying this instrumentation on the function τ_{mod} , defined by (2), whose goal is to approximate the time of closest point of approach of two aircraft. Here and in the rest of this paper, the tilde over a variable, an operator, or a function denotes the fact that it belongs to the floating-point domain.

$$\begin{aligned} \widetilde{\tau_{mod}}(\widetilde{s}_x, \widetilde{v}_x, \widetilde{s}_y, \widetilde{v}_y, \epsilon) & \\ = \text{if } \widetilde{s}_x \widetilde{\sim} \widetilde{v}_x \widetilde{\sim} \widetilde{s}_y \widetilde{\sim} \widetilde{v}_y < -\epsilon & \\ \text{then } (\delta_d \widetilde{\sim} \delta_d \widetilde{\sim} \widetilde{s}_x \widetilde{\sim} \widetilde{s}_x \widetilde{\sim} \widetilde{s}_y \widetilde{\sim} \widetilde{s}_y) \widetilde{\sim} (\widetilde{s}_x \widetilde{\sim} \widetilde{v}_x \widetilde{\sim} \widetilde{s}_y \widetilde{\sim} \widetilde{v}_y) & \\ \text{elseif } \widetilde{s}_x \widetilde{\sim} \widetilde{v}_x \widetilde{\sim} \widetilde{s}_y \widetilde{\sim} \widetilde{v}_y \geq \epsilon \text{ then } -1 & \\ \text{else } \omega & \end{aligned} \quad (14)$$

When the evaluation of $\widetilde{s}_x \widetilde{\sim} \widetilde{v}_x \widetilde{\sim} \widetilde{s}_y \widetilde{\sim} \widetilde{v}_y$ lies in the interval $[-\epsilon, \epsilon)$ the function above signals a warning by returning the value ω . The new argument of the function, ϵ , is expected to be an over-approximation of the round-off error that may occur when computing $\widetilde{s}_x \widetilde{\sim} \widetilde{v}_x \widetilde{\sim} \widetilde{s}_y \widetilde{\sim} \widetilde{v}_y$.

Listing 1 shows the C code and the ACSL annotations generated by PRECiSA for the function $\widetilde{\tau_{mod}}$. The C function `taumod_fp` mimics the definition of $\widetilde{\tau_{mod}}$, while the annotations express the contracts enforcing the properties explained above. The type **double'** is the implementation of a union type consisting of the **double** datatype and the ω warning value⁴. In ACSL, the keywords **requires** and **ensures** are used to describe preconditions and postconditions of a function, respectively. The main precondition of `taumod_fp` (line 9) restricts ϵ to be a non-negative representable numeric value, i.e., it cannot be an infinite or a NaN. The postcondition on line 10 ensures that when the result is not ω , it is the same as the one computed by the floating point version of τ_{mod} (before the instrumentation). The following postcondition (line 11) states that, if additionally to the result not being ω , the argument ϵ is a sound approximation of the round-off error of the guard, then no unstable conditions occur, meaning that the guard has the same value under both floating-point and real-valued arithmetic. This latter condition is expressed by the predicate *stable_paths* _{τ_m} defined on lines 5-7.

⁴For ease of reading no explicit projection of the values in the union type are used.

```

1 /*@
2 double taumod_fp(double  $\widetilde{s_x}, \widetilde{v_x}, \widetilde{s_y}, \widetilde{v_y}$ ) = \let  $\widetilde{g} = \widetilde{s_x} \widetilde{v_x} \widetilde{+} \widetilde{s_y} \widetilde{v_y}$ ;
3    $\widetilde{g} < 0$  ? ( $\delta_d \widetilde{+} \delta_d \widetilde{-} \widetilde{s_x} \widetilde{+} \widetilde{s_y} \widetilde{+} \widetilde{s_y} \widetilde{+} \widetilde{s_y}$ ) /  $\widetilde{g}$  : -1.0;
4
5 predicate stable_paths $\tau_m$ (real  $v_x, v_y, s_x, s_y$ , double  $\widetilde{v_x}, \widetilde{v_y}, \widetilde{s_x}, \widetilde{s_y}$ ) =
6   \let  $\widetilde{g} = \widetilde{s_x} \widetilde{v_x} \widetilde{+} \widetilde{s_y} \widetilde{v_y}$ ; \let  $g = s_x * v_x + s_y * v_y$ ;
7   ( $g < 0 \wedge \widetilde{g} < 0$ )  $\vee$  ( $g \geq 0 \wedge \widetilde{g} \geq 0$ );
8
9 requires: \is_finite?( $\epsilon$ )  $\wedge \epsilon \geq 0$ ;
10 ensures: \result  $\neq \omega \Rightarrow$  \result = taumod_fp( $\widetilde{s_x}, \widetilde{v_x}, \widetilde{s_y}, \widetilde{v_y}$ )
11 ensures:  $\forall$  real  $v_x, v_y, s_x, s_y$ ;
12   \result  $\neq \omega \wedge |(\widetilde{s_x} \widetilde{v_x} \widetilde{+} \widetilde{s_y} \widetilde{v_y}) - (s_x * v_x + s_y * v_y)| \leq \epsilon$ 
13    $\Rightarrow$  stable_paths $\tau_m$ ( $v_x, v_y, s_x, s_y, \widetilde{v_x}, \widetilde{v_y}, \widetilde{s_x}, \widetilde{s_y}$ );
14 */
15 double' taumod_fp(double  $\widetilde{s_x}, \widetilde{v_x}, \widetilde{s_y}, \widetilde{v_y}, \epsilon$ ) {
16   if ( $\widetilde{s_x} \widetilde{+} \widetilde{v_x} \widetilde{+} \widetilde{s_y} \widetilde{+} \widetilde{v_y} < -\epsilon$ )
17     return ( $\delta_d \widetilde{+} \delta_d \widetilde{-} \widetilde{s_x} \widetilde{+} \widetilde{s_y} \widetilde{+} \widetilde{s_y} \widetilde{+} \widetilde{s_y}$ ) / ( $\widetilde{s_x} \widetilde{v_x} \widetilde{+} \widetilde{s_y} \widetilde{v_y}$ );
18   else if ( $\widetilde{s_x} \widetilde{+} \widetilde{v_x} \widetilde{+} \widetilde{s_y} \widetilde{+} \widetilde{v_y} \geq \epsilon$ )
19     return -1.0;
20   else
21     return  $\omega$ ;
22 }

```

Listing 1. C function and annotations generated by PRECiSA for τ_{mod} . Some syntactic simplifications were applied to the code in this listing for ease of reading, e.g., the use of the infix version of some operators and avoiding the repetition of the type of the function parameters, among others.

As already mentioned, PRECiSA is able to compute concrete error bounds for the guards when the user provides specific ranges for the arguments. For instance, let's assume that the ranges for the input variables are the following: $s_x \in [1, 185200]$ meters, $s_y \in [1, 15240]$ meters, $v_x \in [1, 720]$ meters per second, and $v_y \in [2, 720]$ meters per second. If double precision floating-point precision is selected, PRECiSA computes the round-off error bound $\epsilon = 4.58 \times 10^{-8}$ for the expression $\widetilde{s_x} \widetilde{+} \widetilde{v_x} \widetilde{+} \widetilde{s_y} \widetilde{+} \widetilde{v_y}$. Notably, PRECiSA also generates a formal certificate of the validity of this bound, which consists in a theorem that can be mechanically checked in the PVS theorem prover. For the τ_{mod} example, such a theorem can be expressed as it is shown below.

Theorem 2 (Error bound for the guard in τ_{mod}): For all real values v_x, v_y, s_x, s_y and floating-point numbers $\widetilde{v_x}, \widetilde{v_y}, \widetilde{s_x}, \widetilde{s_y}$, if $s_x \in [1, 185200]$, $s_y \in [1, 15240]$, $v_x \in [1, 720]$, $v_y \in [2, 720]$, and each float is the rounding of the respective real, then

$$|(\widetilde{s_x} \widetilde{+} \widetilde{v_x} \widetilde{+} \widetilde{s_y} \widetilde{+} \widetilde{v_y}) - (s_x * v_x + s_y * v_y)| \leq 4.58 \times 10^{-8}.$$

This theorem can be used to prove that one of the hypotheses of the ensures clause in lines 11-13 of Listing 1 holds when velocities and positions are in the specified ranges and ϵ is instantiated with the value from the theorem. Then, under these assumptions, such *ensures* guarantees that float and real flows do not diverge. Furthermore, the accumulated round-off error in the final result of `taumod_fp` is the maximum between the accumulated round-off errors in the expressions of each branch of the if-then-else that does not return a warning (ω). Again, PRECiSA is used to calculate a bound for such an error for every one of these expressions under the same assumption on the input values. In the case of τ_{mod} , these bounds are 6.62×10^{-2} for the first branch and 0 for the second, since -1 is a value that can be exactly representable in floating points. This kind of deduction can be repeated for

```

1 /*@
2 real  $\tau_{mod}$ (real  $s_x, v_x, s_y, v_y$ ) = \let  $g = s_x * v_x + s_y * v_y$ ;
3    $g < 0$  ? ( $\delta_d \widetilde{+} \delta_d \widetilde{-} s_x * s_x + s_y * s_y$ ) /  $g$  : -1;
4
5 ensures:  $\forall$  real  $v_x, v_y, s_x, s_y$ ;
6    $1 \leq v_x \leq 720 \wedge 2 \leq v_y \leq 720 \wedge 1 \leq s_x \leq 185200 \wedge 1 \leq s_y \leq 15240 \wedge$ 
7    $|\widetilde{v_x} - v_x| \leq \frac{ulp(v_x)}{2} \wedge |\widetilde{v_y} - v_y| \leq \frac{ulp(v_y)}{2} \wedge$ 
8    $|\widetilde{s_x} - s_x| \leq \frac{ulp(s_x)}{2} \wedge |\widetilde{s_y} - s_y| \leq \frac{ulp(s_y)}{2} \wedge$ 
9   \result  $\neq \omega$ 
10   $\Rightarrow |\text{result} - \tau_{mod}(s_x, v_x, s_y, v_y)| \leq 6.62 \times 10^{-2}$ ;
11 */
12 double' taumod_num(double  $\widetilde{s_x}, \widetilde{v_x}, \widetilde{s_y}, \widetilde{v_y}$ ) {
13   return taumod_fp( $\widetilde{s_x}, \widetilde{v_x}, \widetilde{s_y}, \widetilde{v_y}, 0x1.897f000000001p-25$ );
14 }

```

Listing 2. Concrete C function generated by PRECiSA for τ_{mod} .

each collection of input ranges provided by the user. PRECiSA summarizes it in a new annotated C function. This kind of function is called *concrete* or *numerical* in the context of this work and it only consists of a call to the function in Listing 1 instantiated with the error estimation computed by PRECiSA ($0x1.897f000000001p-25$ is the hexadecimal representation of the value 4.58×10^{-8}); the latter function, for contraposition, is called *generic*.

Listing 2 shows the concrete function and its associated annotations for τ_{mod} under the assumptions on the inputs described above. The formula on line 6 enforces the restriction on the inputs. Lines 7-8 states the relation between the real and the corresponding floating-point values, as in the hypothesis in Theorem 2. The program contract finishes ensuring that under the mentioned conditions, the difference between the result of the C function and its real-valued specification is at most the estimation computed by PRECiSA.

While Listings 1 and 2 serve as a useful hint to picture the implementation and contracts of more complex functions returning numeric values, the application of the code extraction process to the predicates in the sliced DAIDALUS specification, e.g., $WCV?_{\uparrow}^{\downarrow}$, $WCV?_{\downarrow}^{\uparrow}$, etc., deserves a closer look. For each predicate in the input specification, PRECiSA generates two pairs of C functions. Each of these pairs, as in the case of the functions with numeric return values, consists of a generic and a concrete C function. One of the pairs describes the cases in which the original predicate returns an affirmative answer (true) while the other characterizes the inputs for which a negative answer (false) is obtained. For instance, Listing 3 shows a fragment of the program contracts for the C functions extracted from the predicate $WCV?_{\uparrow}^{\downarrow}$. Actual definitions are omitted because of space limitations. The predicate *wcv_inc_mtn* (line 2) is the real number counterpart of $WCV?_{\uparrow}^{\downarrow}$, while *wcv_inc_mtn_fp* (line 3) is the ACSL floating-point version of $WCV?_{\uparrow}^{\downarrow}$. The predicate *WCVint_inc_mtn_plus* is a new predicate such that if it is satisfied then both *wcv_inc_mtn* and *wcv_inc_mtn_fp* are satisfied. This means that both real and floating-point evaluations of the predicate $WCV?_{\uparrow}^{\downarrow}$ are true. Conversely, *WCVint_inc_mtn_minus* is a new predicate such that if it is satisfied then neither *wcv_inc_mtn* nor *wcv_inc_mtn_fp* are satisfied. This means that both real and floating-point evaluations of the predicate $WCV?_{\uparrow}^{\downarrow}$ are false.

```

1 /*@
2 predicate wcv_inc_mtn (real b, t, v_x, v_y, v_z, s_x, s_y, s_z) = ...;
3 predicate wcv_inc_mtn_fp (double  $\tilde{b}, \tilde{t}, \tilde{s}_x, \tilde{s}_y, \tilde{s}_z, \tilde{v}_x, \tilde{v}_y, \tilde{v}_z$ ) = ...;
4 ...
5 ensures:  $\forall$  real b, t, v_x, v_y, v_z, s_x, s_y, s_z;
6   \result  $\neq \omega \wedge$  \result
7    $\Rightarrow (wcv\_inc\_mtn(b, t, v_x, v_y, v_z, s_x, s_y, s_z) \wedge$ 
8      $wcv\_inc\_mtn\_fp(\tilde{b}, \tilde{t}, \tilde{s}_x, \tilde{s}_y, \tilde{s}_z, \tilde{v}_x, \tilde{v}_y, \tilde{v}_z))$ ;
9 */
10 bool' WCVint_inc_mtn_plus (double  $\tilde{b}, \tilde{t}, \tilde{s}_x, \tilde{s}_y, \tilde{s}_z, \tilde{v}_x, \tilde{v}_y, \tilde{v}_z$ ) { ... }
11
12 ensures:  $\forall$  real b, t, v_x, v_y, v_z, s_x, s_y, s_z;
13   \result  $\neq \omega \wedge$  \result
14    $\Rightarrow (!wcv\_inc\_mtn(b, t, v_x, v_y, v_z, s_x, s_y, s_z) \wedge$ 
15      $!wcv\_inc\_mtn\_fp(\tilde{b}, \tilde{t}, \tilde{s}_x, \tilde{s}_y, \tilde{s}_z, \tilde{v}_x, \tilde{v}_y, \tilde{v}_z))$ ;
16 */
17 bool' WCVint_inc_mtn_minus (double  $\tilde{b}, \tilde{t}, \tilde{s}_x, \tilde{s}_y, \tilde{s}_z, \tilde{v}_x, \tilde{v}_y, \tilde{v}_z$ ) { ... }

```

Listing 3. Excerpt from the program contracts in the generic function generated by PRECiSA for the $wcv?$ predicate.

The return type of these C functions (**bool'**) represents the implementation of the union type between the **bool** datatype and the ω value. Hence, in lines 6 and 13, if the **result** is not a warning, it means that no instability occurred in the functions called by the predicate.

Once each slice of the specification was input to PRECiSA to obtain the corresponding annotated C code, Frama-C was used to verify that this implementation actually fulfills the contracts stated by the annotations. As explained in the paragraphs above, the validity of these contracts is mainly supported by the error-bound certificates generated by PRECiSA, which are output in PVS language and depend on the definitions and properties declared in the axiomatic floating-point formalization from NASALib. For this reason, a particular customization was applied to Frama-C in order to generate the verification conditions in the language of PVS and use the aforementioned floating-point formalization.

B. The top-level function

The process described above generates code for each slice of the specification and verifies its compliance to the corresponding predicate from Table I. Nevertheless, in order to generate code with the same applicability as the original target, i.e., the predicate $wcv?$ from (11), an additional layer of C code is needed. This layer is responsible for selecting the slice activated by the inputs and invoking the corresponding function.

Listing 4 shows an excerpt from the generic top-level function. The postcondition states that if the computation does not raise a warning and the ϵ parameters actually denote bounds for the errors in the conditionals defining the control flow graph of the whole program, then the result is equivalent to the original Well-Clear predicate $wcv?$ defined in (11). The proof of the verification condition generated from this contract relies on the contracts of the invoked functions, e.g., `WCVint_inc_mtn_plus` and `WCVint_inc_mtn_minus` in the excerpt, and the Slicing Correctness Theorem 1. As in the lower layers, accompanying concrete C functions were defined, where the error-bound parameters ϵ are instantiated

```

1 /*@
2 predicate wcv_in_range (real b, t, v_x, v_y, v_z, s_x, s_y, s_z) =
3   // WCV? ((b, t), (v_x, v_y, v_z), (s_x, s_y, s_z)) from Eq. (11)
4 ...
5 requires:  $\backslash is\_finite(\tilde{\epsilon}_0) \wedge \tilde{\epsilon}_0 \geq 0 \wedge \dots \wedge \backslash is\_finite(\tilde{\epsilon}_3) \wedge \tilde{\epsilon}_3 \geq 0$ ;
6 ensures:  $\forall$  real b, t, v_x, v_y, v_z, s_x, s_y, s_z;
7    $|(\tilde{\delta}_z - \tilde{v}_z * \tilde{\delta}_{tcoa}) - (\delta_z - v_z * \delta_{tcoa})| \leq \tilde{\epsilon}_0 \wedge$ 
8    $|(\tilde{t} - coalt\_t\_inc\_vz\_fp(\tilde{s}_z, \tilde{v}_z)) - (t - coalt\_t\_inc\_vz(s_z, v_z))| \leq \tilde{\epsilon}_1 \wedge$ 
9    $|coalt\_b\_inc\_vz\_fp(\tilde{s}_z, \tilde{v}_z) - b - (coalt\_b\_inc\_vz(s_z, v_z) - b)| \leq \tilde{\epsilon}_2 \wedge$ 
10  ...
11  \result  $\neq \omega$ 
12   $\Rightarrow (\backslash result \Leftrightarrow wcv\_in\_range(b, t, v_x, v_y, v_z, s_x, s_y, s_z))$ ;
13 */
14 bool' WCV_interval (double  $\tilde{b}, \tilde{t}, \tilde{s}_x, \tilde{s}_y, \tilde{s}_z, \tilde{v}_x, \tilde{v}_y, \tilde{v}_z, \tilde{\epsilon}_0, \tilde{\epsilon}_1, \tilde{\epsilon}_2, \tilde{\epsilon}_3, \dots$ ) {
15   bool' res;
16   if ( $\tilde{v}_z > 0.0$ ) // increasing vertical separation
17     if ( $\tilde{v}_x == 0.0 \ \&\& \ \tilde{v}_y == 0.0$ ) { // maintaining horizontal separation
18       res = WCVint_inc_mtn_plus( $\tilde{b}, \tilde{t}, \tilde{s}_x, \tilde{s}_y, \tilde{s}_z, \tilde{v}_x, \tilde{v}_y, \tilde{v}_z, \tilde{\epsilon}_0, \tilde{\epsilon}_1, \tilde{\epsilon}_2, \tilde{\epsilon}_3$ );
19       if (res ==  $\omega$  || res) return res;
20       res = WCVint_inc_mtn_minus( $\tilde{b}, \tilde{t}, \tilde{s}_x, \tilde{s}_y, \tilde{s}_z, \tilde{v}_x, \tilde{v}_y, \tilde{v}_z, \tilde{\epsilon}_0, \tilde{\epsilon}_1, \tilde{\epsilon}_2, \tilde{\epsilon}_3$ );
21       if (res ==  $\omega$ ) return  $\omega$ ;
22       if (res) return false;
23       return  $\omega$ ;
24     } else {
25       ...
26     }
27   else {
28     ...
29   }
30 }

```

Listing 4. Excerpt from the generic top-level function.

with concrete values computed by PRECiSA, given user-provided ranges for the rest of the inputs.

One may wonder if the top-level function implementation is subject to conditional instability. However, it can be noticed that the guards used to select the program slice are sign checks on input values that come from an external sensor or data. In these cases, the rounding error corresponds to the representation error on this value which does not affect its sign.

The top-level functions and the accompanying annotations were developed by hand for this case study. Nevertheless, once the criteria to be used to define the slicing is selected, the development of these functions and their annotations is almost mechanic, at least for applications like this one, where quite simple slicing conditions are used. The automation of this stage of the technique is one of the possible extensions to this work.

VI. DISCUSSION

The goal of the work presented in this paper is the extraction and verification of a floating-point C implementation from a proven correct real-valued specification of an algorithmic solution for a safety- and mission-critical problem. When trying to apply the toolchain presented in [7] several practical issues were addressed and new improvements were proposed. This section provides a brief summary of the most significant of them.

The first impediment that prevented the existent toolchain to be applied as in the past was the presence of higher-order elements in the input specification. This issue was addressed by restating several of the declarations into a more concrete form, avoiding the use of higher-order parameters. The level

of effort involved in the application of this simplification could be seen as non-trivial since there were changes in many of the lines of the original specification. Nevertheless, in the majority of the cases, each change was simple and it could be applied in a mechanical way. For this concrete case study, the transformation to first order is a process that could have been performed automatically. In fact, part of the future lines of research is devoted to the development of an automatic procedure to simplify higher-order features from a PVS specification.

After simplifying the original declarations, the resulting first-order specification was fed to PRECiSA. Nevertheless, the process had to be aborted after reaching a time-out of three hours without a response. This impact on the performance could possibly be explained by the number of different flows starting at the top-level function *wcVint*, presented in (10), which provokes the generation of huge error expressions. The manipulation of such expressions deemed the code-generation process to be impractical. The step that allowed pushing PRECiSA beyond its scalability limit was the application of a slicing-based simplification on the first-order specification. The automatic generation of code performed by PRECiSA took less than 15 minutes to finish for the whole collection of slices on the same machine. This improvement is related to the fact that in the DAIDALUS specification some checks on the velocities and positions are repeated along the same branch in the control flow tree. In addition, this phenomenon is repeated in several different branches. The slicing of the specification lifted to the top-level several of these checks, reducing the complexity of each individual slice. While the selection of the slicing criteria would depend on human insight in the general case, once it is decided, the automation of most of the tasks related to the process and integration of the slices into the final analysis is expected to be feasible, at least in examples with a complexity similar to the one presented in this paper.

Another distinguishing feature of this work is the use of a new formalization for floating-point numbers⁵. This formalization is different from the one used in previous works in several aspects. Mainly, it is defined in an axiomatic way, which has a significant impact on the type-checking time of PVS, improving it by a factor of six. Since the verification conditions output by Frama-C are expressed in terms of floating-point and real-valued operations, the PVS libraries where these arithmetic domains are defined need to be type checked. The reduction in the time spent in type-checking improved not only the flow of the work while the proofs for the verification conditions were developed, but also decreased the time needed to rerun such proofs once they were done.

Additionally, this new formalization follows the IEEE-754 standard more closely, including representations for special values such as Not-a-Numbers (NaN) and infinities. While the use of a more detailed model usually complicates its interaction with the rest of the specification, in this work it was possible to reduce such impact to a minimum. For

instance, the only place where a restriction about finiteness of the floating-point representations is explicitly used is for predefined constants and error-bound parameters, as can be seen in the *requires* of all the listings above.

On the bright side, the gain of using this more detailed formalization is at the semantic stance. It is not uncommon to simplify some aspects of the models when a formalization is designed. For the case of floating-point numbers, usually, some aspects of the IEEE-754 standard, such as the special values, are left aside because they complicate the formalization by introducing the need to handle a nontrivial number of special cases. Nevertheless, since the special values are supported by the C language, working with a conceptual model that does not support them could introduce space for flaws undetectable by the analysis.

It is important to note that the almost seamless integration with this new formalization was possible because the check for finiteness was encapsulated in the error-bound certificates generated by PRECiSA. As part of the automatic proof for certificates as the one expressed by Theorem 2, the numeric expressions (including subexpressions) appearing in them are checked to remain in the floating-point representable domain, therefore no infinite values or overflows occur. This check is done by using a branch-and-bound optimization algorithm implemented in the logic of PVS itself [21]. Notably, this process provides hints on *overflow detection* since if the solver cannot decide whether the numeric expressions remain in the representable range for the inputs provided by the user, the proof of the certificate cannot be completed. In other words, if PVS cannot automatically prove the error certificate using the PRECiSA proof strategies, the user is directed to look for a possible overflow condition in their program.

VII. RELATED WORK

Different tools have been proposed to reason about the numerical aspects of C programs. In this work, a combination of PRECiSA, PVS, and Frama-C [10] is used. Support for floating-point round-off error analysis in Frama-C is also provided by the integration with the tool Gappa [22]. However, the applicability of Gappa is limited to straight-line programs without conditionals, and it often requires providing additional ACSL intermediate assertions and hints through annotation that may be unfeasible to generate automatically. The interactive theorem prover Coq can also be applied to prove verification conditions on floating-point numbers thanks to the formalization defined in [23]. Nevertheless, Coq [24] tactics are not available to automatize the verification process.

Several approaches have been proposed for the verification of numerical C code by using Frama-C in combination with Gappa and/or Coq [25]–[30].

In [31], a preliminary version of the technique presented in this paper is used to verify a specific case study of a point-in-polygon containment algorithm. In [7], the verification approach is presented and applied to a small fragment of DAIDALUS. Note, in both [31] and [7] overflow detection is not performed.

⁵Available at https://github.com/nasa/pvslib/tree/master/float/axm_bnd.

Besides Frama-C, other formal methods tools are available to analyze the numerical properties of C code. Fluctuat [32] is a static analyzer that, given a C program with annotations about input bounds and uncertainties on its arguments, produces an estimation of the round-off error of the program. Fluctuat detects the presence of possible unstable guards in the analyzed program, as explained in [33], but does not instrument the program to emit a warning in these cases. The static analyzer Astrée [34] detects the presence of run-time exceptions such as division by zero and under and over-flows employing sound floating-point abstract domains. In contrast to the approach presented here, neither Fluctuat nor Astrée emits proof certificates that an external prover can externally check.

VIII. CONCLUSION AND FUTURE WORK

In this paper, a formal approach is applied to generate and verify a floating-point implementation of the DAIDALUS well-clear specification. This implementation is obtained by manually simplifying and slicing the original specification and then utilizing each slice as input to the PRECiSA code generator. PRECiSA automatically generates a floating-point version of each slice in C syntax enriched with ACSL contracts stating the relationship between the ideal real number specification and the floating-point implementation. In addition, PRECiSA instruments the code to detect control flow divergences due to rounding errors.

The generated C implementation of each slice is analyzed within the Frama-C analyzer. In particular, the WP plugin is used to compute a set of verification conditions that are proved within the PVS theorem prover. These verification conditions ensure that the accumulated rounding error is bounded, all flow divergences are detected, and no overflow occurs.

The verification of the DAIDALUS well-clear C implementation relies on three different tools: the PVS interactive prover, the Frama-C analyzer, and PRECiSA. All of these tools are based on rigorous mathematical foundations and have been used in the verification of industrial and safety-critical systems. The C floating-point transformed program, the PVS verification conditions, and the round-off error bounds are automatically generated. However, a certain level of expertise is needed for proving the PVS verification conditions generated by Frama-C and for proving the equivalence between the original DAIDALUS specification and the simplified and sliced one.

In the future, the authors plan to improve the automation degree of the slicing and top-layer function generation. Static analysis techniques may be used since the slices are built according to the program branches. The authors also plan to simplify the structure of the ACSL contracts generated by PRECiSA to facilitate human inspection and to produce simpler verification conditions. Automatic strategies are already available in PRECiSA to discharge the PVS certificates ensuring the correctness of the rounding error bounds and to prove certain verification conditions generated by the WP analysis. However, additional work needs to be done to fully

automatize this process because of the new extended floating-point formalization used in this paper.




Another line of future research is motivated by the evaluation of the impact of the process of generation of a safer program on its final performance with respect to existing implementations. For instance, the reference implementation for DAIDALUS is expected to outperform the code generated by PRECiSA since some overhead is introduced by checking the stability of every guard. Nevertheless, in aerospace software such as the DAIDALUS library in which no iterative statements are allowed, this kind of overhead could result to be negligible or at least acceptable in real-world deployments, weighting the higher level of safety provided by the code generated by PRECiSA.


REFERENCES

- [1] Advisory Circular, U.S. Dept. of Transportation, Federal Aviation Administration, *AC 90-48D - Pilots' Role in Collision Avoidance*. U.S. Government, 2016.
- [2] U.S. Government, *Aeronautics and Space*. 14 CFR § 91.113, 2004.
- [3] C. Muñoz, A. Narkawicz, G. Hagen, J. Upchurch, A. Dutle, and M. Consiglio, "DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems," in *Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015)*, Prague, Czech Republic, September 2015.
- [4] RTCA DO-365A, *Minimum Operational Performance Standards (MOPS) for Detect and Avoid (DAA) Systems, Appendix H*. RTCA, February 2020.
- [5] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction*, ser. LNCS, vol. 607. Springer, 1992, pp. 748–752. [Online]. Available: https://doi.org/10.1007/3-540-55602-8_217
- [6] A. Narkawicz, C. Muñoz, and A. Dutle, "The MINERVA software development process," in *Automated Formal Methods*, ser. Kalpa Publications in Computing, vol. 5. EasyChair, 2018, pp. 93–108. [Online]. Available: <https://easychair.org/publications/paper/g1Rs>
- [7] L. Titolo, M. Moscato, M. Feliú, and C. Muñoz, "Automatic generation of guard-stable floating-point code," in *Proceedings of the 16th International Conference on Integrated Formal Methods (IFM 2020)*, ser. LNCS, vol. 12546. Springer, 2020, pp. 141–159.
- [8] M. Moscato, L. Titolo, A. Dutle, and C. Muñoz, "Automatic estimation of verified floating-point round-off errors via static analysis," in *Proceedings of the 36th International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2017*. Springer, 2017.
- [9] L. Titolo, M. Feliú, M. Moscato, and C. Muñoz, "An abstract interpretation framework for the round-off error analysis of floating-point programs," in *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 2018, pp. 516–537.
- [10] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Form. Asp. of Comput.*, vol. 27, no. 3, pp. 573–609, 2015.
- [11] C. Muñoz and A. Narkawicz, "Formal analysis of extended well-clear boundaries for unmanned aircraft," in *Proceedings of the 8th NASA FM Symp. (NFM 2016)*, ser. LNCS, vol. 9690. Minneapolis, MN: Springer, June 2016, pp. 221–226.
- [12] C. Muñoz, A. Narkawicz, J. Chamberlain, M. Consiglio, and J. Upchurch, "A family of well-clear boundary models for the integration of UAS in the NAS," in *Proceedings of the 14th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, Georgia, Atlanta, USA, June 2014.
- [13] A. P. Smith, C. Muñoz, A. J. Narkawicz, and M. Markevicius, "A rigorous generic branch and bound solver for nonlinear problems," in *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2015*, 2015, pp. 71–78.
- [14] S. Owre, J. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Proceedings of the 11th International Conference on Automated Deduction (CADE)*. Springer, 1992, pp. 748–752.

- [15] M. D. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*. IEEE Computer Society, 1981, pp. 439–449.
- [16] —, "Program slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.
- [17] G. Canfora, A. Cimitile, A. D. Lucia, and G. A. D. Lucca, "Software salvaging based on conditions," in *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*, H. A. Müller and M. Georges, Eds. IEEE Computer Society, 1994, pp. 424–433.
- [18] J. Q. Ning, A. Engberts, and W. Kozaczynski, "Automated support for legacy code understanding," *Commun. ACM*, vol. 37, no. 5, pp. 50–57, 1994.
- [19] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 12:1–12:41, 2012.
- [20] L. Titolo, C. Muñoz, M. Feliú, and M. Moscato, "Eliminating unstable tests in floating-point programs," in *Proceedings of the 28th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2018)*. Springer, 2018, pp. 169–183.
- [21] A. Narkawicz and C. Muñoz, "A formally verified generic branching algorithm for global optimization," in *Proceedings of the 5th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2013)*, ser. Lecture Notes in Computer Science, E. Cohen and A. Rybalchenko, Eds., vol. 8164. Menlo Park, CA, US: Springer, May 2014, pp. 326–343.
- [22] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Trans. on Computers*, vol. 60, no. 2, pp. 242–253, 2011.
- [23] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq," in *20th IEEE Symposium on Computer Arithmetic, ARITH 2011*. IEEE Computer Society, 2011, pp. 243–252.
- [24] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
- [25] S. Boldo and J. C. Filliâtre, "Formal verification of floating-point programs," in *Proceedings of ARITH18 2007*. IEEE Computer Society, 2007, pp. 187–194.
- [26] S. Boldo and C. Marché, "Formal verification of numerical programs: From C annotated programs to mechanical proofs," *Mathematics in Computer Science*, vol. 5, no. 4, pp. 377–393, 2011.
- [27] S. Boldo, F. Clément, J. C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis, "Wave equation numerical resolution: A comprehensive mechanized proof of a C program," *Journal of Automated Reasoning*, vol. 50, no. 4, pp. 423–456, 2013.
- [28] A. Goodloe, C. Muñoz, F. Kirchner, and L. Correnson, "Verification of numerical programs: From real numbers to floating point numbers," in *Proceedings of the NASA FM Symp. NFM 2013*, ser. LNCS, vol. 7871. Springer, 2013, pp. 441–446.
- [29] C. Marché, "Verification of the functional behavior of a floating-point program: An industrial case study," *Science of Computer Programming*, vol. 96, pp. 279–296, 2014.
- [30] L. Titolo, M. Moscato, C. Muñoz, A. Dutle, and F. Bobot, "A formally verified floating-point implementation of the compact position reporting algorithm," in *Proceedings of the 22nd International Symposium on Formal Methods (FM 2018)*, ser. LNCS, vol. 10951. Springer, 2018, pp. 364–381.
- [31] M. Moscato, L. Titolo, M. Feliú, and C. Muñoz, "Provably correct floating-point implementation of a point-in-polygon algorithm," in *Proceedings of the 23rd International Symposium on Formal Methods (FM 2019)*, ser. LNCS, vol. 11800. Springer, 2019, pp. 21–37.
- [32] E. Goubault and S. Putot, "Static analysis of numerical algorithms," in *Proceedings of SAS 2006*, ser. LNCS, vol. 4134. Springer, 2006, pp. 18–34.
- [33] —, "Robustness analysis of finite precision implementations," in *Proceedings of APLAS 2013*, ser. LNCS, vol. 8301. Springer, 2013, pp. 50–57.
- [34] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and Rival, "The ASTREE Analyzer," in *Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, ser. LNCS, vol. 3444. Springer, 2005, pp. 21–30.

Formal Verification of Correctness and Information Flow Security for an In-Order Pipelined Processor

Ning Dong^{}, Roberto Guanciale^{}, Mads Dam^{}
KTH Royal Institute of Technology, Stockholm, Sweden
{dongn, robertog, mfd}@kth.se

Andreas Lööw^{}
Imperial College London, London, UK
a.loow@imperial.ac.uk

Abstract—We present an in-order pipelined processor and its verification in the HOL4 interactive theorem prover. The processor implements the RISC ISA Silver and features a general 5-stage pipeline. The correctness of the processor is proved by exhibiting a refinement relation between the traces of the pipelined circuit and the Silver ISA. The processor is constructed by using a HOL4 Verilog library for formally verified hardware, and its correctness is guaranteed down to the Verilog implementation. Additionally, we analyze the information flow properties of the processor by utilizing the refinement relation. The notion of conditional noninterference formulates that a processor should not leak more information via its timing channel than what is expected by a leakage model expressed at the ISA level. We establish the conditional noninterference for our processor and demonstrate the adaptability of the information flow methodology to accommodate various processor designs, attacker models, and environments. Our approach to verify processor implementations and enable information flow analysis at the circuit level is suitable for ISAs beyond Silver.

Index Terms—Formal Verification, Information Flow, Pipelined Processor, Interactive Theorem Prover

I. INTRODUCTION

Almost all modern processors use instruction pipelining to improve performance. Since processors are the fundamental hardware component of a computing system, it is important to formally ensure the correctness and security of pipelined processors, as testing may miss important corner cases.

Formal analysis of pipelined processors has been studied for decades [1]–[7]. Of these, early work [1]–[4] used simplified models. Later, more comprehensive verifications were performed, such as the VAMP processor [6] which lacked the guarantee to the hardware description language (HDL) level, and the MIPS pipeline [5] with non-mechanized proof. Importantly, Kami [7] used intermediate models (e.g. a 3-stage pipeline) tied with their specific ISA model to verify a 4-stage pipelined circuit by refinement.

Over the past couple of years, prompted by the emergence of the Spectre [8] family of vulnerabilities, the modelling and verification of processor’s information flow at microarchitecture and hardware levels have received significant attention [9]–[14]. However, these works have not addressed the combination of information flow security and functional correctness systematically.

This work has been supported by the TrustFull project funded by the Swedish Foundation for Strategic Research. Ning Dong is supported by the KTH-CSC joint scholarship programme for his doctoral studies.

In this paper, we present the formal verification of an in-order 5-stage pipelined processor in the HOL4 theorem prover [15]. In-order pipelines are widely used in applications from IoT to autonomous systems, and form the basis for more advanced pipeline designs. We choose the general-purpose RISC ISA Silver [16] as the target due to its simplicity and generality. The former facilitates our circuit implementation and verification, and the latter makes our work more reusable for other ISAs. The processor is implemented with the help of the hardware development library introduced by Lööw et al. [16], [17]. The library is embedded inside HOL4 and targets the Verilog HDL.

In summary, we have the following contributions:

- We verify the correctness of a pipelined processor by establishing a refinement relation between the traces of the processor and the Silver ISA. The correctness proof is mechanized in HOL4 and is guaranteed down to the Verilog implementation of the processor.
- Notably our verification approach can deal with pipeline challenges that have been only partially addressed before, see details in Section III-A.
- We provide non-mechanized analysis of information flows of the processor via observational models. The analysis shows the absence of unexpected side channels via a verification strategy that complements the verification of functional correctness.
- The processor has been successfully synthesized as a small computer system (in combination with other components, e.g., a cache and interrupt handler) for the PYNQ-Z1 FPGA board, by using the Xilinx Vivado toolchain. We evaluate its performance with several software programs compiled for the system.

The formalization in HOL4 is about 21,000 lines of code including circuit definition and correctness proof and took around 14 person months to develop, which is available at <https://doi.org/10.5281/zenodo.8199575>.

II. BACKGROUND

A. Instruction Pipelining

Instruction pipelining divides the processing of instructions into different stages, e.g., fetch, decode, and execute, and thus allows the processor to handle multiple instructions concurrently. Pipelined processors can operate at a higher clock frequency because of their simpler stage circuits compared

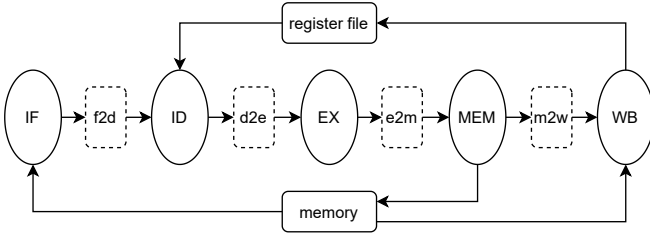


Fig. 1. Simplified view of a 5-stage pipeline

to sequential processors, which handle one instruction at a time. Figure 1 presents a typical 5-stage pipeline. An interface between every two adjacent stages transfers necessary elements to the next stage as input. Instruction fetch (**IF**) uses a branch predictor to produce the program counter (PC) for the next cycle, which for most RISC processors simply increases the current PC by 4. Instruction decode (**ID**) decodes information from instructions and reads the register file to prepare source operands data for the next stages. Execution (**EX**) computes the results of operations that do not depend on external components like memory. The results are not directly written to the register file, which is only updated by the Write back (**WB**) stage. Memory (**MEM**) stage interacts with the external environment, for instance, implements memory loads and stores. When the memory results are ready, they are delivered directly to **WB** to update the register file.

B. HOL4 Verilog Library

The processor is constructed using the HOL4 Verilog library [16], [17], which contains a formal semantics for the HDL Verilog and a proof-producing translator from (shallowly embedded) HOL functions to (deeply embedded) Verilog descriptions. The translator allows users to define circuits as HOL next-state functions and automatically generate corresponding Verilog descriptions (expressed in the Verilog semantics of the library), where each translation is guaranteed to be correct by a HOL proof automatically constructed during the translation process. Since modeling circuits as HOL next-state functions describes the cycle-by-cycle behaviour of circuits, the HOL4 Verilog library ensures that no timing information is lost or introduced when the circuit models are exported to Verilog. Initially, the semantics and the translator were limited to sequential logic, i.e., `always_ff` blocks triggered by the positive edge of the circuit's clock signal. Later, Löw [18] extended the semantics and the translator to support combinational logic, i.e., `always_comb` blocks.

C. Silver ISA

Generally, the semantics of an ISA is modelled by a state transition relation: $s_0 \rightarrow s_1$, which represents the atomic execution of one instruction. Then, the ISA trace σ is produced by \rightarrow from the initial state s_0 , i.e., $\sigma = s_0 \rightarrow s_1 \rightarrow s_2 \dots$, and $\sigma(n) = s_n$.

The Silver ISA [16] is originally defined in L3 [19] which is a domain-specific language for ISA specifications and translated into HOL4 as our circuit specification. The Silver ISA state

is a record, $s = (PC, M, R, CF, OF, DI, DO, ME)$. Here, PC , M , and R are the program counter, memory, and register file respectively. Two flags (CF and OF) are used to record carry and overflow for the ALU (arithmetic logic unit) add and subtraction computations, DI and DO are two data ports for I/O operations, and ME is a trace to record memory states. Silver instructions have fields *opc* determining the operation and *func* determining the functionality for ALU and SHF operations. The difference in the usage of *func* is that SHF only uses the lower 2 bits of *func* but the ALU uses all 4 bits. There are three data resource fields (Ra , Rb , and Rw), which are followed by their corresponding flags (Fa , Fb , and Fw) to indicate it as a register address (flag is 0) or an immediate constant (flag is 1). So, Da , Db , and Dw are the data read from R with their addresses if the flag is 0, otherwise, the constant. Normally, Da and Db are source data, and Rw is the destination register to write the result (Fw usually is 0). Some operations do not write any register, Rw in these instructions could be a source register or immediate constant. For example, Rw is an input address for conditional jump CJMP.

The following operations have non-trivial effects on the pipeline: (1) JMP and CJMP: unconditional and conditional jumps. The JMP instruction computes the jump target using ALU with the current PC and Da as input, and stores the value $PC + 4$ to the register Rw . The CJMP instruction checks if the ALU functionality as determined by *func* applied to Da and Db is 0 or not. The destination is $PC + Dw$ if the condition is true, otherwise, $PC + 4$. (2) MLD and MSTR: bitwise memory load and store. The MLD instruction loads a single byte or a word at the address Da in the memory M to the register Rw . The MSTR instruction updates M with Db as the address ad and Da as the value v . (3) INTR: interrupt. The purpose of INTR is to allow the processor to communicate with the external environment, and the hardware implementation is expected to wait for a reply from the environment for INTR. The Silver ISA does not interact with the environment. The INTR instruction appends the current memory M to the trace ME , which is never used by the ISA for any execution. (4) ACC: acceleration. The ACC instruction computes an addition of the lower and higher 16 bits of input Da . The purpose of ACC is to support a separate accelerator in the processor implementation. The ACC instruction is a placeholder for richer functionalities. (5) DIN: data input operation. The DIN instruction writes the data from the DI port to the register Rw . Because the Silver ISA is defined as a self-contained machine i.e. without any interaction with the external environment, the value of DI remains unaffected after initialization.

D. Security Condition

ISAs serve as the main interface between software and hardware, ensuring the correctness and security of software. However, ISAs do not capture non-functional aspects of systems, like the execution time, that can be utilized by an attacker to infer confidential data. To scope our work, we consider the attacker as an external agent that can monitor the timing channel when outputs are produced by our system. For

Silver, this corresponds to measuring the clock cycles elapsed between INTR. It is usually infeasible to verify resilience against side channels by taking into account both software and processor design at the same time [9], [10]. In practice, these analyses are usually done by using observational models, which extend the ISA with leakage functions that overapproximate what influences the side channels [12]. These models have several benefits: (1) they allow for overapproximation of leakage, permitting scalable verification by using conservative measures; (2) they provide a common interface that can be used (with slight variations) to analyze different types of channels or different hardware designs. The goal is usually to demonstrate a variety of noninterference [20] (see Section V): the results of the leakage functions are independent of confidential data. Recently, variations of noninterference for processor executions have been proposed [10], [12], [21] to capture the idea of hardware execution traces being indistinguishable in terms of side-channel observations (e.g., timing and cache accesses) if leakages of the corresponding ISA-level traces are identical.

III. PIPELINE IMPLEMENTATION

A. Design

Our pipelined processor, called *Silver-Pi*, follows the 5-stage design in Figure 1. The design addresses common pipeline challenges such as data hazards, and is general enough to accommodate other RISC ISAs (see Section VI-A).

a) Data hazards: The pipeline may process interdependent instructions, for example, a program of two instructions `i0: R1 := R0+1; i1: R2 := R1-2;`. The pipeline must prevent `i1` from reading a wrong (old) value for the register `R1` in the **ID** stage, when `i0` is still being processed in the pipeline and its result has not been committed to the register file. Silver-Pi uses pipeline stalling by checking whether each register's address (*Ra*, *Rb*, *Rw*) is affected by instructions in the **EX**, **MEM**, and **WB** stages. If data hazards are identified, a control unit stalls the instruction in **ID** stage until data hazards are resolved.

b) External Delays: Requests issued by the **MEM** stage can take several processor cycles to be answered. Normally any new request to the same external hardware component will be ignored during the waiting cycles. For instance, when the **MEM** stage issues a MLD request to the memory, the pipeline is stalled until the memory replies the result of MLD to the **WB** stage, which is then committed to the register file *R*. The same approach is applied to the other three kinds of Silver instructions that communicate with external components and that can take several hardware cycles to be answered: MSTR, INTR, and ACC.

c) Mispredicted program counters: The pipeline fetches instructions speculatively for the next cycle until the next *PC* is determined. These speculatively fetched instructions can be wrong when an instruction in the pipeline modifies the program counter (i.e., JMP and CJMP). Consider the example in Table I, instructions `i0 - i3` are regular operations, and `i4` is a JMP. The instructions `i4'` and `i4''` are stored in the next two addresses to `i4` in the memory, and `i5` is stored at

	IF	ID	EX	MEM	WB
<i>t</i>	<code>i4 (JMP)</code>	<code>i3</code>	<code>i2</code>	<code>i1</code>	<code>i0</code>
<i>t + 1</i>	<code>i4'</code>	<code>i4 (JMP)</code>	<code>i3</code>	<code>i2</code>	<code>i1</code>
<i>t + 2</i>	<code>i4''</code>	<code>i4'</code>	<code>i4 (JMP)</code>	<code>i3</code>	<code>i2</code>
<i>t + 3</i>	<code>i5</code>	NOP	NOP	<code>i4 (JMP)</code>	<code>i3</code>

TABLE I
PROCESSING A JUMP IN THE PIPELINE

the target address of `i4`. The `i4'` and `i4''` instructions are speculatively fetched at the cycle *t + 1* and *t + 2*. In Silver ISA, target addresses of jumps are determined only after the ALU results are available in the **EX** stage. In addition, the *PC* cannot be affected by external hardware components like memory. For these reasons, we implement a jump handler in the **EX** circuit. For the example in Table I, after the **EX** circuit computes the target of `i4`, the instructions `i4'` and `i4''` are flushed as NOP (no operation) and the proper next instruction `i5` is fetched at *t + 3*. The reason for not handling jumps in the **MEM** or **WB** stage is discussed in Section VI-A.

B. Formal Implementation in HOL4

The circuit state is represented by a record *c* = (*g*, *c_{if}*, *c_{id}*, *c_{ex}*, *c_{mem}*, *c_{wb}*). The component *g* contains general fields like the register file *R_g*, and output fields for interacting with the environment, including a program counter *PC_g*, data address *ad_g*, stored data value *v_g*, a command *cmd_g* issuing fetch/load/store requests to memory, etc. Other components are the states of their relevant pipeline stage, consisting of fields operated by the pipeline stage internally. For instance, *c_{id}* has the fields *opc_{id}*, *func_{id}*, *Rw_{id}*, etc. for decoding ISA level *opc*, *func*, *Rw*, etc.

We formalize an environment state *e* to describe necessary external hardware components such as a memory subsystem. Formally, *e* = (*M*, *DI*, *inst*, *data*, *rdy*, *mirdy*, *iack*). The components *M* and *DI* are identical to the memory and data ports of the ISA state. Other items excluding *M* and *DI* are the environment's outputs to the processor. The signals *inst* and *data* are instruction and data values from memory, and *rdy* indicates the memory request is finished and the memory is able to process the next request. The signal *mirdy* indicates that memory initialization is finished, and *iack* is an acknowledgement of the interrupt handler to the processor to inform that the INTR request is finished.

As required by the HOL4 Verilog library, the circuit is implemented as next-state functions that update the processor's state *c* and communicate with the environment's state *e*. These functions are divided into two lists *fl* and *cl* for the *always_ff* and *always_comb* blocks respectively. The functions in the *fl* list are mainly to transfer data from one stage to the next stage as input. Other functions in the *fl* list include an accelerator to perform the Silver ISA ACC operation, updating the program counter *PC_g* to fetch instructions, writing the register file *R_g*, and a general function managing interactions with the environment.

Functions in the *cl* list are the components of every pipeline stage's internal circuit and a control unit function. The major function in the **IF** stage is a multiplexer to select the next

cycle's PC ($next_PC_{if}$), and another function assigns to the fetched instruction $instr_{if}$ from the memory's reply. Functions in the **ID** stage perform the decoding for the instruction delivered from the **IF** stage and check data hazards. The function checking data hazards does not consider opc_{id} , since opc_{id} does not affect the correctness of such checks. Two main functions in the **EX** stage are responsible for computing the ALU and SHF operations separately. The jump handler is defined with the output jmp_{ex} based on the ALU result and opc_{ex} , as described in Section III-A. Functions in the **MEM** stage identify operations that need to communicate with the external environment or the accelerator. A multiplexer in the **WB** stage determines the result to write the destination register Rw_{wb} at the next cycle, based on the operation code opc_{wb} . The control unit in the cl list is defined to maintain the process between different pipeline stages when pipeline challenges in Section III-A happen.

Finally, the Silver-Pi circuit is defined as follows: $ag\pi = mk_module(procs\ fl)(procs\ cl)\ init_{ag\pi}$. The term $init_{ag\pi}$ executes once in the initial cycle to set up initial values for essential items of the circuit state c . For instance, PC_g is assigned to 0. The functions $procs$ and mk_module are provided by the HOL4 Verilog library to construct the circuit. The function $procs$ sequentially composes a list of next-state functions into one next-state function. The function mk_module constructs a representation of a Verilog module out of a next-state function for sequential logic and a next-state function for combinational logic.

The external environment is consistent with the one used by the non-pipelined Silver circuit [16], see Section IV-A. Given an environment trace $\beta = e \rightarrow e' \rightarrow e'' \dots$, the circuit $ag\pi$ produces the processor's execution trace $\alpha = c \rightarrow c' \rightarrow c'' \dots$ where $\alpha(t)$ is the processor's state at the cycle t . In the following, we use ϕ to range over circuit execution traces, i.e., traces such that exist $\alpha = ag\pi\ \beta$ and $\phi(t) = (\alpha(t), \beta(t))$.

IV. CORRECTNESS PROOF

The correctness of the pipelined circuit is established by exhibiting a trace relation between the circuit and the Silver ISA. Our proof methodology has its roots in the pipeline proof for MIPS ISA [5], [22], but significant changes have been made because the Silver ISA is closer to modern RISC ISAs than MIPS (see Section VI-A) and we use a more realistic environment. Based on the translator in the Verilog library, a Verilog AST is generated for the circuit definition $ag\pi$. The simulation between them is automatically proved by the translator. Correctness between the ISA and $ag\pi$ is lifted to the Verilog AST, as shown in Figure 2.

A. External Environment

The external environment is axiomatized where the environment trace β in the circuit trace ϕ satisfies an assumption AX which consists of four parts: (1) mem_env ; (2) mem_start_env ; (3) $intr_env$; (4) di_env .

The first assumption constrains the memory subsystem to support instantaneous memory accesses, which can reply to

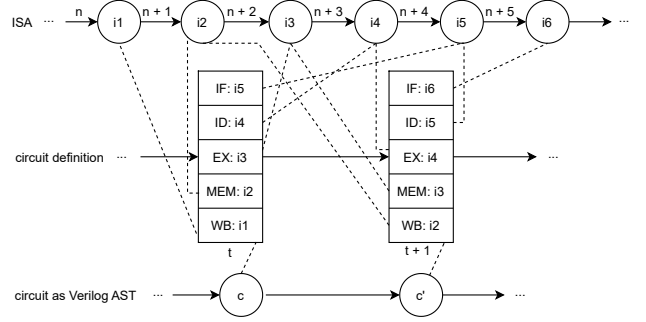


Fig. 2. Correspondence between the ISA and circuits at different levels

memory requests within one cycle. This assumption describes the reactions of the memory subsystem to four memory operations that can be requested by the pipeline: (1) do nothing; (2) fetch an instruction; (3) fetch an instruction and load data; (4) fetch an instruction and store data. For example, as Definition 1 shows, if the memory gets a fetch command at the cycle t and is ready at the previous cycle, then it replies immediately ($m = 0$) or after a finite number of cycles with the right instruction without affecting the memory.

Definition 1.

$$\begin{aligned} \alpha(t).cmd_g = fetch \wedge \beta(t-1).rdy \Rightarrow \\ \exists m. (\forall p \leq m. \beta(t+p).M = \beta(t-1).M) \wedge \\ (\forall p < m. \neg \beta(t+p).rdy) \wedge \beta(t+m).rdy \wedge \\ \beta(t+m).inst = \beta(t).M[\alpha(t).PC_g] \end{aligned}$$

The second assumption states that the memory initialization is eventually completed: $\exists m. \beta(m).mirdy$.

The third assumption constrains the interrupt handler which behaves similarly to the memory subsystem. The handler replies $iack$ to the processor's interrupt request within a finite number of clock cycles. As introduced in Section II-C, the Silver ISA appends the whole memory into ME when $INTR$ happens but never uses ME . Practical interrupt handlers usually do not perform a copying-all-memory operation and the memory states are regulated by mem_env . So, the ME descriptions in $intr_env$ have been removed.

The last assumption constrains the DI port in the environment used by the DIN instruction. The DI port is supposed to be updated by an external controller, rather than the processor itself in the previous work [16]. Since the value of DI in the Silver ISA (see Section II-C) remains unchanged during execution, $\forall t. \beta(t).DI = \beta(0).DI$.

B. Scheduling Function

The pipelined circuit processes instructions concurrently and takes several hardware cycles to complete an instruction. To reason about the correctness of pipeline stages, we use a scheduling function, as inspired by the MIPS work [5], [22]. The scheduling function I maps the processing instruction in a pipeline stage k at cycle t , $n = I(k, t)$. It is inductively defined, for example, if the **EX** stage is enabled for the cycle

t via a field $enable_{ex}$, $I(\mathbf{EX}, t) = I(\mathbf{ID}, t - 1)$, otherwise, $I(\mathbf{EX}, t) = I(\mathbf{EX}, t - 1)$.

As mentioned in Section III-A, $ag\pi$ handles \mathbf{JMP} and \mathbf{CJMP} instructions in the \mathbf{EX} stage. For the program example in Table I, the wrongly fetched instructions ($i4'$ and $i4''$) do not appear at the ISA level when \mathbf{JMP} happens. These instructions are later flushed as \mathbf{NOP} by the circuit. For this reason, the scheduling function is partial. When the scheduling function is undefined (\perp) for the \mathbf{IF} or \mathbf{ID} stage, the corresponding stage is processing an instruction to be discarded and we do not need to argue its correctness. If \perp results for the \mathbf{EX} , \mathbf{MEM} , and \mathbf{WB} stages, the corresponding stage is processing a \mathbf{NOP} instruction that has been inserted by the control unit as a result of a flush. In this case, we must demonstrate that the discarded instruction does not induce any operations on other parts of the circuit. For instance, there is no memory request when \perp appears in the \mathbf{MEM} stage.

C. Trace relation

In order to demonstrate equivalence between the pipelined circuit and the ISA, we introduce a trace relation \sim_I with the help of the scheduling function I . Most circuit fields are related to the ISA level, but a few fields are irrelevant since they are used by the processor internally to maintain the process, e.g., $enable_{ex}$ controlling the \mathbf{EX} stage. We partition the related fields of the circuit according to the pipeline stages, and as either *visible* or *invisible*. A visible field $V(f)$ has a direct counterpart in the ISA state. For example, the carry flag CF_{ex} used by \mathbf{ALU} is related to the ISA's CF and located in the \mathbf{EX} stage. An invisible field cannot be directly observed at the ISA level, since it is an intermediate signal used by the ISA to process instructions. For example, opc_{ex} has no direct counterpart in the ISA state and is extracted from the instruction in memory that is pointed to by the program counter.

Intuitively, if a stage k after t clock cycles, $I(k, t)$ is equal to n , then its visible fields are associated with the corresponding final ISA state, and its invisible fields are associated with the state prior to executing the last instruction. If $I(k, t) = \perp$, the pipeline stage should not have any effect on its visible fields.

Definition 2. $\phi \sim_I \sigma$ if for every $t, k, f \in fields(k)$:

$$\begin{aligned} V(f) &\Rightarrow I(k, t) \neq \perp \Rightarrow \phi(t).f = \pi_f(\sigma(I(k, t))) \wedge \\ \neg V(f) &\Rightarrow I(k, t) \neq \perp \Rightarrow \phi(t).f = \pi_f(\sigma(I(k, t) - 1)) \wedge \\ V(f) &\Rightarrow I(k, t) = \perp \Rightarrow \phi(t).f = \phi(t - 1).f \end{aligned}$$

For visible fields (e.g. CF_{ex} in \mathbf{EX}), π_f returns the corresponding ISA field. For invisible fields (e.g. opc_{ex} in \mathbf{EX}), π_f extracts the relevant information from the ISA state, for example, $decode_opc_{isa}$ extracts opc from the ISA memory pointed to by the ISA program counter.

Finally, we define an initial relation \sim_0 which guarantees that the circuit and ISA start from corresponding initial states: $\phi(0) \sim_0 \sigma(0)$. For example, $\phi(0).M = \sigma(0).M$ and $\phi(0).R_g = \sigma(0).R$.

D. Correctness and Proof

To prevent self modifying programs, any software executing on our pipelined processor must follow the software condition SC in Definition 3 that no instruction is modified in the memory by the previous four instructions being processed in the pipeline.

Definition 3.

$$\begin{aligned} \forall n. is_mem_str_{isa} \sigma(n) \wedge n < i < n + 5 \Rightarrow \\ \sigma(i).PC \neq mem_str_addr_{isa} \sigma(n) \end{aligned}$$

The function $is_mem_str_{isa}$ identifies if the operation is \mathbf{MSTR} for an ISA state and $mem_str_addr_{isa}$ returns the stored address ad . The circuit behaviour is undefined when SC is violated.

The correctness theorem of the pipelined circuit is formulated as follows:

Theorem 1. *If the initial circuit and ISA states are consistent $\phi(0) \sim_0 \sigma(0)$, the external environment satisfies $AX \phi$, and the program satisfies the software condition $SC \sigma$, then the trace relation is met with a unique scheduling function for ϕ and σ such that $\phi \sim_I \sigma$.*

Proof: The theorem is proved by induction on the cycle t . For the initial cycle, the proof is trivial since the initial relation \sim_0 ensures that the circuit states are initialized with the same value as the ISA for visible fields. Invisible fields are ignored in the initial cycle since no instruction has been processed by the pipeline. The following highlights the major difficulties of every pipeline stage for the induction step from t to $t + 1$:

IF: The correctness of the next cycle $PC_{next_PC_{if}}$ is achieved by the proof of the jump handler in the \mathbf{EX} stage. The software condition SC guarantees that the fetched instruction $instr_{if}$ is not affected by any instruction in the pipeline.

ID: The data from the register is correct under the condition that no instruction in the later stages writes to the reading registers at the current cycle. This condition is satisfied by the pipelined circuit, because of signals checking data hazards.

EX: Because of correct \mathbf{ALU} computations and operation code opc_{ex} in the \mathbf{EX} stage, we establish the correctness of the jump handler. For the two visible flags CF_{ex} and OF_{ex} recording the carry and overflow of \mathbf{ALU} , they are updated for specific values of function code, i.e., $func_{ex} = 0/1/2$. The \mathbf{NOP} and \mathbf{SHF} instructions are possible but not allowed to update the visible flags in the \mathbf{EX} stage. To guarantee this, we have proved that $func_{ex}$ has a certain value ($\neq 0/1/2$) when the \mathbf{NOP} or \mathbf{SHF} instruction is in the \mathbf{EX} stage.

MEM and WB: the major proof is to ensure the \mathbf{NOP} instruction does not affect other circuit components, mainly the memory M in the environment and the register file R_g . This is proved by showing a specific value of the operation code opc_{mem} and opc_{wb} . ■

In the following, we use \simeq to represent that ϕ corresponds to σ : $\phi \simeq_I \sigma \triangleq AX \phi \wedge \phi \sim_0 \sigma \wedge \phi \sim_I \sigma$.

V. INFORMATION FLOW

A. Conditional Noninterference

We devise a verification strategy that complements functional correctness to show the absence of undesired timing channels. To characterize the information flows of processors, we extend the ISA model with a leakage function obs which extracts the part of the ISA state that can affect the execution time of a program. For our case, as common observations used for constant time implementations [12], [23], obs_{ag} returns the program counter PC , the addresses of data memory accesses ad , and the condition of CJMP.

The leakage function induces the notion of observation equivalence.

Definition 4. Two ISA states s_1 and s_2 are observation equivalent, $s_1 \approx_{obs} s_2$, if $obs s_1 = obs s_2$.

We extend observation equivalence pointwise to ISA traces $\sigma_1 \approx_{obs} \sigma_2$, which results in an equivalence that resembles synchronous noninterference [20], [24]. Notice that since the attacker observes the PC , observation equivalent traces have the same length, as is common for constant time programming.

The strategy is to take the ISA traces as a reference for permitted information flow, and forbid the attacker from learning any other secrets via the timing channel. That is, the circuit is secure if for any pair of ISA-level traces σ_1 and σ_2 that are indistinguishable by the attacker ($\sigma_1 \approx_{obs} \sigma_2$), their corresponding circuit traces $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$ have the same timing behaviour. To guarantee that, we require $I_1(k, t) = I_2(k, t)$ for all pipeline stages k at every cycle, which means that the two circuit traces process the same instruction at the same stage and therefore have the same timing observations. A functionally correct processor can violate this security requirement and leak secrets, we demonstrate an example later in Section V-B. For our attacker model in Section II-D, we need to ensure $I_1(k, t) = I_2(k, t)$ at the cycle t when INTR happens. To guarantee this, it is necessary to keep track of instruction processing in the circuit over time. As discussed in Section VII, this property has been formulated as a variant of conditional noninterference. Let Σ be the set of valid ISA traces that processors can implement, conditional noninterference is defined as follows:

Definition 5. A pipelined circuit is conditional noninterferent with respect to obs , written $CNI(obs)$, if for any two ISA traces σ_1 and σ_2 in Σ such that $\sigma_1 \approx_{obs} \sigma_2$, for any circuit trace ϕ_1 with a scheduling function I_1 satisfying $\phi_1 \simeq_{I_1} \sigma_1$, there exists a circuit trace ϕ_2 and scheduling function I_2 such that $\phi_2 \simeq_{I_2} \sigma_2$, and $\forall k, t. I_1(k, t) = I_2(k, t)$.

The execution time of processors clearly depends on the behavior of the environment. To reason about the circuit's timing channel, we use an environment constraint EC which requires that two environment traces β_1 and β_2 respond to their processor traces α_1 and α_2 respectively at the same cycle t if all processor's requests before t are identical. For example, after the same sequence of memory operations with the same

addresses, β_1 and β_2 answer the next memory request in the same number of clock cycles. This allows some subsets of addresses to be reliably faster, and also caching-like behaviours where accessing an address can be faster if that address has been accessed before. For our environment AX in Section IV-A, we define EC_{ag} for all components except for di_env , because it does not affect the observation of circuit trace. Definition 6 shows the EC_{ag} constraint for mem_env where fields in α_1 and α_2 are produced by $ag\pi$, and other constraints (for mem_start_env and $intr_env$) in EC_{ag} are comparable:

Definition 6.

$$\begin{aligned} \forall t, t'. t' \leq t \wedge \alpha_1(t').cmd_g &= \alpha_2(t').cmd_g \wedge \\ \alpha_1(t').PC_g &= \alpha_2(t').PC_g \wedge \\ (\alpha_1(t').cmd_g &= load/store \Rightarrow \\ \alpha_1(t').ad_g &= \alpha_2(t').ad_g) \Rightarrow \beta_1(t).rdy = \beta_2(t).rdy \end{aligned}$$

As mentioned in Section IV-C, the trace relation and correctness proof do not argue some processor's internal fields which are used to maintain the pipeline processing. But such fields can either directly affect the scheduling results like $enable_{ex}$ or be observed by the environment like cmd_g , and thus impact the execution time of programs. Therefore, we show the equivalence of these fields between two circuit traces for every cycle t . Formally, $\phi_1 \approx_f \phi_2$ where f extracts necessary fields. Intuitively, by combining \simeq_I and \approx_f , Lemma 1 shows that instructions are processed in the same way by the processor in the two circuit traces if their ISA traces are indistinguishable.

Lemma 1. For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2 , and the circuit traces ϕ_1 and ϕ_2 satisfy $EC_{ag}(\phi_1, \phi_2)$, then $\forall k, t. I_1(k, t) = I_2(k, t)$ and $\phi_1 \approx_f \phi_2$.

Proof: Lemma 1 is proved by induction on the cycle t with the help of Theorem 1. For the initial cycle, $I_1(k, 0) = I_2(k, 0)$ directly, and $\phi_1(0) \approx_f \phi_2(0)$ because of \sim_0 and $\approx_{obs_{ag}}$. The proof for the induction step is divided into two parts: $I_1(k, t+1) = I_2(k, t+1)$ and $\phi_1(t+1) \approx_f \phi_2(t+1)$.

For the scheduling results, the main work is the equivalence of fields handling pipeline challenges in Section III-A at the cycle t , because the current scheduling results depend on the circuit state at the previous cycle. We comment on the lemma $\alpha_1(t).jmp_{ex} = \alpha_2(t).jmp_{ex}$ and omit other lemmas for data hazards and environment delays, since they are proved in the same way. From the assumption $I_1(\mathbf{EX}, t) = I_2(\mathbf{EX}, t)$, if the result is an ISA step m , $\phi_1(t) \simeq_{I_1} \sigma_1(m)$ and $\phi_2(t) \simeq_{I_2} \sigma_2(m)$ hold because of Theorem 1. Since $\sigma_1 \approx_{obs_{ag}} \sigma_2$, $\sigma_1(m) \approx_{obs_{ag}} \sigma_2(m)$. According to obs_{ag} , $\sigma_1(m)$ and $\sigma_2(m)$ have the same opc and condition of CJMP. The two ISA states have the same observation for JMP and CJMP, and therefore $\alpha_1(t).jmp_{ex} = \alpha_2(t).jmp_{ex}$. If the scheduling result is \perp , $\neg \alpha_1(t).jmp_{ex}$ and $\neg \alpha_2(t).jmp_{ex}$ hold as the correctness required. Based on the lemmas for pipeline challenges and the assumption $\phi_1(t) \approx_f \phi_2(t)$, $I_1(k, t+1) = I_2(k, t+1)$.

For \approx_f , the major work is similar to above, i.e., lemmas for pipeline challenges but for the cycle $t+1$, because the current

circuit state determines fields in \approx_f . We use the induction proof in Theorem 1 to prove, for example, $\alpha_1(t+1).jmp_{ex} = \alpha_2(t+1).jmp_{ex}$. Accordingly, $\phi_1(t+1) \approx_f \phi_2(t+1)$. ■

Lemma 2 shows the existence of ϕ_2 and I_2 for σ_2 when ϕ_1 is determined.

Lemma 2. *If $\phi_1 \simeq_{I_1} \sigma_1$ and $SC \sigma_1, \sigma_1 \approx_{obs_{ag}} \sigma_2$ and $SC \sigma_2$, then there exists a circuit trace ϕ_2 and scheduling function I_2 satisfying $\phi_2 \simeq_{I_2} \sigma_2$, and $EC_{ag}(\phi_1, \phi_2)$.*

Proof: To construct ϕ_2 , we compose a processor trace α_2 produced by $ag\pi$ with the following β_2 .

$$\begin{aligned} \beta_2(0) &= \langle |M := \sigma_2(0).M; rdy := \beta_1(0).rdy; \dots| \rangle \wedge \\ \beta_2(t+1) &= \langle |M := \text{let } t' = \text{lvr}(\beta_1, t+1) \text{ in} \\ &\quad \text{if } \alpha_2(t').cmd_g = store \wedge \beta_1(t+1).rdy \\ &\quad \text{then } \beta_2(t).M[\alpha_2(t').ad_g := \alpha_2(t').v_g] \\ &\quad \text{else } \beta_2(t).M; rdy := \beta_1(t+1).rdy; \dots| \rangle \end{aligned}$$

The function lvr returns the cycle when the latest valid memory request happened in β_1 before the cycle $t+1$. We omit other fields in β_2 since they are defined similarly as M and rdy .

The proof mainly concerns AX , because \sim_0 and EC_{ag} are fulfilled by β_2 's definition. Suppose $AX \phi_2$, the correctness and existence of I_2 is proved by Theorem 1. For AX , the main work is mem_env as other constraints are trivial or similar to mem_env . We show the proof for fetch (Definition 1) and other cases for load and store are proved similarly. By using induction on t and \approx_f from Lemma 1, the two processor traces issued the last valid fetch request at the same previous cycle t' . Because of $AX \phi_1$ and the same rdy signal in β_1 and β_2 , we apply the response time m in β_1 to β_2 and then β_2 fulfills the fetch constraint by its definition. ■

Based on Lemma 1 and 2, the verified Silver-Pi is CNI with respect to obs_{ag} if ISA traces in Σ are valid (i.e. satisfying the Definition 3 of SC). As Section IV-D mentioned, the circuit behaviour is undefined when executing self modifying programs.

Theorem 2. *If all ISA traces in Σ satisfy SC i.e. $\forall \sigma. \sigma \in \Sigma \Rightarrow SC \sigma$, then the verified Silver-Pi is CNI(obs_{ag}).*

B. Security Analysis

The verification strategy for conditional noninterference can be easily applied to different ISAs as long as their correctness verification has established a scheduling function. Handling different attacker models requires modifying the definitions of obs and EC . For example, consider a memory subsystem whose data accesses take constant time regardless of the address, then data accesses issued by the processor do not affect the execution time of programs. This condition is more restrictive on the timing channel for the environment but less for the processor. To accommodate this, we redefine a obs'_{ag} by relaxing obs_{ag} to not observe the explicit data address ad . Correspondingly, EC'_{ag} is defined by removing the following restriction in Definition 6 of EC_{ag} : $\alpha_1(t').cmd_g = load/store \Rightarrow \alpha_1(t').ad_g = \alpha_2(t').ad_g$. Theorem 2 remains valid with obs'_{ag} and EC'_{ag} .

To motivate the information flow analysis, a correct processor can still have side channels and violate conditional noninterference. For example, we developed a modified Silver-Pi, named Silver-Pi-v2, that only cleans opc_{id} instead of the entire instruction when flushing the **ID** stage. The functional correctness is unaffected since the NOP instruction does not modify the pipeline irregardless of the operands, but the CNI does not hold due to data hazards. Consider Table I, as mentioned in Section III-B, the processor does not consider opc_{id} when checking data hazards at the cycle $t+3$. Other relevant fields for checking data hazards like Ra_{id} and Fa_{id} are extracted from $i4''$. Because $i4''$ is not a valid ISA-level instruction, the two circuit traces may fetch different $i4''$ and have varying data hazard signals, leading to differences in the execution time of programs. To fix this problem, we can either modify the Silver-Pi-v2 to *not* check data hazards when **ID** stage is flushed, or refine a obs''_{ag} that guarantees ISA states have the same value for $i4''$ after JMP and CJMP. The observation obs''_{ag} is true for our hardware setting since the instruction memory is fixed after initialization. However, obs''_{ag} is not valid for all cases. For instance, uncertain data can be stored after an instruction in the Armv8 architecture. Therefore, this vulnerability can be exploited as a side channel.

We demonstrate the side channel with the program in Table I: $i3$ to $i5$ are $R0:=0$; CJMP $R1=0? i5$; INTR; where $R1$ stores a secret flag, and $i4''$ is uncertain data that can be decoded as e.g., $R2:=R0+1$ or $R2:=0$. The Silver-Pi spends the same time to complete this program despite the value of $i4''$, as it is CNI. But the Silver-Pi-v2 takes more time to execute the program if $i4''$ is $R2:=R0+1$ than $R2:=0$ when the flag in $R1$ is false. The reason is that $R2:=R0+1$ has a data dependency with $i3$. The attacker takes the execution time of the program with $R2:=0$ as a baseline, then observes and compares whether INTR occurs later or not in the interrupt handler when $i4''$ is $R2:=R0+1$, and thus learns $R1$'s flag.

VI. DISCUSSION AND EVALUATION

A. Discussion

The process of verifying correctness has discovered underlying bugs in our initial pipeline. One of these bugs is exemplified by a simple program that consists of two instructions: $i0$: $R1 := SHF R2 R3$; $i1$: $R4 := R5+R6$. The buggy processor decodes $func_{id}$ for SHF and ALU in the same way, although the SHF instruction only uses the lower 2 bits of $func_{id}$. The processor computes a wrong result of $i1$ because the 4 bits $func_{id}$ of $i0$ modifies two visible flags CF_{ex} and OF_{ex} of ALU to a distinct value. We fixed this bug by assigning 3 to the higher 2 bits of SHF's $func_{id}$, which prevents SHF to affect the flags.

A major difference in the pipeline design between Silver-Pi and similar work for RISC-V and MIPS [5], [7] is the jump handler. The Silver ISA requires to use the ALU for processing jump instructions. So, our case does not allow a jump handler in the **ID** stage like the MIPS pipeline. Actually, the MIPS pipeline can determine the next PC in the **ID** stage because of the MIPS ISA's delay slot, which is uncommon for other

ISAs such as RISC-V [25], Armv7 [26], and Armv8-A [27]. A verified pipeline in Kami [7] determines the next PC in the last stage. However, it will waste several hardware cycles to process wrongly fetched instructions compared to our design. Since the Kami work targets a subset of RISC-V RV32I ISA, the pipeline can apply our design and improve its performance.

Our verification approach comprehensively addresses essential issues of pipelined processors: functional correctness and information flow analysis. Following our correctness verification, the information flow analysis is straightforward and flexible for various ISAs (see Section V-B). Our correctness verification can be applied to other RISC ISAs such as RISC-V and MIPS. The RISC-V RV32I specification [25] is largely similar to Silver for supported instructions. As the work inspired us, Lutsyk et al. [5] presented the correctness proof of a pipelined MIPS processor using the trace relation and scheduling function.

On the other hand, it is difficult to directly apply our approach to verify large-scale processors, e.g., a pipeline with 10 stages. The major reasons are: (1) all pipeline stages are indispensable for the scheduling function; (2) the trace relation argues almost all fields in the circuit. A possible solution is to split the pipeline verification into stages. For a specific stage, assuming that necessary inputs from other stages are correct, its functional correctness is checked against the ISA. Finally, the whole pipeline's correctness is guaranteed by proving that internal assumptions are satisfied by other stages.

B. Evaluation

We have produced a full executable Silver-Pi system for an FPGA board PYNQ-Z1 by using Xilinx Vivado Design Suite (version 2020.2). To evaluate our pipeline's performance, we have tested the Silver-Pi and the non-pipelined Silver processor with 4 programs compiled using the verified CakeML compiler [16]: print Hello World, count/sort words of an input file in the memory, and a proof-checker for OpenTheory [28], [29]. The non-pipelined processor performs the fetch and execution of an instruction in different cycles. Once integrated with the other (identical) hardware components (e.g., cache and interrupt handler), the pipelined system can be clocked at 65 MHz and the non-pipelined system at 45 MHz. Table II shows the benchmark results. Given that the time of executing instructions by processors significantly exceeds the time used for handling requests by the environment, the instructions executed per second (IPS, $\frac{IPS_{\pi}}{IPS} \approx \frac{t}{t_{\pi}}$) shows that the pipelined processor is about 25% faster on average than the non-pipelined one. Our work focuses on verification, the pipelined processor has limited improvements in performance. According to the Vivado timing report, the bottleneck in the pipelined system is the cache that prevents the hardware system from higher clock frequency, and the non-pipelined system is the processor itself. Other limitations of the Silver-Pi are the lack of a forwarding unit and the branch predictor that always predicts $PC + 4$.

	hello(ms)	count(ms)	sort(ms)	checker(min)
non-pipelined	23.17	62.03	78.53	8.98
pipelined	17.94	48.48	67.19	7.28

TABLE II
EXECUTION TIME FOR PROGRAMS ON SILVER PROCESSORS

VII. RELATED WORK

A. Processor Verification

Formal verification of processors has generally focused on functional correctness with a long history (back to 1970s) by using various methods, e.g., theorem proving [2], [3], [6], [7], model checking [1], [4], [30], [31], SAT solver [32], etc. Most [1]–[4], [30]–[32] only verified simplified and abstract processor models, and therefore did not guarantee the correctness of actual processor implementations. For example, Manolios et al. [32] verified pipelined machines at *term level* that abstracts away details of processors' operations and instructions like the ALU.

Intuitively, using an interactive theorem prover (ITP) provides proof with high trustworthiness. For example, the *non-pipelined* FM9001 microprocessor [33]–[35] was modeled and verified in Nqthm theorem prover. However, there are only few verified pipelined processors at the circuit level with machine-checked proofs such as [6], [7]. Of these, Beyer et al. [6] verified a processor VAMP for the DLX ISA using the theorem proving system PVS initially, then Isabelle/HOL [36]. They used unverified tools to translate the processor design described in theorem provers to Verilog code. So, their verification is not really guaranteed down to the Verilog implementation. The Kami project [7] developed an in-order pipelined processor. Similar to our work, they verified the pipelined processor at the circuit level but targeted a high-level HDL Bluespec that requires a larger trusted computing base (TCB) than ours (the Bluespec synthesis tool). Other differences in the pipeline design are apparent which weaken the performance of their pipeline: the number of stages (4 vs 5) and jump handler as discussed in Section VI-A. For verification, their correctness is proved by refinement via intermediate modules that are tied to particular ISA and circuits, which makes their work hard to reuse for other ISAs. Their ISA model uses immutable instruction memory that remains unaffected by memory operations, and thus allows the processor to fetch the old instruction even when dealing with self modifying programs.

In contrast to these papers [6], [7], we extended the pipeline verification approach based on a trace relation and scheduling function [5], [22]. Kovalev et al. [22] implemented and verified a pipelined processor for the MIPS ISA, and Lutsyk et al. [5] extended the processor with operating system support. However, their proofs are *not* mechanized, while our proof is machine-checked in HOL4. More importantly, we updated the proof methodology to address challenges of non-MIPS ISAs e.g., refining the scheduling function for mispredicted PC (Section IV-B).

B. Low-level Information Flow

Recently, formal approaches [9]–[14] have been proposed to capture vulnerabilities (e.g., Spectre [8], Meltdown [37] and Foreshadow [38]) at microarchitecture and hardware levels. Since these vulnerabilities are mainly caused by advanced pipeline features like out-of-order (OoO) execution and speculation, most works [10]–[13] focused on the information flow analysis with respect to these features. However, Fadiheh et al. [9] presented the Orc attack caused by pipeline stalling in processors for RAW (read-after-write) hazard in the cache. So, in-order pipelined processors also require security analysis for information flows.

For our analysis, we use the same observational model as many other proposals [9]–[14] that focus on constant time programming. Some of these works [10], [12], [13] aims to analyze programs with security conditions that are variants of (conditional) noninterference on different side channels. For example, the MIL project [10], [13] can detect programs leakage via trace-driven cache side channels for OoO execution and speculation using their specific hardware semantics, but MIL ignored the timing channel because of no accurate time information at microarchitecture level.

Conditional noninterference (Definition 5) represents a variation of other similar noninterference definitions that have emerged recently to address security concerns in the post-spectre era [10], [12], [21]. For example, Guanciale et al. [10] proposed a notion of CNI that captures the new information leaks that may be introduced by the target model (for our case the processor circuit) and ignores any leaks already present in the reference model (for our case the ISA). Spectector [21] presented the speculative noninterference (SNI) that compares information flows of the same program under the speculative and nonspeculative semantics with a policy specifying the allowed leaks. Cauligi et al. [12] summarized relative notions of noninterference to detect secret leakage caused by speculative executions. Since the Silver ISA and our circuit design are deterministic, Definition 5 is a derivative of ignorance-preserving refinement (IPR) [39]. The IPR notion reflects that an observer’s ignorance (an epistemic notion that captures the inability to distinguish two related traces) must be preserved under refinement.

To identify vulnerabilities caused by hardware implementations, Fadiheh et al. [9] proposed a SAT-based model checking methodology UPEC to detect processors’ vulnerabilities at the register transfer level, and later extended the UPEC for checking OoO processors [11]. However, UPEC considers a fixed security property which makes it hard to adjust different ISA-level leakage functions. Counterexamples generated by UPEC are not always caused by hardware design (e.g., because of unreachable states), while the violations in our CNI proof are directly related to the circuit implementation. Wang et al. [14] proposed a verification tool LEAVE for checking processor implementations against ISA-level leakage contracts via an SMT solver but only demonstrated LEAVE with simple 2/3 stage RISC-V processors. To simplify the security verification,

LEAVE utilizes a decoupling theorem that separates security and functional correctness requirements for contract satisfaction and ignores the functional correctness in their verification. By contrast, our work addresses both functional correctness and information flow security for processors, and the functional correctness proof certainly contributes to security verification.

VIII. CONCLUSION

We have presented a verified in-order pipelined processor Silver-Pi for the Silver ISA, proved correct down to its Verilog implementation based on the HOL4 Verilog library. The information flow properties on the circuit’s timing channel are analyzed and the conditional noninterference for our processor is established based on the correctness proof, which shows executing programs on the processor does not leak more information than permitted on the ISA level. We have tested the Silver-Pi on an FPGA board and our benchmark results show that our pipelined processor executes programs faster than the previous non-pipelined Silver processor.

Our verification approach systematically handles the correctness and information flow properties of processor implementations, and is applicable to other ISAs like RISC-V. Therefore, we believe our work can improve the usability of formal verification for processor design and implementation to identify critical correctness and security problems.

REFERENCES

- [1] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessor control,” in *International Conference on Computer Aided Verification*. Springer, 1994, pp. 68–80.
- [2] J. Sawada and W. A. Hunt, Jr., “Processor verification with precise exceptions and speculative execution,” in *International Conference on Computer Aided Verification*. Springer, 1998, pp. 135–146.
- [3] P. Manolios, “Correctness of pipelined machines,” in *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1954. Springer, 2000, pp. 161–178.
- [4] M. N. Velev and R. E. Bryant, “Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction,” in *Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, USA, June 5-9, 2000*. ACM, 2000, pp. 112–117.
- [5] P. Lutsyk, J. Oberhauser, and W. J. Paul, Eds., *A Pipelined Multi-Core Machine with Operating System Support, Hardware Implementation and Correctness Proof*, ser. Lecture Notes in Computer Science. Springer, 2020, vol. 9999.
- [6] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul, “Putting it all together - formal verification of the VAMP,” *Int. J. Softw. Tools Technol. Transf.*, pp. 411–430, 2006.
- [7] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, “Kami: a platform for high-level parametric hardware specification and its modular verification,” *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 24:1–24:30, 2017.
- [8] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *Symposium on Security and Privacy*, 2019, pp. 1–19.
- [9] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, “Processor hardware security vulnerabilities and their detection by unique program execution checking,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 994–999.
- [10] R. Guanciale, M. Balliu, and M. Dam, “Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1853–1869.


- [11] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.
- [12] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical foundations for software spectre defenses," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 666–680.
- [13] K. Palmskog, X. Yao, N. Dong, R. Guanciale, and M. Dam, "Foundations and tools in hol4 for analysis of microarchitectural out-of-order execution," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2022, p. 129.
- [14] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," *arXiv preprint arXiv:2305.06979*, 2023.
- [15] HOL development team, "HOL interactive theorem prover," 2023. [Online]. Available: <https://hol-theorem-prover.org>
- [16] A. Löw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. C. J. Fox, "Verified compilation on a verified processor," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019, pp. 1041–1053.
- [17] A. Löw and M. O. Myreen, "A proof-producing translator for Verilog development in HOL," in *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormalISE@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 2019, pp. 99–108.
- [18] A. Löw, "Reconciling verified-circuit development and Verilog development," in *Conference on Formal Methods in Computer-Aided Design—FMCAD 2022*, 2022, p. 89.
- [19] A. C. J. Fox, "Directions in ISA specification," in *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7406. Springer, 2012, pp. 338–344.
- [20] J. Rushby, *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory Menlo Park, 1992.
- [21] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19.
- [22] M. Kovalev, S. M. Müller, and W. J. Paul, *A Pipelined Multi-core MIPS Machine - Hardware Implementation and Correctness Proof*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 9000.
- [23] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security Symposium*, vol. 16, 2016, pp. 53–70.
- [24] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 355–364.
- [25] A. Waterman and K. Asanovic, "The RISC-V instruction set manual volume I: Unprivileged ISA," *Document Version 20191213*, 2019.
- [26] ARM, "ARM architecture reference manual ARMv7-A and ARMv7-R edition," *Document Version Issue C.d*, 2018.
- [27] —, "Armv8-A instruction set architecture," *Document Version Issue 1.1*, 2020.
- [28] J. Hurd, "The OpenTheory standard theory library," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6617. Springer, 2011, pp. 177–191.
- [29] O. Abrahamsson, "A verified proof checker for higher-order logic," *Journal of Logical and Algebraic Methods in Programming*, 2020.
- [30] R. Jhala and K. L. McMillan, "Microarchitecture verification by compositional model checking," in *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings 13*. Springer, 2001, pp. 396–410.
- [31] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with ISA-Formal," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 2016, pp. 42–58.
- [32] P. Manolios and S. K. Srinivasan, "Automatic verification of safety and liveness for pipelined machines using web refinement," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 3, pp. 1–19, 2008.
- [33] W. A. Hunt, Jr., "Microprocessor design verification," *J. Autom. Reason.*, vol. 5, no. 4, pp. 429–460, 1989.
- [34] W. A. Hunt, Jr. and B. C. Brock, "A formal HDL and its use in the FM9001 verification," *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, pp. 35–47, 1992.
- [35] B. C. Brock and W. A. Hunt, Jr., "The DUAL-EVAL hardware description language and its use in the formal specification and verification of the FM9001 microprocessor," *Formal Methods in System Design*, pp. 71–104, 1997.
- [36] M. A. Hillebrand and S. Tverdyshev, "Formal verification of gate-level computer systems," in *Computer Science - Theory and Applications, Fourth International Computer Science Symposium in Russia, CSR 2009, Novosibirsk, Russia, August 18-23, 2009. Proceedings*, ser. Lecture Notes in Computer Science, vol. 5675. Springer, 2009, pp. 322–333.
- [37] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018, pp. 973–990.
- [38] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium*, 2018, pp. 991–1008.
- [39] C. Baumann, M. Dam, R. Guanciale, and H. Nemati, "On compositional information flow aware refinement," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–16.

Modular System Synthesis

Kanghee Park 

Keith J.C. Johnson 

Loris D’Antoni 

Thomas Reps 

University of Wisconsin–Madison

Madison, USA

{khpark, keithj, loris, reps}@cs.wisc.edu

Abstract—This paper describes a way to improve the scalability of program synthesis by exploiting *modularity*: larger programs are synthesized from smaller programs. The key issue is to make each “larger-created-from-smaller” synthesis sub-problem be of a similar nature, so that the kind of synthesis sub-problem that needs to be solved—and the size of each search space—has roughly the same character at each level. This work holds promise for creating program-synthesis tools that have far greater capabilities than currently available tools, and opens new avenues for synthesis research: how synthesis tools should support modular system design, and how synthesis applications can best exploit such capabilities.

I. INTRODUCTION

In program synthesis, the goal is to automatically (or semi-automatically) create programs that match high-level intents provided by a user—e.g., logical specifications or input-output examples. To date, however, synthesis tools cannot contend with large programs because they require synthesizing (or at least reasoning about) a program in its entirety.

The obvious direction is to try to exploit *compositionality* and synthesize larger programs by having them invoke other (already synthesized) programs. Consider for example the problem of writing a program for a ticket-vendor application that can, among other things, issue and reserve tickets. Building such a system requires creating modules for various data structures—perhaps a stack and queue—and using these modules in a top-level module that processes ticket requests. It is natural to ask whether such modules can be synthesized separately—i.e., in a compositional fashion.

The fundamental question is

Can one address the scalability problem of program synthesis by exploiting compositionality, so that (i) larger programs are synthesized from smaller programs, and (ii) each “larger-created-from-smaller” synthesis sub-problem is of a similar nature, so that the essence of each sub-problem (and the size of each search space) has roughly the same character?

A solution to this question is surprisingly tricky to envisage. Most existing synthesis approaches require having a concrete semantics or implementation in hand when reasoning about modules, components, APIs, etc. [5], [18], [20], and such synthesis tools end up reasoning about the entire program all the way down to its lowest-level components. Not only is this approach in fundamental opposition to the “similar-nature/similar-size” principle articulated above, it makes synthesis increasingly hard as more modules are considered.

Instead, when code is synthesized for some module M , all reasoning about lower-level modules $\{M_i\}$ on which M directly depends should be carried out in a way that is *agnostic* about the implementations of $\{M_i\}$. This observation leads us to pose two related challenges: (i) How can one carry out program synthesis without having in hand details about the implementations of lower-level modules? (ii) How can one ensure that each synthesis problem results in code that is independent of the implementations of lower-level modules?

In this paper, we present the case for the following thesis:

Program synthesis can scale using modular system design.

Modular system design is one of the most important concepts in designing software. A system should be organized in a layered fashion, where information hiding is used to hide implementation choices [16]. The *information-hiding* principle intuitively states that each module exports an interface that does not reveal specific implementation choices used inside the module, and changing the module’s implementation should not force any changes to be made to other modules.

Programmers practice modular system design, or at least aspire to it. In essence, our goal is to provide a level of automation for what good programmers do manually. Of course, we are not trying to automate everything. What is left in the hands of the programmer are architectural decisions and specifications of the intended behavior of individual modules. The programmer is responsible for the overall organization of the system’s design, and must decide such issues as: What are the layers in the system? What are the implementation choices in a given layer (such as choices about data structures and data representations)? What operations are exposed in each layer, and what is the intended behavior of each operation?

We identify *two* opportunities for providing automation for each module and, as a key contribution of this paper, we formally define these synthesis problems.

Module-Implementation Synthesis. Synthesis can be helpful in creating the implementations of the various functions in each module from some specifications. The key difference from traditional synthesis problems is that implementation details of “lower” modules are not available. Instead, one only has access to *implementation-agnostic specifications* of the semantics of such modules.

Module-Specification Synthesis. Because modules can only expose their semantics to other modules in a way that does not reveal their implementation details, it can be challenging

to come up with such semantic definitions. We propose to automate the creation of such implementation-agnostic semantic definitions using synthesis, namely, *synthesis of formulas*.

Note the role of the second kind of synthesis problem: its results provide part of the specification when one moves on to the task of synthesizing the implementation of functions in the next module. By analogy with the Paul Simon lyric “one man’s ceiling is another man’s floor” [19], we have “one module’s semantics is another module’s primitives.”

We call this approach *modular system synthesis* (MOSS). The visibility restrictions of information hiding provide the key for MOSS to achieve the objective of making synthesis scalable via “similar-nature/similar-size” sub-problems: both of our synthesis problems concern a single module of the system, and a single module’s implementation only. By concealing the implementation of lower-level modules, MOSS ensures that the formula representing the semantics of these layers remains independent of the size of the “accumulated” system as we move to higher-level layers. Moreover, MOSS retains the usual benefit of modular system design, namely, it results in software that (usually) can be readily adapted—in this context, re-synthesized—as requirements change.

This paper contributes both a framework and solidifying the concept of contract-based design in the context of program synthesis, which abstracts components or sub-systems based on their interfaces. Notably, the study of interface compatibility and composition has not been extensively explored in the context of program synthesis, opening up many opportunities for future developments. Specifically, using the aforementioned ticket-vending application as an example (§II), it (i) defines modular system synthesis (§III); (ii) defines the two kinds of synthesis problems that arise in MOSS (§IV); and (iii) describes a proof-of-concept system, called MOSSKIT, that achieves these goals (§V).

MOSSKIT is based on two existing program-synthesis techniques: JLIBSKETCH [14] a program-sketching tool that supports algebraic specifications, and SPYRO [15] a tool for synthesizing precise specifications from a given codebase. We used MOSSKIT to carry out case studies based on two-layer modular synthesis problems from Mariano et al. [14], which demonstrated that concealing lower-level components can be advantageous in reducing the complexity of the synthesis problem. Expanding upon their work, our case study in §V-B further explored scenarios involving multiple layers. MOSS exhibits even better scalability compared to scenarios where executable semantics for all lower layers are exposed. A further case study based on Mariano et al. in §V-D also highlights the challenges of writing correct specifications. Our framework and the act of performing synthesis for both the implementations and specifications of the modules unveiled bugs in the modules synthesized by Mariano et al. and in the module’s specifications, which they manually wrote.

§VI discusses related work. §VII concludes.

II. ILLUSTRATIVE EXAMPLE

We present an experiment that illustrates the various aspects of MOSS. The problem to be solved is as follows: Synthesize a simple ticket-vendor application that supports the operations `prepSales`, `resTicket`, `issueTicket`, `soldOut`, `numTicketsRem`, and `numWaiting`. (To simplify matters, we assume it is not necessary to cancel a reservation.)

A. A Modular TicketVendor Implementation

We decompose the system into three modules (Fig. 1):

Module 3: The `TicketVendor` module uses a `Queue` of reservations to implement the aforementioned operations.

Module 2: The `Queue` module implements the operations `emptyQ`, `enq`, `front`, `deq`, `sizeQ`, and `isEmptyQ`. In our setting, a `Queue` is implemented using two stacks [12].¹

Module 1: The `Stack` module implements the operations `emptyS`, `push`, `top`, `pop`, `sizeS`, and `isEmptyS`. In our setting, a `Stack` is implemented using linked-list primitives of the programming language.

Moreover, the implementation of each module is to abide by the principle of information hiding: (i) The `TicketVendor` module can use operations exposed by `Queue`, but their actual implementations are hidden in Module 2. (ii) The `Queue` module can use operations exposed by `Stack`, but their actual implementations are hidden in Module 1.

B. The Input of Modular TicketVendor Synthesis

A MOSSKIT user supplies the following information:

Architectural-design choices:

- The decomposition of the problem into `TicketVendor`, `Queue`, and `Stack` modules (gray boxes in Fig. 1).
- Which operations are to be exposed by each module, denoted by $\mathcal{P}[\text{module}]$ —e.g., in Fig. 1, the `Queue` module exposes $\mathcal{P}[\text{Queue}]$, which contains `enq` and `deq` operations, but not `push` and `pop` operations on the underlying stacks.

Data-structure/data-representation choices:

Module 3: `TicketVendor` uses a `Queue`.

Module 2: A `Queue` is implemented using two `Stack`s.

Module 1: A `Stack` is implemented using a linked list.

These choices are shown by the green boxes underneath each module in Fig. 1. For example, the `Queue` module is built on top of the `Stack` module. However, only the `Stack` interface—i.e., the function symbols in $\mathcal{P}[\text{Stack}]$ and its (potentially synthesized) implementation-agnostic specification $\varphi_{\text{sem}}^{\text{Stack}}$ —is accessible by the `Queue` module.

Specifications of the module-specific synthesis problems:

Module 3: Specifications of the behaviors of `prepSales`, `resTicket`, `issueTicket`, `soldOut`, `numTicketsRem`, and `numWaiting` in terms of the exposed `Queue` operations (and possibly other `TicketVendor` operations). For example, the implementation-specific specifications for the

¹The invariant is that the second `Stack` holds a prefix of the `Queue`’s front elements, with the top element of the second `Stack` being the `Queue`’s front-most element. The first `Stack` holds the `Queue`’s back elements—with the top element of the first `Stack` being the `Queue`’s back-most element.

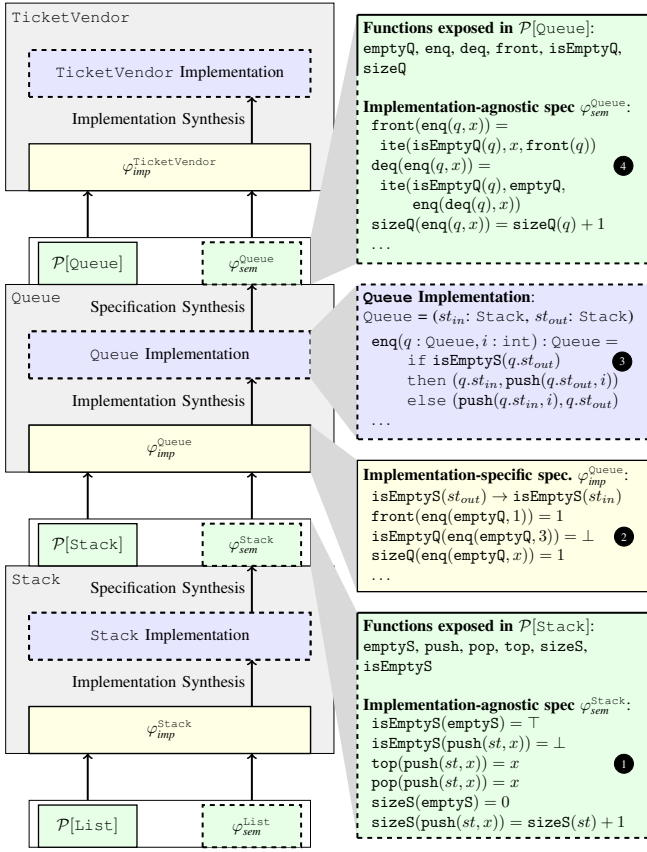


Fig. 1. Organization of the modular TicketVendor synthesis problem: user-supplied inputs are shown in solid boxes; synthesized outputs are shown in dashed boxes. On the right, the Queue module’s specifications and implementation are expanded; the other modules would have similar details.

TicketVendor module, denoted by the yellow box labeled $\varphi_{\text{imp}}^{\text{TicketVendor}}$ in Fig. 1, might constrain `issueTicket` to dequeue a buyer from the underlying Queue module, but only if `soldOut` (a TicketVendor operation) is false.

Module 2: Specifications of the behaviors of the Queue operations in terms of the exposed Stack operations (and possibly other Queue operations). For example, the implementation-specific specification for the Queue module ($\varphi_{\text{imp}}^{\text{Queue}}$), shown in Fig. 1, contains, among others, constraints that state that (i) if the first stack st_{in} is empty, so is the second stack st_{out} , (ii) enqueueing 1 on an empty queue and then retrieving the front of the queue yields 1.

Module 1: Specifications of the behaviors of the Stack operations in terms of the programming language’s linked-list operations (and possibly other Stack operations). For example, the implementation-specific specification of the Stack module ($\varphi_{\text{imp}}^{\text{Stack}}$) might specify that `push` adds an element on the front of the stack’s underlying linked list.

A user must also specify a search space of possible implementations. In MOSSKIT, this is done using a SKETCH file.

C. The Output of Modular TicketVendor Synthesis

Using the MOSS framework, we synthesize three module implementations: the TicketVendor module implementation, which satisfies $\varphi_{\text{imp}}^{\text{TicketVendor}}$ (and uses Queue);

the Queue module implementation, which satisfies $\varphi_{\text{imp}}^{\text{Queue}}$ (and uses Stack); and the Stack module implementation, which satisfies $\varphi_{\text{imp}}^{\text{Stack}}$ (and uses lists). However, to synthesize the TicketVendor module implementation, we need an *implementation-agnostic specification* of Queue, denoted by $\varphi_{\text{sem}}^{\text{Queue}}$. The same can be said for the Queue module implementation, for which we need an implementation-agnostic specification of Stack, denoted by $\varphi_{\text{sem}}^{\text{Stack}}$ ².

The user could write $\varphi_{\text{sem}}^{\text{Queue}}$ and $\varphi_{\text{sem}}^{\text{Stack}}$ manually, but it is more convenient to synthesize these specifications from the Queue and Stack module implementations, respectively. The MOSS methodology is to start with the bottom-most module and work upward, alternately applying two synthesis procedures: first synthesizing the implementation of a module M and then synthesizing M ’s implementation-agnostic specification φ_{sem}^M , which gets exposed to the next higher module.

For the modular TicketVendor-synthesis problem, we start with Stack, the bottommost module, and synthesize a Stack module implementation—a set of $\mathcal{P}[\text{List}]$ programs—that satisfies the implementation-specific specification $\varphi_{\text{imp}}^{\text{Stack}}$. (In MOSSKIT, this step is done using program sketching and the tool JLIBSKETCH [14].) This step is depicted in Fig. 1 as the Implementation Synthesis problem in the Stack module. We then switch to the Specification Synthesis problem for Stack, and synthesize $\varphi_{\text{sem}}^{\text{Stack}}$, an implementation-agnostic specification of Stack. (In MOSSKIT, this step is done by providing a grammar of possible properties and by using the tool SPYRO [15].) For the Stack module, the resultant $\varphi_{\text{sem}}^{\text{Stack}}$ is the conjunction of the equalities shown at ① in Fig. 1.

Using $\varphi_{\text{sem}}^{\text{Stack}}$ (①), together with the implementation-specific specification $\varphi_{\text{imp}}^{\text{Queue}}$ (②), we now synthesize the Queue module implementation (③)—a set of $\mathcal{P}[\text{Stack}]$ programs—and the implementation-agnostic specification $\varphi_{\text{sem}}^{\text{Queue}}$ (④) via the same two-step process.

Finally, using $\varphi_{\text{sem}}^{\text{Queue}}$ and the implementation-specific specification $\varphi_{\text{imp}}^{\text{TicketVendor}}$, we synthesize the TicketVendor module implementation. (If needed by a further client, we would then synthesize the implementation-agnostic specification $\varphi_{\text{sem}}^{\text{TicketVendor}}$.) Thus, the last output of the synthesis procedure, shown in Fig. 1, consists of implementations of Stack, Queue, and TicketVendor, and the implementation-agnostic specifications $\varphi_{\text{sem}}^{\text{Stack}}$ and $\varphi_{\text{sem}}^{\text{Queue}}$.

D. Benefits of Modular System Synthesis

At some point, we might want to decide to modify the implementation of the Queue module to use directly the linked-list primitives provided by the language (shown in Fig. 2). Information hiding allows us to do so in a compartmentalized way—i.e., by only changing the specific Queue module. Importantly, the module’s interface, composed of the function

²Technically, List is part of the programming language; however, so that all sub-problems have the same form, we assume—as shown in Fig. 1—that we also have available an implementation-agnostic specification of List, denoted by $\varphi_{\text{sem}}^{\text{List}}$. In our evaluation, we synthesize $\varphi_{\text{sem}}^{\text{List}}$ automatically.

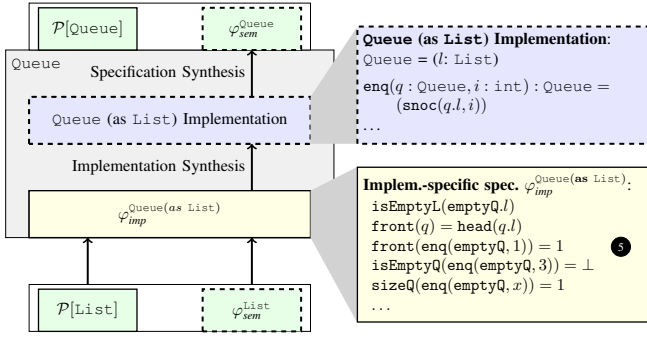


Fig. 2. Alternative implementation of the Queue module using list primitives instead of two stacks. $\mathcal{P}[\text{Queue}]$ and $\varphi_{\text{sem}}^{\text{Queue}}$ are the same as in Fig. 1.

symbols in $\mathcal{P}[\text{Queue}]$ and its implementation-agnostic specification $\varphi_{\text{sem}}^{\text{Queue}}$, does not change when the implementation of the Queue module changes. Because this interface is what the TicketVendor module was synthesized with respect to, changes to the Queue implementation are not visible to TicketVendor.

III. MODULAR SYSTEM DESIGN

In this section, we formally define modular system design and the corresponding specification mechanisms. A system is organized in modules, and each module exports a module interface MI and a specification $\varphi_{\text{sem}}^{MI}$ of the semantics of the module interface. Both MI and $\varphi_{\text{sem}}^{MI}$ hide the module's implementation. A module's implementation can also have a set of private functions PF , which can only be used within the module. A program is constructed by stacking layers of such modules.³ For instance, the example in Fig. 1 has three modules: Stack, Queue, and TicketVendor. (None of those modules have private functions.)

In the following, we assume a programming language \mathcal{P} (e.g., C with its core libraries), and use $\mathcal{P}[MI]$ to denote \mathcal{P} extended with the functions exposed by module MI .

Definition 1 (Modular System Design): A system is **implemented modularly** if it is partitioned into disjoint sets of functions $PF_1, MI_1, PF_2, MI_2, \dots, PF_n, MI_n$, such that for each $f \in PF_i \cup MI_i$, f is implemented using $\mathcal{P}[MI_{i-1} \cup PF_i \cup MI_i]$ —i.e., f only uses operations in \mathcal{P} , and calls to functions in the interface exported from layer $i-1$, to private functions of layer i , and to functions in the interface exported from layer i .

To reduce notational clutter, we will ignore private functions, and only discuss the functions in module interfaces.

As we saw in §II, we need to abide by the principle of *information hiding*—i.e., changing the implementations of any function in MI_{i-1} should not require changing the implementations of functions in MI_i . With this principle in mind, we now describe the different natures of the specification for the module implementation at a given layer i (§III-A) and the specification exposed to layer $i+1$ (§III-B).

³In general, the structure of the dependencies among layers can form a directed acyclic graph. However, to reduce notational clutter, throughout the paper we assume that the layers have a strict linear order.

A. Implementation-specific Specifications

When synthesizing specific implementations of the functions MI_i at layer i , the specifications are allowed to use symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$ —i.e., the specification can refer to the functions we are specifying and to the ones in the interface exported from the previous layer—as well as implementation-specific details from layer i (e.g., data-structure declarations).

Definition 2: An **implementation-specific specification** for a set of functions MI_i at layer i is a predicate $\varphi_{\text{imp}}^{MI_i}$ that only uses symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$.

Example 1: In the implementation-specific specification of Queue from Fig. 1, where Queue is implemented using two Stacks, one of the properties is as follows:

$$\text{isEmptyQ}(q) \iff \text{isEmptyS}(q.st_{in}) \wedge \text{isEmptyS}(q.st_{out}).$$

For the version from Fig. 2, where Queue is implemented using a List, the analogous property is

$$\text{isEmptyQ}(q) \iff \text{isEmptyL}(q.l).$$

A specification might also contain a set of examples, e.g., $\text{front}(\text{enq}(\text{emptyQ}, 1)) = 1$ and $\text{front}(\text{enq}(\text{enq}(\text{emptyQ}, 1), 2)) = 1$.

B. Implementation-agnostic Specifications

While implementation-specific details are needed to converge on an implementation with which the programmer is happy, when exposing the specification of MI_i at layer $i+1$, to abide to the principle of information hiding, one cannot provide specifications that involve function symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$, but only those in $\mathcal{P}[MI_i]$.

Definition 3: An **implementation-agnostic specification** for a set of functions MI_i at layer i is a predicate $\varphi_{\text{sem}}^{MI_i}$ that only uses symbols in $\mathcal{P}[MI_i]$.

Example 2: Because of the vocabulary restrictions imposed by Def. 3, it is natural for implementation-agnostic specifications to take the form of algebraic specifications [7], [9], [10], [13], [23]. For instance, for the Queue module, the conjunction of the following equalities is an implementation-agnostic specification $\varphi_{\text{sem}}^{\text{Queue}}$ for Queue:

$$\begin{aligned} \text{isEmptyQ}(\text{emptyQ}) &= \top & \text{isEmptyQ}(\text{enq}(q, x)) &= \perp \\ \text{sizeQ}(\text{emptyQ}) &= 0 & \text{sizeQ}(\text{enq}(q, x)) &= \text{sizeQ}(q) + 1 \\ \text{front}(\text{enq}(q, x)) &= \text{ite}(\text{isEmptyQ}(q), x, \text{front}(q)) \\ \text{deq}(\text{enq}(q, x)) &= \text{ite}(\text{isEmptyQ}(q), q, \text{deq}(\text{enq}(q, x))) \end{aligned} \quad (1)$$

Note that Eq. (1) serves as $\varphi_{\text{sem}}^{\text{Queue}}$ both for the version of Queue from Fig. 1, where Queue is implemented using two Stacks, and for the version of Queue from Fig. 2, where Queue is implemented using a List.

IV. SYNTHESIS IN MODULAR SYSTEM SYNTHESIS

In this section, we define the *implementation-synthesis* (§IV-A) and *specification-synthesis* (§IV-B) problems that enable our scheme for modular system synthesis.

A. Synthesis of Implementations

The obvious place in which synthesis can be helpful is in synthesizing the implementations of the various functions at each layer from their implementation-specific specifications. For example, in Fig. 1, an implementation of `Queue` (the function `enq` is shown in the second box on the right) is synthesized from the implementation-agnostic specification φ_{sem}^{Stack} of `Stack`, and an implementation-specific specification φ_{imp}^{Queue} that is allowed to talk about how the two `Stacks` used to implement a `Queue` are manipulated (e.g., `isEmptyS(stout)` \rightarrow `isEmptyS(stin)`).

Definition 4 (Implementation synthesis): For module interface MI_i , the **implementation-synthesis problem** is a triple $(S_i, \varphi_{sem}^{MI_{i-1}}, \varphi_{imp}^{MI_i})$, where

- S_i is the set of possible implementations we can use for MI_i (every program in S_i uses only symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$).
- $\varphi_{sem}^{MI_{i-1}}$ is an implementation-agnostic specification of the module-interface functions in MI_{i-1} .
- $\varphi_{imp}^{MI_i}$ is an implementation-specific specification that uses only symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$.

A solution to the implementation-synthesis problem is an implementation of MI_i in S_i that satisfies $\varphi_{imp}^{MI_i}$.

This particular form of synthesis where one draws a program from a search space to match a specification is fairly standard in the literature. However, we observe that a particular aspect of modular system design makes most synthesis approaches inadequate—i.e., the specification $\varphi_{sem}^{MI_{i-1}}$ can talk about functions in MI_{i-1} only in an implementation-agnostic way. For example, when synthesizing functions in `Queue`, we do not have direct access to a stack implementation—i.e., we cannot actually execute the implementation. Instead, we have access to the semantics of `Stack` through implementation-agnostic properties such as `isEmptyS(push(st, x)) = \perp` .

We are aware of only one tool, `JLIBSKETCH`, that can perform synthesis with algebraic specifications [14], and we use it in our evaluation. In `JLIBSKETCH`, one provides S_i as a program sketch (i.e., a program with integer holes that need to be synthesized), $\varphi_{sem}^{MI_{i-1}}$ as a set of rewrite rules over the functions in MI_{i-1} , and $\varphi_{imp}^{MI_i}$ as a set of assertions.

B. Synthesis of Implementation-agnostic Specifications

Because the implementation of layer $i-1$ is hidden when performing synthesis at layer i , the user has to somehow come up with implementation-agnostic specifications like the ones shown in Fig. 1. Our next observation is that such specifications can also be synthesized! With this observation, modular system design becomes a fairly automatic business where the programmer mostly has to decide how to structure modules and provide implementation-specific specifications and search spaces (typically as regular-tree grammars [3]).

In Fig. 1, the implementation-agnostic specification φ_{sem}^{Queue} of `Queue` is synthesized from the `Queue` implementation. (The same φ_{sem}^{Queue} , or one equivalent to it, is synthesized from the alternative `Queue` implementation of Fig. 2.)

Definition 5 (Specification synthesis): For module interface MI_i , a **specification-synthesis problem** is a pair (F_i, Φ_i) where

- F_i is a set of programs, written in $\mathcal{P}[MI_{i-1} \cup MI_i]$, that is a concrete implementation of MI_i .
- Φ_i is the set of possible properties we can use for $\varphi_{sem}^{MI_i}$ (every property in Φ_i uses only symbols in $\mathcal{P}[MI_i]$). (Typically, Φ_i is given as a regular-tree grammar for a fragment of logic in which terms can only use symbols in $\mathcal{P}[MI_i]$.)

A solution to the specification-synthesis problem is a set of properties $\varphi_{sem}^{MI_i} \subseteq \Phi_i$ such that for every $\alpha \in \varphi_{sem}^{MI_i}$:

Soundness: The implementation F_i satisfies α .

Precision: There is no property $\alpha' \in \Phi_i$ that implies α and such that the implementation F_i satisfies α' .

In general, there might not be just one answer to this synthesis problem because there could be multiple ways to build the set of properties $\varphi_{sem}^{MI_i}$. Furthermore, it can be the case that there are infinitely many properties in Φ_i that are sound, precise, and mutually incomparable. While in this paper we do not worry about these details, the tool we use in our evaluation `SPYRO` is always guaranteed to find a maximal set of properties in Φ_i whenever such a set is finite (`SPYRO` uses a regular-tree grammar to describe the set of possible properties Φ_i , but requires such a set to be finite.) In practice, even when the set is infinite, one can build tools that find a “good” set of properties and stop without trying to find an exhaustive set.

Discussion. When the goal is to build a system structured in a modular fashion, modular system synthesis enables defining “small” synthesis problems of similar nature that concern only a single module’s implementation.

While implementation-agnostic specifications can be synthesized via the synthesis problem defined in Def. 5, one should be aware that there is additional flexibility to be gained if one is willing to write implementation-agnostic specifications manually. In particular, if all of the implementation-agnostic specifications are synthesized, then it is necessary to create the system *bottom-up*, synthesizing the module implementations in the order MI_1, MI_2, \dots, MI_n (interleaved with the synthesis of $\varphi_{sem}^{MI_1}, \varphi_{sem}^{MI_2}, \dots, \varphi_{sem}^{MI_n}$). In contrast, when the user is willing to write the implementation-agnostic specifications manually (in addition to the implementation-specific specifications $\{\varphi_{imp}^{MI_i}\}$), then the module implementations for MI_1, MI_2, \dots, MI_n can be synthesized in any order.

V. IMPLEMENTATION AND CASE-STUDY EVALUATION

We carried out case studies of `MOSS` for the simple three-layer system that has been used as a running example and for some of the modular-synthesis problems presented in the paper that introduced `JLIBSKETCH` [14].

A. Implementation

Our implementation, called `MOSSKIT`, uses `JLIBSKETCH` [14] to synthesize the implementation code for each layer k (from the implementation-specific specification for layer k)

```

1 void snoc(list l, int val, ref list ret_list) {
2   boolean is_empty_ret;
3
4   ret_list = new list();
5   is_empty(l, is_empty_ret);
6   if (is_empty_ret) {
7     ret_list.hd = val;
8     nil(ret.tl);
9   } else {
10    ret_list.hd = l.hd;
11    snoc(l.tl, val, ret.tl);
12  }
13 }

```

Fig. 3. Implementation of `snoc` supplied to SPYRO. Returning a value from a function is done by storing the value into a reference parameter of the function.

and SPYRO [15] to synthesize the implementation-agnostic specification for use at layer $k + 1$.

JLIBSKETCH is a program-synthesis tool for Java that allows libraries to be described with collections of algebraic specifications. Similar to its popular C counterpart SKETCH [22], JLIBSKETCH allows one to write programs with holes and assertions, and then tries to find integer values for the holes that cause all assertions to hold. Each specification is a rewrite rule of the form $pattern \Rightarrow result$. For instance, one of the rewrite rules in the specification of a stack could be $pop(push(st, k)) \Rightarrow st$. To prevent infinite rewrite loops, a set of rewrite rules provided to JLIBSKETCH must not form a cycle. For instance, the rule $a + b \Rightarrow b + a$ is not allowed. The synthesis problem that JLIBSKETCH addresses is to find a program that is correct for any program input, for any library implementation that satisfies the algebraic specifications.

SPYRO addresses the problem of synthesizing specifications automatically, given an implementation. SPYRO takes as input (i) a set of function definitions Σ , and (ii) a domain-specific language \mathcal{L} —in the form of a grammar—in which the extracted properties are to be expressed. Properties that are expressible in \mathcal{L} are called \mathcal{L} -properties. SPYRO outputs a set of \mathcal{L} -properties $\{\varphi_i\}$ that describe the behavior of Σ . Moreover, each of the φ_i is a *best* \mathcal{L} -property for Σ : there is no other \mathcal{L} -property for Σ that is strictly more precise than φ_i . Furthermore, the set $\{\varphi_i\}$ is *exhaustive*: no more \mathcal{L} -properties can be added to it to make the conjunction $\bigwedge_i \varphi_i$ more precise. SPYRO uses SKETCH as the underlying program synthesizer—i.e., it generates a number of synthesis problems in the form of SKETCH files and uses SKETCH to solve such problems.

Although SPYRO is built on top of SKETCH (instead of JLIBSKETCH), in our case study we manually implemented the term-rewriting approach used by the JLIBSKETCH solver in the SKETCH files used by SPYRO to synthesize implementation-agnostic specifications that only depend on algebraic specifications of lower layers. That is, we replace every function call f appearing in a SKETCH file with a function $normalize(f)$, where $normalize$ is a procedure that applies the rewrite rules from the algebraic specification.

MOSSKIT inherits the limitations of JLIBSKETCH and

```

1 var {
2   int v1;
3   int v2;
4   list l;
5   list cons_out;
6   list snoc_out;
7 }
8 relation {
9   cons(v1, l, cons_out);
10  snoc(cons_out, v2, snoc_out);
11 }
12 generator {
13   boolean AP -> !GUARD || RHS;
14   boolean GUARD -> true
15     | is_empty(l) | !is_empty(l);
16   boolean RHS -> equal_list(snoc_out, L);
17   int I -> v1 | v2;
18   list L -> l | nil()
19     | snoc(l, I) | cons(I, L);
20 }

```

Fig. 4. Grammar for the domain-specific language in which SPYRO is to express an extracted List property. The relation definition in lines 8-11 specifies that the variables `snoc_out` `l`, `v1` and `v2` are related by $snoc_out = snoc(cons(l, v1), v2)$. From the grammar (“generator”) in lines 12-20, SPYRO synthesizes best implementation-agnostic properties of form $GUARD \rightarrow snoc_out = L$ (implicitly conjoined with $snoc_out = snoc(cons(v1, l), v2)$). In this case, the only expression for `GUARD` that succeeds is `!true`, and the property synthesized is $snoc_out = cons(v1, snoc(l, v2))$ (with the additional implicit conjunct $snoc_out = snoc(cons(v1, l), v2)$).

SPYRO—i.e., the synthesized implementations and specifications are sound up to a bound. Despite this limitation, the authors of JLIBSKETCH and SPYRO have shown that these tools typically do not return unsound results in practice. §V-E provides a detailed discussion of the limitations of MOSS and MOSSKIT.

B. Ticket-vendor Case Study

Our first benchmark is the ticket-vending application described throughout the paper. Our goal is to synthesize the four module implementations in Fig. 1 (except the bottom one), as well as the specification of each module that needs to be exposed to a higher-level module.

When synthesizing specifications, due to the scalability limitations of SPYRO, we called SPYRO multiple times with different smaller grammars instead of providing one big grammar of all possible properties of each module. In each call to SPYRO, we provided a grammar in which we fixed a left-hand-side expression of an equality predicate, and asked SPYRO to search for a right-hand-side expression for the equality. We allowed the right-hand-side expression to contain a conditional where the guard can be selected from the outputs of Boolean operators in the module, their negation, or constants. For instance, Figures 3 and 4 illustrate two inputs provided to SPYRO to solve the specification-synthesis problem for `List`: (i) a program describing the implementation of `List` (Fig. 3), and (ii) a grammar describing the set of possible properties (Fig. 4).

Because we wanted to use the synthesized equalities as input to JLIBSKETCH when synthesizing the implementation


```

1 public void enq(int x) {
2     Stack st_in = this.st_in;
3     Stack st_out = this.st_out;
4
5     assume !st_out.isEmpty() || st_in.isEmpty();
6
7     if (genGuard(st_in, st_out)) {
8         st_in = genStack2(st_in, st_out, x);
9         st_out = genStack2(st_in, st_out, x);
10    } else {
11        st_in = genStack2(st_in, st_out, x);
12        st_out = genStack2(st_in, st_out, x);
13    }
14
15    assert !st_out.isEmpty() || st_in.isEmpty();
16
17    this.st_in = st_in;
18    this.st_out = st_out;
19 }

```

Fig. 5. JLIBSKETCH sketch of `enq`. Lines 5 and 15 assert the implementation-specific property $\text{isEmptyS}(st_{out}) \rightarrow \text{isEmptyS}(st_{in})$. JLIBSKETCH generates an expression to fill in each occurrence of the generators, `genStack2` and `genGuard`—the reader can think of each of these generators as being grammars from which JLIBSKETCH can pick an expression. For these generators, expressions can be variables or single function calls to functions of the appropriate type—e.g., `genStack2` can generate expressions such as `st_in`, `st_out`, `st_in.pop()`, `st_out.pop()`, etc.

of the next higher-level module, we provided grammars of equalities that avoided generating cyclic rewrite rules. We addressed this issue by limiting the search space for the right-hand-side expression. The function symbols permitted in the right-hand-side expression are one of the functions in the left-hand-side expression, functions used in the implementation of a function in the left-hand-side expression, or constants. Also, the outermost function symbol of the left-hand side can only be applied to a strictly smaller term.

To illustrate some of the properties synthesized by MOSSKIT (that are not shown in Fig. 1) the complete set of equalities in the implementation-agnostic specification φ_{sem}^{List} synthesized by SPYRO is the following:

```

head(cons(hd, tl)) = tl    isEmptyL(nil) =  $\top$ 
tail(cons(hd, tl)) = hd   isEmptyL(cons(hd, tl)) =  $\perp$ 
sizeL(nil) = 0            snoc(nil, x) = cons(x, nil)
sizeL(cons(hd, tl)) = sizeL(tl) + 1
snoc(cons(hd, tl), x) = cons(hd, snoc(tl, x))

```

When considering the cumulative time taken to synthesize the algebraic specification of each module, SPYRO took 41 seconds for φ_{sem}^{List} (longest-taking property 7 seconds), 34 seconds for φ_{sem}^{Stack} (longest-taking property 7 seconds), and 44 seconds for φ_{sem}^{Queue} (longest-taking property 13 seconds).

We used JLIBSKETCH to synthesize implementations of the modules. In addition to the implementation-agnostic specification of the module below the one we were trying to synthesize, we provided an implementation-specific specification of the module to be synthesized. For example, the φ_{imp}^{Stack} specification involved JLIBSKETCH code with 17 assertions, and the following examples are an excerpt from the φ_{imp}^{Stack} specification (x, y , and z are universally quantified integers

that are allowed to be in the range 0 to 10):

```

top(push(emptyS, x)) = x    top(push(push(emptyS, x), y)) = y
sizeS(emptyS) = 0          sizeS(push(emptyS, x)) = 1

```

Besides the assertions, we provided JLIBSKETCH with a fairly complete sketch of the structure of the implementation: we provided loops and branching structures, and only asked JLIBSKETCH to synthesize basic statements and expressions. For example, the sketch provided for the operation `enq` of module `Queue` = $(st_{in} : \text{Stack}, st_{out} : \text{Stack})$ is shown in Fig. 5. This sketch of `enq` of module `Queue` uses two Stacks: st_{in} , which stores elements in the rear part of the queue, and st_{out} , which stores elements in the front part of the queue. Stack st_{in} holds the rearmost element on top, and Stack st_{out} stores the frontmost element on top. To make the `front` operation more efficient, we decided to make sure that the frontmost element is always at the top of st_{out} . This implementation decision is expressed as assertions in lines 5 and 15, constituting an implementation-specific specification φ_{imp}^{Queue} , shown as 2 in Fig. 1.

Afterward, based on the implementation synthesized by JLIBSKETCH, SPYRO was able to solve each `Queue` specification-synthesis problem within 40 seconds, yielding the following implementation-agnostic specification φ_{sem}^{Queue} :

```

isEmptyS(emptyQ) =  $\top$     isEmptyQ(enq(q, i)) =  $\perp$ 
sizeQ(emptyQ) = 0
sizeQ(enq(q, i)) = sizeQ(q) + 1
isEmptyQ(q)  $\rightarrow$  front(enq(q, i)) = i
¬isEmptyQ(q)  $\rightarrow$  front(enq(q, i)) = front(q)
isEmptyQ(q)  $\rightarrow$  deq(enq(q, i)) = q
¬isEmptyQ(q)  $\rightarrow$  deq(enq(q, i)) = enq(deq(q), i)

```

A `TicketVendor` is implemented using a `Queue`, which stores the id numbers of clients who have reserved tickets. Each issued ticket contains the id of the buyer. The implementation-specific specification $\varphi_{imp}^{TicketVendor}$ consisted of JLIBSKETCH code with 24 assertions, and contains multiple examples, such as the following (again, x and y are universally quantified integers that are allowed to be in the range 0 to 10):

```

numTicketsRem(preSales(2)) = 2
numWaiting(preSales(2)) = 0
numWaiting(resTicket(preSales(2), x)) = 1
issueTicket(resTicket(preSales(2), x)).owner = x

```

Again, we provided JLIBSKETCH with a fairly complete sketch of the program structure, and JLIBSKETCH was able to synthesize the implementations of all the `TicketVendor` functions within 10 seconds. For example, the function `prepSales` for `TicketVendor` = $(num_{ticket} : \text{int}, q_{waiting} : \text{Queue})$ was synthesized as `prepSales` ($n : \text{int}$) := (n, emptyQ) .

We compared the time needed to synthesize each module from the algebraic specification of the previous module to the time needed to synthesize using the implementation of all previous modules. Synthesizing `Stack` from the specification φ_{sem}^{List} took 3 seconds instead of the 2 seconds needed when the implementation of `List` was provided. Synthesizing `Queue` from the specification φ_{sem}^{Stack} took 188

seconds instead of the 799 seconds needed when the concrete implementations of `Stack` and `List` were provided. Synthesizing `TicketVendor` from the specification φ_{sem}^{Queue} took 7 seconds, but `JLIBSKETCH` crashed when the concrete implementations of `Queue`, `Stack` and `List` were provided.

Key finding: This experiment shows that modular synthesis takes 1-5 minutes per module, whereas the time taken to synthesize a module from the underlying module implementations grows with the number of modules—to the point where synthesis is unsuccessful with existing tools.

As discussed in §II-D, we also synthesized an implementation of `Queue` that uses `List` instead of two `Stack`s. The `List` holds the oldest element of the `Queue` at its head. The implementation-specific specification $\varphi_{imp}^{Queue (as List)}$ consisted of `JLIBSKETCH` code with 19 assertions, including examples similar to those shown at 5 in Fig. 2. We used `JLIBSKETCH` to verify whether the specification φ_{sem}^{Queue} still held true for the new implementation. Because it did (confirmation took <1 second), `TicketVendor` does not need to be changed to use the `Queue (as List)` implementation.

C. Case Studies from Mariano et al. [14]

Our second set of benchmarks is collected from the paper that introduced synthesis from algebraic specifications via `JLIBSKETCH` [14]. In that work, Mariano et al. used a number of benchmarks that involve two modules—e.g., synthesizing a backend cryptographic component for a tool that brings `NuCypher` to `Apache Kafka`, using `ArrayList` and `HashMap` as underlying modules. The goal of their paper was to show that in `JLIBSKETCH` it was easier/faster to synthesize the module at layer 1 when the module of layer 0 was exposed through an algebraic specification (rather than a concrete implementation). The current implementation of `MOSSKIT` does not support strings, so we used only the benchmarks for which the algebraic specifications for the layer-0 module (i) did not use `string` operations, and (ii) did not use auxiliary functions that were not in the signature of the module. In total, we considered four layer-0 modules: `ArrayList`, `TreeSet`, `HashSet`, and `HashMap`. Each `JLIBSKETCH` benchmark consisted of (i) an algebraic specification of the layer-0 module (written by hand), (ii) a `SKETCH`-like specification of the layer-1 module, and (iii) a mock implementation of the layer-0 module—i.e., a simplified implementation that mimics the module’s intended behavior (e.g., `HashSet` is implemented using an array). The mock is not needed by `JLIBSKETCH`, but allowed Mariano et al. to compare synthesis-from-algebraic-specifications against synthesis-from-mocks [14, §5].

We used these items in a different manner from the `JLIBSKETCH` experiments. From just the mock implementation of layer 0, we asked `MOSSKIT` to synthesize a most-precise algebraic specification, which we compared with the algebraic specification manually written by Mariano et al. From that algebraic specification and the `SKETCH`-like specification of the layer-1 module, we asked `MOSSKIT` to synthesize the implementation of layer 1. (The second step essentially replicated the algebraic-synthesis part of the `JLIBSKETCH` experiments.)

For the layer-0 synthesis step of each benchmark, we synthesized algebraic specifications using grammars similar to the ones used in §V-B.

When considering the time taken to synthesize the entire algebraic specification of each module, `SPYRO` took 626 seconds for $\varphi_{sem}^{ArrayList}$, 54 seconds for $\varphi_{sem}^{HashSet}$, and 1,732 seconds for $\varphi_{sem}^{HashMap}$. Because mock implementations are simplified versions of actual implementations, the mock implementation of `TreeSet` is identical to the mock implementation of `HashSet`—i.e., they both represent sets as arrays. Furthermore, the two implementations have the same algebraic specifications—i.e., $\varphi_{sem}^{HashSet} = \varphi_{sem}^{TreeSet}$ —which can thus be synthesized in the same amount of time.

Key finding: For all but two benchmarks, the \mathcal{L} -conjunctions synthesized by `MOSSKIT` were equivalent to the algebraic properties manually written by Mariano et al. For the mock implementation of `HashMap` and `ArrayList` provided in `JLIBSKETCH`, for specific grammars, `MOSSKIT` synthesized empty \mathcal{L} -conjunctions (i.e., the predicate `true`) instead of the algebraic specifications provided by Mariano et al.—i.e., $k_1 = k_2 \Rightarrow \text{get}(\text{put}(m, k_1, v), k_2) = v$ and $i = j \Rightarrow \text{get}(\text{set}(l, i, v), j) = v$, for `HashMap` and `ArrayList`, respectively. Upon further inspection, we discovered that `JLIBSKETCH`’s mock implementation of `HashMap` was incorrect, and did not satisfy the specification that Mariano et al. gave, due to an incorrect handling of hash collision! After fixing the bug in the mock implementation of `HashMap`, we were able to synthesize the expected algebraic specification. However, when inspecting the implementation of `ArrayList`, we found that for this benchmark the implementation was correct but the algebraic specification provided by Mariano et al. was incorrect! After modifying the grammar, we could synthesize the correct algebraic specification $(i = j) \wedge (0 \leq i) \wedge (i \leq \text{sizeL}(l)) \Rightarrow \text{get}(\text{set}(l, i, v), j) = v$. However, this modification revealed a bug in one of the implementations of `HashMap` that Mariano et al. had synthesized from the earlier erroneous specification! We discuss this finding further in the next section.

This finding illustrates how modular system synthesis can help to *identify* and *avoid* bugs in module implementations.

D. Additional Case Studies Based on Mariano et al. [14]

We noticed that the `JLIBSKETCH` benchmarks provided an opportunity to build a more complicated benchmark that involved 3 modules (instead of 2). In particular, two of the benchmarks involved synthesizing the implementation of a (layer-1) `HashMap` module from a (layer-0) algebraic specification of `ArrayList`. (The two benchmarks synthesized different implementations that handled collisions differently and we refer to the corresponding modules as `HashMap1` and `HashMap2`.) The third benchmark involved synthesizing the implementation of a (layer-2) `Kafka` from a (layer-1) algebraic specification of `HashMap`. Thus, we built two 3-layer benchmarks in which the goal was to synthesize `Kafka` using an implementation of `HashMap` that used an implementation of `ArrayList`. For us, each 3-layer benchmark involved four

synthesis problems: (1) the algebraic specification $\varphi_{sem}^{ArrayList}$ of `ArrayList` (from the mock); (2) the implementation of either `HashMap1` or `HashMap2`; (3) the algebraic specification of `HashMap`; and (4) the implementation of `Kafka` (this part was already synthesized in [14]).

As discussed in the previous section, we identified a bug in the specification $\varphi_{sem}^{ArrayList}$ manually provided by Mariano et al., and were able to use to MOSSKIT to synthesize a correct algebraic specification—i.e., step (1). For step (2), the implementation synthesized by Mariano et al. for `HashMap2` was still correct, and we could also use MOSSKIT to synthesize it from the corrected specification $\varphi_{sem}^{ArrayList}$. However, the implementation of `HashMap1` synthesized by JLIBSKETCH was incorrect because it depended on the original, erroneous specification $\varphi_{sem}^{ArrayList}$ for `ArrayList`—(1) put could store values to negative indices; and (2) get could search key from incorrect index after rehashing. We manually changed the implementation of the rehashing function in the sketch of `HashMap1` to fix the bug, but the change was large enough that we did not attempt to rewrite the program sketch needed to synthesize this specification (i.e., we manually wrote the implementation of `HashMap1` instead of synthesizing it). Synthesis problem (3) is at the heart of handling a multi-module system in a modular fashion: we used MOSSKIT to synthesize algebraic specifications of `HashMap1` and `HashMap2`—in each case, giving MOSSKIT access to the (correct) implementations of `HashMap1` and `HashMap2` and the (correct) algebraic specification of `ArrayList` (but not an implementation of `ArrayList`).

Key finding: MOSSKIT failed to synthesize the same algebraic specification we had obtained for `HashMap` in §V-C when attempting to synthesize a specification for `HashMap1` and `HashMap2`. When inspecting the synthesized properties, we realized that the algebraic specification $\varphi_{sem}^{ArrayList}$ exposed by `ArrayList` still had a problem! In particular, $\varphi_{sem}^{ArrayList}$ was too weak to prove the algebraic specifications needed by `HashMap1` and `HashMap2`—i.e., $\varphi_{sem}^{ArrayList}$ did not characterize properties that were needed by `HashMap1` and `HashMap2` to satisfy the algebraic specification $\varphi_{sem}^{HashMap}$. We used Sketch itself to produce a violation of the algebraic specification $\varphi_{sem}^{HashMap}$ for `HashMap1` under the weaker assumption that `ArrayList` only satisfied the specification $\varphi_{sem}^{ArrayList}$, and used the violations generated by SKETCH to identify what properties we needed to add to strengthen $\varphi_{sem}^{ArrayList}$. In particular, $\text{sizeL}(\text{ensureCapacity}(l, n)) = \text{sizeL}(l)$ and $\text{get}(\text{ensureCapacity}(l, n), i) = \text{get}(l, i)$ were added to describe the behavior of `ensureCapacity`. We were then able to modify the grammar used to synthesize algebraic specifications for $\varphi_{sem}^{ArrayList}$ and synthesize the missing property. After obtaining $\varphi_{sem}^{ArrayList}$, we successfully synthesized the full algebraic specification for `HashMap2` (i.e., $\varphi_{sem}^{HashMap}$) and most of the algebraic specification for `HashMap1`. Because the corrected implementation of `HashMap1` was particularly complicated—e.g., each call to `put` requires rehashing when the load factor is greater than a predefined value—MOSSKIT timed out while synthesizing every property, with the excep-

tion of the property $\text{get}(\text{emptyMap}, k) = \text{err}$.

This finding illustrates how modular system synthesis can help identify when module specifications are not strong enough to characterize the behavior of other modules.

E. Limitations of MOSSKIT

JLIBSKETCH and SPYRO represent the algebraic specifications of modules as rewrite rules for algebraic datatypes (ADTs). Reasoning about ADTs is a challenging problem, and to the best of our knowledge, SKETCH and JLIBSKETCH are only frameworks capable of handling problems involving ADTs effectively. Therefore, MOSSKIT uses them as the underlying solver and inherits limitations of SKETCH.

The primary limitation of MOSSKIT is its bounded soundness guarantee. SKETCH ensures soundness only for a bounded number of loop/recursion unrollings, and bounded input sizes. Verifying the unbounded correctness of the synthesized programs poses a significant challenge, as semantics of lower-level modules are represented as rewrite rules on ADTs. As a future direction, we plan to integrate MOSSKIT with verifiers such as Dafny to perform full verification, as was done in [15] for the properties synthesized by SPYRO. However, it is worth noting that MOSSKIT has already been useful in finding bugs in existing implementations: specification synthesis has helped find implementation errors in the case studies of Mariano et al. [14], as demonstrated in §V-C and §V-D.

Although the case studies in §V-B and reference [14] show satisfactory performance of SKETCH for most problems, scalability issues persist. In particular, unrolling nested loops significantly increases the number of holes of the SKETCH problem, which increases the problem’s difficulty.

Besides the limitations inherited from SKETCH, MOSS has a specific requirement for the system’s modular structure, which should be a directed acyclic graph (DAG)—i.e., the implementation-agnostic specifications of all dependent modules must be provided to synthesize a particular module. MOSS addresses the challenges in writing accurate specifications by using the synthesis of implementation-agnostic specifications. However, in this approach one needs to synthesize all dependent modules and their specifications before attempting to synthesize a new module. Alternatively, to synthesize higher-level modules without the lower-level implementations, the user can manually supply the implementation-agnostic specifications of the lower-level modules.

VI. RELATED WORK

A problem related to ours is that of component-based synthesis (CBS), where the goal is *assembling* pre-existing components/APIs to generate more complex programs. Many existing approaches for solving CBS problems scale reasonably well [5], [18], [20], but require the individual components to be executable. In our setting, this approach is not possible because the details of lower-level components (e.g., how a `Stack` is implemented) need not be observable.

A few tools have abstracted components and modules using specifications. JLIBSKETCH [14] uses algebraic properties to

represent the semantics of modules and is a key component of our implementation. (CL)S [2] and APIphany [8] use types to represent the behavior of components and can be used in tandem with specialized type-directed synthesizers. The key differences between our work and these tools is that MOSS provides two well-defined synthesis primitives that support composing multiple modules, rather than synthesizing just one implementation for one module. Furthermore, the aforementioned types are limited in how they can represent relations between multiple components in an implementation-agnostic way, thus making us opt for algebraic specifications.

Many synthesis tools perform some kind of “compositional” synthesis by breaking an input specification into sub-specifications that are used to separately synthesize sub-components of a target program [1], [17]. This notion of “compositionality” is orthogonal to ours, and is more of a divide-and-conquer approach to solving *individual* synthesis problems. MOSS can make use of such a divide-and-conquer approach when synthesizing a module’s implementation.

For the task of synthesizing an algebraic specification, MOSSKIT uses SPYRO. Besides SPYRO, there are a number of works about discovering specifications from code, based on both static techniques [6], [21] and dynamic techniques [4], [11]. The static approaches mostly target predicates involving individual functions (instead of algebraic properties and equalities involving multiple functions). The dynamic techniques are flexible and can identify algebraic specifications (e.g., for Java container classes [11]), but require some “bootstrapping” inputs, and only guarantee soundness with respect to behaviors that are covered by the tests that the inputs exercise.

VII. CONCLUSION

Conceptual contributions. At the conceptual level, this paper contributes both a framework and a new way to think about program synthesis that opens many research directions. Specifically, the paper introduces MOSS, a framework for using synthesis to perform modular system synthesis. The main contribution of this paper is not an immediate solution to the modular-synthesis problem, but rather the identification of two key synthesis primitives that are required to realize MOSS in practice: 1) synthesis from an implementation-agnostic specification, and 2) synthesis of an implementation-agnostic specification. While our tool implements both of these primitives using tools based on SKETCH (thus inheriting its limitations), an interesting research directions is whether other synthesis approaches (enumeration, CEGIS, etc.) can be extended to handle our synthesis problems, perhaps by leveraging the popular `egg` framework [24] which allows one to reason about equivalence of terms with respect to a term-rewriting system—i.e., our algebraic specifications.

Experimental Contributions. We created MOSSKIT, a proof-of-concept implementation of MOSS based on two existing program-synthesis tools: JLIBSKETCH [14], a program-sketching tool that supports algebraic specifications, and SPYRO [15], a tool for synthesizing precise specifications

from code. The case studies carried out with MOSSKIT show that (i) modular synthesis is faster than monolithic synthesis, and (ii) performing synthesis for both implementations and specifications of the modules can prevent subtle bugs.

ACKNOWLEDGEMENT

Supported, in part, by a Microsoft Faculty Fellowship, a gift from Rajiv and Ritu Batra; by ONR under grant N00014-17-1-2889; and by NSF under grants CCF-1750965, 1763871, 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] R. Alur, P. Cerný, and A. Radhakrishna. Synthesis through unification. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2015.
- [2] J. Bessai, A. Dudenhefner, B. Döder, M. Martens, and J. Rehof. Combinatory logic synthesizer. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2014.
- [3] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2008.
- [4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
- [5] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612, 2017.
- [6] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.
- [7] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Abstract data-types as initial algebras and correctness of data representations. In *Proceedings Conference on Computer Graphics, Pattern Recognition and Data Structure*, May 1975.
- [8] Z. Guo, D. Cao, D. Tjong, J. Yang, C. Schlesinger, and N. Polikarpova. Type-directed program synthesis for restful apis. In R. Jhala and I. Dillig, editors, *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 122–136. ACM, 2022.
- [9] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, Computer Systems Research Group, Univ. of Toronto, Toronto, Canada, Sept. 1975.
- [10] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.
- [11] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Trans. Software Eng.*, 33(8):526–543, 2007.
- [12] R. Hood and R. Melville. Real-time queue operation in pure LISP. *Inf. Process. Lett.*, 13(2):50–54, 1981.
- [13] B. H. Liskov and S. N. Zilles. Specification techniques for data abstractions. *IEEE Trans. Software Eng.*, 1(1):7–19, 1975.
- [14] B. Mariano, J. Reese, S. Xu, T. Nguyen, X. Qiu, J. S. Foster, and A. Solar-Lezama. Program synthesis with algebraic library specifications. *Proc. ACM Program. Lang.*, 3(OOPSLA):132:1–132:25, 2019.
- [15] K. Park, L. D’Antoni, and T. Reps. Synthesizing specifications. *CoRR*, abs/2301.11117, 2023.

- [16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, 1972.
- [17] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the 24th International Conference on Artificial Intelligence, IJCAI’15*, page 792–800. AAAI Press, 2015.
- [18] K. Shi, J. Steinhardt, and P. Liang. FrAngel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL):73:1–73:29, 2019.
- [19] P. Simon. One man’s ceiling is another man’s floor, May 1973. T-700.050.850-1 BMI, ISWC, JASRAC.
- [20] R. Singh, R. Singh, Z. Xu, R. Krosnick, and A. Solar-Lezama. Modular synthesis of sketches using models. In K. L. McMillan and X. Rival, editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 395–414. Springer, 2014.
- [21] J. L. Singleton, G. T. Leavens, H. Rajan, and D. R. Cok. Inferring concise specifications of APIs. *CoRR*, abs/1905.06847, 2019.
- [22] A. Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013.
- [23] J. M. Spitzen and B. Wegbreit. The verification and synthesis of data structures. *Acta Informatica*, 4:127–144, 1974.
- [24] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panckekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

Modelling and Verification of Security-Oriented Resource Partitioning Schemes

Adwait Godbole^{*}, Leiqi Ye[†], Yatin A. Manerkar[†], Sanjit A. Seshia^{*}

^{*}University of California Berkeley, Berkeley, USA

{adwait, ssesia}@berkeley.edu

[†]University of Michigan, Ann Arbor, USA

{yeleiqi, manerkar}@umich.edu

Abstract—Side channel attacks such as Spectre and Meltdown exploit on-chip resources such as caches and buffers shared between the victim and the attacker in order to leak secret information from the victim. Previous works aim to mitigate these attacks by partitioning these vulnerable resources and allocating disjoint partitions to mutually untrusting process domains. While disjoint allocation prevents the attacker from gaining direct visibility of victim’s partitions, secret information can also be leaked through the book-keeping state implementing the replacement/allocation policy. Proofs of security must reason about the partitions as well as the policy.

In this work, we develop an abstract formal model for a generic security-oriented resource partitioning scheme, and formulate a corresponding attacker model. We then develop *conditional equality*-based relational invariants that enable unbounded proofs of security of the partitioning scheme with respect to the attacker model. These invariants allow us to reason about the state of the partitioning policy, which, as we discuss, can be more challenging than reasoning about the partitions themselves. We use our framework to model two resource partitioning approaches: DAWG and COLORIS. We demonstrate that using our invariants leads to verification performance improvements over other, more automated, model-checking approaches such as BMC and PDR.

I. INTRODUCTION AND EXAMPLE

Transient execution attacks such as Spectre [1], Meltdown [2], and the more recent MDS attacks [3], [4], [5], [6] exploit microarchitectural features such as caches, buffers, and functional units in order to leak secret data (e.g. cryptographic private keys). These features form *side channels* that allow the attacker to observe execution artefacts such as cache accesses, execution time and power consumption. The attacker can infer the secret data based on these observations. For instance, cache-based side channels [7], are based on the fact that victim’s accesses to specific cache lines are observable to the attacker through a timing-based side channel [8]. Timing measurements can then be used to reconstruct accessed memory addresses, and consequentially, leak data that these addresses depend on.

While the microarchitectural features exploited by these attacks vary (e.g. caches [1], [2], [9], TLB [10], load, store and line-fill buffers [3], [4], [5], [6]), the central theme is exploiting a cache-like *resource* which is *shared* between the attacker and victim. In order to mitigate these attacks, there are approaches (e.g. [11], [12], [13], [14]) that partition these shared resources, and enforce allocation of *disjoint* partitions

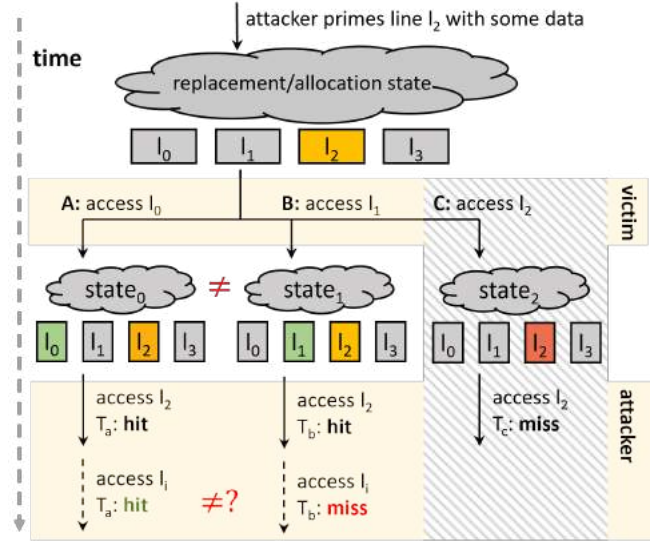


Fig. 1. Example illustrating a Prime+Probe style attack. The grey cloud represents the policy state and rectangles represent the resource partitions (cache lines). (A, B) indicate behaviours permitted by a partitioning scheme which has allocated lines l_0, l_1 to the victim and line l_2 to the attacker. (C, textured) indicates a behaviour which is possible on a non-partitioned cache, but is not possible on a partitioned cache with the above allocation.

to mutually distrusting processes. Disjoint partitions disallow an attacker from affecting (modifying/observing) the victim’s partitions, thus eliminating side channels formed by resource entries (e.g. cache lines). However, data leakage may still be possible through the state that implements the replacement/partitioning policy (e.g. [15] demonstrates an attack which leaks data through the Least-Recently-Used replacement policy state). Hence, proofs of security for designs implementing resource partitioning must reason over the partitioning policy in addition to the partition contents themselves.

In this work, we develop an abstract formal model for resource partitioning schemes, and a corresponding attacker model, capturing cache-based timing side channels. With the goal of proving the security of resource-partitioning approaches, we formulate *conditional-equality invariants*, which are a form of relational invariants [16]. These invariants enable unbounded proofs of the system against a non-interference-based formulation of the attacker model.

Previous works (e.g. [17], [18]) use pure equality-based relational invariants. While pure equality-based invariants suffice for reasoning over partition contents, the policy state requires more nuanced reasoning which is enabled by conditional equality invariants. We now illustrate this in the context of a Prime+Probe [8] attack (ref. Fig. 1).

Prime+Probe on a Resource Partitioned Cache:

As a warm-up, we begin by discussing how a Prime+Probe attack operates on an unpartitioned cache. For simplicity, consider a segment of the cache consisting of four cache lines: $\{l_0, \dots, l_3\}$. These are illustrated at the top of Fig. 1. Initially the attacker *primes* the cache by loading a value into a specific line, say l_2 (highlighted in orange in Fig. 1).

Following this, the victim performs a memory operation leading to a cache access (victim tab in Fig. 1). We consider three scenarios where the victim accesses either l_0, l_1 or l_2 (denoted A, B, and C respectively). In cases (A, B) the victim access (highlighted in green) does not evict the attacker primed line l_2 while in case (C) it does (highlighted in red).

Now, if the attacker accesses (*probes*) the cache on line l_2 , there are two possible outcomes: in case (C) the access results in a cache miss (since the victim evicted the attacker's primed data), while cases (A, B) result in a cache hit (since the attacker's data was untouched). This difference manifests in the execution time of the attacker's probing access and allows the attacker to deduce the cache line accessed by the victim earlier. Since the cache line is indexed by the memory address, the attacker can infer this address, and as a consequence, any potentially secret data that the address depend on.

Cache partitioning schemes avoid this by allocating disjoint partitions to untrusting processes. In our example, a partitioning scheme can allocate l_0, l_1 to the victim process and l_2 to the attacker process. Since the victim is not allocated l_2 , (C) is not a valid execution for the partitioned cache. Since both remaining executions (A, B) have the same (hit) result on the attacker's probing access, it cannot distinguish between these, and consequently, cannot infer the victim's loaded address.

Leakage through the policy state: While disjointness of victim and attacker partitions prevents a hit/miss timing side-channel leakage for the *first* probing access, leakage may still be possible through the replacement/allocation policy state (cloud in Fig. 1). The possible victim accesses, (A, B), may still lead to differing policy states ($state_0$) and ($state_1$). While this difference is not visible on the first probing access, it could potentially manifest after *some* number of attacker accesses.

Policy states need not be equal: In order to prove that an arbitrary number of attacker accesses lead to the same (hit/miss) outcome, relational model-checking [19], [16] based approaches (e.g. [18], [17], [20]) develop invariants relating possible resource states (A, B in Fig. 1). While the contents of the attacker-observable partition (l_2) itself must be equal, (else the attacker observes different outcomes), for the policy state, enforcing exact equality may be too strong. Policy states which are not fully equivalent, but still related in *some way* may be sufficient to ensure identical attacker access outcomes. Our model allows us to formulate *conditional* equality-based

invariants, that are more nuanced than exact equality, and hence support inductive proofs of non-interference.

Related work: We formulate the security of partitioning schemes as a non-interference-based [21] hyperproperty [22]. Our verification approach is based on the well-known translation of non-interference to 2-safety [23], [24]. We verify the resulting 2-safety property with symbolic model checking [25], [26] (e.g. BMC, k-Induction [27], [28], PDR [29]).

There are several approaches that perform verification of non-interference-based properties, both on RTL (e.g. [30], [31]) and on abstract models (e.g. [32], [33], [34]). These approaches focus on proving programs secure against specific vulnerabilities by using techniques such as bounded model checking (e.g. [32]) and fuzzing (e.g. [31]). Our focus, on the other hand, is developing an abstract model specialized for resource partitioning schemes (implemented in either SW/HW) and proving unbounded security through invariants.

Our work is closest to the approaches performing relational symbolic execution/model checking (e.g. [18], [19], [17]). These too verify security-based hyperproperties on the self-composition of the model, by identifying relational invariants. These works consider different models than ours (e.g. [18] uses a simple programming language) or make use of different (often simpler) relational invariants (e.g. [17] considers pure equality-based invariants). Our focus is identifying specialized invariants in the context of resource partitioning-based models (for which pure equality-based invariants are inadequate).

Our contributions are as follows: (1) *Formal model for security-oriented resource partitioning:* We develop a formal model for security-oriented resource partitioning schemes with a corresponding attacker model that captures cache-based timing side channels. (2) *Conditional equality invariants:* We formulate relational invariants that are customized to this model and which enable us to reason about the partitions as well as the policy states. (3) *Evaluation:* We use our approach to model two partitioning schemes, DAWG [11] and COLORIS [13], and demonstrate that inductive proofs using our invariants can have much better performance as compared to bounded (BMC) and unbounded (PDR) techniques.

Outline: In §II we formulate the resource partitioning model and the corresponding threat model. In §III we develop our conditional-equality invariants that support inductive proofs of security. In §IV we discuss our case studies and experimental results, and §V concludes.

II. MODELLING RESOURCE PARTITIONING SCHEMES

A. Resource model

Our model captures an abstract shared *resource* (e.g. cache lines, cache-ways, memory pages) that is being partitioned. For simplicity, we assume that there is only a single resource, but, this can easily be extended to multiple resources.

1) *Resource:* A resource is a collection of *cells*, each cell representing one unit of the resource. In Fig. 1, each line (l_i) is a cell. We assume that cells are indexed by an index set \mathbb{I} , and hold a value from the set \mathbb{V} . The value $\perp \in \mathbb{V}$ represents


```

1 if ( $\exists i. r(i) = \alpha \wedge d = a(i)$ ) {
2   // Is a hit, let i be the hit
   index
3    $p \leftarrow f_{\text{updH}}(a, p, d, i)$ 
4 } else {
5   // Not a hit
6    $i \leftarrow f_{\text{evict}}(a, p, d)$ 
7    $r(i) \leftarrow \alpha$ 
8    $p \leftarrow f_{\text{updM}}(a, p, d, i)$ 
9 }

```

Fig. 2. Semantics of an access $d : \alpha$, from configuration $c = \langle a, r, p \rangle$.

the NULL value. The map r defines the mapping from cell indices to values that they contain, $r : \mathbb{I} \rightarrow \mathbb{V}$.

2) *Domains*: A set of protection domains, denoted by \mathbb{D} , share the resource. In Fig. 1, the victim and attacker processes are domains. Each cell in the resource is allocated to a domain. This allocation is specified by an *allocation map*, identifying the domain that has access to a cell, $a : \mathbb{I} \rightarrow \mathbb{D}$. We denote the set of possible allocations as $A = \mathbb{I} \rightarrow \mathbb{D}$. For allocation a , we denote the set of all cell indices allocated to d by $a \downarrow_d \subseteq \mathbb{I}$.

3) *Policies*: We consider an abstract policy with states P . Each policy state $p \in P$ is an assignment to *policy elements*, $p : E \rightarrow \mathbb{V}_p$. In the Prime+Probe example of Fig. 1, the policy elements E are the bits in the replacement/allocation state (e.g. PLRU tree bits for a Tree-PLRU policy [35]).

Modelling the state as a collection of elements, E , as opposed to monolithically, allows us to develop conditional equality invariants in §III.

We identify three functions defining the policy behaviour:

$$\begin{aligned}
f_{\text{evict}} &: A \times P \times \mathbb{D} \rightarrow \mathbb{I} \\
f_{\text{updM}} &: A \times P \times \mathbb{D} \times \mathbb{I} \rightarrow P \\
f_{\text{updH}} &: A \times P \times \mathbb{D} \times \mathbb{I} \rightarrow P
\end{aligned}$$

The function f_{evict} , given the current allocation, policy state, and the domain performing the access identifies the cell index that is chosen for replacement by the policy. The function f_{updM} identifies the new policy state that results when an access (by a given domain) is a miss, while f_{updH} identifies the state when the access is a hit (the hitting index is a function input).

4) *Overall configuration*: The overall configuration is a tuple $c = \langle a, r, p \rangle$ of these components. We denote the set of configurations as C . Initial configurations are of the form $\langle a, r_{\text{init}}, p_{\text{init}} \rangle$, where a is arbitrary, the resource is empty, $r_{\text{init}} = \lambda i. \perp$, and $p_{\text{init}} \in P$ is the initial partitioning state.

5) *Resource access semantics*: At each step, a domain $d \in \mathbb{D}$ performs an access operation with argument $\alpha \in \mathbb{V}$, denoted as $d : \alpha$ (e.g. in a cache resource, the address is the argument). Fig. 2 provides the semantics of an operation $d : \alpha$, which are determined by the functions $f_{\text{evict}}, f_{\text{updM}}, f_{\text{updH}}$.

In case of a hit (i.e some allocated cell contains the accessed argument), the replacement state is updated according to f_{updH}

(line 3). On a miss, the replaced index i is determined (line 6), following which the cell at i , and the replacement state is updated (lines 7, 8). We denote the transition relation (defined in Fig. 2) as $\delta_{\text{acc}} : C \times \{d : \alpha \mid d \in \mathbb{D}, \alpha \in \mathbb{V}\} \rightarrow C$, which for a previous state and operation $d : \alpha$, gives the resultant state.

6) *Executions*: A trace of the model is a sequence of configurations, $\pi = c_0 \cdot c_1 \cdots$ where c_0 is an initial configuration (§II-A4), and at each step j , some access operation ($d_j : \alpha_j$) is performed: $\delta_{\text{acc}}(c_j, d_j : \alpha_j) = c_{j+1}$. For a trace π , we denote $\text{op}_\pi[j]$ as the operation performed at step j . We denote the set of executions of the model as Π .

B. Attacker Model and Security Property

We consider a timing-based attacker that can observe differences between the hit/miss outcomes of its accesses. Hence the hit/miss outcomes should not depend on the arguments (α s) of accesses by other domains. Otherwise, the attacker could infer these arguments, constituting an information leak. Operation ($d : \alpha$) results in a hit if $\text{isHit}(c, d : \alpha) = \exists i. r(i) = \alpha \wedge a(i) = d$. Configurations c_1, c_2 are (single-access) indistinguishable to the attacker (with domain $d^\#$) if the following holds:

$$\phi_{\text{indist}}(c_1, c_2) \equiv \forall \alpha. \text{isHit}(c_1, d^\# : \alpha) \iff \text{isHit}(c_2, d^\# : \alpha)$$

We formulate security as a non-interference property, where the attacker allocations and accesses are public inputs and the attacker hit/miss outcome is the public output. Non-interference requires that for any two traces, if the public inputs are equal, then so must be the public output. In our setting, two traces π_1, π_2 have the same public inputs (denoted $\pi_1 =_L \pi_2$) if the attacker allocation is identical ($a(\pi_1[0]) \downarrow_{d^\#} = a(\pi_2[0]) \downarrow_{d^\#}$) and attacker accesses are identical:

$$\forall j. \forall \alpha. \text{op}_{\pi_1}[j] = (d^\# : \alpha) \iff \text{op}_{\pi_2}[j] = (d^\# : \alpha)$$

Finally, identical public outputs (hit/miss outcomes) are captured by ϕ_{indist} . Hence, the overall non-interference-based hyperproperty is formulated as:

$$\Phi_{\text{sec}} \equiv \forall \pi_1, \pi_2 \in \Pi. \pi_1 =_L \pi_2 \implies \forall j. \phi_{\text{indist}}(\pi_1[j], \pi_2[j])$$

III. INVARIANTS FOR RESOURCE PARTITIONING SCHEMES

We aim to prove the hyperproperty Φ_{sec} using relational model checking [16] by developing a relational invariant $\phi_{\text{inv}}(c_1, c_2)$ (that relates states from the two traces π_1, π_2).

A. Conditions on ϕ_{inv}

We begin by listing conditions on ϕ_{inv} . As the base case, we get (by $\pi_1 =_L \pi_2$):

$$a_1 \downarrow_{d^\#} = a_2 \downarrow_{d^\#} \implies \phi_{\text{inv}}(\langle a_1, r_{\text{init}}, p_{\text{init}} \rangle, \langle a_2, r_{\text{init}}, p_{\text{init}} \rangle) \quad (\text{base})$$

Next, we want ϕ_{inv} to be inductive, both w.r.t. attacker (Eq. ind-d $^\#$) and non-attacker (Eq. ind-non-d $^\#$) accesses. For the attacker accesses, the access argument should be identical.

$$\begin{aligned} \forall \alpha. \phi_{\text{inv}}(c_1, c_2) &\implies \\ \phi_{\text{inv}}(\delta_{\text{acc}}(c_1, d^\# : \alpha), \delta_{\text{acc}}(c_2, d^\# : \alpha)) &\quad (\text{ind-d}^\#) \end{aligned}$$

$$\begin{aligned} \forall d_1, d_2 \neq d^\#. \forall \alpha_1, \alpha_2. \phi_{\text{inv}}(c_1, c_2) &\implies \\ \phi_{\text{inv}}(\delta_{\text{acc}}(c_1, d_1 : \alpha_1), \delta_{\text{acc}}(c_2, d_2 : \alpha_2)) &\quad (\text{ind-non-d}^\#) \end{aligned}$$

Finally, we want the invariant to imply indistinguishability for an attacker access (ϕ_{indist}):

$$\forall c_1, c_2. \phi_{inv}(c_1, c_2) \implies \phi_{indist}(c_1, c_2) \quad (\text{indist})$$

It is straightforward to see that if some ϕ_{inv} satisfies Eqns. base, ind- $d^\#$, ind-non- $d^\#$, indist, we get a proof of Φ_{sec} .

B. Shape of ϕ_{inv} invariants

In this section, we specialize the form of the invariant ϕ_{inv} considered. We require that ϕ_{inv} enforce (a) that cells allocated to $d^\#$ are identical, and (b) that contents of the $d^\#$ -allocated cells are equal. Formally, we require $\phi_{inv} \supseteq \phi_1$, where,

$$\phi_1 \equiv (a_1 \downarrow_{d^\#} = a_2 \downarrow_{d^\#}) \wedge (\forall i. i \in a_1 \downarrow_{d^\#} \implies r_1(i) = r_2(i))$$

Note that $\phi_1 \implies \phi_{indist}$. While ϕ_1 constrains allocations and their contents such that single-step indistinguishability is guaranteed, different policy states (p_1, p_2) may lead to ϕ_1 not being inductive. Hence ϕ_{inv} additionally needs to relate the policy states from the two traces. However, unlike the resource contents (r_1, r_2), the policy states may not be fully equivalent, and yet the scheme may be secure. Hence the invariant ϕ_{inv} must be more nuanced when relating c_1 with c_2 .

In order to develop more nuanced invariants we make use of the fact that the policy state is composed of elements (§II-A3). We constrain that p_1, p_2 must only agree on some elements from E . The choice of these elements depends on the attacker-allocated indices ($a \downarrow_{d^\#}$) and is identified by a filtering function $filter : 2^{\mathbb{I}} \mapsto 2^E$. The equality of p_1, p_2 is conditioned to only the elements in $filter(a \downarrow_{d^\#})$. The invariant ϕ_{inv} is defined as:

$$\phi_{inv} \equiv \phi_1 \wedge \forall e \in filter(a \downarrow_{d^\#}). p_1(e) = p_2(e)$$

The first term (ϕ_1) enforces equivalence of the $d^\#$ -allocated cells and their contents while the second enforces equality of the $filter$ -identified elements of the policy state.

Importantly, the set of elements in $filter(a \downarrow_{d^\#})$ is not known statically since $a \downarrow_{d^\#}$ can be arbitrary (we want to verify security for arbitrary allocations). Hence conditional equality-based invariants cannot be subsumed by pure equality based relational invariants.

IV. EXPERIMENTAL EVALUATION

We evaluate our modelling and verification approach on two case studies based on previously proposed partitioning schemes: (1) DAWG [11] and (2) COLORIS [13]. We cast both of these into our formal model (§II-A). We then formulate conditional equality-invariants by manually identifying the filter function (§III-B) and then perform verification w.r.t. the property Φ_{sec} (§II-B). We discuss details of modelling and verification in §IV-A and §IV-B respectively. Our experimentation is performed on a server machine running on an Intel i7-13700k processor at 5.2 GHz with 20GB of RAM. Our case study examples including models, invariants, and proof scripts can be found at https://github.com/lichye/sec_resource_partitioning.

TABLE I
DAWG VERIFICATION RUN TIMES

Policy	Verification approach	Runtime
PLRU	k -Ind ($k = 10$)	2.789s
	BMC ($d = 12$)	42.6s
	BMC ($d = 20$)	145m43s
	PDR	24m10s
NRU	k -Ind ($k = 10$)	2.674s
	BMC ($d = 12$)	1m12s
	BMC ($d = 20$)	179m16s
	PDR	Timeout

A. Dynamically Allocated Way Guard (DAWG)

DAWG [11] proposes a technique for secure *way partitioning* of set associative structures, and develops an implementation of a way-partitioned cache. It allows privileged software to allocate *cache-ways* to processes based on resource utilization, and aims to provide isolation between cache-ways allocated to mutually untrusting processes.

1) *Eviction Policy*: We implement both Pseudo-Least Recently Used (PLRU) and Not Recently Used (NRU) eviction policies in DAWG. Similar to standard PLRU or NRU policies, DAWG's PLRU and NRU policies include metadata that is used to determine the victim way and to record the order in which addresses are accessed. PLRU employs a pointer to a metadata tree, while NRU uses access bits. The metadata tree and access bits form the policy elements (E) in our model in the case of PLRU and NRU respectively. In order to ensure isolation, DAWG constrains updates to the metadata and the identification of victim ways to the allocated partitions. Correspondingly, the hardware implementation must ensure that accesses performed by one domain do not modify metadata visible to another domain, as this could alter the hit/miss outcomes in other domains, potentially leading to timing-based information leakage.

2) *Formal Modelling and Verification*: We consider an 8-way DAWG cache design and implement Verilog modules for PLRU and NRU policies. Since DAWG performs way-partitioning, each way is a cell in our model, and hence, $|\mathbb{I}| = 8$ for both cases. For the PLRU policy, the policy elements contain the PLRU-tree bits, $|E_{PLRU}| = 7$ and for NRU they are the access bits $|E_{NRU}| = 8$. We perform a self-composition of this module and formulate Φ_{sec} as a safety property over this self-composition. For this, we use the standard encoding of 2-safety as a safety property [23]. We use the Yosys [36] based SymbiYosys (SBY) [37] model checker with Boolector [38] and ABC [29] as backend solvers to verify this property.

3) *Result and Analysis*: We apply three approaches: k -induction and BMC (using SMTBMC), and PDR (using ABC). For k -induction, we formulate invariants as discussed in §III-B. In Table I, we present the proof runtimes observed for the PLRU and the NRU policies. We observe that both PDR proof runtimes and BMC runtimes (for larger depths) are significantly higher than a k -inductive proof with our invariants. Our approach shows a significant speedup of 15.2x to 4022x in the runtime.

COLORIS [13] performs cache partitioning based on page colouring. In COLORIS, the OS kernel assigns colours to each memory page based on its address bits. Consequently, memory pages with different colours map to different cache sets in a physically indexed cache. By allocating different colours to each process COLORIS aims to improve performance in scenarios where cache contention occurs.

1) *Formal Modelling and Verification*: While COLORIS performs colour-based partitioning, it does allow colours to be shared between processes under certain scenarios. Hence, in full generality, it may allocate non-disjoint partitions to mutually untrusting processes, which puts it at risk of security leaks. However, in our experiments, we assume that the allocated colours are in fact disjoint, making the scheme secure.

We develop a model for a 4-way associative cache with a page allocation scheme, implemented in UCLID5 [39]. UCLID5 allows us to develop a model that partially abstracts the replacement policy by using uninterpreted functions to model policy functions (§II-A3), while constraining the policy to enforce disjointness of allocations. This abstraction allows us to verify an arbitrary policy that enforces disjointness.

2) *Result and Analysis*: We use both BMC and induction-based approaches to verify the model in UCLID5. For the proof by induction, we formulate Eqns. base, ind-d#, ind-non-d#, and indist as separate proofs using UCLID5. The cumulative runtime of the inductive proof (summing individual proof runtimes) is 16m33s. On the other hand, a BMC proof of security does not terminate even for a depth of three.

V. CONCLUSION

In this work, we develop a formal model for resource partitioning schemes and a corresponding attacker model that captures information leakage through timing-based side-channel attacks. We develop conditional equality-based relational invariants that enforce equality of state elements conditioned on some dynamic preconditions, and are more expressive than pure equality-based invariants. These invariants can support inductive proofs of security for resource partitioning schemes against a non-interference-based characterization of the attacker model. We model two partitioning schemes using our approach and demonstrate that conditional equality invariant-based proofs, while requiring manual specification of the invariants, can be much faster than other model-checking approaches. For future work, it would be interesting to develop an algorithm for automated synthesis of these invariants by utilizing their structure. Abstract models and invariants, such as the one we propose, that are specialized to design features, can provide scalable and trustworthy security guarantees.

ACKNOWLEDGEMENTS

This work was supported in part by Intel under the Scalable Assurance program, DARPA contract FA8750-20-C0156 and NSF grant 1837132.

- [1] Paul C. Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Michael Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.
- [2] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul C. Kocher, Daniel Genkin, Yuval Yarom, and Michael Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.
- [3] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-Flight Data Load. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019.
- [4] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 339–354, 2021.
- [5] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
- [6] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
- [7] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, 2016.
- [8] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.
- [9] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Net-Spectre: Read Arbitrary Memory over Network. *ArXiv*, abs/1807.10535, 2019.
- [10] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.
- [11] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, 2018.
- [12] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stappf. Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures. *ArXiv*, abs/2110.08139, 2021.
- [13] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A dynamic cache partitioning system using page coloring. *2014 23rd International Conference on Parallel Architecture and Compilation (PACT)*, pages 381–392, 2014.
- [14] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection. *2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.
- [15] Wenjie Xiong and Jakub Szefer. Leaking Information Through Cache LRU States. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 139–152, 2020.
- [16] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational Verification Using Product Programs. In *World Congress on Formal Methods*, 2011.
- [17] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. Lazy Self-composition for Security Verification. In *International Conference on Computer Aided Verification*, 2018.
- [18] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038, 2019.
- [19] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. Relational Symbolic Execution. *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, 2017.

- [20] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a Doubt: Testing for Divergences between Software Versions. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1181–1192, 2016.
- [21] Joseph A. Goguen and José Meseguer. Unwinding and Inference Control. *1984 IEEE Symposium on Security and Privacy*, pages 75–75, 1984.
- [22] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.
- [23] Tachio Terauchi and Alexander Aiken. Secure Information Flow as a Safety Problem. In *Sensors Applications Symposium*, 2005.
- [24] Gilles Barthe, P. D’Argenio, and Tamara Rezk. Secure information flow by self-composition. *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 100–114, 2004.
- [25] Kenneth L. McMillan. Symbolic model checking. In *International Conference on Computer Aided Verification*, 1993.
- [26] Edmund M. Clarke and David E. Long. Model checking, abstraction, and compositional verification. 1993.
- [27] Per Bjesse and Koen Claessen. SAT-Based Verification without State Space Traversal. In *Formal Methods in Computer-Aided Design*, 2000.
- [28] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design*, 2000.
- [29] Alan Mischenko et al. Berkeley ABC tool. <https://github.com/berkeley-abc/abc>, 2022.
- [30] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark W. Barrett, Subhasish Mitra, and Wolfgang Kunz. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking. *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 994–999, 2018.
- [31] Jaewon Hur, Suhwan Song, Dongup Kwon, Eun-Tae Baek, Jangwoo Kim, and Byoungyoung Lee. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303, 2021.
- [32] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A Formal Approach to Secure Speculation. *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 288–28815, 2019.
- [33] Marco Guarnieri, Boris Köpf, José Francisco Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled Detection of Speculative Information Flows. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2020.
- [34] Musard Balliu, Mads Dam, and Roberto Guanciale. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.
- [35] David A. Patterson and John L. Hennessy. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. 2013.
- [36] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-A Free Verilog Synthesis Suite. <https://github.com/YosysHQ/yosys>, 2013.
- [37] Claire Wolf, et. al. Symbiosys. <https://github.com/YosysHQ/sby>, 2022.
- [38] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2009.
- [39] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis. In *34th International Conference on Computer Aided Verification (CAV)*, volume 13371 of *Lecture Notes in Computer Science*, pages 538–551. Springer, 2022.

Lift-off: Trustworthy ARMv8 semantics from formal specifications

Kait Lam^{} and Nicholas Coughlin^{}

Defence Science and Technology Group, Australia

School of EECS, University of Queensland, Brisbane, Australia

{kait.lam, n.coughlin}@uq.edu.au

Abstract—Disassembly and lifting tools are essential in the verification of software binaries. However, existing tools are difficult to validate and hence not suitable when high levels of assurance are needed. We address this by deriving a trustworthy lifter for ARMv8, named ASLp, based on ARM’s official machine-readable architecture files. ASLp is capable of extracting usable semantics for a large subset of ARMv8, covering almost all integer, control flow, memory and vector instructions.

We demonstrate the utility of ASLp by integrating it with the CMU Binary Analysis Platform. Furthermore, we translate the lifter’s output into LLVM IR and compare the resulting semantics with those from existing lifters Remill and RetDec, leveraging the trustworthiness afforded by our lifter to find a number of major and minor bugs in their outputs.

I. INTRODUCTION

Binary analysis techniques enable program verification over executable machine code, in contrast to reasoning over the abstract programming languages from which it may be derived. These techniques are essential in high assurance domains where software development toolchains represent a liability [1] and may obfuscate hardware behaviours of critical concern [2], [3], [4]. Due to the complexity of binary executables, multiple tools have been developed to provide a foundation for further analysis [5], [6], [7]. These tools *disassemble* binaries, identifying contents such as its machine instructions and static data, and then *decompile* them, lifting the behaviours of machine instructions into machine-generic imperative code. This is advantageous as domain-specific analyses can then be applied to this generic, simplified representation.

To soundly perform this transformation, these foundational tools require trustworthy semantic models of each architecture they aim to support. For disassembly, the architecture’s instruction encoding logic [8] is required to identify instructions, along with some limited understanding of the control flow implications of instructions [7]. To then decompile them, detailed knowledge of the architecture’s state and the effects of instructions on this state is essential [9].

While the information required for disassembly is widely available in the form of reusable decoding libraries [7], the task of specifying detailed instruction semantics for decompilation presents considerable difficulty. Modern architectures support thousands of instructions, with frequent additions to address performance and security issues [10], [11], [12]. Manually encoding each instruction’s behaviour is a time consuming and error-prone task [13]. There are limited alternatives to a

manual approach however, as instruction semantics are generally only specified as informal prose in large instructions set architecture (ISA) manuals [14], [15]. Correctly interpreting all behaviours described in these documents is a difficult task [16], with some projects deferring to incomplete hardware testing to derive semantics instead [17].

An additional concern is the fidelity of these semantic models. Encodings are generally simplified and optimised for a particular application given assumptions over the architecture behaviour, such as ignoring privileged execution modes. While this may benefit the implementation, it limits model reuse between tools due to over-specialisation to an intended purpose [18]. Furthermore, these implicit assumptions may not be clearly documented, potentially invalidating the soundness of any subsequent analysis. Evidently, these issues may negate the high assurance benefits that motivate binary analysis.

Trustworthy architecture models are required in a variety of other domains, such as compiler verification [19], [20], hardware verification [21], [22], and emulation [23]. Given this common motivation, multiple efforts have been made to develop formal architecture models for use across verification projects [24], [25], [18], [26]. While these models have seen use in certain binary analysis applications [27], [28], they have not been broadly used as a semantic underpinning for decompilation tools. This can be attributed to the significant semantic gap between these formal architecture models and the semantic encodings expected by these tools. For instance, formal models may exploit language features such as dependent types, recursive functions and exceptions. Such specification styles are not supported by decompilation tools, which instead encode semantics in simple imperative languages [29], [30].

In this paper, we propose the application of *partial evaluation* to bridge this semantic gap, specialising and translating the formal architecture semantics for each instruction to an encoding suitable for decompilation tools. In Section II, we detail an implementation of partial evaluation for a formal semantics of ARMv8 [25]. Following this, we describe two distinct use cases of the partial evaluator. First, in Section III, we demonstrate the feasibility of its direct use by integrating it into the CMU Binary Analysis Platform [5] to obtain a full binary analysis toolchain. Second, in Section IV, we compare the instruction semantics from two decompilation tools [29], [31] with those derived from the partial evaluator, leveraging existing translation validation techniques to automate the com-

parison [32]. Finally, we explore related work and conclude in Sections V and VI respectively.

II. APPROACH

The foundation of this work is the machine-readable architecture (MRA) published by ARM [33]. This is the formal specification of the ARM architecture, used internally for verification and validation of their hardware. The MRA specification is a comprehensive description of the architecture’s intricacies and behaviours, describing registers, memory behaviour (including faults and translation), interrupts, and the behaviour of exceptions. For our purposes, we are most interested in the opcode decoder and instruction semantics. These are expressed within the MRA using ARM’s architecture specification language (ASL).

ASL [34] is domain-specific imperative language for specification of instruction behaviours and architectural details. Some notable features are its arbitrary-precision integer and real types, dependently-sized bitvector types, pattern matching, and exception handling.

We make use of ASLi, an open-source library for interacting with ASL [35] which provides a lexer, parser, interpreter, and AST transformer. We extend ASLi with a static transformation process to extract instruction semantics and simplify them using *partial evaluation*.

In doing so, we produce the semantics of individual instructions in a proper subset of ASL called *reduced ASL*. Reduced ASL represents instruction semantics with restricted control flow statements and a minimal set of primitive operations based on SMT-LIB’s theory of bitvectors. This allows for easy integration with other tools and straightforward translation into other intermediate languages for binary analysis.

Our extension of ASLi with partial evaluation, which we call *ASLp*, is introduced in Section II-B.

A. Machine-readable architecture example

The MRA specification provides a comprehensive description of the hardware’s behaviours. To detail these behaviours concisely, the specification groups instructions into broad classes based on their function and addressing mode. Specifically, a single **__encoding** in the specification handles several mnemonics, disambiguating them by fields extracted from the opcode. As an example, the encoding in Listing 1 describes the semantics for **add** and **sub** with shifted operands. The pseudocode contains considerable complexity with branches and subroutines to handle flags, different data sizes, and various bitshift options. Even more details are contained within the function calls and overloaded array operations.

In the encoding, **__field** defines slices of the opcode, the **__decode** section extracts information from fields, and the **__execute** block gives the operational semantics of the instruction. When lifting a single opcode, the **__decode** block can be evaluated ahead of time and combined with the **__execute** statements to produce a simplified summary of the instruction’s behaviour. This is explained in the next section, Section II-B.

```
__encoding aarch64_integer_arith_add_sub_shiftedreg
__field Rd 0 +: 5
[...]
__decode
integer d = UInt(Rd); // destination operand
integer n = UInt(Rn); // first operand
integer m = UInt(Rm); // second operand
integer datasize = if sf == '1' then 64 else 32;
boolean sub_op = (op == '1'); // add or sub
boolean setflags = (S == '1'); // set flags?

if shift == '11' then UNDEFINED;
if sf == '0' && imm6[5] == '1' then UNDEFINED;

// logical/arithmetic, left/right shift
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

__execute
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 =
    ShiftReg(m, shift_type, shift_amount);
bits(4) nzcvc;
bit carry_in;

if sub_op then
    operand2 = NOT(operand2);
    carry_in = '1';
else
    carry_in = '0';
(result, nzcvc) = AddWithCarry(operand1,
                                operand2, carry_in);

if setflags then
    PSTATE.[N,Z,C,V] = nzcvc;
X[d] = result;
```

Listing 1: ASL encoding of integer add/subtract with a shifted register operand, from the `arch_instrs.asl` file.

To support a common specification across architecture versions and optional extensions, instruction behaviours are often guarded by *feature flags* denoting differences in their behaviours across possible hardware implementations. This is accomplished with stub functions, i.e. functions declared without implementations, that are later overridden to select the desired behaviour. Some examples of these are:

```
boolean HasArchVersion(ArchVersion version);
boolean HaveEL(bits(2) el); // exception levels
boolean HaveSVE(); // scalable vectors
boolean HaveBTIExt(); // branch target ident.
boolean HaveDITExt(); // data indep. timing
```

The feature flags offer substantial control over the specification, permitting it to be tailored to the underlying hardware as necessary. We leverage this functionality to make our assumptions over the hardware precise and explicit, overriding these feature flags as a form of configuration to our partial evaluation process.

B. Partial evaluation implementation

The key to extracting useful semantics from the MRA specification is *partial evaluation*, a program transformation applicable in contexts where a subset of the program’s inputs are known in advance. These known inputs are propagated throughout the program body, permitting the early evaluation of computations and simplification of language structures


```
X[1] = ZeroExtend(
  (X[2][0 +: 32] + (X[3][0 +: 32] << 4)), 64);
```

Listing 2: Residual program for `add w1, w2, w3, LSL 4`

based on identified constraints. The *residual* program produced by this transform consumes any remaining unknown inputs to generate a result equivalent to that of the original program [36].

For example, Listing 2 is the residual program of `add w1, w2, w3, LSL 4`, from the partial evaluation of Listing 1. Since this specifies various inputs ahead of time (e.g. register usage, bitvector widths and operation mode), the program can be significantly simplified. This process extends to function calls, such as `AddWithCarry()`, which are inlined and simplified down to primitive operations (e.g. `ZeroExtend()`, `<<` and `+`). The final residual program succinctly represents the instruction’s effects in terms of unknown inputs—here, values held in the register array `x`.

We implement our ASL partial evaluator (ASLP)¹ by augmenting the existing ASLi [35] to perform *online* partial evaluation [37]. This approach preserves the structure of the existing interpreter but extends it to consider a *symbolic* state, in which variables map to one of the following:

- 1) Known v : A known concrete value v
- 2) Expr e : The result of a simplified pure expression e
- 3) Unknown: An unknown value

Known encodes inputs and intermediary calculations that are known ahead of time, with Unknown encoding the inverse. Expr encodes intermediary calculations over Unknown variables and is used to identify simplifications.

Our partial evaluator `aslp(prog, sym)` produces a residual program for the program `prog` given the symbolic state `sym`. We define correctness of `aslp` in terms of the existing interpreter, `eval`, such that the residual program will produce an equivalent final state as the original program given agreement between `sym` and the initial concrete state, `st`:

$$\begin{aligned} &\forall prog, st, sym. \\ &(\forall x, e. sym(x) = Expr\ e \implies st(x) = eval(e, st)) \wedge \\ &(\forall x, v. sym(x) = Known\ v \implies st(x) = v) \implies \\ &eval(aslp(prog, sym), st) = eval(prog, st) \end{aligned}$$

where `eval(prog, st)` returns the final state for program `prog` and `eval(e, st)` returns the value of expression e .

The partial evaluator maintains the symbolic state through a forward traversal of the program, evaluating language structures where possible and building the residual program otherwise. We list some of the applied partial evaluation techniques:

1) *Expression Simplification*: Rewrite rules are applied during the construction of Expr e terms. These transforms are critical to matching the simplicity of existing lifter outputs as, without them, the abstract nature of ASL introduces redundant operations. For instance, bitvector calculations may include redundant slicing, concatenation and extension operations due to the use of shared code paths. These are aggressively

rewritten, generally by distributing slicing operations over sub-expressions to identify additional simplification opportunities.

2) *Aggregate Values*: ASL enables various aggregate values, such as tuples, records and arrays. We unpack these structures into their individual components and transform operations over them accordingly. As an example, consider ASL’s syntax for a destructuring assignment to multiple fields of a record in `PSTATE`. `[N, Z, C, V] = nzcv`. In this operation, the 4 bits of the bitvector `nzcv` are extracted into the corresponding fields (N,Z,C,V) of the `PSTATE` record. The partial evaluation process lowers this operation into four assignments to the individual fields of the appropriate slice of `nzcv`.

3) *Function Inlining*: ASL supports functions to implement common functionality. We inline all calls to ensure we emit a single code sequence for an instruction, excluding those to a configurable set of primitive functions. Inlining is implemented by introducing a fresh variable to represent the function result and stitching the callee and caller bodies together appropriately. This stitching process is complicated by the limited control flow expressible in ASL, demanding transforms detailed later in Item II-B5.

The primitive function set enables the abstraction of complex processor features. For instance, memory accesses in the MRA specification are complex, including details such as virtual address translation. As these complexities are generally ignored during binary analysis, the inlining process is configured to treat various memory operations as primitives. These function calls are later translated into corresponding primitives in binary analysis tools, as detailed in Sections III and IV. A similar technique is applied to model floating-point operations.

4) *Iteration*: Loops are widely used in instruction specifications, notably to encode operations over vector elements. The bounds on these loops are generally known during partial evaluation, permitting their elimination via unrolling. While this increases program size, it significantly simplifies subsequent analysis. When loop bounds are unknown, all iterating language structures are lowered into `while` loops and emitted into the residual program.

5) *Branching Control Flow*: ASL supports various branching control flow structures, such as `if` statements, ternary operators and pattern matching. Often, branch conditions can be resolved during partial evaluation, eliminating the branch. If not, these structures are lowered into `if` statements and all possible branches are explored, merging their symbolic states when control flow eventually rejoins.

As the state merge process can cause loss of analysis precision, we may defer the control flow join by duplicating the statements appearing after an `if` statement into both branches. Moreover, this transform is necessary when inlining a function with a `return` statement within a branch, as a means to represent the execution of the inlined function’s body after the conditional `return`.

Note that this transform may result in an exponential increase in code size, given a sequence of branches. Consequently, it is not applied to branching structures derived from an unrolled loop, as seen in various vector instructions

¹ Available at: <https://github.com/UQ-PAC/aslp>

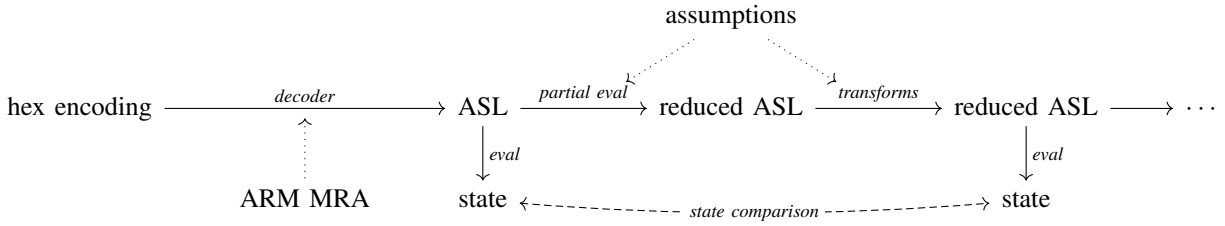


Fig. 1. Overview of ASLp’s partial evaluation pipeline. Dotted lines indicate some external input and dashed lines represent state comparison.

with conditional behaviours per vector element. It is applied in a majority of cases however, as residual programs for instructions generally contain at most a single branch.

C. Transforms

After partial evaluation, transformations are applied to the output to further simplify its representation. This includes dead code elimination and common sub-expression elimination to remove unnecessary and duplicated calculations introduced during partial evaluation.

Furthermore, the MRA specification makes extensive use of arbitrary-precision integers, a feature rarely considered in binary analysis frameworks. Consequently, we convert these operations into bounded equivalents to successfully integrate residual programs into these frameworks.

This is possible as the processor state is specified using types of a bounded size, e.g. a 64-bit register’s value is known to fit within 64 bits. Therefore, while instructions may be specified with arbitrary-precision operations, only some bounded component of their effects will ultimately interact with the processor state. We determine these bounds using a simple interval analysis over the residual program. Given these intervals, we transform all integer variables and operations into corresponding bitvector equivalents of the required size.

D. Testing

To validate the partial evaluator, we implement differential testing [38] between the original ASL specification and its reduced form (denoted by the “state comparison” dashed arrow in Figure 1). Although this is not a formal proof of correctness, it increases our confidence in the validity of our implementation.

We perform this testing on a subset of the ARMv8 instructions, chosen to be a representative sample of application-level opcodes with predictable behaviours. Specifically, we include:

- integer and arithmetic instructions,
- floating-point instructions,
- branch instructions,
- vector instructions,
- memory load/store instructions, and
- atomic instructions (sequential semantics only).

To implement differential testing, we leverage the existing ASL interpreter to evaluate both the original specification and the partial evaluator’s output for a particular instruction

encoding. We then compare the final states for the two executions, with a mismatch indicating a potential bug in the partial evaluator. For each ARMv8 mnemonic, we test various combinations of register operands and flag values with a randomly initialised machine state for the interpreter.

The partial evaluator passes these tests for almost all instruction families, with errors limited to uncommon ASL features which we currently do not support. Notably, multiple bugs in the existing ASL interpreter were identified during this testing, which we detail in Appendix A.

III. BINARY ANALYSIS TOOLCHAIN

The CMU Binary Analysis Platform (BAP) is a toolkit and library supporting the analysis of binary programs. Given a binary file, it provides disassembly, control flow reconstruction, and instruction semantics in its BAP intermediate language (BIL). Moreover, it is built with a modular plugin system, enabling the development of additional analyses and lifters.

We develop a plugin to integrate our ARMv8 semantics into BAP by translating ASLp’s reduced ASL into BIL and interfacing with BAP’s knowledge base. With this, we are able to leverage the existing BAP machinery and combine it with our derived semantics, easily obtaining a full binary analysis toolchain.

To demonstrate the viability of the resulting lifter, we compare its output with that of an existing ARMv8 plugin² across a series of programs, summarised in Table I. The ASLp variant of BAP successfully lifts a superset of instructions relative to those supported by the existing plugin, capturing additional memory and vector operations. Notably, the ASLp variant successfully lifts all instructions in the example binaries except for 2508 instructions implementing floating-point operations, for which the conversion to BAP’s semantics is unclear.

To evaluate the complexity of the produced programs, we compare the line counts of their outputs and average line length. The ASLp variant consistently produced a shorter representation with an average line length of 29.45 characters, comparable to the 29.47 characters of the existing implementation. A manual inspection of the results attributes these differences to alternative representations of flag calculations and branching conditions. Moreover, multiple semantic errors in the existing lifter were identified, such as incorrect operation

²We use the following, as it is the most comprehensive ARMv8 plugin to our knowledge: <https://github.com/BinaryAnalysisPlatform/bap/pull/1546>

TABLE I
BAP COMPARISON

Program	Lifted Instructions		Time (s)		Size (lines)	
	BAP	ASLp	BAP	ASLp	BAP	ASLp
bzip2	25254	25275	8.55	10.60	37658	35727
cntlm	15899	15908	7.59	8.64	34691	33064
gcc	152036	153673	63.88	77.43	391241	375623
gzip	17501	17554	6.82	8.00	35283	33682
oggenc	56227	56817	18.70	24.67	106448	101464

widths and memory address calculations. We do not consider a detailed semantic comparison of the two outputs, instead focusing such efforts on other lifters in Section IV.

The ASLp variant introduces additional overhead, increasing lifting time by roughly 20%. This is unsurprising, due to the additional analysis required to reduce and translate the ASL specification into BAP’s representation. We believe this is an acceptable trade-off, as the ASL implementation provides greater coverage of the ARMv8 architecture with a stronger argument for correctness. Moreover, these benefits extend to any other architectures specified in ASL, avoiding the substantial effort needed to manually encode such semantics in BAP’s existing infrastructure.

IV. SEMANTIC COMPARISON OF LIFTERS

Ours is not the first project to provide semantics for the ARMv8 architecture. Although we have trustworthy semantics from the architecture model, there are many existing lifters in active use with their own instruction semantics. Instead of replacing these, we can validate their semantics by comparing them with the ASL lifter. In this way, we can gain a level of confidence in their instruction semantics over and above fuzz testing or hardware testing.

The semantic comparison is done using the translation validation tool from Alive2 [32] which verifies that a given LLVM IR program refines a source program—that is, the target program’s behaviours are a subset of the source program’s behaviours. The tool was developed to verify compiler and optimisation passes, but here it is used to test for equivalence of the semantics from different lifters. Alive2 supports this by performing its refinement checks bidirectionally.

To test the output of ASLp, which emits reduced ASL, we developed a translator³ from reduced ASL to LLVM IR and compare its result with other lifters that produce LLVM IR. We choose RetDec and Remill to demonstrate this process.

Although these lifters share a common output language, the representation of registers, memory, and other hardware-level state differs in each. To compare them, we translate the lifter outputs into a unified “dialect” of LLVM IR. This dialect needs to be simple to aid Alive2’s reasoning while capturing enough of the machine state to faithfully represent the semantics we wish to compare.

We design this unified state representation as follows. Registers are modelled as global integer variables of various sizes:

- 31 64-bit general purpose registers, `x0` to `x30`,
- 32 128-bit vector registers, `v0` to `v31`,

³Available at: <https://github.com/UQ-PAC/llvm-translator>

```
declare noundef i8 @load_8(i64 noundef)
    inaccessiblememoryonly nounwind willreturn
    readonly
```

```
declare noundef void @store_8(i64 noundef, i8)
    inaccessiblememoryonly nounwind willreturn
```

Listing 3: Memory load/store functions. 8, 16, 32, and 64-bit versions are defined.

- 4 1-bit flag registers, `nf`, `zf`, `cf`, and `vf`,
- a 64-bit `pc` register, and a 64-bit `sp` register.

Representing memory requires more careful consideration, since many instructions can load/store memory at arbitrary addresses. In LLVM, this is conventionally done with a `inttoptr` (“integer to pointer”) cast, but Alive2 cannot reason about these operations. Instead, we approximate these by modelling memory as uninterpreted functions, as seen in Listing 3. In order for two programs to verify as equivalent, the source and target must have identical calls in the same order. This is an overapproximation of memory behaviours but is sufficient for verifying a single instruction’s semantics.

To reduce the overapproximation, LLVM attribute tags are used on each declarations to constrain the effects of these intrinsics on memory and global variables.

The `inaccessiblememoryonly` tag indicates the functions only read from or write to memory not visible to the caller. This is well-suited to representing the memory and its effects (virtual address translation, alignment, etc.) and indicates that the loads/stores are independent of register values but interacts with other loads and stores. Load functions are additionally tagged as `readonly`, indicating that the inaccessible memory is not modified.

The `willreturn` tag indicates that these functions will terminate (i.e. not loop forever). Combined with `nounwind`, it means the function will terminate without raising an exception (i.e. without jumping back up the call stack).

As a consequence of using Alive2 for validation, the semantic comparison in this work is done with respect to the formal semantics of LLVM IR [39]. However, LLVM IR is designed for use within optimising compilers as a compilation target of higher-level languages. As such, it is intentionally under-specified; some details are left as undefined behaviour (UB) to allow for different implementations. Moreover, it introduces “undefined” and “poison” values into each type so compilers may exploit particular instances of undefined behaviour for optimisation. These features are useful in compilation but less suitable for our purposes.

Formal semantics of instruction-internal behaviours, including the ARMv8 model we consider, should be precise and not exhibit any undefined behaviour. Undefined and poison values do not occur naturally within the architecture specification language. To handle these, we annotate many LLVM operations as `noundef` and `nonnull` to assert values are never undefined/poison or null. These will invoke undefined behaviour when their assumption is violated, which is acceptable since two programs will verify as long as the UB occurs in the same way in both cases.

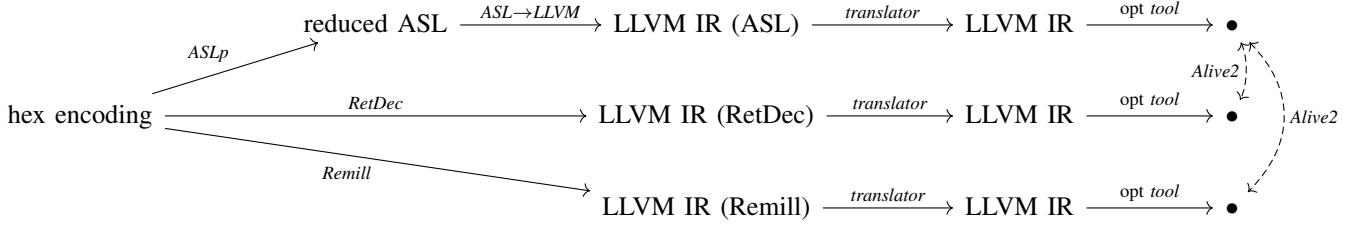


Fig. 2. Overview diagram of the lifter evaluation process. ASLp indicates the pipeline of Figure 1. Dashed lines indicate semantic comparison, and filled dots are the same type as their predecessor.

TABLE II
EVALUATION RESULTS

	Count	RetDec Equivalent	Mismatch	Timeout	Unsupported	Remill Equivalent	Mismatch	Timeout	Unsupported
Branch	162	74	9	10	69	120	1	37	4
Integer	14442	10147	608	415	3272	12058	15	1238	1131
Memory	6414	3864	618	0	1932	5057	88	0	1269
Vector binary	19170	264	426	0	18480	2997	0	0	16173
Vector unary	1098	9	153	0	936	333	0	0	765
Total	41286	14358	1814	425	24689	20565	104	1275	19342

With the comparison framework set up as above, evaluation of the lifters was conducted for each opcode, as outlined in Figure 2. For each opcode of interest, we perform the following:

- 1) Separately disassemble the instruction with each lifter. The ASL lifter produces a reduced ASL program which is translated to LLVM IR by a pattern-matching translator.
- 2) The LLVM IR from each lifter is transformed into the unified state and memory representation described above.
- 3) The LLVM optimiser `opt` is run on each output to simplify the resulting structures and allow for easier comparisons.
- 4) For each lifter under test, `alive-tv` checks for equivalence between its output and ASLp’s LLVM IR.
- 5) If Alive2 can prove the equivalence, it reports the two outputs are equivalent. Otherwise it reports a mismatch or time out. A mismatch may be a difference in memory states or undefined behaviour. Where lifters do not match, Alive2 gives input values which cause the mismatch and we investigate the discrepancies by manually comparing the LLVM IR with the ARM ISA.

We used a subset of the integer, logical, branching and vector instructions tested in Section II-D to evaluate the three lifters. The results are summarised in Table II, organised by classes of instructions. Both RetDec and Remill were compared with the ASL lifter with a timeout of 20 seconds. The “Equivalent” column indicates both are semantically equivalent, and “Mismatch” indicates a difference in memory state or undefined behaviours. The “Unsupported” instructions are those supported by the ASL lifter but not the lifter under test.

A. RetDec

RetDec [31], developed by Avast Software, is a retargetable decompiler with plugins for IDA Pro and radare2. Here, we analyse its Capstone2LLVMIR component which provides its instruction semantics in LLVM IR. The produced IR is similar to our unified representation: registers are mutable global variables, memory operations used `inttopttr` instructions, and intrinsic functions handle program flow and branching. Since the goal of Capstone2LLVMIR is to lift to higher-level C/C++, it “does not aim to fully translate (give meaning/semantics to) all assembly instructions” [31].

Despite this, RetDec lifted a large fraction of the instructions tested. However, it had some inaccuracies in key implementation details and shortcomings with vector instructions. These are explained in more detail below.

Overflow flag computation: The overflow flag (`vf`) computation is incorrect for instructions which incorporate the carry flag (`cf`), e.g., `adcs`, `sbc`s. In these cases, `vf` is set when adding the carry *twice* would result in an overflow, when it should only be considered once. For example, this occurs with `adcs xzr, x0, xzr` (bytecode `1f001fba`) when `cf` is set and `x0` is $2^{63} - 2$. This computation would not overflow, but RetDec’s semantics indicate `vf` would be set. This error affects approximately 240 integer opcodes tested.

lshl/lshr/ashr poison: The LLVM instructions `shl`, `lshr` and `ashr` for bitvector shifts are defined to return a poison value when the shift amount is equal to or greater than the register size. However, ARMv8 shift instructions, such as `lsl`, `lsr`, `asr` and `ror`, are well defined in such scenarios, shifting by the desired amount modulo the size. RetDec ignores this difference, naively converting between the two.

For example, RetDec lifts `lsr x1, x1, x0` (2124c09a) to an LLVM snippet containing `lshr i64, %X1, %X0`. This

operation will return poison when `%x0` exceeds 63, where the ARMv8 specification would return 0. These inaccuracies make up 72 mismatches of integer opcodes.

Moreover, RetDec generates invalid shifts in various other cases, such as `extr` mnemonics and instructions with register operand rotations. These cases contain shifts that will always return poison, e.g., `shl i32 %3, 32`. This affects 162 opcodes, such as `and x0, x0, xzr, ror #0` (0000df8a).

clz poison: `clz` counts the number of zero bits before the first one bit in a bitvector, starting at the most significant bit. RetDec uses LLVM’s `@llvm.ctlz.*` intrinsic functions to implement this behaviour. However, these calls are configured such that they will return poison when the bitvector is zero, instead of returning its width. For example, when `x0` is zero `clz x0, x0` (0010c0da) should set `x0` to 64, but the RetDec result will produce poison.

uxtw/sxtw truncating to 32-bit: Various ARMv8 mnemonics accept register extension modes to specify how registers of different widths should be extended prior to applying an operation. For instance, `uxtw` specifies a zero-extension from 32-bit to 64-bit and `sxtw` specifies a signed-extension from 32-bit to 64-bit. Due to encoding quirks, it is possible to encode various modes that are effectively no-ops, such as `uxtx` for a zero-extension from 64-bit to 64-bit. However, RetDec lifts such cases incorrectly, truncating 64-bit registers down to 32-bit and then extending back to the original size. This error affects 220 opcodes, such as `add x0, x1, x0, uxtx` (2060208b).

Shifted uxtw/sxtw truncation: ARMv8 allows for the specification of shifted 32-bit offsets in memory address calculations. For instance, `str w0, [x0, w0, uxtw #2]` (005820b8), will perform a store to `x0 + (ZeroExtend(w0, 64) << 2)`, where `w0` is a 32-bit register. However, RetDec lifts these address calculations such that the shift is applied before the appropriate sign- or zero-extension. This results in the loss of `w0`’s upper two bits and, subsequently, an incorrect address calculation. This affects 360 opcodes with `sxtw` operands, and 126 with `uxtw`.

sxtw extension: When specifying a memory access with a 32-bit offset, ARMv8 permits the application of either a zero- or a sign-extension to pad the value to 64-bit. RetDec always produces a zero-extension however, leading to incorrect addresses for `sxtw`. For example, `str w0, [x1, w0, sxtw]` (20c820b8) exhibits this behaviour. This error affects the same 360 `sxtw` opcodes as above.

udiv/sdiv by zero: LLVM’s `udiv` and `sdiv` integer division instructions trigger undefined behaviour when the denominator is zero, instead of a zero result as defined in the ARMv8 specification. RetDec fails to account for this mismatch, for example, lifting `udiv x0, x0, x0` (0008c09a) to an LLVM snippet that triggers undefined behaviour when `x0` is zero.

Pre-increment address: Load/store pair instructions load or store two words at adjacent locations in memory, given the address of the first word and an increment. With pre-increment addressing, the address register should be incremented by the given offset prior to the memory access. However, the increment is added to the address of the second word in-

stead of the first, leading to incorrect values in the updated address register. For example, `stp xzr, xzr, [x0, #16]!` (1f7c81a9) increments `x0` by 24 instead of 16. This affects 102 variants of `stp`. Other paired memory operations appear to lift correctly.

SIMD instructions: For the majority of vector instructions, RetDec returns incorrect semantics; it treats the operands as ordinary registers and does not vectorise operations. For example, `add v0.8b, v1.8b, v1.8` (2084210e) produces:

```
%0 = load i128, i128* @v1
%1 = load i128, i128* @v1
%2 = add i128 %0, %1
store i128 %2, i128* @v0
```

whereas the correct operation would consider `v1` as 8 separate bytes, adding each byte elementwise. The same deficiency affects all vector instructions, leading to mismatches for all such opcodes. There are 579 such opcodes across the binary and unary vector instructions. The few ‘equivalent’ results for vector instructions occur when this discrepancy does not affect correctness, e.g. bitwise logical operations and subtraction of a register from itself.

B. Remill

Remill [29], developed by Trail of Bits, is a library providing LLVM IR instruction semantics for various architectures. As it “focuses on accurately lifting instructions” [29], it has seen wide adoption both individually and as part of the McSema binary decompiler [40]. Remill was found to have much fewer discrepancies than RetDec when compared with the ASL semantics.

sdiv overflow: When performing signed division over n -bit two’s complement integers, the calculation $(-2^{n-1})/(-1)$ will overflow, as 2^{n-1} is not representable in n -bit two’s complement. Under ARMv8, this is defined as returning -2^{n-1} , i.e. the truncation of 2^{n-1} . However, Remill lifts this operation to LLVM’s `sdiv` instruction, which treats an overflow as undefined behaviour. For example, this error manifests in `sdiv x1, x0, x1` (010cc19a).

ldp/stpb with writeback: ARMv8’s memory addressing modes support incrementing an address register before or after memory is accessed, in a process called *writeback*. When the data and address registers of these operations overlap, the implementation is permitted to select one of several acceptable behaviours. For instance, loads with overlapping registers may: skip the writeback operation, writeback an unknown value, treat the operation as a no-op or consider the instruction undefined. For stores with overlaps, it may: store the original value of the overlapping register (before writeback), store an unknown value, or act as a no-op or undefined opcode.

As an example, `ldp x1, x0, [x1], #-8` (2180ffa8) accesses a pair of words at memory address `x1` then decrements `x1` by 8. However, this overlaps with the use of `x1` as a data register, holding the first loaded value. The ASL lifter is explicitly configured to skip the writeback, i.e., the decrement, keeping the loaded value. Remill does the opposite, overwriting the loaded value with the decremented

address. The correct outcome here is ultimately dependent on the hardware implementation, potentially leading to either interpretation being correct. However, the ASLp approach makes such configuration explicit, whereas the behaviour is silently assumed by Remill.

`strb w0, [x0], #-1 (00f41f38)` is a more concerning example, featuring an overlapping post-indexed store. The ASL lifter stores the original value of `w0` to the memory address `x0`, and then decrements `x0` by 1. Remill, however, performs the store and skips the writeback operation. According to the specification, this is not an allowed behaviour, indicating an invalid lifting.

These inconsistencies affect overlapping `ldp`, `ldpsw`, `strb`, and `strh` opcodes. These make up the 88 mismatching memory instructions. Other variants, notably `ldr` and `str`, did not have these discrepancies.

smaddl: When lifting fused multiply-add instructions, Remill includes “no signed wrap” annotations on the corresponding LLVM operations, returning poison in the case of signed overflow. However, this is unnecessary, as these ARMv8 instructions are specified to implement standard truncation behaviour in such scenarios. This accounts for all 15 of Remill’s mismatched integer instructions.

br xzr: The `e0031fd6` opcode should disassemble to `br xzr`, semantically a jump to address 0. However, the semantics produced by Remill jump to the value held in `sp` instead. This is the only mismatch on branch instructions.

C. Bugs found

We reported the above inconsistencies to the relevant projects. Additionally, we identified bugs in Alive2:

- There was a soundness issue caused by an incorrect peephole optimisation in an integer comparison operation.
- Type punning in LLVM (loading from a pointer that stores a different type) is defined to return poison, but this was not implemented by Alive’s semantics due to an incorrect optimisation.

These Alive2 bugs were reported and fixed by its maintainers during the course of our work.

V. RELATED WORK

The field of binary decompilation and analysis is vast and represents decades of ongoing research. Similar to this work, standalone hardware architecture models have been developed for use in decompilation tools. Notable examples of this include Remill and GHIDRA’s sleigh library [30]. While these libraries are immediately applicable to many decompilation tasks and support multiple ISAs, they lack a formal foundation. For instance, the formal semantics of their output language is unclear [41], [42] and the derivation of their models is rarely documented.

Various approaches to decompilation have been proposed that build on trustworthy architecture models. For instance, *Decompilation into Logic* [27] leverages architecture encodings specified within the HOL4 theorem prover [43]. These

specifications are simplified, using the theorem prover’s rewriting engines, to derive concise semantics for individual instructions. Similar techniques have been applied in other theorem provers [44], [45], [28], potentially leveraging symbolic execution to further improve the rewriting process [46]. The produced representations are suitable for reasoning within theorem provers, but cannot be easily integrated into decompilation tools, due to the use of abstract logic constructs in their results. An exception to this is HolBA [47], which successfully converts these logic constructs into an imperative language, at the expense of significantly slower lifting times.

Existing work has explored the application of partial evaluation to decompilation. For instance, Gómez-Zamalloa et al. [48] develop a lifter from Java bytecode to Prolog via the partial evaluation of an interpreter for the former written in the latter. While their approach features similar implementation details to ours, such as careful inlining configuration, they consider a wholistic perspective, partially evaluating whole programs with the intention to directly validate the residual Prolog representation.

In recent years, attention has turned to the validation of decompilers. The concept of differential testing for lifted IRs was first explored by Kim et al. [13] with the MeanDiff tool. In this work, three unproven x86 lifters (PyVEX [49], BAP [5], and BINSEC [50]) were compared to each other using an equivalent approach to Section IV. While similar to our work, we benefit from a trustworthy ISA semantics for ARMv8, providing a higher level of assurance and greater instruction coverage.

Dasgupta et al. [26] apply a similar technique to validate instruction semantics derived from Remill with respect to their own model of x86-64, written in K [51]. The validated Remill instruction semantics are subsequently concatenated to decompile the entire program. It is unclear whether instruction semantics could be directly derived from their trusted model, as done in Section III.

VI. CONCLUSION

This work documents the wide-reaching benefits and applications of a canonical, accurate, and comprehensive ISA specification. With architecture specifications provided by ARM, we can provide a solid, trustworthy foundation for binary disassembly and lifting. Partial evaluation allows us to summarise these semantics into a minimal IR, for easy integration with other analysis tools, with our BAP ARMv8 plugin as an example. Moreover, we demonstrate that the produced representation is no more complex than that of existing, manually encoded lifters. We also show the effectiveness of formal semantics for validating the results of established binary lifters using existing LLVM analysis and reasoning tools. The amount of errors we found, in some cases central to whole families of opcodes, demonstrates the importance of a reference semantics that can be instrumented for automated checkers. Altogether, this leads to more trustworthy binary lifting with promising future applications, providing trustworthiness in a field which demands high levels of assurance.

Acknowledgements: We would like to thank James Paterson and Andrew Brown, who both contributed to the implementation of this work.

REFERENCES

- [1] K. Thompson, “Reflections on trusting trust,” *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984. [Online]. Available: <https://doi.org/10.1145/358198.358210>
- [2] V. D’Silva, M. Payer, and D. X. Song, “The correctness-security gap in compiler optimization,” in *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE Computer Society, 2015, pp. 73–87. [Online]. Available: <https://doi.org/10.1109/SPW.2015.33>
- [3] G. Barthe, S. Blazy, R. Hutin, and D. Pichardie, “Secure compilation of constant-resource programs,” in *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 2021, pp. 1–12. [Online]. Available: <https://doi.org/10.1109/CSF51468.2021.00020>
- [4] G. Balakrishnan and T. W. Reps, “WYSINWYX: What You See Is Not What You eXecute,” *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, 2010. [Online]. Available: <https://doi.org/10.1145/1749608.1749612>
- [5] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “BAP: A binary analysis platform,” in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469.
- [6] F. Wang and Y. Shoshitaishvili, “Angr - the next generation of binary analysis,” in *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*. IEEE Computer Society, 2017, pp. 8–9. [Online]. Available: <https://doi.org/10.1109/SecDev.2017.14>
- [7] A. Flores-Montoya and E. M. Schulte, “Datalog disassembly,” in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1075–1092. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya>
- [8] N. Ramsey and M. F. Fernandez, “Specifying representations of machine instructions,” *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 3, pp. 492–524, 1997. [Online]. Available: <https://doi.org/10.1145/256167.256225>
- [9] C. Cifuentes and S. Sendall, “Specifying the semantics of machine instructions,” in *6th International Workshop on Program Comprehension (IWPC ’98), June 24-26, 1998, Ischia, Italy*. IEEE Computer Society, 1998, pp. 126–133. [Online]. Available: <https://doi.org/10.1109/WPC.1998.693332>
- [10] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “PACMAN: attacking ARM pointer authentication with speculative execution,” in *ISCA ’22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 685–698. [Online]. Available: <https://doi.org/10.1145/3470496.3527429>
- [11] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, “CHERI: A hybrid capability-system architecture for scalable software compartmentalization,” in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37. [Online]. Available: <https://doi.org/10.1109/SP.2015.9>
- [12] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker, “The ARM scalable vector extension,” *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. [Online]. Available: <https://doi.org/10.1109/MM.2017.35>
- [13] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, “Testing intermediate representations for binary analysis,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 353–364.
- [14] ARM, “ARM Architecture Reference Manual for A-profile architecture,” 2023.
- [15] Intel Corporation, “Intel A64 and IA-32 Architectures Software Developer’s manual,” 2023.
- [16] A. Reid, “Who guards the guards? Formal validation of the Arm v8-M architecture specification,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 88:1–88:24, 2017. [Online]. Available: <https://doi.org/10.1145/3133912>
- [17] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, “Stratified synthesis: automatically learning the x86-64 instruction set,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krantz and E. D. Berger, Eds. ACM, 2016, pp. 237–250. [Online]. Available: <https://doi.org/10.1145/2908080.2908121>
- [18] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, “ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: <https://doi.org/10.1145/3290384>
- [19] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, “CakeML: a verified implementation of ML,” in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 179–192. [Online]. Available: <https://doi.org/10.1145/2535838.2535841>
- [20] X. Leroy, “A formally verified compiler back-end,” *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, 2009. [Online]. Available: <https://doi.org/10.1007/s10817-009-9155-4>
- [21] K. Nienhuis, A. Joannou, T. Bauereiss, A. C. J. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell, “Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process,” in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1003–1020. [Online]. Available: <https://doi.org/10.1109/SP40000.2020.00055>
- [22] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell, “Verified security for the Morello capability-enhanced prototype Arm architecture,” in *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, ser. Lecture Notes in Computer Science, I. Sergey, Ed., vol. 13240. Springer, 2022, pp. 174–203. [Online]. Available: https://doi.org/10.1007/978-3-030-99336-8_7
- [23] D. Lockhart, B. Ilbeyi, and C. Batten, “Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers,” in *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*. IEEE Computer Society, 2015, pp. 256–267. [Online]. Available: <https://doi.org/10.1109/ISPASS.2015.7095811>
- [24] A. C. J. Fox, “Formal specification and verification of ARM6,” in *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, ser. Lecture Notes in Computer Science, D. A. Basin and B. Wolff, Eds., vol. 2758. Springer, 2003, pp. 25–40. [Online]. Available: https://doi.org/10.1007/10930755_2
- [25] A. Reid, “Trustworthy specifications of ARM® v8-A and v8-M system level architecture,” in *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD ’16. Austin, Texas: FMCAD Inc, 2016, p. 161–168.
- [26] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, “A complete formal semantics of x86-64 user-level instruction set architecture,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1133–1148. [Online]. Available: <https://doi.org/10.1145/3314221.3314601>
- [27] M. O. Myreen, M. J. C. Gordon, and K. Slind, “Decompilation into logic - Improved,” in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, G. Cabodi and S. Singh, Eds. IEEE, 2012, pp. 78–81. [Online]. Available: <https://ieeexplore.ieee.org/document/6462558/>
- [28] M. Sammler, A. Hammond, R. Lepigre, B. Campbell, J. Pichon-Pharabod, D. Dreyer, D. Garg, and P. Sewell, “Islaris: Verification of machine code against authoritative ISA semantics,” in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming*

Language Design and Implementation, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 825–840. [Online]. Available: <https://doi.org/10.1145/3519939.3523434>

[29] Trail of Bits, “lifting-bits/remill: Library for lifting machine code to LLVM bitcode,” <https://github.com/lifting-bits/remill>, 2022.

[30] National Security Agency, “Sleigh,” <https://github.com/NationalSecurityAgency/ghidra>, 2022.

[31] Avast Software, “avast/retdec: RetDec is a retargetable machine-code decompiler based on LLVM,” <https://github.com/avast/retdec>, 2022.

[32] N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr, “Alive2: bounded translation validation for LLVM,” in *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 65–79. [Online]. Available: <https://doi.org/10.1145/3453483.3454030>

[33] ARM, “ARM developer exploration tools,” 2023.

[34] A. Reid, “ARM’s architecture specification language,” Aug 2016. [Online]. Available: https://alastairreid.github.io/specification_languages/

[35] —, “Using ASLi with Arm’s V8.6-A ISA specification,” Jan 2020. [Online]. Available: <https://alastairreid.github.io/using-asli/>

[36] Y. Futamura, “Partial computation of programs,” in *RIMS Symposia on Software Science and Engineering*. Springer Berlin Heidelberg, 1983, pp. 1–35. [Online]. Available: https://doi.org/10.1007/3-540-11980-9_13

[37] E. Sumii and N. Kobayashi, “A hybrid approach to online and offline partial evaluation,” *High. Order Symb. Comput.*, vol. 14, no. 2-3, pp. 101–142, 2001. [Online]. Available: <https://doi.org/10.1023/A:1012984529382>

[38] W. M. McKeeman, “Differential testing for software,” *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: <https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf>

[39] LLVM Project, “LLVM language reference manual,” <https://llvm.org/docs/LangRef.html>, 2022.

[40] Trail of Bits, “lifting-bits/mcsema: Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode,” <https://github.com/lifting-bits/mcsema>, 2022.

[41] N. Naus, F. Verbeek, D. Walker, and B. Ravindran, “A formal semantics for P-Code,” in *Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022, Trento, Italy, October 17-18, 2022, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. Lal and S. Tonetta, Eds., vol. 13800. Springer, 2022, pp. 111–128. [Online]. Available: https://doi.org/10.1007/978-3-031-25803-9_7

[42] L. Li and E. L. Gunter, “K-LLVM: A relatively complete semantics of LLVM IR,” in *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, ser. LIPIcs, R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 7:1–7:29. [Online]. Available: <https://doi.org/10.4230/LIPIcs.ECOOP.2020.7>

[43] K. Slind and M. Norrish, “A brief overview of HOL4,” in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, ser. Lecture Notes in Computer Science, O. A. Mohamed, C. A. Muñoz, and S. Tahar, Eds., vol. 5170. Springer, 2008, pp. 28–32. [Online]. Available: https://doi.org/10.1007/978-3-540-71067-7_6

[44] I. Roessle, F. Verbeek, and B. Ravindran, “Formally verified big step semantics out of x86-64 binaries,” in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, A. Mahboubi and M. O. Myreen, Eds. ACM, 2019, pp. 181–195. [Online]. Available: <https://doi.org/10.1145/3293880.3294102>

[45] F. Verbeek, P. Olivier, and B. Ravindran, “Sound C code decompilation for a subset of x86-64 binaries,” in *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020. Proceedings*, ser. Lecture Notes in Computer Science, F. S. de Boer and A. Cerone, Eds., vol. 12310. Springer, 2020, pp. 247–264. [Online]. Available: https://doi.org/10.1007/978-3-030-58768-0_14

[46] A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell, “Isla: Integrating full-scale ISA semantics and axiomatic concurrency models,” in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021. Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12759. Springer, 2021, pp. 303–316. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_14

[47] A. Lindner, R. Guanciale, and R. Metere, “TrABin: Trustworthy analyses of binaries,” *Sci. Comput. Program.*, vol. 174, pp. 72–89, 2019. [Online]. Available: <https://doi.org/10.1016/j.scico.2019.01.001>

[48] M. Gómez-Zamalloa, E. Albert, and G. Puebla, “Modular decompilation of low-level code by partial evaluation,” in *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China*. IEEE Computer Society, 2008, pp. 239–248. [Online]. Available: <https://doi.org/10.1109/SCAM.2008.35>

[49] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalace - automatic detection of authentication bypass vulnerabilities in binary firmware,” 2015.

[50] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, “The BINCOA framework for binary code analysis,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 165–170. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_13

[51] G. Rosu and T. Serbanuta, “An overview of the K semantic framework,” *J. Log. Algebraic Methods Program.*, vol. 79, no. 6, pp. 397–434, 2010. [Online]. Available: <https://doi.org/10.1016/j.jlap.2010.03.012>


APPENDIX


A. ASL Interpreter Bugs


We identified multiple bugs in the existing ASL interpreter in the process of testing our partial evaluator. Note that the ASL interpreter, as open-sourced by ARM, offered no guarantee of correctness. We fixed these issues in our partial evaluation fork¹.

- Simultaneous assignments to multiple record fields were evaluated in the wrong order. For example, `PSTATE.[N,Z,C,V] = nzcvcv` should assign bit 0 of `nzcvcv` to the `V` field, bit 1 to `C` and so on. Instead, the reverse order was used, e.g. bit 0 was incorrectly assigned to `N`.
- ASL defines reference parameters which allow functions to modify their arguments directly. This was implemented in the parser but its semantics were not handled in the evaluation code, instead treating them as regular parameters.
- The interpreter’s evaluation function would become stuck after breaking from a ‘while’ loop, due to a parsing ambiguity.

Cycle and Commute: Rare-Event Probability Verification for Chemical Reaction Networks

Landon Taylor 
Utah State University
Logan, Utah, USA
landon.jeffrey.taylor@usu.edu

Bryant Israelsen 
Utah State University
Logan, Utah, USA
bryant.israelsen@usu.edu

Zhen Zhang 
Utah State University
Logan, Utah, USA
zhen.zhang@usu.edu

Abstract—In synthetic biological systems, rare events can cause undesirable behavior leading to pathological effects. Due to their low observability, rare events are challenging to analyze using existing stochastic simulation methods. Chemical Reaction Networks (CRNs) are a general-purpose formal language for modeling chemical kinetics. This paper presents a fully automated approach to efficiently construct a large number of concurrent traces by expanding a sample of known traces. These traces constitute a partial state space containing only traces leading to a rare event of interest. This state space is then used to compute a lower bound for the rare event’s probability. We propose a novel approach for the analysis of highly concurrent CRNs, including a CRN reaction independence analysis and an algorithm that exploits CRN concurrency to rapidly enumerate parallel traces. We then present a novel algorithm to add cycles to a partial state space to further increase the rare event’s probability lower bound to its actual value. The resulting prototype tool, RAGTIMER, demonstrates improvement over stochastic simulation and probabilistic model checking.

Index Terms—concurrency, rare events, chemical reaction networks

I. INTRODUCTION

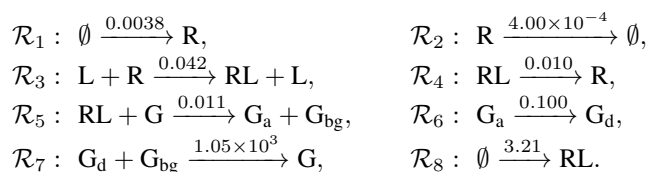
Chemical Reaction Networks (CRNs) are a general-purpose language for modeling chemical kinetics in genetic regulatory networks [1], molecular programming [2], and biochemical reaction systems [3]. Probabilistic behavior is inherent to many systems modeled in CRNs. For example, gene and protein expressions include reactions that occur simultaneously with distinct probabilities. Further, noisy biological systems can easily introduce unexpected and erroneous behavior. In these systems, rare events are often highly relevant, as they can represent infrequent but undesirable behavior that may lead to pathological consequences. Obtaining reliability guarantees is thus essential for CRNs. Existing formal verification techniques, such as *probabilistic model checking* (PMC), can provide provable guarantees to quantify a rare event’s probability in CRNs. In practice, it is often necessary to generate a large number of traces to guarantee an accurate lower bound for a rare-event probability, as a single trace or a small number of traces often yields an insufficient estimate. Existing PMC tools are often unable to enumerate large or infinite state spaces to gather traces and verify a rare event’s probability [4]. The computation of a rare event’s probability can easily become intractable in this case.

This paper presents a *fully* automated approach to exploit a CRN model’s concurrency to rapidly expand a small sample of traces into a partial state space that *only* includes traces leading to a rare event of interest. This partial state space guarantees a lower bound for the rare-event probability. We first propose an independence relation analysis for CRN reactions. It enables a novel parallel trace discovery algorithm that effectively expands a small number of traces. Additionally, we present a novel algorithm to detect and add productive cycles to explored states. Together, the constructed partial state space is used to compute the rare event’s probability lower bound.

In benchmarking tests, a prototype implementation, *Cycle & Commute* expansion of the *Random Assume Guarantee Testing Induced Model Executions for Reachability* (RAGTIMER) tool [5], demonstrates encouraging results for several challenging CRN models. We believe that this unique combination of parallel trace discovery and cycle addition has not been proposed elsewhere and is a fully automated, effective, and user-friendly alternative to existing rare-event simulation approaches for the analysis of CRN rare-event properties.

II. MOTIVATING EXAMPLE

The *modified yeast polarization* model [6] was modified from the pheromone-induced G-protein cycle in *Saccharomyces cerevisia* [7] with a constant ligand population that keeps it away from reaching equilibrium [8], as follows:



This CRN has eight chemical reactions interacting with the species vector $[R, L, RL, G, G_a, G_{bg}, G_d]$. All reaction propensities are in molecules per second. The initial state $s_0 = [50, 2, 0, 50, 0, 0, 0]$ represents the corresponding molecule count. This model incurs a large state space due to its highly concurrent nature, e.g., \mathcal{R}_1 and \mathcal{R}_8 are both independent of all other reactions. Also, by inspection, one can see that at least 100 reactions must execute to reach a state where $G_{bg} = 50$. As discussed in Section VIII, this model challenges several cutting-edge probabilistic model checking tools.

III. PRELIMINARIES

1) *Chemical Reaction Networks (CRNs)*: A CRN is a tuple \mathcal{M} composed of m chemical species $\mathfrak{X} = \{\mathcal{X}_0, \dots, \mathcal{X}_{m-1}\}$, n reactions $\mathfrak{R} = \{\mathcal{R}_0, \dots, \mathcal{R}_{n-1}\}$, an initial state $s_0 : \mathfrak{X}^m \rightarrow \mathbb{Z}_{\geq 0}$, and a vector of all species' initial molecule count, where $m, n \in \mathbb{Z}_{\geq 0}$ and $m, n < \infty$. A CRN is represented as a *Vector Addition System (VAS)* as follows, adapted from [9]. A reaction tuple $\mathcal{R}_i = \langle \mathbf{rv}_i, \mathbf{pv}_i, k_i, \theta_i \rangle$ includes the following: a reactant vector $\mathbf{rv}_i \in \mathbb{Z}_{\geq 0}^m$ representing the stoichiometry of reactants, a product vector $\mathbf{pv}_i \in \mathbb{Z}_{\geq 0}^m$ representing the stoichiometry of products, a reaction rate coefficient $k_i \in \mathbb{R}^+$, and a *propensity function* $\theta_i : \mathbb{Z}_{\geq 0}^m \rightarrow \mathbb{R}^+$ representing the probability that \mathcal{R}_i occurs in a state. The *state change vector*, $\lambda_i = \mathbf{pv}_i - \mathbf{rv}_i$, represents the molecule count update for each species involved in \mathcal{R}_i . In this work, all CRN models follow the *Stochastic Chemical Kinetic (SCK)* assumption, which requires that each reaction \mathcal{R}_i occurs nearly instantaneously, practically limiting elements of λ_i to the values of $0, \pm 1, \pm 2$ and at most three reactants in one reaction [1].

2) *CRN Semantics*: The underlying model of a CRN is a *Continuous-time Markov Chain (CTMC)*, where state updates occur in discrete amounts and the probability of state change is a function of time. Formally, a CTMC is a tuple $\mathcal{C} = \langle \mathbf{S}, s_0, \mathbf{R}, \mathbf{L} \rangle$ where \mathbf{S} is a finite state set called the *state space*; $s_0 \in \mathbf{S}$ is the initial state; $\mathbf{R} : \mathbf{S} \times \mathbf{S} \rightarrow \mathbb{R}_{\geq 0}$ is the transition rate matrix; and $\mathbf{L} : \mathbf{S} \rightarrow 2^{AP}$ is a state labeling function with atomic proposition set AP . A reaction \mathcal{R}_i is *enabled* in state s if its propensity function $\theta_i(s)$ evaluates to a positive value. The propensity function is the product of k_i and the number of possible combinations of reactant molecules: $\theta_i(s) = k_i \prod_{\mathcal{X}_j \in \text{Reactant}_i} (s[j])$. $\text{Reactant}_i \subseteq \mathfrak{X}$ is the set of reactants for \mathcal{R}_i : $\text{Reactant}_i = \{\mathcal{X}_\alpha \mid \mathbf{rv}_i[\alpha] > 0, \forall 0 \leq \alpha < m\}$. The propensity function is the transition rate $\mathbf{R}(s, s')$ from state s to s' in the CTMC \mathcal{C} induced by a CRN. The probability that reaction \mathcal{R}_i is selected to occur out of many reactions is $p(s, s') = \frac{\mathbf{R}(s, s')}{E(s)}$, where the *exit rate* $E(s) = \sum_{s' \in \text{post}(s)} \mathbf{R}(s, s')$ is the sum of all enabled reaction rates in s . A CTMC has a non-zero probability of staying in a state. The probability of exiting a state s in time interval $[0, t]$ is $1 - e^{-E(s) \cdot t}$, where $t \in \mathbb{R}_{\geq 0}$ represents real time. For example, \mathcal{R}_3 is executed from the given initial state in the motivating example to reach $s_1 = [49, 2, 1, 50, 0, 0, 0]$. In this state, \mathbf{rv}_5 and \mathbf{pv}_5 for \mathcal{R}_5 are $[0, 0, 1, 1, 0, 0, 0]$ and $[0, 0, 0, 0, 1, 1, 0]$, respectively; k_5 is 0.011; and $\theta_5(s_1) = k_5(s_1[2])(s_1[3]) = 0.011 \cdot 1 \cdot 50 = 0.55 > 0$, indicating that \mathcal{R}_5 is enabled in s_1 . The state change vector λ_5 is $[0, 0, -1, -1, 1, 1, 0]$. Additional enabled reactions and their propensities at this state are \mathcal{R}_1 (0.0038), \mathcal{R}_2 (0.0196), \mathcal{R}_3 (4.116), \mathcal{R}_4 (0.01), and \mathcal{R}_8 (3.21). The exit rate $E(s_1)$ is 7.9094 and the probability that \mathcal{R}_5 executes is $0.55/7.9094 \approx 0.0695$.

3) *Time-bounded Reachability Property and Target States*: In *Continuous Stochastic Logic (CSL)* [10], [11], the non-nested time-bounded transient reachability probability is specified as $P_{=?}(\Diamond^{[0, T]} \Psi)$. It represents the probability of reaching rare-event Ψ -states within a time bound of T . In this work, a

target Ψ is an equality condition on exactly one species and is not satisfied in s_0 . Formally, let condition Ψ be $\mathcal{X}_\psi = C_\psi$, where $C_\psi \in \mathbb{Z}_{\geq 0}$ and $s_0(\mathcal{X}_\psi) \neq C_\psi$. A state s_i is a *target state* s_ψ if and only if $s_i \models \Psi$. This work provides a guaranteed lower bound on the solution to $P_{=?}(\Diamond^{[0, T]} \Psi)$.

4) *Model Execution*: Denote an execution of reaction \mathcal{R}_i from state s_k as $s'_k = s_k + \lambda_i$. Denote “reaction \mathcal{R}_i is enabled to execute at state s_k ” as $\forall 0 \leq \alpha < m, s_k[\alpha] + \lambda_i[\alpha] \geq 0$. Let a *run* Ξ indicate a sequence of reactions. Reactions \mathcal{R}_i and \mathcal{R}_j are *adjacent* if \mathcal{R}_j immediately follows \mathcal{R}_i in Ξ . Run Ξ is a *valid run* from a state s_i (i.e., $\text{Valid}(\Xi)$ holds for s_i) if no reaction in Ξ is disabled when Ξ 's execution begins at state s_i . A *trace* ρ indicates a valid run starting with s_0 and terminating at a target state s_ψ . A *seed trace* is a trace used as an input for the methods presented in this paper. Note that CRNs are often provided without upper bounds on the species count, which creates an infinite-state CTMC. However, because this work explores only finite traces from s_0 to Ψ -states, the partial state space constructed from these traces is finite.

In this work, seed traces are generated using the trace generation feature in RAGTIMER. RAGTIMER uses compositional testing with assume-guarantee reasoning to rapidly generate many shortest traces.

IV. RELATED WORK

A CRN can be represented as a Vector Addition System (VAS) [9], sometimes described as a Petri net [12]. Reachability analysis, cycle detection, and other properties make VAS a convenient formalism to represent a CRN [13]–[16].

Rare-event properties often found in CRNs pose a challenge to modern stochastic simulation and probabilistic verification methods due to their extremely low observability. The effectiveness of the *weighted Stochastic Simulation Algorithm (wSSA)* [17] heavily relies on a user-specified probability biasing scheme to favor reactions leading to a rare event. Extensions of wSSA (e.g., [18]–[20]) have substantially improved its efficiency. As an alternative to wSSA, the *weighted ensemble (WE)* technique [21], [22] has been used to sample CRN rare events [23], [24]. Existing *statistical model checking (SMC)* techniques (e.g., [25], [26]) integrate rare-event methods. *Importance sampling* [27], [28] weighs the rare-event probability to bias simulation in order to increase the likelihood of encountering rare events of interest. It then compensates for the loss to yield an unbiased probability. In *importance splitting* [29]–[31], an importance function, potentially constructed manually, is used to reward or terminate simulation traces to divide a model's state space into contiguous levels ordered by increasing likelihood of reaching a rare event [32], [33]. Authors of [34] present an automated importance function derivation technique and recently re-implemented the extended RESTART with the *prolonged retrials* importance technique [35], [36] in the SMC engine modes [34], [37], available in the MODEST TOOLSET [38].

The proposed method is *fully* automated and does not require expert knowledge of the CRN model. It is less computationally intensive than other rare-event analysis methods, as

it neither requires rare-event biasing computations nor wastes computational effort pursuing runs that do not lead to a rare event. Lastly, it yields a probability lower-bound with provable guarantees instead of a probability estimate.

V. CRN INDEPENDENCE AND COMMUTABILITY

CRNs are intrinsically highly concurrent. Consider the motivating example in Section II. Reactions \mathcal{R}_1 and \mathcal{R}_8 are *always* enabled, regardless of the current state of the CRN. By leveraging properties of the VAS representation of a CRN, we present a novel analysis of the independence relation among CRN reactions, enabling effective state space exploration.

A. Independence Relation for CRN Reactions

The study of action independence can be traced back to the work of Lipton [39] and Mazurkiewicz [40] on commuting concurrent actions. Mazurkiewicz traces are equivalent classes of action sequences. Action independence has also been the foundation of partial order reduction techniques (e.g. [41]–[43]) for verifying concurrent system correctness. We propose an independence relation specific to reactions in a CRN.

Definition 1 (Independence of CRN Reactions): Two adjacent reactions \mathcal{R}_i and \mathcal{R}_j (defined in Section III-4) are *independent* and *enabled* at state s_k if and only if:

- 1) \mathcal{R}_i and \mathcal{R}_j can execute in either order from s_k :
 $(s_k + \lambda_i) + \lambda_j = (s_k + \lambda_j) + \lambda_i$.
- 2) \mathcal{R}_j is enabled after \mathcal{R}_i executes at s_k :
 $\forall 0 \leq \alpha < m, (s_k + \lambda_i)[\alpha] + \lambda_j[\alpha] \geq 0$.
- 3) \mathcal{R}_i is enabled after \mathcal{R}_j executes at s_k :
 $\forall 0 \leq \alpha < m, (s_k + \lambda_j)[\alpha] + \lambda_i[\alpha] \geq 0$.

If \mathcal{R}_i and \mathcal{R}_j are not independent, they are *dependent*.

Because a VAS representation of a CRN reaction involves only vector addition, condition (1) is true in every state for which both conditions (2) and (3) hold. That is, because vector addition is commutative and associative, firing a series of enabled reactions from a designated state in any order *always* results in the same final state. Conditions (2) and (3) thus become sufficient and necessary conditions for the independence of CRN reactions.

B. Commutability of Reactions

Conditions (2) and (3) described above enable reaction independence (and thus commutability) to be further categorized. We propose three classes to represent commutability between adjacent reactions: trivially, semi-trivially, and conditionally commutable pairs. Adjacent reactions \mathcal{R}_i and \mathcal{R}_j are a *trivially commutable pair* iff $\forall 0 \leq \alpha < m, \lambda_i[\alpha], \lambda_j[\alpha] \in \mathbb{Z}_{\geq 0}$. That is, \mathcal{R}_i and \mathcal{R}_j are trivially commutable at *all states* if they require no reactants to produce their products. \mathcal{R}_i and \mathcal{R}_j are a *semi-trivially commutable pair* iff $\forall 0 \leq \alpha < m, \mathbf{rv}_i[\alpha] = 0 \vee \mathbf{rv}_j[\alpha] = 0$. That is, \mathcal{R}_i and \mathcal{R}_j are semi-trivially commutable at s_k if they share no reactants and are both enabled at s_k . Intuitively, reactions \mathcal{R}_2 and \mathcal{R}_4 in the motivating example are semi-trivially independent because they share no reactants. If a state s_k provides sufficient R to enable \mathcal{R}_2 and sufficient RL to enable \mathcal{R}_4 , then it is always

the case that \mathcal{R}_2 and \mathcal{R}_4 are enabled to execute in any order from s_k . If \mathcal{R}_i and \mathcal{R}_j are neither trivially nor semi-trivially commutable, they are *conditionally commutable*.

Trivially and semi-trivially commutable pairs do not require explicitly checking conditions (2) or (3), enabling state exploration to bypass the need to simulate adjacent reactions to determine commutability. In trivially commutable pairs, λ_i and λ_j contain only non-negative integers, so it is always the case that $\lambda_i + \lambda_j$ contains only non-negative integers. In semi-trivially commutable pairs, each element of $\lambda_i + \lambda_j$ contains at least the lowest negative value in reactants of either λ_i or λ_j , because \mathcal{R}_i and \mathcal{R}_j do not share reactants. Thus, one reaction in a semi-trivially commutable pair cannot disable the other, so if Equation 1 holds, conditions (2) and (3) must also hold. Checking Equation 1 removes the need to simulate semi-trivially commutable reactions directly, conserving effort while exploring the state space. Conditionally commutable pairs require conditions (2) and (3) to be checked explicitly.

$$\forall 0 \leq \alpha < m, (s_k[\alpha] + \lambda_i[\alpha] \geq 0 \wedge s_k[\alpha] + \lambda_j[\alpha] \geq 0) \quad (1)$$

C. Sequences of Conditionally Commutable Reactions

Given a run consisting of a sequence of κ (potentially repeating) reactions $\Xi = \mathcal{R}_0, \mathcal{R}_1, \dots, \mathcal{R}_{\kappa-1}$, it may be desirable to check that firing a sequence of reactions Ξ from s_0 produces a valid run (i.e. each reaction in Ξ is enabled when Ξ is executed in order). If Ξ contains conditionally commutable pairs of reactions, Equation 2 checks that Ξ is a valid run from a state s_x . In the motivating example, a valid run from the initial state s_0 is $\mathcal{R}_8, \mathcal{R}_5$; while an invalid run from s_0 is $\mathcal{R}_5, \mathcal{R}_8$ because \mathcal{R}_5 is not enabled from the initial state, but firing \mathcal{R}_8 enables the execution of \mathcal{R}_5 .

$$\text{Valid}(\Xi) := (\forall j \leq \kappa, 0 \leq i < m, s_x + \sum_{\alpha=0}^j \lambda_\alpha[i] \geq 0) \quad (2)$$

VI. PARALLEL TRACES VIA COMMUTATION

Exploring the inherent concurrency in CRN models helps to discover traces contributing to a rare event's probability. These traces may be obtained by various methods. Results presented in this paper use traces generated by the prototype tool RAGTIMER. CRN models often contain a large volume of parallel traces (i.e., traces that differ by a small number of reactions or arrive at the same state while passing through alternative intermediate states).

To obtain a lower bound for the rare event's probability, we desire to accumulate probability from a large number of traces to a rare event as efficiently as possible. We suggest parallel traces are an efficient way to accumulate probability for rare events. Algorithm 1 finds parallel traces using Equations 1 and 2 to discover pairs of independent, commutable reactions. For example, interrupting a seed trace from the motivating example by firing \mathcal{R}_1 at a random state forms a nearly-identical trace and increases the overall probability lower bound relative to the seed trace alone.

Figure 1 illustrates this principle on a small toy example. In this example, the seed trace (blue) contains reactions $\mathcal{R}_0, \mathcal{R}_1$,

and \mathcal{R}_2 . By interrupting this seed trace with a universally-enabled reaction \mathcal{R}_a , it is possible to obtain many parallel traces and increase the lower-bound of the probability of reaching a target state. This particular example shows two unique target states with four additional traces, so the probability of reaching a target is increased compared to the probability of the seed trace alone. In some models, parallel traces arrive at the same target state as the seed trace via an alternative reaction sequence. Having two target states, as is the case in Figure 1, is allowed but not required for parallel trace exploration; only target state s'_ψ is required.

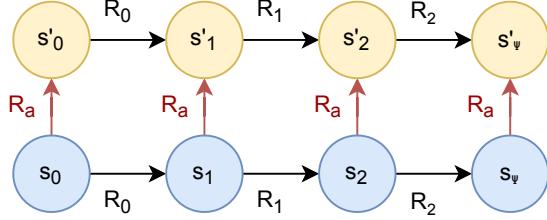


Fig. 1: Parallel trace construction via transition commutation.

A. Trace Commutation Algorithm

Algorithm 1 details the procedure for exploring parallel traces contained in the “Traces” set, which contains seed traces generated by RAGTIMER or a user’s method of choice. As the main procedure, BUILDTRACES builds a partial state space for each seed trace in “Traces”, then calls the recursive function COMMUTE on each trace, which recursively explores traces parallel to each seed trace and builds a partial state space as it explores. Using commutability conditions presented in Section V, Algorithm 1 attempts to find commutable reactions along the entire length of a seed trace. To efficiently explore traces leading to a rare event, it attempts to commute reactions that are enabled from *every* state along the seed trace.

In Figure 1, for instance, line 2 of Algorithm 1 selects the seed trace $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2$ as ρ . In lines 3 and 4, it builds a partial state space for the seed trace, then discovers $\mathcal{R}_a \in E$. In COMMUTE, it executes the prefix (line 11), which is initially empty but is extended during recursion to list the sequence of commuted reactions to fire before firing the reactions from the seed trace. In lines 12 and 13, states along the parallel trace ρ' (shown in yellow on the top of Figure 1) are discovered. In lines 14 and 15, ρ' is built from the commuted transition \mathcal{R}_a . Finally, in line 16, the function recursively attempts to commute transitions along the parallel trace ρ' , which now includes prefix \mathcal{R}_a . This recursive process is shown in Figure 2. The seed trace, shown in blue with s_0 , leads to the discovery of three parallel traces shown in yellow. These traces, with s_b, s_c , and s_d , are then recursively analyzed. For instance, it may lead to the discovery of two more parallel traces, shown in green with states s_e and s_f .

Algorithm 1 Commuting universally enabled transitions

Require: $\mathcal{M} = \langle \mathcal{X}, \mathcal{R}, s_0 \rangle, \Psi, \text{Traces}$.

```

1: procedure BUILDTRACES
2:   for Trace  $\rho$  in Traces do
3:     Build the state space for states along  $\rho$ 
4:      $E \leftarrow$  enabled reactions along  $\rho$ 
5:     COMMUTE( $\emptyset, \rho, E$ )
6:   BUILDCYCLES  $\triangleright$  Defined formally in Algorithm 2
7:   Clean up the model to save time and memory
8:   Export explicit state-transition matrices
9: procedure COMMUTE(Prefix,  $\rho, E$ )
10:  for  $\mathcal{R}_a \in E$  do
11:    Execute Prefix.
12:    Fire  $\mathcal{R}_a$  from each state in  $\rho$  to find  $\rho'$ 
13:     $E' \leftarrow$  enabled reactions along  $\rho'$ .
14:    Execute  $\mathcal{R}_a$  from  $s_0$  of  $\rho$  to find  $s'_0$  in  $\rho'$ .
15:    Execute reactions in  $\rho$  from  $s'_0$  to construct  $\rho'$ .
16:    COMMUTE((Prefix.append( $\mathcal{R}_a$ )),  $\rho', E'$ )

```

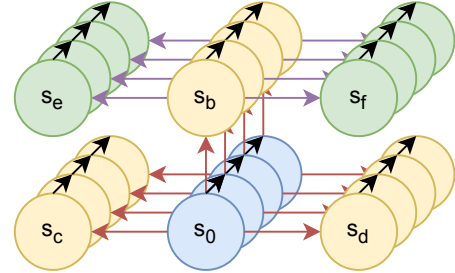


Fig. 2: Recursively-commuted partial state space.

B. Termination Conditions on Algorithm 1

Algorithm 1 does not necessarily terminate. Thus, we propose two methods for determining when to terminate commutation recursion:

- 1) The user can specify a maximum recursion depth. This is a naive approach, but it guarantees termination and gives flexibility to advanced users; or
- 2) The algorithm can terminate based on the time bound T from the model’s CSL property $P_{=?}(\Diamond^{[0,T]} \Psi)$. The mean state residence time for s_i is provided by $MRT(s_i) = 1/E(s_i)$. The sum of mean state residence times along a trace provides the average duration for that trace. If the average trace duration exceeds or approaches the property time bound, it is likely not worth exploring further as traces will become increasingly unlikely. This approach requires less understanding of the model and algorithm, so it provides less flexibility but a more streamlined user experience.

Algorithm 1 is implemented as an extension of the RAGTIMER tool. In this implementation, a user can specify if they prefer to terminate by recursion depth or by analyzing average trace durations. Approach (1) terminates trivially. Termination of approach (2) is justified because each reaction adds a

positive amount of time to the total average trace duration. The algorithm will thus either explore the entire available state space or the average duration will eventually increase to meet the termination threshold. It is our experience that a very small time or depth bound is sufficient to obtain a significant probability boost in the seed traces.

C. Exporting Explicit Models

After exploring parallel traces, each unexplored enabled reaction at any state is replaced by an absorbing reaction (i.e., a new reaction transitioning to an absorbing state) with an equivalent probability. Formally, let $En(s_i)$ represent the set of all reactions enabled in state s_i . Let $Disc(s_i)$ represent the set of all reactions added to the explicit state space, such that $Disc(s_i) \subseteq En(s_i)$. Let $Undisc(s_i)$ represent the set reactions in $En(s_i)$ but not included in the explicit state space, such that $Undisc(s_i) = En(s_i) - Disc(s_i)$.

Let $A(s_i)$, defined in Equation 3, indicate the sum of transition rates directed from state s_i to an abstract absorbing state. The absorbing state preserves probability correctness by consuming any probability that would have been directed to an unexplored portion of the state space. To obtain a probabilistic lower-bound, it is assumed that the absorbing state does not satisfy Ψ . An explicit state space can then be exported for model checking in a tool such as PRISM [44] or Storm [45].

$$A(s_i) = \sum_{R_j \in Undisc(s_i)} \theta(s_j) \quad (3)$$

D. Lower-Bound Probability Guarantee

Because the presented method explicitly enumerates traces, the probability obtained by performing probabilistic model checking on the explicit state graph is guaranteed to be a lower bound. The seed trace is known to reach a target state, so finding parallel traces through commutation also produces traces leading to the same target state. This method is efficient because every state and reaction (except the absorbing state) is guaranteed to contribute to a rare event's probability.

VII. CYCLES FOR PROBABILITY RECAPTURE

CRN models often contain cyclic behavior. Including cycles in a state space is an effective way to increase the total number of explored traces without greatly increasing the total number of states explored. In many models, the exploration of cycles can increase the probability lower bound by redirecting some of the probability that would otherwise be redirected to an absorbing state (see Section VI-C) to a target state.

While a number of cycle exploration methods have been explored (in [13], for instance), we found a simple combinatorial analysis of reactions sufficient to efficiently generate a large number of cycles for the purposes of this work. This approach involves testing multisets of reactions up to a user-specified bound and selecting multisets of reactions such that the sum of state change vectors corresponding to reactions within each multiset is equal to the zero vector.

Formally, a cycle c_i is a κ -multiset containing κ reactions such that the sum of all reaction state change vectors in

c_i is the zero vector. Let "CycleList" be a set of known cycles. Algorithm 2 presents an approach to augmenting the probability lower bound by adding cycles into a partial state space. Let $\omega(c_i)$ represent a permutation of reactions in c_i , with $\Omega(c_i)$ defined as the set of all possible $\omega(c_i)$. Define $\min(\omega(c_i))$ as a vector of length m (i.e., a vector with one element per species) such that $\forall 0 \leq \alpha < m$, $\min(\omega(c_i))[\alpha] = \min_{R_j \in \omega(c_i)} \sum_{k=0}^j \lambda_k[\alpha]$. In Line 4 of Algorithm 2, this is achieved via a vector copy operation. Intuitively, the minimal value of species α in $\omega(c_i)$ is either non-negative, indicating species α is never consumed, or it is negative. If $\min(\omega(c_i))[\alpha] = -\gamma$, at some point, $\omega(c_i)$ has consumed and has not replenished γ molecules of species l . In Algorithm 2, $\min(\omega(c_i))$ determines which states are candidates for the addition of $\omega(c_i)$. If a state s_i does not provide enough of a given reactant to execute $\omega(c_i)$, i.e. if $\text{Valid}(\omega(c_i))$ does not hold at state s_i , cycle $\omega(c_i)$ cannot be added to s_i . By finding the minimal value for a molecule count during a cycle, it becomes unnecessary to simulate a cycle from every state to determine if it is possible to add the cycle to the state. This saves computational effort while enabling cycles to be added to every allowable state. Maximum cycle *lengths* are specified by users, and *all* allowable cycles up to the user-specified length are added to the state space.

In the motivating example, executing \mathcal{R}_2 followed by \mathcal{R}_1 constitutes a permutation $\omega(c)$ of the cycle with length two, i.e., $c = \{\mathcal{R}_1, \mathcal{R}_2\}$. Because this cycle causes a degradation of R followed by a generation of R , $\min(\omega(c))[0] = -1$. That is, $\omega(c)$ can only be added to state s_i if $s_i[0] \geq 1$.

Algorithm 2 Adding cycles to a partial state space

Require: $\mathcal{M} = \langle \mathcal{X}, \mathcal{R}, s_0 \rangle$, CycleList.

```

1: procedure BUILD_CYCLES
2:   for Cycle  $c_i$  in CycleList do
3:     for Cycle permutation  $\omega(c_i)$  in  $\Omega(c_i)$  do
4:        $\min(\omega(c_i)) \leftarrow \min_{R_j \in \omega(c_i)} \sum_{k=0}^j \lambda_k$ 
5:       for State  $s_x$  in discovered state space do
6:         for  $\alpha \in [0, m)$  do
7:           if  $s_x[\alpha] + \min(\omega(c_i))[\alpha] \geq 0$  then
8:             Add  $\omega(c_i)$  to  $s_x$ 

```

Adding even one cycle to a trace can increase the probability of that trace. Because Algorithm 1 produces an explicit state space, the task of evaluating the overall probability impact of cycles is given to a probabilistic model checker. This enables only a few additional states in the explicit state space to influence the probability of the model by providing a larger number of traces. For example, Figure 3 shows the partial state space from Figure 1. An arbitrary cycle (represented by three green states) is enabled to be executed from five states, so rather than direct reactions from those states to an absorbing state, the cycles redirect part of the probability back into the trace leading to the target rare-event state. Note that in this example, the cycle is not added to state s_1 . In a realistic model, this happens when s_1 does not provide sufficient reactants to

enable the full execution of the cycle.

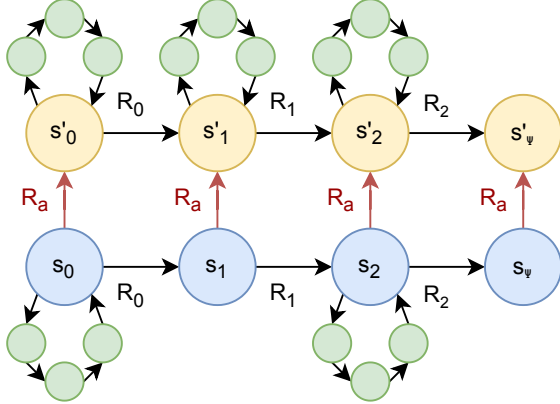


Fig. 3: Cycles added to five states in a partial state space.

It is occasionally the case that cycles add states and transitions (and thus computation time and memory for probabilistic model checking) to a state space without contributing significantly to the rare event’s probability bound. This is largely the case when one or more states along a cycle has a large absorbing rate relative to its other outgoing rates (i.e., when it is more likely that starting a cycle will lead to an absorbing state than return to the original trace). If enough of the probability that flows into the cycle does not flow back toward the target state, the cycle is not sufficiently valuable and need not be added. Given a desirable threshold \mathbb{T} (in our tests, a high threshold of around 0.98) of the ratio $A(s_i)/E(s_i)$, Lines 7-8 of Algorithm 2 may be adapted to include a cycle addition benefit heuristic as shown in Equation 4. The ratio $A(s_i)/E(s_i)$ intuitively represents how much probability is directed to an absorbing state versus into a trace.

if $s_x[y] + \min(\omega(c_i))[y] \geq 0 \wedge A(s_i)/E(s_i) < \mathbb{T}$
then Add $\omega(c_i)$ to s_x (4)

VIII. RESULTS AND DISCUSSION

The parallel trace exploration and cycle addition methods presented in this paper are implemented as part of a prototype tool, RAGTIMER, which interfaces with the PRISM API [44]. Prototype versions of RAGTIMER and its Cycle & Commute expansion are freely available¹. This tool quickly generates many seed traces. The benchmarking results presented in this paper were obtained on an AMD Ryzen Threadripper 12-Core 3.5 GHz Processor and 132 GB of RAM, running Ubuntu 22.04 LTS. We allocated one CPU and 16 GB of RAM to test our approach on all four challenging case studies and compared our method’s results to those of other probabilistic verification tools. In each case study, “Default Cycle & Commute” indicates that the default settings for RAGTIMER are used. The default settings include generating 100 shortest

¹RAGTIMER v0.0 (trace generation) and v0.1 (Cycle & Commute) are available as releases at <https://github.com/fluentverification/ragtimer/tags>.

traces, using a fixed recursion bound of 20 (i.e., limit the number of calls to the COMMUTE function in Algorithm 1 to 20), and adding cycles of two reactions after state space construction. These default settings give users an acceptable result for most models, while “Optimized Cycle & Commute” indicates custom settings for each model.

1) *Single Species Production-Degradation Model*: The model describes a production-degradation interaction between two species [17]: $\mathcal{R}_1 : S_1 \xrightarrow{1.0} S_1 + S_2, \mathcal{R}_2 : S_2 \xrightarrow{0.025} \emptyset$. The initial state for the species vector $[S_1, S_2]$ is $s_0 = [1, 40]$, while the desired CSL property is $P_{=?}(\Diamond^{[0,100]} S_2 = 80)$. Figure 4a shows an increase in the lower bound of the model’s probability as the parallel trace exploration recursion bound increases during exploration of a *single* seed trace. The probability bound asymptotically approaches the actual rare-event probability, which is 3.0631×10^{-7} [17]. Figure 4b shows that while the probability increases exponentially, the number of states increases linearly. Thus, we argue this method explores a productive part of the state space. Partial state space exploration required less than 4 seconds at any recursion depth.

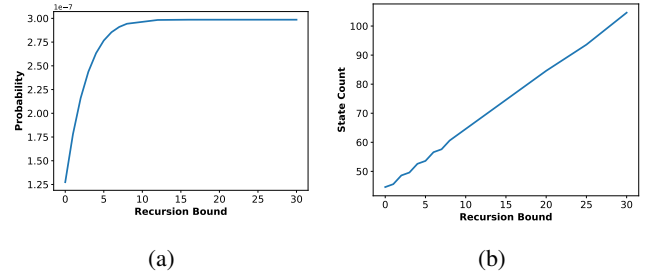


Fig. 4: Single Species Production-Degradation Model.

It is interesting to compare the methods presented in this paper to simple trace generation, which RAGTIMER is already capable of. In our benchmarks, RAGTIMER generated a single seed trace for this model with a probability of 1.03×10^{-16} in 13.26 seconds. It then generated 43 additional traces, increasing the state space’s probability to 1.34×10^{-16} in 31.35 seconds. By expanding the seed trace using methods presented in this paper, however, RAGTIMER achieved a probability bound of 2.99×10^{-7} in 21.21 seconds, including the duration of trace generation, commuting, and model checking. This is a reasonable lower bound to the true probability of the model (3.0631×10^{-7}). Table I summarizes these results. In all result tables, “Default Commuting Options” indicates that the default options implemented in RAGTIMER are selected; “Optimized Commuting Options” indicates that configurations were modified to produce an improved result. In this model, the default settings produced the best probability bound.

2) *Enzymatic Futile Cycle Model*: A futile cycle interaction is modeled in this CRN with six species reacting through six reactions [17]:

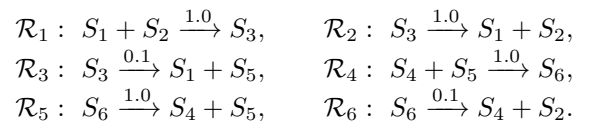


TABLE I: Single-Species Production-Degradation Model.

Method	Probability	Runtime (s)
Generate 1 Trace	$\geq 1.03 \times 10^{-16}$	13.26
Generate 44 Traces	$\geq 1.34 \times 10^{-16}$	31.35
Default Cycle & Commute	$\geq 2.99 \times 10^{-7}$	23.21
Optimized Cycle & Commute	$\geq 2.99 \times 10^{-7}$	23.21

The initial molecule count for species vector $[S_1, S_2, S_3, S_4, S_5, S_6]$ forms the initial state: $s_0 = [1, 50, 0, 1, 50, 0]$ and the rare-event property of interest is $P_{=?}(\Diamond^{[0,100]} S_5 = 25)$. Figure 5 shows the probability and state count for this model’s partial state space as the recursion depth increases for a *single* seed trace. The probability bound sharply increases while the state space grows linearly, illustrating that for this model, states that are considered in the partial state space contribute significantly to the probability bound. State space construction required less than four seconds for all recursion depths for this model. RAGTIMER generated one shortest seed trace for this

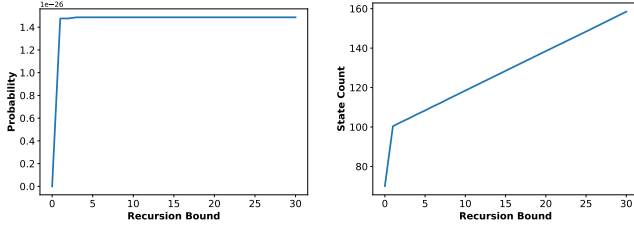


Fig. 5: Enzymatic Futile Cycle Model.

model with a probability of 1.73×10^{-78} in 15.52 seconds. Generating 99 more traces increased the probability bound to 2.71×10^{-64} in 41.1 seconds. By expanding 100 seed traces using methods presented in this paper, RAGTIMER achieved a probability bound of 4.32×10^{-18} in 31.69 seconds, including time for trace generation, commuting, and model checking. This is an improvement of 60 orders of magnitude that requires less runtime than generating a small sample of additional traces. Table II summarizes these results. This model’s results appear to be influenced by the quality of the seed traces used to generate the partial state space. Its optimized settings involve asking RAGTIMER to generate a set of short but unique seed traces. After generating 42 unique traces, it explored a recursion depth of 10 and added all possible cycles of length two to the state space. Further, cycle

TABLE II: Enzymatic Futile Cycle Model.

Method	Probability	Runtime (s)
Generate 1 Trace	$\geq 1.73 \times 10^{-78}$	15.52
Generate 100 Traces	$\geq 2.71 \times 10^{-64}$	41.10
Default Cycle & Commute	$\geq 1.45 \times 10^{-26}$	27.92
Optimized Cycle & Commute	$\geq 4.32 \times 10^{-18}$	31.69

addition boosts the probability of this model without requiring significant additional time. We ran 36 tests to account for this method’s stochastic nature and found that adding only cycles of length two to the model as described in Section VII

increased the average discovered lower probability bound by two orders of magnitude (from 2.28×10^{-21} to 4.92×10^{-19}) while increasing the average total runtime by less than one second (from 25.6 to 26.4 seconds).

3) *Modified Yeast Polarization Model*: The rare event of interest for our motivating example is the rapid build-up of G_{bg} . This is described by the probability of the molecule count of G_{bg} increasing from 0 to 50 within 20 seconds: $P_{=?}(\Diamond^{[0,20]} G_{bg} = 50)$. When this model is simulated using the standard *stochastic simulation algorithm* (SSA) implemented in the PRISM probabilistic model checking tool, the total probability for over 500,000 traces is rounded to 0 due to floating-point precision limitations, indicating that SSA alone produced a probability lower than 4.9×10^{-324} . However, when two seed traces are expanded, the probability is found to be greater than 5.8×10^{-72} after a recursion depth of 6 and with a state space consisting of about 3500 states, produced in less than 10 seconds, as can be seen in Figure 6. Because of this model’s infinite state space, the state count

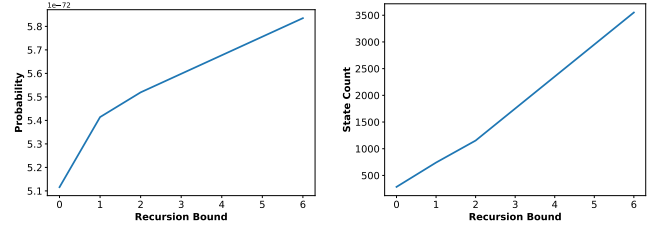


Fig. 6: Modified Yeast Polarization Model.

and probability both appear to increase nearly-linearly with recursion depth. Therefore, the states in this model’s partial state space contribute significantly to its probability bound.

RAGTIMER generated 100 seed traces for which PRISM is unable to compute a nonzero probability (due to floating-point constraints). By expanding these seed traces using methods presented in this paper, however, RAGTIMER achieved a probability of 5.26×10^{-26} in 125.64 seconds, including trace generation, commuting, and model checking time. These results are summarized in Table III. Optimized settings for this model are identical to default settings, but the optimal test used a set of 100 higher-probability seed traces due to the stochastic nature of RAGTIMER trace generation.

TABLE III: Modified Yeast Polarization Model.

Method	Probability	Runtime (s)
Generate 1 Trace	≥ 0.0	23.34
Generate 100 Traces	≥ 0.0	66.78
Default Cycle & Commute	$\geq 1.01 \times 10^{-32}$	167.11
Optimized Cycle & Commute	$\geq 5.26 \times 10^{-26}$	125.64

4) *Simplified motility regulation model*: This model consists of nine species reacting through twelve reactions and represents the genetic mechanism which regulates flagella

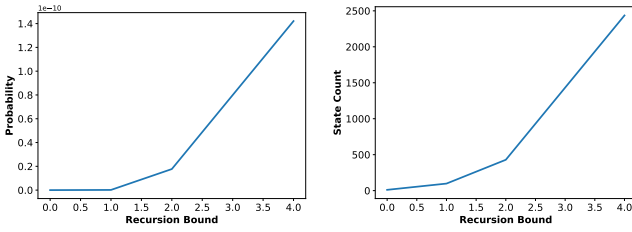
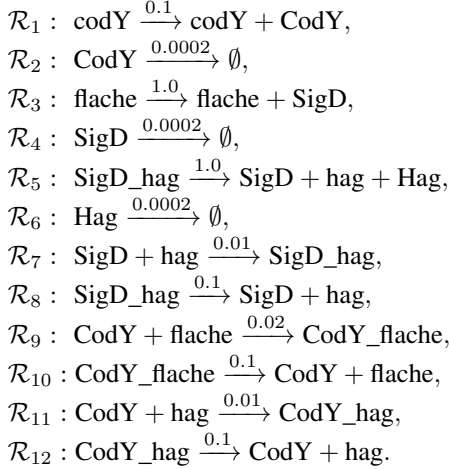


Fig. 7: Simplified Motility Regulation Model.

formation in *Bacillus subtilis* [46]:



The initial molecule count for species vector [codY, flache, SigD_hag, CodY, CodY_flache, hag, CodY_hag, SigD, Hag] forms the initial state $s_0 = [1, 1, 1, 10, 1, 1, 1, 10, 10]$. The rare event property is $P_{=?}(\Diamond^{[0,10]} \text{CodY} = 20)$. Figure 7 shows that while a single seed trace's probability is found to be zero, expanding a single seed trace quickly increases the probability bound to 1.45×10^{-10} . Because the probability and state count growth both appear to grow exponentially relative to the recursion bound, it suggests the states explored by this method contribute efficiently to the rare event probability.

Similarly to the Modified Yeast Reaction Model, generating 100 seed traces produced a low probability bound that was rounded to zero. By expanding the seed trace using methods presented in this paper, however, RAGTIMER achieved a probability of 1.42×10^{-9} in 34.67 seconds, including trace generation, commuting, and model checking time. Results from this model are summarized in Table IV. Due to this model's complexity, its default recursion depth is 2. Increasing the recursion depth to 10 results in the optimized probability.

5) Comparison to modes rare-event simulation engine:

The *modes* statistical model checking tool in the MODEST TOOLSET was able to compute rare-event probabilities efficiently for the presented case studies, and the reported probabilities closely match those reported in [17] and [23]. However, *modes* requires a compositional importance function for rare-event simulation, which limits the use of global variables shared between multiple components. While manual modifications to the model's importance function can be made

to circumvent this, it requires user intervention and an in-depth understanding of the CRN model and MODEST language.

TABLE IV: Simplified Motility Regulation Model.

Method	Probability	Runtime (s)
Generate 1 Trace	≥ 0.0	13.39
Generate 100 Traces	≥ 0.0	17.16
Default Cycle & Commute	$\geq 1.77 \times 10^{-11}$	28.05
Optimized Cycle & Commute	$\geq 1.42 \times 10^{-9}$	34.67

6) *Comparison to probabilistic model checking tools:* We attempted to verify the modified yeast polarization model's rare event property with all species' molecule counts bounded by the reasonably large range of $[0, 150]$ in the probabilistic model checker Storm with the SYLVAN library [47]. Although Storm completed symbolic state space construction quickly, it failed to complete the CTMC analysis of the model within 30 days due to the task of converting a symbolic state space to a sparse matrix representation for time-bounded transient analysis. In another test, the state-truncation probabilistic model checker STAMINA [48] produced a probability bound of $[1.64 \times 10^{-6}, 23.01 \times 10^{-6}]$ on the same model after 2 days.

7) *Discussion:* We claim that our method can compete effectively against these tools because it requires no in-depth understanding of a CRN model, a formal modeling language, or a verification tool. Rather, it requires only a single trace (obtainable from the implemented functionality in RAGTIMER or another user-selected method). We argue that our method is effective because increasing the recursion bound and number of cycles in our benchmarks reliably provides an improved probability bound, demonstrating this method explores an effective region of a state space. This method can also be used to gain insights to guide model synthesis, as it can report information about which reactions and cycles cause a rare event to be more likely. We firmly believe that this model analysis is a useful tool to guide design decisions and reveal design flaws, and that in many cases, it is more useful to a user than a probability report alone.

IX. CONCLUSION

This paper presents a fully-automated approach to expand a small sample of traces and build a partial state space containing only states and transitions leading to a rare event of interest in a CRN model. We propose CRN-specific independence conditions accompanied by an algorithmic method to effectively discover parallel traces that are guaranteed to reach the rare event of interest. This increases the lower bound for the rare event's probability. Adding cycles to a partial state space further increases the rare-event probability lower bound. The promising results from the prototype tool RAGTIMER demonstrate it as an effective and user-friendly method for CRN model analysis. Future work may include further investigation of the properties of cycles in CRN explicit state spaces, integration with other existing probabilistic model checking tools, and improvement on seed trace generation.

Acknowledgment: We thank Arnd Hartmanns (U. Twente) for helping with MODEST TOOLSET; Chris Winstead (Utah State U.), Chris Myers (U. Colorado Boulder), and Hao Zheng (U. South Florida) for their feedback. This work was supported by the National Science Foundation under Grant No. 1856733. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

- [1] C. J. Myers, *Engineering Genetic Circuits*, 1st ed., ser. Chapman & Hall/CRC Mathematical and Computational Biology. Chapman & Hall/CRC, July 2009.
- [2] D. Soloveichik, G. Seelig, and E. Winfree, "Dna as a universal substrate for chemical kinetics," *Proceedings of the National Academy of Sciences*, vol. 107, no. 12, pp. 5393–5398, 2010. [Online]. Available: <https://www.pnas.org/doi/abs/10.1073/pnas.0909380107>
- [3] V. Chellaboina, S. P. Bhat, W. M. Haddad, and D. S. Bernstein, "Modeling and analysis of mass-action kinetics," *IEEE Control Systems Magazine*, vol. 29, no. 4, pp. 60–78, 2009.
- [4] L. Buecherl, R. Roberts, P. Fontanarrosa, P. J. Thomas, J. Mante, Z. Zhang, and C. J. Myers, "Stochastic hazard analysis of genetic circuits in iBioSim and STAMINA," *ACS Synthetic Biology*, vol. 10, no. 10, pp. 2532–2540, 2021, pMID: 34606710. [Online]. Available: <https://doi.org/10.1021/acssynbio.1c00159>
- [5] B. Israelsen, L. Taylor, and Z. Zhang, "Efficient trace generation for rare-event analysis in chemical reaction networks," in *Model Checking Software*, G. Caltais and C. Schilling, Eds. Cham: Springer Nature Switzerland, 2023, pp. 83–102.
- [6] B. J. Daigle, M. K. Roh, D. T. Gillespie, and L. R. Petzold, "Automated estimation of rare event probabilities in biochemical systems," *The Journal of Chemical Physics*, vol. 134, no. 4, p. 044110, Jan. 2011.
- [7] B. Drawert, M. J. Lawson, L. Petzold, and M. Khammash, "The diffusive finite state projection algorithm for efficient simulation of the stochastic reaction-diffusion master equation," *The Journal of Chemical Physics*, vol. 132, no. 7, p. 074101, 2010. [Online]. Available: <https://doi.org/10.1063/1.3310809>
- [8] M. K. Roh, D. T. Gillespie, and L. R. Petzold, "State-dependent biasing method for importance sampling in the weighted stochastic simulation algorithm," *The Journal of Chemical Physics*, vol. 133, no. 17, p. 174106, Nov. 2010.
- [9] M. Česka and J. Křetínský, "Semi-quantitative abstraction and analysis of chemical reaction networks," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 475–496.
- [10] A. Aziz, K. A. Sanwal, V. Singhal, and R. Brayton, "Model-checking continuous-time Markov chains," *ACM Transactions on Computational Logic*, vol. 1, no. 1, pp. 162–170, Jul. 2000.
- [11] M. Kwiatkowska, G. Norman, and D. Parker, *Stochastic Model Checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 220–270.
- [12] I. Koch, "Petri Nets – A Mathematical Formalism to Analyze Chemical Reaction Networks," *Molecular Informatics*, vol. 29, no. 12, pp. 838–843, 2010.
- [13] J. Leroux, "Polynomial Vector Addition Systems With States," in *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, July 9–13, 2018, Prague, Czech Republic, ser. LIPIcs, I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, Eds., vol. 107. Prague, Czech Republic: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Jul. 2018, pp. 134:1–134:13. [Online]. Available: <https://hal.science/hal-01711089>
- [14] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, "Modelling with Generalized Stochastic Petri Nets," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 2, p. 2, Aug. 1998.
- [15] D. Angeli, P. De Leenheer, and E. D. Sontag, "A Petri net approach to the study of persistence in chemical reaction networks," Dec. 2007.
- [16] W. Czerwinski, S. Lasota, R. Łazić, J. Leroux, and F. Mazowiecki, "Reachability in fixed dimension vector addition systems with states," May 2020.
- [17] H. Kuwahara and I. Mura, "An efficient and exact stochastic simulation method to analyze rare events in biochemical systems," *The Journal of Chemical Physics*, vol. 129, no. 16, p. 165101, Oct. 2008.
- [18] C. Jegourel, A. Legay, and S. Sedwards, "Cross-entropy optimisation of importance sampling parameters for statistical model checking," in *Proceedings of the 24th international conference on Computer Aided Verification*, ser. CAV'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 327–342.
- [19] M. Roh, B. J. J. Daigle, D. T. Gillespie, and L. R. Petzold, "State-dependent doubly weighted stochastic simulation algorithm for automatic characterization of stochastic biochemical rare events," in *Journal of Chemical Physics*, vol. 135. American Institute of Physics, 2011.
- [20] M. K. Roh and B. J. Daigle, "Sparse++: improved event-based stochastic parameter search," *BMC Systems Biology*, vol. 10, no. 1, p. 109, 2016. [Online]. Available: <https://doi.org/10.1186/s12918-016-0367-z>
- [21] B. W. Zhang, D. Jasnow, and D. M. Zuckerman, "Efficient and verified simulation of a path ensemble for conformational change in a united-residue model of calmodulin," *Proceedings of the National Academy of Sciences*, vol. 104, no. 46, pp. 18 043–18 048, 2007. [Online]. Available: <https://www.pnas.org/doi/abs/10.1073/pnas.0706349104>
- [22] J. L. Adelman and M. Grabe, "Simulating rare events using a weighted ensemble-based string method," *The Journal of Chemical Physics*, vol. 138, no. 4, p. 044105, 2013. [Online]. Available: <https://doi.org/10.1063/1.4773892>
- [23] R. M. Donovan, A. J. Sedgewick, J. R. Faeder, and D. M. Zuckerman, "Efficient stochastic simulation of chemical kinetics networks using a weighted ensemble of trajectories," *The Journal of Chemical Physics*, vol. 139, no. 11, p. 115105, Sep. 2013.
- [24] D. M. Zuckerman and L. T. Chong, "Weighted ensemble simulation: Review of methodology, applications, and software." *Annu Rev Biophys*, vol. 46, pp. 43–57, May 2017.
- [25] M. Okamoto, "Some inequalities relating to the partial sum of binomial probabilities," *Annals of the Institute of Statistical Mathematics*, vol. 10, no. 1, pp. 29–35, 1959. [Online]. Available: <https://doi.org/10.1007/BF02883985>
- [26] A. Wald, "Sequential tests of statistical hypotheses," *The Annals of Mathematical Statistics*, vol. 16, no. 2, pp. 117–186, 1945. [Online]. Available: <http://www.jstor.org/stable/2235829>
- [27] H. Kahn, "Random sampling (monte carlo) techniques in neutron attenuation problems—I," *Nucleonics*, vol. 6, no. 5, p. 27; passim, May 1950.
- [28] H. Kahn and A. W. Marshall, "Methods of reducing sample size in monte carlo computations," *Journal of the Operations Research Society of America*, vol. 1, no. 5, pp. 263–278, 1953. [Online]. Available: <https://doi.org/10.1287/opre.1.5.263>
- [29] H. Kahn and T. E. Harris, "Estimation of particle transmission by random sampling," *National Bureau of Standards applied mathematics series*, vol. 12, pp. 27–30, 1951.
- [30] M. N. Rosenbluth and A. W. Rosenbluth, "Monte carlo calculation of the average extension of molecular chains," *The Journal of Chemical Physics*, vol. 23, no. 2, pp. 356–359, 1955. [Online]. Available: <https://doi.org/10.1063/1.1741967>
- [31] M. Villén-Altamirano, J. Villén-Altamirano *et al.*, "Restart: a method for accelerating rare event simulations," *Queueing, Performance and Control in ATM (ITC-13)*, pp. 71–76, 1991.
- [32] P. L'Ecuyer, F. Le Gland, P. Lezaud, and B. Tuffin, "Splitting Techniques," in *Rare Event Simulation Using Monte Carlo Methods*. John Wiley & Sons, Ltd, 2009, ch. 3, pp. 39–61.
- [33] M. Villén-Altamirano and J. Villén-Altamirano, *The Rare Event Simulation Method RESTART: Efficiency Analysis and Guidelines for Its Application*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 509–547.
- [34] C. E. Budde, P. R. D'Argenio, and A. Hartmanns, "Automated compositional importance splitting," *Science of Computer Programming*, vol. 174, pp. 90–108, Apr. 2019.
- [35] J. Villén-Altamirano, "Restart vs splitting: A comparative study," *Performance Evaluation*, vol. 121–122, pp. 38–47, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0166531616300839>
- [36] —, "An improved variant of the rare event simulation method restart using prolonged retrials," *Operations Research Perspectives*, vol. 6, pp. 1–9, 2019. [Online]. Available: <http://hdl.handle.net/10419/246387>
- [37] C. E. Budde and A. Hartmanns, "Replicating RESTART with prolonged retrials: An experimental report," in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, ser.

- Lecture Notes in Computer Science, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 373–380. [Online]. Available: https://doi.org/10.1007/978-3-030-72013-1_21
- [38] A. Hartmanns and H. Hermanns, “The Modest Toolset: An integrated environment for quantitative modelling and verification,” in *TACAS*, ser. LNCS, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 593–598.
 - [39] R. J. Lipton, “Reduction: A method of proving properties of parallel programs,” *Commun. ACM*, vol. 18, no. 12, pp. 717–721, dec 1975. [Online]. Available: <https://doi.org/10.1145/361227.361234>
 - [40] A. Mazurkiewicz, “Trace theory,” in *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 278–324.
 - [41] D. Peled, “All from one, one for all: on model checking using representatives,” in *Computer Aided Verification*, C. Courcoubetis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 409–423.
 - [42] P. Godefroid, “Using partial orders to improve automatic verification methods,” in *Computer-Aided Verification*, E. M. Clarke and R. P. Kurshan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 176–185.
 - [43] A. Valmari, “Stubborn sets for reduced state space generation,” in *Advances in Petri Nets 1990*, G. Rozenberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 491–515.
 - [44] M. Kwiatkowska, G. Norman, and D. Parker, “Prism 4.0: Verification of probabilistic real-time systems,” in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 585–591.
 - [45] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, “The probabilistic model checker Storm,” *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 4, pp. 589–610, Aug. 2022.
 - [46] D. B. Kearns and R. Losick, “Cell population heterogeneity during growth of *bacillus subtilis*,” *Genes & development*, vol. 19 24, pp. 3083–94, 2005.
 - [47] T. Dijk and J. Pol, “Sylvan: Multi-core framework for decision diagrams,” *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 6, pp. 675–696, nov 2017. [Online]. Available: <https://doi.org/10.1007/s10009-016-0433-2>
 - [48] R. Roberts, T. Neupane, L. Buecherl, C. J. Myers, and Z. Zhang, “STAMINA 2.0: Improving scalability of infinite-state stochastic model checking,” in *Verification, Model Checking, and Abstract Interpretation*, B. Finkbeiner and T. Wies, Eds. Cham: Springer International Publishing, 2022, pp. 319–331.

Conformance Testing for Stochastic Cyber-Physical Systems

Xin Qin¹, Navid Hashemi¹, Lars Lindemann¹, and Jyotirmoy V. Deshmukh¹

¹Thomas Lord Department of Computer Science, University of Southern California, Los Angeles, USA
{xinqin, navidhas, llindema, jdeshmukh}@usc.edu

Abstract—Conformance is defined as a measure of distance between the behaviors of two dynamical systems. The notion of conformance can accelerate system design when models of varying fidelities are available on which analysis and control design can be done more efficiently. Ultimately, conformance can capture distance between design models and their real implementations and thus aid in robust system design. In this paper, we are interested in the conformance of stochastic dynamical systems. We argue that probabilistic reasoning over the distribution of distances between model trajectories is a good measure for *stochastic conformance*. Additionally, we propose the *non-conformance risk* to reason about the risk of stochastic systems not being conformant. We show that both notions have the desirable *transference* property, meaning that conformant systems satisfy similar system specifications, i.e., if the first model satisfies a desirable specification, the second model will satisfy (nearly) the same specification. Lastly, we propose how stochastic conformance and the non-conformance risk can be estimated from data using statistical tools such as conformal prediction. We present empirical evaluations of our method on an F-16 aircraft, an autonomous vehicle, a spacecraft, and Dubin's vehicle.

I. INTRODUCTION

Cyber-physical systems (CPS) are usually designed using a model-based design (MBD) paradigm. Here, the designer models the physical parts and the operating environment of the system and then designs the software used for perception, planning, and low-level control. Such closed-loop systems are then rigorously tested against various operating conditions, where the quality of the designed software is evaluated against *model properties* such as formal design specifications (or other kinds of quantitative objectives). Examples of such property-based analysis techniques include requirement falsification [1]–[5], nondeterministic and statistical verification [6]–[13], and risk analysis [14], [15].

MBD is a fundamentally iterative process in which the designer continuously modifies the software to tune performance or increase safety margins, or change plant models to perform design space exploration [16], e.g., using model abstraction or simplification [17]–[19], or to incorporate new data [20]. Any change to the system model, however, requires repeating the property-based analyses as many times as the number of system properties. The fundamental problem that we consider in this paper is that of conformance [21]–[25]. The notion of conformance is defined w.r.t. the input-output behavior of a model. Typically, model inputs include exogenous disturbances or user-inputs to the model, user-controllable design parameters, and initial operating conditions. For a given input u , let $y = S(u)$ denote the observable behavior of the model S .

Furthermore, let $d(y_1, y_2)$ be a metric defined over the space of the model behaviors. For deterministic models, two models S_1, S_2 are said to be δ -conformant if for all inputs u it holds that $d(y_1, y_2) < \delta$ where $y_1 = S_1(u)$ and $y_2 = S_2(u)$ [22], [23], [25]. This notion of deterministic conformance is useful to reason about worst-case differences between models. However, most CPS applications use components that exhibit stochastic behavior; for example, sensors have measurement noise, actuators can have manufacturing variations, and most physical phenomena are inherently stochastic. The central question that this paper considers is: *What is the notion of conformance between two stochastic CPS models?*

There are some challenges in comparing stochastic CPS models; even if two models are repeatedly excited by the same input, the pair of model behaviors that are observed may be different for every such simulation. Thus, the observable behavior of a stochastic model is more accurately characterized by a distribution over the space of trajectories. A possible way to compare two stochastic models is to use measure-theoretic techniques to compare the distance between the trajectory distributions. A number of divergence measures such as the f-divergences, e.g., the Kullback-Leibler divergence and the total variation distance, or the Wasserstein distance may look like candidate tools to compare the trajectory distributions. However, we argue in this paper that a divergence is not the right notion to use to compare stochastic CPS models. There can be two stochastic models whose output trajectories are very close using any trajectory space metric, but the divergence between their trajectory distributions can be infinite. On the other hand, there can be two trajectory distributions with zero divergence for which the distance between trajectories can be arbitrarily far apart.

This raises an interesting question: how do we then compare two stochastic models? In this paper, we argue that probabilistic bounds derived from the distribution of the distances between model trajectories (excited by the same input) gives us a general definition of conformance that has several advantages, as outlined below. We complement this probabilistic viewpoint further and capture the risk that the distribution of the distances between model trajectories is large leveraging risk measures [26].

First, we show that two stochastic systems that are conformant under our definition inherit the property of *transference* [22]. In simple terms, transference is the property that if the first model has certain logical or quantitative properties, then

the second model also satisfies the same (or nearly same) properties. This property brings several benefits. Consider the scenario where probabilistic guarantees that a model has certain quantitative properties have been established after an extensive and large number of simulations. Ordinarily, if there were any changes made to the model, establishing such probabilistic guarantees would require repeating the extensive simulation-based procedure. However, transference allows us to potentially sample from existing simulations for the first model and sample a small number of simulations from the modified model to establish stochastic conformance between the models, thereby allowing us to establish probabilistic guarantees on the *second* model. We demonstrate examples of such transference w.r.t. quantitative properties arising from quantitative semantics of temporal logic specifications and control-theoretic cost functions.

Next, we show how we can efficiently compute these probabilistic bounds using the notion of *conformal prediction* [27], [28] from statistical learning theory. At a high-level, conformal prediction involves computing quantiles of the empirical distribution of non-conformity scores over a validation dataset to obtain prediction intervals at a given confidence threshold.

The contributions of this paper are summarized as follows:

- We define *stochastic conformance* as a probabilistic bound over the distribution of distances between model trajectories. We also define the *non-conformance risk* to detect systems that are at risk of not being conformant.
- We show that both notions have the desirable transference property, meaning that conformant systems satisfy similar system specifications.
- We show how stochastic conformance and the non-conformance risk can be estimated using statistical tools from risk theory and conformal prediction.

II. PROBLEM STATEMENT AND PRELIMINARIES

Consider the probability space (Ω, \mathcal{F}, P) where Ω is the sample space, \mathcal{F} is a σ -algebra¹ of Ω , and $P : \mathcal{F} \rightarrow [0, 1]$ is a probability measure. In this paper, our goal is to quantify conformance of stochastic systems, i.e., systems whose inputs and outputs form a probability space with an appropriately defined measure. Let the two stochastic systems be denoted by S_1 and S_2 . The inputs and outputs of stochastic systems are *signals*, i.e., functions from a bounded interval of positive reals known as the *time domain* $\mathbb{T} \subseteq \mathbb{R}_{\geq 0}$ to a metric space, e.g., the standard Euclidean metric. Each stochastic system S_i then describes an input-output relation $S_i : \mathcal{U} \times \Omega \rightarrow \mathcal{Y}$ where \mathcal{U} and \mathcal{Y} denote the sets of all input and output signals. We allow input signals² to be stochastic, and we use the notation $U : \mathbb{T} \times \Omega \rightarrow \mathbb{R}^m$ to denote a stochastic input signals.³ Modeling stochastic systems this way provides great flexibility, and S_i

can e.g., describe the motion of stochastic hybrid systems, Markov chains, and stochastic difference equations.

Assume now that we apply the input signal $U : \mathbb{T} \times \Omega \rightarrow \mathbb{R}^m$ to systems S_1 and S_2 , and let the resulting output signals be denoted by $Y_1 : \mathbb{T} \times \Omega \rightarrow \mathbb{R}^n$ and $Y_2 : \mathbb{T} \times \Omega \rightarrow \mathbb{R}^n$, respectively. We assume that the functions S_1 , S_2 , and U are measurable so that the output signals Y_1 and Y_2 are well-defined stochastic signals. One can hence think of Y_1 and Y_2 to be drawn from the distributions \mathcal{D}_1 and \mathcal{D}_2 , respectively, which are functions of the probability space (Ω, \mathcal{F}, P) as well as the functions S_1 , S_2 , and U . In this paper, we make no restricting assumptions on the functions S_1 , S_2 , and U , and consequently we make no assumptions on the distributions \mathcal{D}_1 and \mathcal{D}_2 .

Informal Problem Statement. Let Y_1 and Y_2 be stochastic output signals of the stochastic systems S_1 and S_2 , respectively, under the stochastic input signal U . How can we quantify closeness of the stochastic systems S_1 and S_2 under U ? To answer this question, we will explore different ways of defining system “closeness” of Y_1 and Y_2 , and we will present algorithms to compute these stochastic notions of closeness. A subsequent problem that we consider is related to transference of properties from one system to another system. Particularly, given a signal temporal logic specification, can we infer guarantees about the satisfaction of the specification of one system from another system if the systems are close under a suitable definition of closeness?

A. Distance Metrics and Risk Measures

To define a general framework for quantifying closeness of stochastic systems, we will use i) different signal metrics to capture the distance between individual realizations $y_1 := Y_1(\cdot, \omega)$ and $y_2 := Y_2(\cdot, \omega)$ of the stochastic signals Y_1 and Y_2 where $\omega \in \Omega$ is a single outcome, and ii) probabilistic reasoning and risk measures to capture stochastic conformance and non-conformance, respectively, under these signal metrics.

We first equip the set of output signals \mathcal{Y} with a function $d : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$ that quantifies distance between signals. A natural choice of d is a signal metric that results in a metric space (\mathcal{Y}, d) . We use general signal metrics such as the metric induced by the L_p signal norm for $p \geq 1$. Particularly, define $d_p(y_1, y_2) := \left(\int_{\mathbb{T}} \|y_1(t) - y_2(t)\|^p dt \right)^{1/p}$ so that the L_∞ norm can also be expressed as $d_\infty(y_1, y_2) := \sup_{t \in \mathbb{T}} \|y_1(t) - y_2(t)\|$.

It is now easy to see that a signal metric $d(Y_1, Y_2)$ evaluated over the stochastic signals Y_1 and Y_2 results in a distribution over distances between realizations of Y_1 and Y_2 . To reason over properties of $d(Y_1, Y_2)$, we will use probabilistic reasoning but we will also consider risk measures [26] as introduced next.

A risk measure is a function $R : \mathfrak{F}(\Omega, \mathbb{R}) \rightarrow \mathbb{R}$ that maps the set of real-valued random variables $\mathfrak{F}(\Omega, \mathbb{R})$ to the real numbers. Typically, the input of R indicates a cost. There exist various risk measures that capture different characteristic of the distribution of the cost random variable, such as the mean or the variance. However, we are particularly interested in tail risk measures that capture the right tail of the cost distribution, i.e., the potentially rare but costly outcomes.

¹A σ -algebra on a set Ω is a nonempty collection of subsets of Ω closed under complement, countable unions, and countable intersections.

²Probability spaces over signals are defined by standard notions of cylinder sets [6].

³We will instead of the probability measure P , defined over (Ω, \mathcal{F}) , use more generally the notation Prob to be independent of the underlying probability space that we induce, e.g., as a result of transformations via U .

In this paper, we particularly consider the value-at-risk VaR_β and the conditional value-at-risk $CVaR_\beta$ at risk level $\beta \in (0, 1)$. The VaR_β of a random variable $Z : \Omega \rightarrow \mathbb{R}$ is defined as

$$VaR_\beta(Z) := \inf\{\alpha \in \mathbb{R} \mid \text{Prob}(Z \leq \alpha) \geq \beta\},$$

i.e., $VaR_\beta(Z)$ captures the $1 - \beta$ quantile of the distribution of Z from the right. Note that there is an obvious connection between value-at-risk and chance constraints, i.e., it holds that $\text{Prob}(Z \leq \alpha) \geq \beta$ is equivalent to $VaR_\beta(Z) \leq \alpha$. The $CVaR_\beta$ of Z , on the other hand, is defined as

$$CVaR_\beta(Z) := \inf_{\alpha \in \mathbb{R}} (\alpha + (1 - \beta)^{-1} E([Z - \alpha]^+))$$

where $[Z - \alpha]^+ := \max(Z - \alpha, 0)$ and $E(\cdot)$ indicates the expected value. When the function $\text{Prob}(Z \leq \alpha)$ is continuous (in α), it holds that $CVaR_\beta(Z) = E(Z \mid Z \geq VaR_\beta(Z))$, i.e., $CVaR_\beta(Z)$ is the expected value of Z conditioned on the outcomes where Z is greater or equal than $VaR_\beta(Z)$. Finally, note that it holds that $VaR_\beta(Z) \leq CVaR_\beta(Z)$, i.e., $CVaR_\beta$ is more risk sensitive.

For our risk transference results that we present later, we will require that R is *monotone*, *positive homogeneous*, and *subadditive*:

- For two random variables $Z, Z' \in \mathfrak{F}(\Omega, \mathbb{R})$, the risk measure R is monotone if $Z(\omega) \leq Z'(\omega)$ for all $\omega \in \Omega$ implies that $R(Z) \leq R(Z')$.
- For a random variable $Z \in \mathfrak{F}(\Omega, \mathbb{R})$, the risk measure R is positive homogeneous if, for any constant $H \geq 0$, it holds that $R(HZ) = HR(Z)$.
- For two random variables $Z, Z' \in \mathfrak{F}(\Omega, \mathbb{R})$, the risk measure R is subadditive if $R(Z + Z') \leq R(Z) + R(Z')$.

We remark that the VaR_β and the $CVaR_\beta$ satisfies all three properties [26].

B. System specifications

To express specifications, we use Signal Temporal Logic (STL). Let $y : \mathbb{T} \rightarrow \mathbb{R}^n$ be a deterministic signal, e.g., a realization of the stochastic signal Y . The atomic elements of STL are predicates that are functions $\mu : \mathbb{R}^n \rightarrow \{\text{True}, \text{False}\}$. For convenience, the predicate μ is often defined via a predicate function $h : \mathbb{R}^n \rightarrow \mathbb{R}$ as $\mu(y(t)) := \text{True}$ if $h(y(t)) \geq 0$ and $\mu(y(t)) := \text{False}$ otherwise. The syntax of STL is recursively defined as

$$\phi ::= \text{True} \mid \mu \mid \neg\phi' \mid \phi' \wedge \phi'' \mid \phi' U_I \phi'' \quad (1)$$

where ϕ' and ϕ'' are STL formulas. The Boolean operators \neg and \wedge encode negations (“not”) and conjunctions (“and”), respectively. The until operator $\phi' U_I \phi''$ encodes that ϕ' has to be true until ϕ'' becomes true at some future time within the time interval $I \subseteq \mathbb{R}_{\geq 0}$. We derive the operators for disjunction ($\phi' \vee \phi'' := \neg(\neg\phi' \wedge \neg\phi'')$), eventually ($F_I \phi := \top U_I \phi$), and always ($G_I \phi := \neg F_I \neg \phi$).

To determine if a signal y satisfies an STL formula ϕ that is imposed at time t , we can define the semantics as a relation \models , i.e., $(y, t) \models \phi$ means that ϕ is satisfied. While the STL semantics are fairly standard [29], we recall them in Appendix A in [30]. Additionally, we can define robust

semantics $\rho^\phi(y, t) \in \mathbb{R}$ that indicate how robustly the formula ϕ is satisfied or violated [31], [32], see Appendix A in [30]. Larger and positive values of $\rho^\phi(y, t)$ hence indicate that the specification is satisfied more robustly. Importantly, it holds that $(y, t) \models \phi$ if $\rho^\phi(y, t) > 0$.

III. CONFORMANCE FOR STOCHASTIC INPUT-OUTPUT SYSTEMS

Our goal is now to quantify closeness of two stochastic systems S_1 and S_2 under the input U . We present our definitions for stochastic conformance and non-conformance risk upfront, and provide motivation for these afterwards.

Definition 1. Let $U : \mathbb{T} \times \Omega \rightarrow \mathbb{R}^m$ be a stochastic input signal, $S_1, S_2 : \mathcal{U} \times \Omega \rightarrow \mathcal{Y}$ be stochastic systems, and $Y_1, Y_2 : \mathbb{T} \times \Omega \rightarrow \mathbb{R}^n$ be stochastic output signals with $Y_1 := S_1(U, \cdot)$ and $Y_2 := S_2(U, \cdot)$. Further, let $\epsilon \in \mathbb{R}$ be a conformance threshold, $\delta \in (0, 1)$ be a failure probability, and $d : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$ be a signal metric. Then, we say that the systems S_1 and S_2 under the input U are (ϵ, δ) -conformant if

$$\text{Prob}(d(Y_1, Y_2) \leq \epsilon) \geq 1 - \delta. \quad (2)$$

Additionally, let $R : \mathfrak{F}(\Omega, \mathbb{R}) \rightarrow \mathbb{R}$ be a risk measure and $r \in \mathbb{R}$ be a risk threshold. Then, S_1 and S_2 under the input U are at risk of being r -non-conformant if

$$R(d(Y_1, Y_2)) > r. \quad (3)$$

Eq. (2) is referred to as stochastic conformance and Eq. (3) as non-conformance risk. Let us now motivate and discuss these two definitions. While the definition of conformance in equation (2) appears natural at first sight, there are at least two competing ways of defining stochastic conformance. First, as Y_1 and Y_2 are distributions, it would be possible to define conformance as $D(Y_1, Y_2)$ where D is a distance function that measures the difference between two distributions, such as a divergence (Kullback–Leibler or f -divergence). However, our definition provides an intuitive interpretation in the signal space where system specifications are typically defined, while it is usually difficult to provide such an interpretation for divergences between distributions. Additionally, the divergence between Y_1 and Y_2 may be unbounded (or zero) even when equation (2) holds (does not hold).

Proposition 1. There exist stochastic systems S_1 and S_2 and distance metrics d where equation (2): i) is satisfied for $\epsilon > 0$ and $\delta = 0$, i.e., w.p. 1, but where the divergence between the systems is unbounded, and ii) is not satisfied for any given $\epsilon > 0$ and $\delta \in (0, 1)$, but where the divergence between the systems is zero.

Proof. Let us first prove i). For simplicity, consider systems S_1 and S_2 where the stochastic input and output signals are defined over the time domain $\mathbb{T} := \{t_0, \dots, t_T\}$. Further, for all $t \in \mathbb{T}$ let $y_1(t) := 0$ and $y_2(t) := \epsilon$. Clearly, equation (2) is satisfied, e.g., for d_∞ . The distributions D_1 and D_2 (joint distributions of $Y_1(t)$ and $Y_2(t)$, respectively) are Dirac distributions centered at 0 and ϵ , respectively. The Kullback–Leibler divergence between these two distributions is ∞ [33].

Let us now prove ii). Let \mathbb{T} consist of a single time point for simplicity so that Y_1 and Y_2 are random variables defined over a sample space \mathbb{R} . Let D_1 and D_2 be the *same* uniform distribution over $[0, a]$. Clearly, the divergence between D_1 and D_1 is zero. We know that the distribution of $Y := Y_1 - Y_2$ has support on $[-a, a]$, and that the probability density function of Y is $p(y) := 1/a - |y|/a^2$. We can now compute $\text{Prob}(|y| \leq \epsilon) = 2\epsilon/a - \epsilon^2/a^2$. Given $\epsilon > 0$ and $\delta \in (0, 1)$, we pick an $\bar{\delta} \in (0, 1)$ such that $\bar{\delta} > \delta$. We then solve the quadratic equation $2\epsilon/a - \epsilon^2/a^2 = 1 - \bar{\delta}$ subject to the constraint that $\epsilon \leq a$. Consequently, we find that $a \geq \epsilon/(1 - \bar{\delta})(1 + \sqrt{\bar{\delta}})$ results in $\text{Prob}(|y| \leq \epsilon) < 1 - \delta$ so that (2) is not satisfied. \square

Another way of defining stochastic conformance was presented in [34] where the authors consider a task-specific definition of stochastic conformance where satisfaction probabilities are required to be approximately equal. In other words, two stochastic systems are called c -approximately probabilistically conformant if $|\text{Prob}((Y_1, \tau) \models \phi) - \text{Prob}((Y_2, \tau) \models \phi)| \leq c$. In this definition, it may happen that two systems are c -approximately probabilistically conformant for a small value of c , while the systems produce completely different behaviors and individual realizations y_1 and y_2 are vastly different. Additionally to not being task specific, our definition covers the risk of being r -non-conformant in equation (3).

Finally, we would like to remark that the definition of conformance in equation (2) is related to the definition of non-conformance risk in equation (3). In fact, when the risk measure R is the value-at-risk VaR_β , then we know that

$$VaR_\beta(d(Y_1, Y_2)) > r \Leftrightarrow \text{Prob}(d(Y_1, Y_2) \leq r) < \beta$$

since $VaR_\beta(d(Y_1, Y_2)) \leq r$ is equivalent to $\text{Prob}(d(Y_1, Y_2) \leq r) \geq \beta$ according to Section II. Consequently, if $\beta := 1 - \delta$ and $r := \epsilon$ then $VaR_\beta(d(Y_1, Y_2)) > r$ implies that the systems S_1 and S_2 under U are not (ϵ, δ) -conformant.

The notion of conformance in Definition 1 is useful when the input U describes internal inputs such as system parameters (an unknown mass), exogeneous disturbances from known sources, or initial system conditions. In other words, the distribution U is known, making U a *known unknown*. However, in case of external inputs that could be manipulated (e.g. user inputs that represent rare malicious attacks), the input U may be unknown, making U an *unknown unknown*. We therefore provide an alternative definition of conformance.

Definition 2. Let $U \in \mathcal{U}$ be an unknown deterministic input signal, $S_1, S_2 : \mathcal{U} \times \Omega \rightarrow \mathcal{Y}$ be stochastic systems, and $Y_1, Y_2 : \mathbb{T} \times \Omega \rightarrow \mathbb{R}$ be stochastic output signals with $Y_1 := S_1(U, \cdot)$ and $Y_2 := S_2(U, \cdot)$. Further, let $\epsilon \in \mathbb{R}$ be a conformance threshold, $\delta \in (0, 1)$ be a failure probability, and $d : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$ be a signal metric. Then, we say that the systems S_1 and S_2 are (ϵ, δ) -conformant if

$$\text{Prob}\left(\sup_{U \in \mathcal{U}} d(Y_1, Y_2) \leq \epsilon\right) \geq 1 - \delta. \quad (4)$$

Additionally, let $R : \mathfrak{F}(\Omega, \mathbb{R}) \rightarrow \mathbb{R}$ be a risk measure and $r \in \mathbb{R}$ be a risk threshold. Then, we say that the systems

S_1 and S_2 under the input U are at risk of being r -non-conformant if

$$R\left(\sup_{U \in \mathcal{U}} d(Y_1, Y_2)\right) > r. \quad (5)$$

Based on this definition, note that it will be inherently more difficult to verify Definition 2 compared to Definition 1 due to the sup-operator.

IV. TRANSFERENCE OF SYSTEM PROPERTIES UNDER CONFORMANCE

We expect two systems S_1 and S_2 that are (ϵ, δ) -conformant in the sense of Definitions 1 and 2 to have similar behaviors with respect to satisfying a given system specification. Therefore, we will define the notion of transference with respect to a performance function $C : \mathcal{Y} \rightarrow \mathbb{R}$ that measures how well a signal $y \in \mathcal{Y}$ satisfies this system specification. Towards capturing similarity between S_1 and S_2 with respect to C , the signal metric d has to be chosen carefully.

Definition 3. Let $d : \mathcal{Y} \times \mathcal{Y}$ be a signal metric and $C : \mathcal{Y} \rightarrow \mathbb{R}$ be a performance function. Then, we say that C is Hölder continuous w.r.t. d if there exists constants $H, \gamma > 0$ such that, for any two signals $y_1, y_2 : \mathbb{T} \rightarrow \mathbb{R}^n$, it holds that

$$|C(y_1) - C(y_2)| \leq H d(y_1, y_2)^\gamma \quad (6)$$

A specific example of the performance function C is the robust semantics ρ^ϕ of an STL specification ϕ . In fact, the robust semantics ρ^ϕ are Hölder continuous w.r.t. the sup-norm d_∞ for constants $H = 1$ and $\gamma = 1$ [35, Lemma 2]. For the convenience of the reader, we state the proof of [35, Lemma 2] with the notation used in this paper in Appendix B in [30]. The robust semantics are also Hölder continuous w.r.t. the Skorokhod metric, see Appendix C in [30]. A commonly used performance function in control is $C(y) = \int_0^T y(t)^\top y(t) dt$, and we note that this choice of C is Hölder continuous w.r.t. d_1 as shown in Appendix D in [30]. Finally, note that the Hölder continuity condition in equation (6) implies that, for any constants $c, \epsilon \in \mathbb{R}$, it holds that

$$(C(y_1) \geq c \wedge d(y_1, y_2) \leq \epsilon) \Rightarrow C(y_2) \geq c - H\epsilon^\gamma. \quad (7)$$

A. Transference under stochastic conformance

With the definition of C being Hölder continuous w.r.t. d , we can now derive a stochastic transference result under stochastic conformance as per Definition 1.

Theorem 1. Let the premises in Definitions 1 and 3 hold. Further, let the systems S_1 and S_2 under the input U be (ϵ, δ) -conformant so that equation (2) holds and let C be Hölder continuous w.r.t. d so that equation (6) holds. Then, it holds that

$$\begin{aligned} \text{Prob}(C(Y_1) \geq c) &\geq 1 - \bar{\delta} \Rightarrow \\ \text{Prob}(C(Y_2) \geq c - H\epsilon^\gamma) &\geq 1 - \delta - \bar{\delta}. \end{aligned}$$

Proof. By assumption, it holds that $\text{Prob}(d(Y_1, Y_1) \leq \epsilon) \geq 1 - \delta$ and $\text{Prob}(C(Y_1) \geq c) \geq 1 - \bar{\delta}$ so that we know that

$\text{Prob}(d(Y_1, Y_1) > \epsilon) \leq \delta$ and $\text{Prob}(C(Y_1) < c) \leq \bar{\delta}$. We can now apply the union bound over these two events so that

$$\text{Prob}(d(Y_1, Y_1) > \epsilon \vee C(Y_1) < c) \leq \delta + \bar{\delta}.$$

From here, we can simply see that

$$\text{Prob}(d(Y_1, Y_1) \leq \epsilon \wedge C(Y_1) \geq c) \geq 1 - \delta - \bar{\delta}.$$

Since C is Hölder continuous w.r.t. d , which implies that equation (7) holds, it is easy to conclude that $\text{Prob}(C(Y_2) \geq c - H\epsilon^\gamma) \geq 1 - \delta - \bar{\delta}$. \square

Theorem 1 tells us that i) (ϵ, δ) -conformance of systems S_1 and S_2 under U , and ii) Hölder continuity of the performance function C w.r.t. the metric d enables us to derive a probabilistic lower bound for the performance of system S_2 w.r.t. C from the performance of system S_1 .

We can derive a transference result similar to Theorem 1 when we assume that the systems S_1 and S_2 are (ϵ, δ) -conformant in the sense of Definition 2 instead of Definition 1.

Theorem 2. *Let the premises in Definitions 2 and 3 hold. Further, let the systems S_1 and S_2 be (ϵ, δ) -conformant so that equation (4) holds and let C be Hölder continuous w.r.t. d so that equation (6) holds. Then, it holds that*

$$\begin{aligned} \text{Prob}\left(\inf_{U \in \mathcal{U}} C(Y_1) \geq c\right) &\geq 1 - \bar{\delta} \Rightarrow \\ \text{Prob}\left(\inf_{U \in \mathcal{U}} C(Y_2) \geq c - H\epsilon^\gamma\right) &\geq 1 - \delta - \bar{\delta} \end{aligned}$$

Proof. By assumption, it holds that $\text{Prob}(\sup_{U \in \mathcal{U}} d(Y_1, Y_1) \leq \epsilon) \geq 1 - \delta$ and $\text{Prob}(\inf_{U \in \mathcal{U}} C(Y_1) \geq c) \geq 1 - \bar{\delta}$ so that we know that $\text{Prob}(\sup_{U \in \mathcal{U}} d(Y_1, Y_1) > \epsilon) \leq \delta$ and $\text{Prob}(\inf_{U \in \mathcal{U}} C(Y_1) < c) \leq \bar{\delta}$. We can now apply the union bound over these two events so that

$$\text{Prob}\left(\sup_{U \in \mathcal{U}} d(Y_1, Y_1) > \epsilon \vee \inf_{U \in \mathcal{U}} C(Y_1) < c\right) \leq \delta + \bar{\delta}.$$

From here, we can simply see that

$$\text{Prob}\left(\sup_{U \in \mathcal{U}} d(Y_1, Y_1) \leq \epsilon \wedge \inf_{U \in \mathcal{U}} C(Y_1) \geq c\right) \geq 1 - \delta - \bar{\delta}.$$

This equation tells us that, for each $U \in \mathcal{U}$, we have

$$\text{Prob}(d(Y_1, Y_1) \leq \epsilon \wedge C(Y_1) \geq c) \geq 1 - \delta - \bar{\delta}.$$

Since C is Hölder continuous w.r.t. d , we know that equation (6) holds for each $U \in \mathcal{U}$. Consequently, we can conclude that $\text{Prob}(\inf_{U \in \mathcal{U}} C(Y_2) \geq c - H\epsilon^\gamma) \geq 1 - \delta - \bar{\delta}$. \square

B. Transference under non-conformance risk

On the other hand, by considering the notion of r -non-conformance risk, we expect that two systems S_1 and S_2 that are not at risk of being r -non-conformant have a similar risk of violating a specification. Here, we define the risk of violating a specifications by following ideas from [14] as $R(-C(Y_1))$ and $R(-C(Y_2))$.

Theorem 3. *Let the premises in Definitions 1 and 3 hold. Further, let the systems S_1 and S_2 under the input U not be at risk of being r -non-conformant so that equation (3) does not hold (i.e., $R(d(Y_1, Y_2)) \leq r$) and let C be Hölder*

continuous w.r.t. d with $\gamma = 1$ so that equation (6) holds. If the risk measure R is monotone, positive homogeneous, and subadditive, it holds that

$$R(-C(Y_2)) \leq R(-C(Y_1)) + Hr.$$

Proof. We can derive the following chain of inequalities

$$\begin{aligned} R(-C(Y_2)) &\stackrel{(a)}{\leq} R(-C(Y_1) + Hd(Y_1, Y_1)) \\ &\stackrel{(b)}{\leq} R(-C(Y_1)) + R(Hd(Y_1, Y_1)) \\ &\stackrel{(c)}{=} R(-C(Y_1)) + HR(d(Y_1, Y_1)) \\ &\stackrel{(d)}{\leq} R(-C(Y_1)) + Hr \end{aligned}$$

where (a) follows since C is Hölder continuous w.r.t. d and since R is monotone, (b) follows since R is subadditive, and (c) follows since R is positive homogeneous, while the inequality (d) follows since S_1 and S_2 under U are not at risk of being r -non-conformant, i.e., $R(d(Y_1, Y_2)) \leq r$. \square

This result implies that the risk of system S_2 w.r.t. the performance function C is upper bounded by the risk of system S_1 w.r.t. C if the systems S_1 and S_2 are not at risk of being r -non-conformant. We remark that a similar result appeared in our prior work [35]. Here, we present these results in the more general context of conformance and extend the result as we use general performance functions C , which additionally requires R to be positive homogeneous. Additionally, we derive a transference result similar to Theorem 3 when we assume that the systems S_1 and S_2 are not at risk of being r -non-conformant in the sense of Definition 2 instead of Definition 1.

Theorem 4. *Let the premises in Definitions 2 and 3 hold. Further, let the systems S_1 and S_2 not be at risk of being r -non-conformant so that equation (5) does not hold (i.e., $R(\sup_{U \in \mathcal{U}} d(Y_1, Y_2)) \leq r$) and let C be Hölder continuous w.r.t. d with $\gamma = 1$ so that equation (6) holds. If the risk measure R is monotone, positive homogeneous, and subadditive, it holds that*

$$R(-\inf_{U \in \mathcal{U}} C(Y_2)) \leq R(-\inf_{U \in \mathcal{U}} C(Y_1)) + Hr.$$

Proof. We can derive the following chain of inequalities

$$\begin{aligned} R(-\inf_{U \in \mathcal{U}} C(Y_2)) &\stackrel{(a)}{\leq} R(-\inf_{U \in \mathcal{U}} C(Y_1) + H \sup_{U \in \mathcal{U}} d(Y_1, Y_1)) \\ &\stackrel{(b)}{\leq} R(-\inf_{U \in \mathcal{U}} C(Y_1)) + R(H \sup_{U \in \mathcal{U}} d(Y_1, Y_1)) \\ &\stackrel{(c)}{=} R(-\inf_{U \in \mathcal{U}} C(Y_1)) + HR(\sup_{U \in \mathcal{U}} d(Y_1, Y_1)) \\ &\stackrel{(d)}{\leq} R(-\inf_{U \in \mathcal{U}} C(Y_1)) + Hr \end{aligned}$$

where (a) follows since $-\inf_{U \in \mathcal{U}} C(Y_2) = \sup_{U \in \mathcal{U}} -C(Y_2)$, since C is Hölder continuous w.r.t. d , and since R is monotone, (b) follows since R is subadditive, and (c) follows since R is positive homogeneous. The inequality (d) follows since S_1 and S_2 are not at risk of being r -non-conformant, i.e., $\sup_{U \in \mathcal{U}} R(d(Y_1, Y_2)) \leq r$. \square

V. STATISTICAL ESTIMATION OF STOCHASTIC CONFORMANCE

We propose algorithms to compute stochastic conformance and the non-conformance risk. In practice, note that one will be limited to discrete-time stochastic systems to apply these algorithms.

A. Estimating stochastic conformance

To estimate stochastic conformance, we use conformal prediction which is a statistical tool introduced in [28], [36] to obtain valid uncertainty regions for complex prediction models without making assumptions on the underlying distribution or the prediction model [27], [37]–[40]. Let $Z, Z^{(1)}, \dots, Z^{(k)}$ be $k + 1$ independent and identically distributed random variables modeling a quantity known as the *nonconformity score*. Our goal is to obtain an uncertainty region for Z based on $Z^{(1)}, \dots, Z^{(k)}$, i.e., the random variable Z should be contained within the uncertainty region with high probability. Formally, given a failure probability $\delta \in (0, 1)$, we want to construct a valid uncertainty region over Z (defined in terms of a value \bar{Z}) that depends on $Z^{(1)}, \dots, Z^{(k)}$ such that $\text{Prob}(Z \leq \bar{Z}) \geq 1 - \delta$.

By a surprisingly simple quantile argument, see [27, Lemma 1], one can obtain \bar{Z} to be the $(1 - \delta)$ th quantile of the empirical distribution of the values $Z^{(1)}, \dots, Z^{(k)}$ and ∞ . By assuming that $Z^{(1)}, \dots, Z^{(k)}$ are sorted in non-decreasing order, and by adding $Z^{(k+1)} := \infty$, we can equivalently obtain $\bar{Z} := Z^{(p)}$ where $p := \lceil (k + 1)(1 - \delta) \rceil$ with $\lceil \cdot \rceil$ being the ceiling function.

We can now use conformal prediction to estimate stochastic conformance as defined in Definition 1 by setting $Z := d(Y_1, Y_2)$. We therefore assume that we have access to a calibration dataset D_{cal} that consists of realizations $y_1^{(i)}$ and $y_2^{(i)}$ from the stochastic signals $Y_1 \sim \mathcal{D}_1$ and $Y_2 \sim \mathcal{D}_2$, respectively.

Theorem 5. *Let the premises of Definition 1 hold and D_{cal} be a calibration dataset with datapoints $(y_1^{(i)}, y_2^{(i)})$ drawn from $\mathcal{D}_1 \times \mathcal{D}_2$. Further, define $Z^{(i)} := d(y_1^{(i)}, y_2^{(i)})$ for all $i \in \{1, \dots, |D_{\text{cal}}|\}$ and $Z^{(|D_{\text{cal}}|+1)} := \infty$, and assume that the $Z^{(i)}$ are sorted in non-decreasing order. Then, it holds that $\text{Prob}(d(Y_1, Y_2) \leq \bar{Z}) \geq 1 - \delta$ with \bar{Z} defined as $\bar{Z} := Z^{(p)}$ where $p := \lceil (|D_{\text{cal}}| + 1)(1 - \delta) \rceil$. Thus, the systems S_1 and S_2 under the input U are (ϵ, δ) -conformant if $\bar{Z} \leq \epsilon$.*

We see that checking stochastic conformance as defined in Definition 1 is computationally simple when we have a calibration dataset D_{cal} . Checking stochastic conformance as defined in Definition 2, however, is more difficult due to the existence of the sup-operator. To compute this notion of conformance, we make two assumptions: i) the set \mathcal{U} is compact, and ii) for every realization $\omega \in \Omega$, the function $d(Y_1(\cdot, \omega), Y_2(\cdot, \omega))$ is Lipschitz continuous with Lipschitz constant L . While knowledge of the Lipschitz constant L would presume knowledge about the closeness of the systems S_1 and S_2 , it would only provide a conservative over-approximation. We will, however, not need to know the Lipschitz constant L and estimate L instead along with probabilistic guarantees.

Algorithm 1 Conformance Estimation as per Definition 2

Input: Failure probability $\delta \in (0, 1)$ and grid size $\kappa > 0$
Output: \bar{Z} such that $\text{Prob}(\sup_{U \in \mathcal{U}} d(Y_1, Y_2) \leq \bar{Z} + L\kappa) \geq 1 - \delta$

- 1: Construct κ -net $\bar{\mathcal{U}}$ of \mathcal{U}
- 2: **for** $\bar{U} \in \bar{\mathcal{U}}$ **do**
- 3: Obtain calibration set $D_{\text{cal}}^{\bar{U}}$ consisting of realizations $(y_1^{(i)}, y_2^{(i)})$ under \bar{U}
- 4: Compute $\bar{Z}_{\bar{U}} := Z^{(p)}$ by applying Theorem 5 but instead using dataset $D_{\text{cal}}^{\bar{U}}$
- 5: $\bar{Z} := \max_{\bar{U} \in \bar{\mathcal{U}}} \bar{Z}_{\bar{U}}$

Our approach is summarized in Algorithms 1 and 2. Algorithm 1 computes \bar{Z} such that $\text{Prob}(\sup_{U \in \mathcal{U}} d(Y_1, Y_2) \leq \bar{Z} + L\kappa) \geq 1 - \delta$ when L is known and where κ is a gridding parameter, while Algorithm 2 estimates the Lipschitz constant. We present a description of these algorithms upfront and state their theoretical guarantees afterwards.

In line 1 of Algorithm 1, we construct a κ -net $\bar{\mathcal{U}}$ of \mathcal{U} , i.e., we construct a finite set $\bar{\mathcal{U}}$ so that for each $U \in \mathcal{U}$ there exists a $\bar{U} \in \bar{\mathcal{U}}$ such that $\bar{d}(U, \bar{U}) \leq \kappa$ where $\bar{d} : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}$ is a metric. For this purpose, simple gridding strategies can be used as long as the set \mathcal{U} has a convenient representation. Alternatively, randomized algorithms can be used that sample from \mathcal{U} [41]. In lines 2-4, we apply Theorem 5 for each element $\bar{U} \in \bar{\mathcal{U}}$. Therefore, we obtain realizations $(y_1^{(i)}, y_2^{(i)})$ from $\mathcal{D}_1 \times \mathcal{D}_2$ under \bar{U} (line 3). We then compute $\bar{Z}_{\bar{U}}$ so that $\text{Prob}(d(Y_1(\bar{U}, \cdot), Y_2(\bar{U}, \cdot)) \leq \bar{Z}_{\bar{U}}) \geq 1 - \delta$ (line 4). Finally, we set $\bar{Z} := \max_{\bar{U} \in \bar{\mathcal{U}}} \bar{Z}_{\bar{U}}$ (line 5).

In Algorithm 2, we compute \bar{L} such that $\text{Prob}(L \leq \bar{L}) \geq 1 - \delta_L$. We uniformly sample control inputs (U', U'') (line 2), obtain realizations (y_1', y_2') from $\mathcal{D}_1 \times \mathcal{D}_2$ under U' and realizations (y_1'', y_2'') from $\mathcal{D}_1 \times \mathcal{D}_2$ under U'' (line 3), and compute the non-conformity score $L^{(i)}$ (line 4). In line 5, we obtain an estimate \bar{L} of the Lipschitz constant L that holds with a probability of $1 - \delta_L$ over the randomness introduced in Algorithm 1.

Theorem 6. *Let the premises of Definition 2 hold. If the Lipschitz constant L of $d(Y_1(\cdot, \omega), Y_2(\cdot, \omega))$ is known uniformly over $\omega \in \Omega$, then, for a gridding parameter $\kappa > 0$, the output \bar{Z} of Algorithm 1 ensures that*

$$\text{Prob}(\sup_{U \in \mathcal{U}} d(Y_1, Y_2) \leq \bar{Z} + L\kappa) \geq 1 - \delta$$

Thus, the systems S_1 and S_2 are (ϵ, δ) -conformant if $\bar{Z} + L\kappa \leq \epsilon$. Otherwise, let $\delta_L \in (0, 1)$ be a failure probability, then the output \bar{L} of Algorithm 2 ensures that

$$\text{Prob}(\sup_{U \in \mathcal{U}} d(Y_1, Y_2) \leq \bar{Z} + \bar{L}\kappa) \geq 1 - \delta - \delta_L$$

where Prob is defined over the randomness introduced in Algorithm 2.

Proof. From line 4 of Algorithm 1 we know that $\text{Prob}(d(Y_1(\bar{U}, \cdot), Y_2(\bar{U}, \cdot)) \leq \bar{Z}_{\bar{U}}) \geq 1 - \delta$ for each $\bar{U} \in \bar{\mathcal{U}}$.

Algorithm 2 Lipschitz Constant Estimation of L

Input: Failure probabilities $\delta_L \in (0, 1)$, grid size $\kappa > 0$, calibration size $K_L > 0$

Output: \bar{L} such that $\text{Prob}(\sup_{U \in \mathcal{U}} d(Y_1, Y_2) \leq \bar{L} + \bar{L}\kappa) \geq 1 - \delta - \delta_L$

- 1: **for** i from 1 to K_L **do**
- 2: Sample (U', U'') uniformly from $\mathcal{U} \times \mathcal{U}$
- 3: Obtain realizations (y'_1, y'_2) under U' and (y''_1, y''_2) under U''
- 4: Compute $L^{(i)} := |d(y'_1, y'_2) - d(y''_1, y''_2)| / \bar{d}(U', U'')$
- 5: Compute $\bar{L} := L^{(p)}$ where $p := \lceil (K_L + 1)(1 - \delta'') \rceil$

Due to Lipschitz continuity, we can conclude that for each $U \in \mathcal{U}$ that is such that $\bar{d}(U, \bar{U}) \leq \kappa$ it holds that

$$\text{Prob}(d(Y_1, Y_2) \leq \bar{Z}_{\bar{U}} + L\kappa) \geq 1 - \delta.$$

Since $\bar{\mathcal{U}}$ is a κ -net of \mathcal{U} , it follows that $\text{Prob}(\sup_{U \in \mathcal{U}} d(Y_1, Y_2) \leq \bar{Z} + L\kappa) \geq 1 - \delta$.

For the second part of the proof, note that from line 5 of Algorithm 2 we know that $\text{Prob}(L \leq \bar{L}) \geq 1 - \delta_L$. We can now union bound over this event and $\text{Prob}(d(Y_1(\bar{U}, \cdot), Y_2(\bar{U}, \cdot)) \leq \bar{Z}_{\bar{U}}) \geq 1 - \delta$ so that

$$\text{Prob}(d(Y_1(\bar{U}, \cdot), Y_2(\bar{U}, \cdot)) \leq \bar{Z}_{\bar{U}} \wedge L \leq \bar{L}) \geq 1 - \delta - \delta_L.$$

The rest of the proof follows as in the first part. \square

B. Estimating non-conformance risk

We next briefly summarize how to estimate the value-at-risk and the conditional value-at-risk following standard results such as from [14], [42] and [43], respectively.

Proposition 2. *Let the premises of Definition 1 hold and D_{cal} be a calibration dataset with datapoints $(y_1^{(i)}, y_2^{(i)})$ drawn from $\mathcal{D}_1 \times \mathcal{D}_2$. Let $\beta \in (0, 1)$ be a risk level and $\gamma \in (0, 1)$ be a failure threshold. Define $Z^{(i)} := d(y_1^{(i)}, y_2^{(i)})$ for each $i \in \{1, \dots, |D_{cal}|\}$ and assume that $\text{Prob}(Z \leq \alpha)$ is continuous in α . Then,*

$$\text{Prob}(\underline{VaR}_\beta \leq VaR_\beta(d(Y_1, Y_2)) \leq \overline{VaR}_\beta) \geq 1 - \gamma.$$

where $\underline{VaR}_\beta := \inf \left\{ \alpha \in \mathbb{R} \mid \widehat{\text{Prob}}(Z \leq \alpha) - \sqrt{\frac{\ln(2/\gamma)}{2|D_{cal}|}} \geq \beta \right\}$ and $\overline{VaR}_\beta := \inf \left\{ \alpha \in \mathbb{R} \mid \widehat{\text{Prob}}(Z \leq \alpha) + \sqrt{\frac{\ln(2/\gamma)}{2|D_{cal}|}} \geq \beta \right\}$ with the empirical cumulative distribution function $\widehat{\text{Prob}}(Z \leq \alpha) := \frac{1}{|D_{cal}|} \sum_{i=1}^{|D_{cal}|} \mathbb{I}(Z^{(i)} \leq \alpha)$ and the indicator function \mathbb{I} .

For estimating $CVaR_\beta(Z)$, we assume that the random variable $d(Y_1, Y_2)$ has bounded support, i.e., that $\text{Prob}(d(Y_1, Y_2) \in [a, b]) = 1$. Note that $d(Y_1, Y_2)$ is usually bounded from below by $a := 0$ if d is a metric. To obtain an upper bound, we assume that the distance function saturated at b , e.g., by clipping values larger than b to b . In practice, this means that realizations that are far apart already have a large distance and are capped to b .

Proposition 3. *Let the premises of Definition 1 hold and D_{cal} be a calibration dataset with datapoints $(y_1^{(i)}, y_2^{(i)})$ drawn from $\mathcal{D}_1 \times \mathcal{D}_2$. Let $\beta \in (0, 1)$ be a risk level and $\gamma \in (0, 1)$ be a failure threshold. Define $Z^{(i)} := d(y_1^{(i)}, y_2^{(i)})$ for each $i \in \{1, \dots, |D_{cal}|\}$ and assume that $\text{Prob}(d(Y_1, Y_2) \in [a, b]) = 1$. Then, it holds that*

$$\text{Prob}(\underline{CVaR}_\beta \leq CVaR_\beta(d(Y_1, Y_2)) \leq \overline{CVaR}_\beta) \geq 1 - \gamma.$$

where $\overline{CVaR}_\beta := \widehat{CVaR}_\beta + \sqrt{\frac{5 \ln(3/\gamma)}{|D_{cal}|(1-\beta)}}(b - a)$ and $\underline{CVaR}_\beta := \widehat{CVaR}_\beta - \sqrt{\frac{11 \ln(3/\gamma)}{|D_{cal}|(1-\beta)}}(b - a)$ where the empirical estimate of $CVaR_\beta(Z)$ is $\widehat{CVaR}_\beta := \inf_{\alpha \in \mathbb{R}} (\alpha + (|D_{cal}|(1 - \beta))^{-1} \sum_{i=1}^{|D_{cal}|} [Z^i - \alpha]^+)$.

As a consequence of these two lemmas, we know that with a probability of $1 - \gamma$ the systems S_1 and S_2 under the input U are at risk of not being conformant if $\underline{VaR}_\beta \geq \alpha$ or $\overline{CVaR}_\beta \geq \alpha$ based on the risk measure of choice.

VI. CASE STUDIES

We now demonstrate the practicality of stochastic conformance and risk analysis through various case studies. For validation, if we obtain the value \bar{Z} using a conformal prediction procedure for a nonconformity score defined by the random variable Z , i.e., such that $\text{Prob}(Z \leq \bar{Z}) \geq 1 - \delta$. Then, given a test set D_{test} , the validation score is defined as $VS(Z) := |\{z \in D_{test} \mid z \leq \bar{Z}\}| / |D_{test}|$.

A. Dubin's car.

Dubin's car models the motion of a point mass vehicle. The state variables are the x and y position, θ denotes the steering angle and v the longitudinal velocity. While both θ and v are typically assumed to be control inputs, we adapt the case study from [44] where the angular velocity $\omega(t)$ at each time t is assumed to be given so that $\theta(t) := T_s \pi + \sum_{i=1}^t \omega(i) T_s$ where $T_s := 0.1s$. In this example, we assume that $\omega(i) := \frac{\pi}{50T_s}$ for $i \in [1, 25]$, and $\omega(i) := -\frac{\pi}{50T_s}$ for $i \in [26, 50]$. The velocity $v(t)$ is provided by a feedback controller. The dynamics are assumed to have additive white Gaussian noise $\eta^x(t), \eta^y(t) \sim \mathcal{N}(0, 0.005)$. The dynamical equations of motion are as described below:

$$\begin{aligned} x(t+1) &= x(t) + T_s v(t) \cos(\theta(t)) + \eta^x(t) \\ y(t+1) &= y(t) + T_s v(t) \sin(\theta(t)) + \eta^y(t) \end{aligned}$$

The two systems that we compare have two different feedback controllers. The first feedback controller uses the method from [44], [45] and the second controller uses the method from [46]. We plot a set of sampled trajectories in Fig. 1a. This figure also shows the set of initial states $\mathcal{I} := [-1, 0] \times [-1, 0]$. The controller aims to ensure that the system trajectory stays within a series of sets \mathcal{T}_1 through \mathcal{T}_{50} , the corresponding STL specification is $\phi_{dubin} := \bigwedge_{i=1}^{50} F_{[i-1, i]}([x_i \ y_i] \in \mathcal{T}_i)$. For the experiments that follow, we uniformly sampled initial states from \mathcal{I} and noise η^x, η^y from the described Gaussian distribution.

Effect of calibration set size. In the first experiment, we wish to benchmark the effect of the size of the calibration set D_{cal}

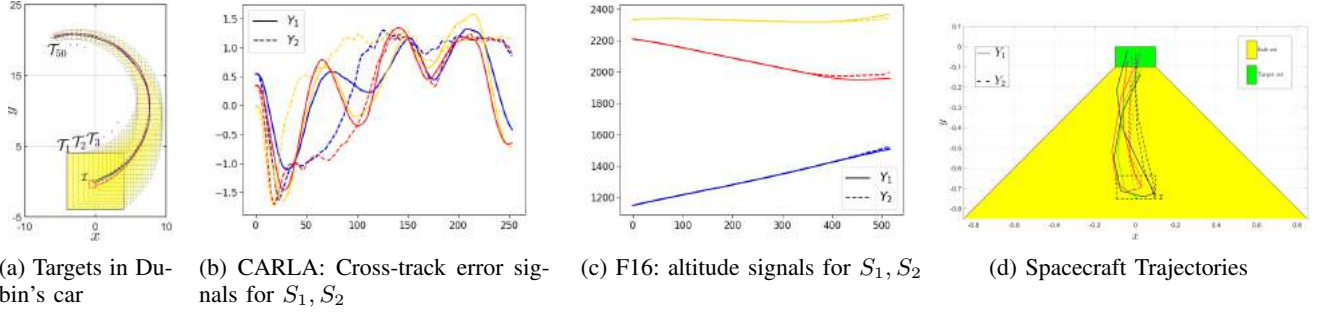


Fig. 1: The solid lines refer to Y_1 and the dashed lines refer to Y_2 ; in each of the displayed plots, the initial condition for each pair of realizations is the same.

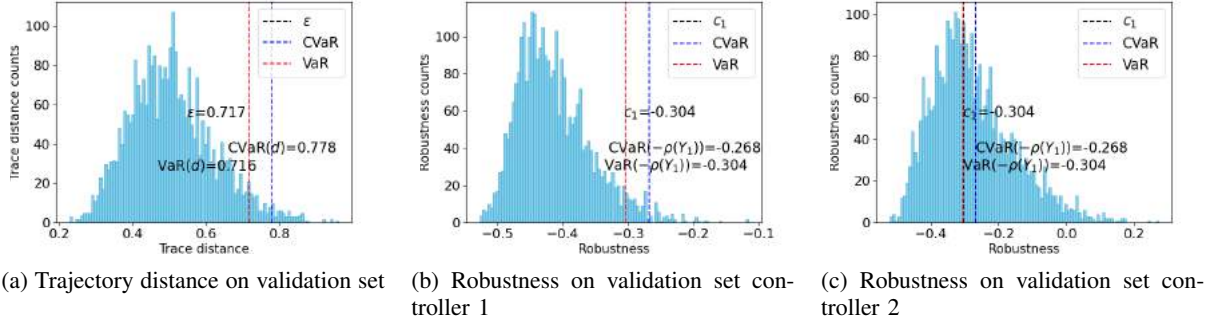


Fig. 2: Distance and robustness histogram for Dubin's car with $\delta = \bar{\delta} = 0.05$. We use $CVaR(d)$ to denote $CVaR(d(Y_1, Y_2))$. The c_1 and ϵ are the values of conformal prediction on the calibration set of $\rho^{\phi_{dubin}}(Y_1)$ and $d_\infty(Y_1, Y_2)$.

for various distance metrics. The results are shown in Table I. The table shows that with smaller sizes of the calibration set, we get a more conservative ϵ for d_∞ (which translates into a higher validation score). The VaR is almost identical to the value of ϵ at larger D_{cal} sizes. We note that the $CVaR$ values change with the value of VaR . A similar trend can be observed the Skorokhod distance and the L_2 -metric.

Empirical evaluation of transference. We empirically demonstrate that Theorem 1 holds. We use $C(Y) = \rho^{\phi_{dubin}}(Y)$, i.e., the robust semantics w.r.t. the property ϕ_{dubin} , and the L_∞ signal metric d_∞ . The results are shown in Table II. We can see that the predicted upper bound for the robustness of realizations of Y_2 w.r.t. ϕ_{dubin} is negative ($c_1 - \epsilon$), so it is not possible to conclude that the second system satisfies ϕ_{dubin} with probability greater than $1 - \delta - \bar{\delta}$. However, we note that c_2 is indeed greater than the bound ($c_1 - \epsilon$). Similarly, we show that Theorem 3 is also empirically validated by computing the $CVaR$ values for the first system and the risk measure on $d_\infty(Y_1, Y_2)$. We show the empirical distributions of $d_\infty(Y_1, Y_2)$, and $\rho^{\phi_{dubin}}(Y_i)$ for $i = 1, 2$ in Figure 2.

Empirical evaluation of Theorem 2. We next apply Algorithms 1 and 2 to this case study. We grid the initial set of states evenly into 25 cells with a grid size of $\kappa = 0.02$. We sample 650 trajectories on each cell to obtain their calibration sets. Algorithm 2 gives $\bar{Z} = 0.7562$ and $L\kappa = 0.0687$, giving $\bar{Z} + L\kappa = 0.8249$. We then evaluate on two test sets of unseen initial conditions with $|D_{test}| = 1000, 2500$. The success rate

on the test sets are 0.9996 and 1.0, with the goal success rate being 0.9. The experiments demonstrate the effectiveness of Theorem 6.

B. F-16 aircraft.

The F-16 aircraft control system from [47] uses a 13-dimensional non-linear plant model based on a 6 d.o.f. airplane model, and its dynamics describe force equations, moments, kinematics, and engine behavior. We alter the original system S_1 from [47] to a modified version S_2 by changing the controller gains. We evaluate the performance of the two systems on the ground collision avoidance scenario with the specification $\phi_{gas} := G_{[0,T]}(h \geq 1000)$ where T is the mission time and h is the altitude. For data collection, we perform uniform sampling of the initial states. We assume that the x-center of gravity (xcg) of the aircraft is a stochastic parameter with uniform distribution on $[0, 0.8]$. We obtain a calibration set D_{cal} of size 1000 by uniform sampling of the initial states and the xcg parameter. We separately sample 3000 signals for D_{test} . The results of transference and risk estimates are shown in Table III.

C. Autonomous Driving using the CARLA simulator.

CARLA is a high-fidelity simulator for testing of autonomous driving systems [48]. We consider two learning-based lane-keeping controllers from [14], one being an imitation learning controller (S_1) and another being a learned

Distance	$ D_{cal} $	ϵ	$d(Y_1, Y_2)$		
Metric			$VS(d(Y_1, Y_2))$	$Var(d(Y_1, Y_2))$	$CVaR(d(Y_1, Y_2))$
d_∞	50	0.7825	0.987	0.7183	0.7947
	1000	0.7163	0.956	0.7148	0.7647
	2000	0.7122	0.952	0.712	0.7814
	3000	0.7118	0.952	0.7117	0.7862
d_{sk} (Skorokhod Distance)	50	0.6723	0.953	0.6517	0.7181
	1000	0.6722	0.972	0.6711	0.7156
	2000	0.6645	0.96	0.6639	0.7106
	3000	0.6619	0.952	0.6613	0.7079
d_2	50	2.6086	0.937	2.503	2.612
	1000	2.7339	0.96	2.732	3.048
	2000	2.7071	0.944	2.706	3.044
	3000	2.7238	0.955	2.722	3.0929

TABLE I: Effect of calibration set size on the validation score and risk measures. The size of the test set, i.e., $|D_{test}|$, is 1000. We use the conformal prediction procedure from Section V to obtain ϵ as defined in Definition 1 for $\delta = 0.05$.

	$ D_{cal} $	c_1	ϵ	$VS(\rho_1)$	$VS(d_\infty)$	c_2	Thm 1	$CVaR$			Thm.3
							valid?	$-d_\infty$	$-\rho_1$	$-\rho_2$	valid?
$\delta = 0.2,$	100	0.31	0.59	0.95	0.76	0.21	Y	0.90	-0.28	0.00	Y
$\bar{\delta} = 0.05$	3K	0.30	0.60	0.95	0.79	0.20	Y	0.93	-0.27	0.03	Y
$\delta = 0.1,$	1K	0.30	0.67	0.96	0.92	0.15	Y	0.79	-0.27	0.02	Y
$\bar{\delta} = 0.05$	3K	0.30	0.66	0.95	0.91	0.15	Y	0.81	-0.27	0.03	Y
$\delta = 0.05,$	2K	0.31	0.71	0.94	0.95	0.11	Y	0.78	-0.27	0.02	Y
$\bar{\delta} = 0.05$	3K	0.30	0.71	0.95	0.95	0.11	Y	0.79	-0.27	0.03	Y

TABLE II: Empirical evaluation of transference. Let ρ_i be short-hand for $\rho^{\varphi_{dubin}}(Y_i)$ for $i = 1, 2$, and d_∞ be short-hand for $d_\infty(Y_1, Y_2)$. Using Theorem 5, we show $\text{Prob}(\rho_1 \geq c_1) > 1 - \bar{\delta}$, and $\text{Prob}(d_\infty \leq \epsilon) > 1 - \delta$. The validity scores for each guarantee on a test set D_{test} with 1000 samples are shown. The value c_2 is obtained using Theorem 5 on ρ_2 and observe that it exceeds $c_1 - \epsilon$, validating Theorem 1. Similarly, we report the $CVaR$ values for $-\rho_1$ and d_∞ , and $CVaR(-\rho_1) + CVaR(d_\infty) \geq CVaR(-\rho_2)$ for all cases, validating Theorem 3.

Case Study	Spec	$ D_{cal} $	$ D_{test} $	$VS(\rho_1)$	$VS(d_\infty)$	ϵ	$Var(d_\infty)$
F-16	ϕ_{gcas}	1K	3K	0.95	0.98	200	200
CARLA	ϕ_{cte}	700	300	0.94	0.96	1.88	1.87
Satellite	ϕ_{sat}	7K	3K	0.96	0.97	0.18	0.18

TABLE III: Transference results for various case studies. We use $\delta = 0.05$ and $\bar{\delta} = 0.05$. As before, ρ_1 is used as short-hand for $\rho^\phi(Y_1)$ for each spec, and d_∞ is used as short-hand for $d_\infty(Y_1, Y_2)$.

barrier function controller (S_2). We obtain 1000 trajectories from each controller during a 180 degree left turn, and we use 700 of them for calibration and 300 for testing. In this data, the initial states (c_e, θ_e) are drawn uniformly from $[-1, 1] \times [-0.4, 0.4]$ where c_e is the deviation from the center of the lane center (cross track error) and θ_e is the orientation error. The STL specification $\phi_{cte} := G(|c_e| \leq 2.25)$ restricts $|c_e|$ to be bounded by 2.25. The results are shown in Table III.

D. Spacecraft Rendezvous

Next, we consider a spacecraft rendezvous problem from [49]. Here, a deputy spacecraft is to rendezvous with a master spacecraft while staying within a line-of-sight cone. The system is a 4D model $s = [x, y, v_x, v_y]^\top$ where $x, y \in \mathbb{R}$ are the relative horizontal and vertical distances between the two spacecrafts and $v_x, v_y \in \mathbb{R}$ are the relative vertical and horizontal velocities. There are two different feedback controllers, using the same control algorithms we used in Dubin's car example (i.e., the controllers from [44], [45] and [46]). The STL specification is a reach-

Case Study	Spec	$ D_{cal} $	$ D_{test} $	δ	$CVaR$		
					d_∞	$-\rho_1$	$-\rho_2$
F-16	ϕ_{gcas}	1K	3K	0.01	200.3	-62.3	-62.3
CARLA	ϕ_{quad}	7K	3K	0.01	2.04	-0.31	0.88
Satellite	ϕ_{sat}	7K	3K	0.01	0.19	0.0	0.08

TABLE IV: Empirical validation of risk transference for all case studies. As before, ρ_i is short-hand for $\rho^\phi(Y_i)$, and d_∞ is short-hand for $d_\infty(Y_1, Y_2)$. Here, we set the risk level $\beta = \delta$ in each case.

avoid specification (visually depicted in Fig. 1d), which requires the system to always stay in the yellow region and eventually reach the target rectangle \mathcal{T} shown: $\phi_{sat} := G_{[1,5]}(y, |y|, |v_x|, |v_y| \leq -|x|, y_{max}, v_{x,max}, v_{y,max}) \wedge F_{[1,5]}^s \in \mathcal{T}$. The set of initial states is $\mathcal{I} = [-0.1, 0.1] \times [-0.1, 0.1]$. The system is assumed to have additive Gaussian process noise with zero mean and a diagonal covariance matrix with variances $10^{-4}, 10^{-4}, 5 \times 10^{-8}, 5 \times 10^{-8}$. We uniformly sample 100 different initial states from \mathcal{I} and 100 noise values sampled from the noise distribution. We divide the dataset into D_{cal} and D_{test} with sizes 7K and 3K respectively. The results are shown in Table III.

Discussion on results for Transference across case studies. We omit the column for Table III that shows the proportion of D_{test} of the realizations of Y_2 for which the bound $c_1 - \epsilon$ exceeds c_2 , where the c_i 's are the conformal bounds on ρ_i 's. For all case studies this ratio was either 1.0 or close to 1.0, establishing the empirical validity of Theorem 1. We also observe that above results show that it is feasible to use stochastic conformance in a control improvisation loop, where we want to change a system controller (perhaps for optimizing a performance objective) while allowing only some degradation on probabilistic safety guarantees.

VII. RELATED WORK

Conformance has found applications in cyber-physical system design [50], [51] as well as in drug testing and other applications [52]–[54]. Our work is inspired by existing works for **conformance of deterministic systems** by which we mean that systems are non-stochastic, see [21], [55] for surveys. The authors in [23]–[25] considered conformance testing between hybrid system. To capture distance between hybrid system trajectories that may exhibit discontinuities, signal metrics were considered that simultaneously quantify distance in space and time, resembling notions of system closeness in the hybrid systems literature [56], [57]. For instance, [23] proposes $(T, J, (\tau, \epsilon))$ -closeness where τ and ϵ capture both timing distortions and state value mismatches, respectively, and where T and J quantify limits on the total time and number of discontinuities, respectively. A stronger notion compared to $(T, J, (\tau, \epsilon))$ -closeness was proposed in [22] by using the Skoroghod metric. The benefit of [22] over the other notion is that it preserves the timing structure. All these works derive transference results with respect to timed linear temporal logic or metric interval temporal logic specifications.

Conformance of stochastic systems has been less studied. The authors in [58] propose precision and recall conformance measures based on the notion of entropy of stochastic automata. The authors in [59] use the Wasserstein distance to quantify distance between two stochastic systems, which is fundamentally different from our approach. (Bi)simulation relations for stochastic systems were studied in [60]–[62]. Such techniques can define behavioral relations for systems [63], [64], and they can be used to transfer verification results between systems [65]. The authors in [66] utilize such behavioral relations to verify RL policies between a concrete and an abstract system. We remark that bisimulations are difficult to compute, see e.g., [67], unlike our approach. Probably closest to our work is [34]. However, in this paper conformance is task specific which allows two systems to be conformant w.r.t. a system specification even when the systems produce completely different trajectories. Additionally, we consider a worst-case notion of conformance where no information about the input that excites both stochastic systems is available.

VIII. CONCLUSION

We studied conformance of stochastic dynamical systems. Particularly, we defined conformance between two stochastic systems as probabilistic bounds over the distribution of distances between model trajectories. Additionally, we proposed the non-conformance risk to reason about the risk of stochastic systems not being conformant. We showed that both notions have the transference property, meaning that conformant systems satisfy similar system specifications. Lastly, we showed how stochastic conformance and the non-conformance risk can be estimated from data using statistical tools such as conformal prediction.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their feedback. The National Science Foundation supported this work through the following grants: CAREER award (SHF-2048094), CNS-1932620, CNS-2039087, FMITF-1837131, CCF-SHF-1932620, the Airbus Institute for Engineering Research, and funding by Toyota R&D and Siemens Corporate Research through the USC Center for Autonomy and AI.

REFERENCES


- [1] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, “Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications,” *Lectures on Runtime Verification: Introductory and Advanced Topics*, pp. 135–175, 2018.

- [2] Q. Thibault, J. Anderson, A. Chandratte, G. Pedrielli, and G. Fainekos, "Psy-taliro: A python toolbox for search-based test generation for cyber-physical systems," in *Formal Methods for Industrial Critical Systems: 26th International Conference, FMICS 2021, Paris, France, August 24–26, 2021, Proceedings* 26. Springer, 2021, pp. 223–231.
- [3] T. Akazaki, S. Liu, Y. Yamagata, Y. Duan, and J. Hao, "Falsification of cyber-physical systems using deep reinforcement learning," in *Formal Methods: 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15–17, 2018, Proceedings* 22. Springer, 2018, pp. 456–465.
- [4] J. V. Deshmukh and S. Sankaranarayanan, "Formal techniques for verification and testing of cyber-physical systems," *Design Automation of Cyber-Physical Systems*, pp. 69–105, 2019.
- [5] X. Qin, N. Aréchiga, A. Best, and J. Deshmukh, "Automatic testing with reusable adversarial agents," *arXiv preprint arXiv:1910.13645*, 2021.
- [6] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.
- [7] Y. Wang, M. Zarei, B. Bonakdarpour, and M. Pajic, "Statistical verification of hyperproperties for cyber-physical systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–23, 2019.
- [8] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Proceedings of the International conference on runtime verification*, St. Julians, Malta, November 2010, pp. 122–135.
- [9] A. Legay and M. Viswanathan, "Statistical model checking: challenges and perspectives," *International Journal on Software Tools for Technology Transfer*, vol. 17, pp. 369–376, 2015.
- [10] G. Agha and K. Palmisano, "A survey of statistical model checking," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 28, no. 1, pp. 1–39, 2018.
- [11] A. Abate, A. Edwards, M. Giacobbe, H. Punchihewa, and D. Roy, "Quantitative verification with neural networks for probabilistic programs and stochastic systems," 2023.
- [12] Y. Zhao and K. Y. Rozier, "Probabilistic model checking for comparative analysis of automated air traffic control systems," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014, pp. 690–695.
- [13] X. Qin, Y. Xia, A. Zutshi, C. Fan, and J. V. Deshmukh, "Statistical verification of cyber-physical systems using surrogate models and conformal inference," in *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPs)*, 2022, pp. 116–126.
- [14] L. Lindemann, L. Jiang, N. Matni, and G. J. Pappas, "Risk of stochastic systems for temporal logic specifications," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 3, pp. 1–31, 2023.
- [15] P. Akella, A. Dixit, M. Ahmadi, J. W. Burdick, and A. D. Ames, "Sample-based bounds for coherent risk measures: Applications to policy synthesis and verification," *arXiv preprint arXiv:2204.09833*, 2022.
- [16] A. D. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Design & Test*, vol. 34, no. 1, pp. 77–90, 2016.
- [17] W. H. Schilders, H. A. Van der Vorst, and J. Rommes, *Model order reduction: theory, research aspects and applications*. Springer, 2008, vol. 13.
- [18] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas, "Discrete abstractions of hybrid systems," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 971–984, 2000.
- [19] C. Belta, B. Yordanov, and E. A. Gol, *Formal methods for discrete-time dynamical systems*. Springer, 2017, vol. 15.
- [20] A. S. Polydoros and L. Nalpantidis, "Survey of model-based reinforcement learning: Applications on robotics," *Journal of Intelligent & Robotic Systems*, vol. 86, no. 2, pp. 153–173, 2017.
- [21] H. Roehm, J. Oehlerking, M. Woehrle, and M. Althoff, "Model conformance for cyber-physical systems: A survey," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 3, pp. 1–26, 2019.
- [22] J. V. Deshmukh, R. Majumdar, and V. S. Prabhu, "Quantifying conformance using the skorokhod metric," in *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part II* 27. Springer, 2015, pp. 234–250.
- [23] H. Abbas, B. Hoxha, G. Fainekos, J. V. Deshmukh, J. Kapinski, and K. Ueda, "Conformance testing as falsification for cyber-physical systems," *arXiv preprint arXiv:1401.5200*, 2014.
- [24] H. Abbas, H. Mittelmann, and G. Fainekos, "Formal property verification in a conformance testing framework," in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2014, pp. 155–164.
- [25] H. Y. Abbas, *Test-based falsification and conformance testing for cyber-physical systems*. Arizona State University, 2015.
- [26] A. Majumdar and M. Pavone, "How should a robot assess risk? towards an axiomatic theory of risk in robotics," in *Robotics Research*. Springer, 2020, pp. 75–84.
- [27] R. J. Tibshirani, R. Foygel Barber, E. Candès, and A. Ramdas, "Conformal prediction under covariate shift," in *Proceedings of the Conference on Neural Information Processing Systems*, vol. 32, Vancouver, Canada, December 2019.
- [28] V. Vovk, A. Gammerman, and G. Shafer, *Algorithmic learning in a random world*. Springer Science & Business Media, 2005.
- [29] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proceedings of the Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, Grenoble, France, September 2004, pp. 152–166.
- [30] X. Qin, N. Hashemi, L. Lindemann, and J. V. Deshmukh, "Conformance testing for stochastic cyber-physical systems," *arXiv preprint arXiv:2308.06474*, 2023.
- [31] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems*, Klosterneuburg, Austria, September 2010, pp. 92–106.
- [32] G. E. Fainekos and G. J. Pappas, "Robustness of temporal logic specifications for continuous-time signals," *Theoretical Computer Science*, vol. 410, no. 42, pp. 4262–4291, 2009.
- [33] R. M. Gray, *Entropy and information theory*. Springer Science & Business Media, 2011.
- [34] Y. Wang, M. Zarei, B. Bonakdarpour, and M. Pajic, "Probabilistic conformance for cyber-physical systems," in *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems*, 2021, pp. 55–66.
- [35] M. Cleaveland, L. Lindemann, R. Ivanov, and G. J. Pappas, "Risk verification of stochastic systems with neural network controllers," *Artificial Intelligence*, vol. 313, p. 103782, 2022.
- [36] G. Shafer and V. Vovk, "A tutorial on conformal prediction," *Journal of Machine Learning Research*, vol. 9, no. 3, 2008.
- [37] A. N. Angelopoulos and S. Bates, "A gentle introduction to conformal prediction and distribution-free uncertainty quantification," *arXiv preprint arXiv:2107.07511*, 2021.
- [38] M. Fontana, G. Zeni, and S. Vantini, "Conformal prediction: A unified review of theory and new challenges," *Bernoulli*, vol. 29, no. 1, pp. 1 – 23, 2023.
- [39] J. Lei, M. G'Sell, A. Rinaldo, R. J. Tibshirani, and L. Wasserman, "Distribution-free predictive inference for regression," *Journal of the American Statistical Association*, vol. 113, no. 523, pp. 1094–1111, 2018.
- [40] M. Cauchois, S. Gupta, A. Ali, and J. C. Duchi, "Robust validation: Confident predictions even when distributions shift," *arXiv preprint arXiv:2008.04267*, 2020.
- [41] R. Vershynin, *High-dimensional probability: An introduction with applications in data science*. Cambridge university press, 2018, vol. 47.
- [42] P. Massart, "The tight constant in the dvoretzky-kiefer-wolfowitz inequality," *The annals of Probability*, pp. 1269–1283, 1990.
- [43] Y. Wang and F. Gao, "Deviation inequalities for an estimator of the conditional value-at-risk," *Operations Research Letters*, vol. 38, no. 3, pp. 236–239, 2010.
- [44] A. P. Vinod and M. M. Oishi, "Affine controller synthesis for stochastic reachability via difference of convex programming," in *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, 2019, pp. 7273–7280.
- [45] K. Lesser, M. Oishi, and R. S. Erwin, "Stochastic reachability for control of spacecraft relative motion," in *52nd IEEE Conference on Decision and Control*. IEEE, 2013, pp. 4705–4712.
- [46] M. P. Vitis and C. J. Tomlin, "On feedback design and risk allocation in chance constrained control," in *2011 50th IEEE Conference on Decision and Control and European Control Conference*. IEEE, 2011, pp. 734–739.
- [47] P. Heidlauf, A. Collins, M. Bolender, and S. Bak, "Verification challenges in f-16 ground collision avoidance and other automated maneuvers," in *ARCH@ ADHS*, 2018, pp. 208–217.
- [48] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the Conference on robot learning*. Mountain View, California: PMLR, November 2017, pp. 1–16.
- [49] A. P. Vinod, J. D. Gleason, and M. M. Oishi, "Sreachtools: a matlab stochastic reachability toolbox," in *Proceedings of the 22nd ACM*

international conference on hybrid systems: computation and control, 2019, pp. 33–38.

- [50] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts, “Benchmarks for model transformations and conformance checking,” in *1st International Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH)*, 2014.
- [51] H. Araujo, G. Carvalho, M. Mohaqeqi, M. R. Mousavi, and A. Sampaio, “Sound conformance testing for cyber-physical systems: Theory and implementation,” *Science of Computer Programming*, vol. 162, pp. 35–54, 2018.
- [52] R. Dimitrova, M. Gazda, M. R. Mousavi, S. Biewer, and H. Hermanns, “Conformance-based doping detection for cyber-physical systems,” in *Formal Techniques for Distributed Objects, Components, and Systems: 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings 40*. Springer, 2020, pp. 59–77.
- [53] S. Biewer, R. Dimitrova, M. Fries, M. Gazda, T. Heinze, H. Hermanns, and M. R. Mousavi, “Conformance relations and hyperproperties for doping detection in time and space,” *arXiv preprint arXiv:2012.03910*, 2020.
- [54] R. Dimitrova, M. Gazda, M. R. Mousavi, S. Biewer, and H. Hermanns, “Conformance-based doping detection for cyber-physical systems,” in *Formal Techniques for Distributed Objects, Components, and Systems*, A. Gotsman and A. Sokolova, Eds. Cham: Springer International Publishing, 2020, pp. 59–77.
- [55] N. Khakpour and M. R. Mousavi, “Notions of conformance testing for cyber-physical systems: Overview and roadmap,” in *26th International Conference on Concurrency Theory (CONCUR 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
- [56] R. Goebel, R. G. Sanfelice, and A. R. Teel, “Hybrid dynamical systems,” *IEEE control systems magazine*, vol. 29, no. 2, pp. 28–93, 2009.
- [57] —, “Hybrid dynamical systems: modeling stability, and robustness,” Princeton, NJ, USA, 2012.
- [58] S. J. Leemans and A. Polyvyanyy, “Stochastic-aware conformance checking: An entropy-based approach,” in *Advanced Information Systems Engineering: 32nd International Conference, CAiSE 2020, Grenoble, France, June 8–12, 2020, Proceedings 32*. Springer, 2020, pp. 217–233.
- [59] S. J. Leemans, A. F. Syring, and W. M. van der Aalst, “Earth movers’ stochastic conformance checking,” in *Business Process Management Forum: BPM Forum 2019, Vienna, Austria, September 1–6, 2019, Proceedings 17*. Springer, 2019, pp. 127–143.
- [60] A. A. Julius, A. Girard, and G. J. Pappas, “Approximate bisimulation for a class of stochastic hybrid systems,” in *2006 American Control Conference*. IEEE, 2006, pp. 6–pp.
- [61] A. A. Julius and G. J. Pappas, “Approximations of stochastic hybrid systems,” *IEEE Transactions on Automatic Control*, vol. 54, no. 6, pp. 1193–1203, 2009.
- [62] S. Haesaert and S. Soudjani, “Robust dynamic programming for temporal logic control of stochastic systems,” *IEEE Transactions on Automatic Control*, vol. 66, no. 6, pp. 2496–2511, 2020.
- [63] G. Bian and A. Abate, “On the relationship between bisimulation and trace equivalence in an approximate probabilistic context,” in *Foundations of Software Science and Computation Structures: 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 20*. Springer, 2017, pp. 321–337.
- [64] A. A. Julius and A. Van Der Schaft, “Bisimulation as congruence in the behavioral setting,” in *Proceedings of the 44th IEEE Conference on Decision and Control*. IEEE, 2005, pp. 814–819.
- [65] K. Zhang and M. Zamani, “Infinite-step opacity of nondeterministic finite transition systems: A bisimulation relation approach,” in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2017, pp. 5615–5619.
- [66] F. Delgrange, A. Nowe, and G. A. Pérez, “Wasserstein auto-encoded mdps: Formal verification of efficiently distilled rl policies with many-sided guarantees,” *arXiv preprint arXiv:2303.12558*, 2023.
- [67] A. Girard and G. J. Pappas, “Approximate bisimulation: A bridge between computer science and control theory,” *European Journal of Control*, vol. 17, no. 5–6, pp. 568–578, 2011.

MediK: Towards Safe Guideline-based Clinical Decision Support

Manasvi Saxena 
University of Illinois
Urbana, IL, United States
msaxena2@illinois.edu

Shuang Song
University of Illinois
Urbana, IL, United States
shuang3@illinois.edu

Lui Sha
University of Illinois
Urbana, IL, United States
lrs@illinois.edu

Abstract—Clinical Best Practice Guidelines (BPGs) are systematically developed, evidence-based statements published by medical institutions and associations that standardize diagnosis and treatment for various clinical scenarios. When expressed in an executable medium, BPGs can be utilized to build systems that assist healthcare professionals (HCPs) with situation-specific advice. Such systems, known as Guideline-based Clinical Decision Support Systems (CDSSs), have been shown to improve patient outcomes.

Several Domain-Specific Languages (DSLs) have been proposed to facilitate expressing BPGs in a computer-interpretable format that is easily comprehensible to HCPs. Given the safety-critical nature of CDSSs, the need for such languages to have complete formal semantics and an ecosystem of formal analysis tools has been recognized. Moreover, since these languages evolve over time to accommodate complexities in modeling BPGs, tools for them must also be adaptable to changes. But, existing languages lack complete formal semantics, or analysis tools derived from them.

This work introduces MediK: a new DSL for expressing BPGs with a complete executable formal semantics, and formal analysis tools, including a model checker, symbolic execution engine, and deductive verifier. As MediK's tools are derived from its semantics, any update to the language is automatically reflected across all tools. To evaluate our approach, we collaborated with a major pediatric hospital to develop a MediK-based CDSS for the screening and management of Pediatric Sepsis and validated that it satisfies desired safety properties. Our CDSS is Institutional Review Board (IRB) approved and is slated to undergo clinical simulations.

Index Terms—Semantics, Model checking

I. INTRODUCTION

Preventable Medical Errors (PMEs) characterized by incorrect intended treatment, or incorrect executions of intended treatment present a significant challenge in Healthcare [1]. According to a seminal report on the subject [2], in 1997, between 44,000 and 98,000 deaths were estimated to have been caused by PMEs in the United States alone. A more recent study analyzed data from the eight-year period between 2000 and 2008, and estimated that in 2013, the number of deaths caused by PMEs was more than 250,000, making PMEs the third-leading cause of death in the United States [3]. The adverse effects of PMEs extend beyond patient outcomes. One study estimated the financial burden of PMEs to the United

States to be 19.5 billion dollars in 2008 [4]. According to the authors of [1], PMEs caused psychological effects such as anger and guilt in healthcare providers (HCPs), adversely impacting their mental health.

One strategy to mitigate PMEs is to utilize evidence-based statements published by hospital and medical associations that codify recommended interventions for various clinical scenarios called Best Practice Guidelines (BPGs) [5]. High quality guidelines are routinely updated to account for results from clinical trials and advances in medicine, and make the latest diagnosis and treatment information accessible to providers [6].

While BPGs have the potential to reduce medical errors, their effectiveness hinges on the adherence of healthcare providers to them. For example, consider Advanced Cardiac Life Support (ACLS): a BPG published by the American Heart Association (AHA) for management of a life threatening condition called in-hospital cardiac arrest (IHCA) [7], [8]. Studies suggest that management of IHCA in 30% of adult, and 17% of pediatric cases deviates from the AHA-prescribed BPG, resulting in worse patient outcomes [9]–[13].

While BPG-adherence is difficult to achieve in practice [14], [15], integrating BPGs with existing patient care-flow, and making them readily-accessible when required can improve adherence [16]. To this end, hospitals commission computerized Decision Support Systems (CDSSs) that codify BPGs and support HCPs with situation-specific advice. Such systems have been shown to improve BPG-adherence [17], [18], and evidence from multi-center clinical trials suggests that they reduce PMEs [19], [20]. Thus, guideline-based CDSSs are now considered imperative to the future of medical decision making in general [21].

A guidelines-based CDSS usually consists of: (a) a translation of the guideline to an executable medium, called the knowledge-base, (b) an interface for user-interaction, and, (c) additional infrastructure that integrates with external data sources such as sensors, health records [22]. Typically, to develop a CDSS, domain experts in medicine collaborate with computer scientists to develop requirements documentation that presents the BPGs's semantics in a manner amenable to software development [23]. This documentation is then utilized to develop the knowledge-base, which is subsequently integrated with data sources (such as patient-parameter sensors

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1545002

and health records), and a User Interface (UI) to obtain a complete system. Thus, the BPG serves as a functional specification for the CDSS's knowledge-base. But, the aforementioned process has several limitations. First, the implementation, i.e., the knowledge-base may not concur with its specification, i.e., the text-based BPG. BPGs are specified as long, complex textual documents, where the exact meaning of terms may not be explicitly stated, and recommendations may be ambiguous [24]. Capturing and communicating these complexities via requirements documentation is challenging, and incorrect or incomplete documentation has resulted in failed implementations [25]. Second, as BPGs evolve to reflect new evidence or local adaptations, corresponding updates must be made to the CDSS as well. However, due to the gap between the BPG and the knowledge-base, effort must be expended into bringing the knowledge-base to reflect said updates.

To address above mentioned limitations, several Domain Specific Languages (DSLs) for directly expressing knowledge-base as Computer Interpretable Guidelines (CIGs) have been introduced. By providing mechanisms to facilitate representation of medical knowledge, such DSLs allow the CIG to serve as both the system specification, i.e. the BPG, and implementation, i.e. the knowledge-base. This ensures that there is no gap between the BPG and its executable counterpart. Given the safety-critical nature of CDSSs, the need for formally verified execution engines and analysis tools has been recognized. To this end, some existing DSLs have partially-defined semantics and support for verification via model-checking. However, as identified by the authors of [26], [27], existing languages lack a complete formal and executable semantics, interpreters or compilers with correctness guarantees, and a comprehensive suite of accompanying tools such as model-checkers, symbolic-execution engines, and deductive verifiers. The difficulties of formal analysis are further compounded by the fact that CDSSs are concurrent systems involving interactions with heterogeneous external agents such as sensors and HCPs, making their analysis challenging. We address these by introducing MediK (pronounced Medi-kay), a DSL for expressing a CDSS's knowledge-base as concurrently-executing state machines. MediK provides:

- 1) A *complete executable formal semantics* specified in the K semantics framework.
- 2) A *correct-by-construction interpreter*, and *analysis tools* such as a model-checker and deductive verifier.
- 3) A uniform way of modeling *heterogeneous agents* for both *execution* and *analysis*.

To evaluate our approach, we worked with the Children's Hospital of Illinois at OSF St. Francis Medical Center (referred to as OSF in the remainder of this work) to develop a CDSS for their pediatric sepsis management guidelines. The MediK-based system expresses the guideline *succinctly*, and allows establishing desired *safety* properties. To the best of our knowledge, ours is the first system for sepsis management with a set of safety guarantees.

We briefly describe the organization of this paper. In section II, we present a real-world BPG for management of sepsis, and use it to illustrate requirements that a DSL for encoding clinical guidelines must satisfy. In section III, we describe the MediK DSL, and illustrate how it addresses aforementioned requirements. To evaluate our approach, we utilized MediK to implement a real-world CDSS for pediatric sepsis management, which we describe in Section IV. In section V, we discuss how MediK builds on existing work, mention directions for future work in VI, and conclude in section VII.

II. MOTIVATING EXAMPLE

In this section, we introduce a real world BPG for management of sepsis in pediatric cases to motivate the need for Guidelines-based Clinical Decision Support Systems, and to illustrate characteristics that are desired of a DSL for such systems.

Sepsis is life-threatening condition caused by the body's extreme response to an infection [28], and is a major cause of morbidity and mortality in children [29]. Adverse outcomes can, however, be mitigated through timely identification and prompt treatment with antibiotics and intravenous (IV) fluids [30], [31]. BPGs for screening and management of sepsis in pediatric Emergency Departments (EDs) have shown effectiveness in screening and management of sepsis [29], leading to their adoption in many pediatric EDs [32], [33].

In Fig. 1, we present a simplified version of the screening section of OSF's sepsis management guideline. In essence, when a patient arrives at the ED with a fever or an infection, the HCP is supposed to obtain (a) the patient's age, (b) any conditions, such as cancer, immunosuppression, etc., that increase likelihood of sepsis, and (c) the patient's vital signs,

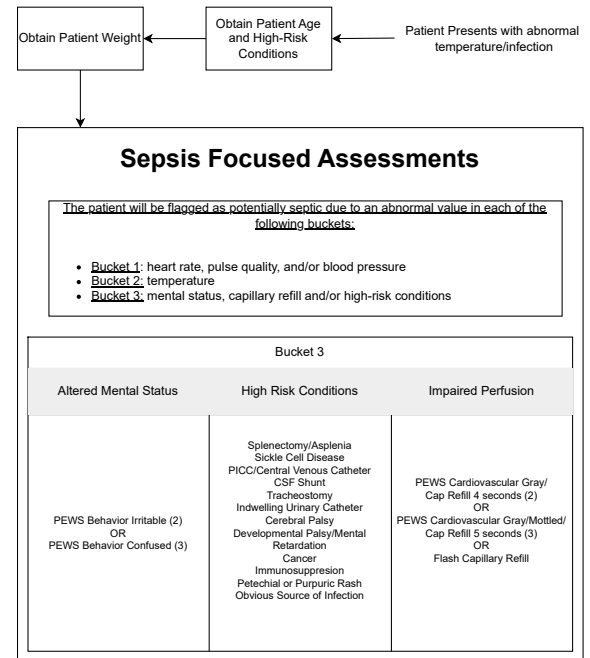


Fig. 1: Pediatric sepsis screening BPG

Age	Heart Rate	Systolic BP	Temp
$0d - 1m$	> 205	< 60	< 36 or > 38
$\geq 1m - 3m$	> 205	< 70	< 36 or > 38
$\geq 3m - 1y$	> 190	< 70	< 36 or > 38.5
...
$\geq 13y$	> 100	< 90	< 36 or > 38.5

TABLE I: Vital Signs Chart

such as heart rate, systolic blood pressure, respiratory rate, etc.

This information is then used to check for abnormalities in clusters of linked information, called “buckets”. For instance, if the patient’s heart rate is abnormal, then “bucket 1” is said to have an abnormal value. Checking for such abnormalities often involves the use of tables, such as TABLE I, that contain normal ranges indexed by *age*. If the patient has at least one abnormal value in every “bucket”, then he/she is flagged as potentially septic.

The BPG-recommended treatment for sepsis involves multiple concurrent workflows, such as screening for septic shock, fluid resuscitation, and administering antibiotics. In Fig. 2, we provide a version of the fluid resuscitation guideline used at OSF. Briefly, if the patient is flagged as potentially septic, the guideline suggests (i) obtaining any fluid-overload risks, (ii) administering normal saline (typically over a period of 15 minutes), where the dosage is dictated by risks determined in previous step, (iii) assessing signs of fluid-overload, (iv) evaluating patient responsiveness to normal saline upon completion of the administering process, and, (v) determining whether another fluid bolus should be administered based on information from previous steps.

This real-world BPG exhibits characteristics common across

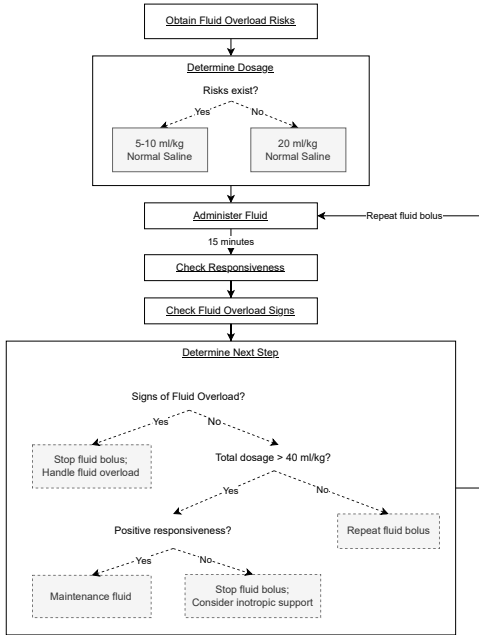


Fig. 2: Fluid Resuscitation Guideline

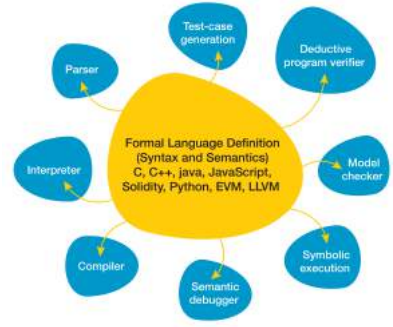


Fig. 3: \mathbb{K} Overview

many BPGs. Specifically BPGs typically:

- Involve *concurrent* workflows, such as administering drugs, monitoring vitals, performing treatment, etc. There may also be inter-workflow interactions. For instance, a diagnosis of sepsis during the screening may require modifications to an ongoing course antibiotics.
- Often specified in a *flowchart-like* notation. See [34] and [35] for other flowchart-based BPGs for management of *cardiac arrest*, and screening, risk-reduction, treatment and survivorship in cancer care respectively.
- Require communication between *heterogeneous agents* such as monitors and Electronic Health Records (EHRs).
- Often use *tables* indexed by parameters such as age, weight, etc to present normal/abnormal ranges for measurements, or recommended dosages for drugs.

Note that the aforementioned characteristics are *not* specific to one guideline. According to a review paper on CIGs [24], such DSLs should additionally (a) be formally defined, i.e. have a formal syntax and semantics, and (b) have an execution engine to provide decision support.

In the following sections, we describe how these characteristics dictate the design philosophy behind MediK. We argue that this philosophy makes MediK both intuitive to HCPs, and suitable for expressing complex guidelines.

III. MEDIK

In this section, we introduce the MediK DSL for expressing CIGs. MediK has designed to describe knowledge-base used in safety-critical systems. Thus, it is vital that:

- The interpreter is correct w.r.t. the formal semantics.
- The language has a comprehensive suite of formal program analysis tools.
- New features based on HCP feedback can be implemented quickly, conveniently, and correctly.

We achieve this by defining MediK (i.e., its syntax and semantics) in \mathbb{K} . \mathbb{K} is a rewriting-based framework for defining executable semantics of languages, type systems and formal analysis tools. It has been successfully used to define executable semantics of many real world languages such as C [36], Java [37], Javascript [38], and the Ethereum Virtual Machine [39]. We will introduce \mathbb{K} by need while discussing

MediK. For more details on K, we refer the reader to [40] [41].

The K ecosystem provides a suite of tools, such as an interpreter, model-checker, and deductive verifier that are parametric over the language’s semantics, as shown in Fig. 3. Thus, by defining the semantics of MediK in K, we obtain aforementioned tools for it without any extra effort. Additionally:

- The K-based interpreter for MediK essentially executes the language’s semantics rules, it is correct-by-construction.
- Incorporating changes to MediK only requires updating the semantics. Since the tools are derived from the semantics, they’re automatically updated.

The remainder of this section introduces MediK and describes how it’s designed around characteristics of BPGs from Section II. Recall that BPGs typically involve concurrent workflows, often expressed using a flowchart-like notation that may involve inter-workflow interactions. To ensure MediK programs are comprehensible to HCPs, they must be representable in a flowchart-like notation that HCPs are already comfortable with, and be capable of expressing inter-workflow interactions succinctly. To address these diverse requirements, we borrow from existing state-of-art languages for modeling large concurrent systems, like P [42], but make adaptations to make expressing and validating BPGs easier. We explore the differences to existing techniques in section V. In MediK, like in P, programs are expressed as concurrently executing instances of state machines that communicate via passing messages. Given a BPG where each workflow is expressed as a flowchart, we express said flowcharts as State Machines in MediK. Each flowchart node in the BPG is represented as a state in a state machine, and edges are represented as state transitions. During execution, instances of these machines are created, which interact with each other by passing events. Note the distinction between machine and its instance. A machine is analogous to an Object Oriented Programming (OOP) class, whereas its instance is analogous to an OOP object.

Next, we describe MediK using its K-framework definition. The K definition of a language has two components. The first is the language’s syntax, which is defined using a BNF-like notation. K utilizes this grammar to generate a parser for programs in the language. We describe MediK’s syntax in depth in Section III-A. The second is the semantics, which is defined using a K-configuration and rewrite rules. The K-configuration organizes the program’s execution state. Rewrite rules that operate over said configuration dictate the evolution of program state during execution. We describe the semantics in greater depth in Section III-B¹

A. Syntax

We use the skeleton of a MediK machine, and use it to describe the syntax. Note that we use [...] to denote

optional constructs, <...> for mandatory constructs, lowercase for terminals, and uppercase for non-terminals.

```

1 [init] machine <IDENTIFIER>
2   receives <IDENTIFIER_LIST> {
3     vars <IDENTIFIER_LIST>;
4
5   [init] state <IDENTIFIER> {
6     entry [(IDENTIFIER_LIST)] {
7       <STMT> // entry block
8     }
9     on <IDENTIFIER> [(IDENTIFIER_LIST)] do {
10      <STMT> // event handler
11    }
12  }
13 }
```

A MediK program consists of a set of machine definitions. A machine definition starts with the keyword `machine`, followed by its name (line 1). On line 2, following the `receives` keyword, is a comma-separated list of identifiers signifying the events that the machine can receive from other machines. One machine in a program can be prefixed with the `init` keyword. This machine is referred to as the initial machine. On line 3, following the keyword `vars`, another comma-separated list of identifiers signifies the instance-variables. During execution, each instance maintains a mapping from these variables to values. Each machine defines a set of states, such as the one in lines 5-11. A state has a name, an optional entry block (lines 6-8), and a set of event handlers (lines 9-11). The entry block begins with the keyword `entry`, and may contain a list of variables that are bound to values when the state is entered during execution. One state in the machine may be prefixed with `init`, specifying the initial state. When execution begins, an implicit instance of the initial machine is created, and the `entry` block of its initial state is executed. When an instance of a machine is dynamically created during runtime, the `entry` block of its initial state is executed. Event handlers within a state begin with `on` followed by the event name and an optional list of variables. When the event handler is executed, data from the received event’s payload is bound to aforementioned variables which can be used in the code block that follows the `do` keyword.

Within the entry and event handler code blocks, there may be statements. Below, we give a simplified version of the K-grammar for statements. In K, productions are defined using the keyword `syntax` (lines 1, 7). Terminals are enclosed in quotes (""), and non-terminals begin with an uppercase character.

```

1 syntax Exp ::= Id | Val | "this"
2 | Exp "." Exp
3 | "obtainFrom" "(" Exp "," Exp ")"
4 | "interval" "(" Exp "," Exp ")"
5 | Exp "in" Exp
6
7 syntax Stmt ::= Exp "=" Exp ";"
8 | "if" "(" Exp ")" Block "else" Block
9 | "new" Id "(" Exps ")" ";"
10 | "createFromInterface" "(" Id "," String ")" ";"
11 | "sleep" "(" Exp ")" ";"
12 | "send" Exp "," Id "," "(" Exps ")" ";"
13 | "broadcast" Id "," "(" Exps ")" ";"
14 | "goto" Id "(" Exps ")" ";"
15 | Exp "in" "{" CaseDecl "}"
```

Lines 1-5 define the syntax of MediK expressions. Line 1 defines basic expressions such as identifiers (denoted by the

¹The complete executable semantics is available at [43].

builtin \mathbb{K} production `Id`), values such as booleans, or rationals, or “this”, which enables an instance to refer to itself. Line 2 defines the usual dot operator (`.`), which can be used to access members of an instance. `obtainFrom` (line 3), `interval` (line 4) and `in` (line 5) are useful in context of defining BPGs, and are described through an example in section IV. Apart from these, Medi \mathbb{K} also supports common expressions such as `+`, `-`, `>`, `>=` over rationals and `&&`, `||` over booleans.

In lines 7-15, we define syntax for Medi \mathbb{K} statements. Some of these, such as variable assignment (line 7), `if-else` (line 8) and `new Id(..);` (line 9) are commonly found in other languages, and have expected meanings. The remaining statements (lines 10-15) have nuanced meanings in context of state machines. We shall go over these while discussing Medi \mathbb{K} ’s semantics in section III-B.

B. Semantics

Semantics of a language defined in \mathbb{K} has two components: (1) description of program state via \mathbb{K} -configurations, and (2) \mathbb{K} rules that dictate state evolution. Next we describe these components in detail.

1) **\mathbb{K} -Configuration:** \mathbb{K} represents program execution state using \mathbb{K} -configurations. A \mathbb{K} -configuration is an unordered list of (potentially nested) *cells*, specified using an XML-like notation. When declaring rules (as rewrites) over this state, any subset of the cells present in the configuration can be mentioned. This allows specifying only necessary parts of the state for a given rule, letting \mathbb{K} assume that the rest of the configuration remains unchanged. The following configuration defines the initial state for any Medi \mathbb{K} program:

```

1 configuration
2   <instance multiplicity="*" type="Map"> ...
3     <k> createMachineDefs ($PGM)
4       ~> createInitInstances </k>
5     <env> .Map </env>
6     <env> .Map </env>
7     <inBuffer> .List </inBuffer>
8     <activeState> . </activeState>
9   </instance>
10  <machine multiplicity="*" type="Map"> ...
11    <machineName> . </machineName>
12    <states>
13      <state multiplicity="*" type="Map">
14        <stateName> . </stateName>
15        <entryBlock> . </entryBlock>
16        <eventHandlers> ... </eventHandlers>
17      </state>
18    </states>
19  </machine>

```

The keyword `configuration` (line 1) defines a \mathbb{K} -configuration, followed by xml-like notation for the \mathbb{K} -cells. For example `<foo> ... </foo>` corresponds to a \mathbb{K} -cell with the name `foo`. The `<instance>` cell (lines 2-9) contains state of each Medi \mathbb{K} machine instance during execution. Each instance manages its instance variables using a map in the `<env>` cell (line 5), a buffer of incoming events in the `<inBuffer>` cell (line 7) and the currently executing code in the `<k>` cell (lines 3-4).² When a Medi \mathbb{K} program is executed, \mathbb{K} replaces `$PGM` (line 3) with the Abstract Syntax

Tree (AST) of the program, obtained by parsing the program using the syntax from section III-A. The `createMachineDefs` constructs is defined (using rewrite rules) to traverse the program AST and populate the configuration with information related to each machine. The `createInitInstances` creates an instance for the machine with the `init` keyword, leading to execution of the initial machine’s entry block. Note that `~>` symbol (line 3) is interpreted by \mathbb{K} as “followed-by”, i.e., execution of `createMachineDefs` is followed by execution of `createInitInstances`. The attribute `multiplicity="*" on lines 2 and 10 signifies that multiple copies of the corresponding cells, in this case <machine> and <instance> cells, can exist in the configuration during execution. This allows, during execution, for multiple machine definitions, each with multiple instances, to exist. The <machine> cell (lines 10-19) holds information relevant to a machine definition, such as the name in the <machineName> cell (line 11) and states in the <states> cell (lines 12-18). The <state> (lines 13-17) holds information relevant to a state, such as the entry block in cell <entryBlock> (line 15) and event handlers in cell <eventHandlers> (line 16).`

2) **\mathbb{K} -Rules:** \mathbb{K} -rules operate over the configuration and define the evolution of program state during execution. A \mathbb{K} -rule begins with the keyword `rule`, and is a statement of the form $\varphi \Rightarrow \psi$, where φ and ψ are patterns over configuration terms and \mathbb{K} -variables. We say φ is the *LHS* and ψ is the *RHS* of the rule. Let substitution θ be a map from \mathbb{K} -variables to terms. Say, for given pattern φ and substitution θ , $\varphi\theta$ be the term obtained by replacing each variable v in φ with $\theta(v)$. During execution, if the current configuration C , i.e. program execution state, matches φ with substitution θ , then it is rewritten to $\psi\theta$. We say pattern φ matches configuration C iff there exists a substitution θ s.t. $C = \varphi\theta$. For example, consider the following rule for updating the value of a local program variable.

```

1 rule <k> I:Id = V:Val => V ... </k>
2   <env> (I |-> Loc) ... </env>
3   <store> Store => Store[Loc <- V] </store>

```

Here, `I`, `V`, `Loc`, and `Store` are \mathbb{K} -variables. Note the distinction between program variables and \mathbb{K} -variables: while program variables are simply identifiers, \mathbb{K} -variables have logical meaning. The `...` is used to denote parts of the configuration not relevant to the rule. Typically, the top of the `k` cell contains the statement currently being executed. Suppose we’re executing the statment `i = 2;`. In this case, the current configuration will have a `k` cell of the form `<k> i = 2 ... </k>`, an environment cell `env` where variable `i` maps to some pointer `p`, and a store cell `store` containing a map `M` with some value pointed-to by `p`. The *LHS* matches with substitution $\theta = (I \mapsto i, V \mapsto 2, Loc \mapsto p, Store \mapsto M)$, resulting in the top of the `k` cell to be rewritten to the value 2, and pointer `p` updated to point to 2 in `M`. Note if there exist multiple rules that can match the current configuration, then one rule is non-deterministically chosen and applied. An execution is a sequence of rule applications that continues until no rule matches the configuration.

²For brevity, we present a simplified version of the configuration. See [43] for the entire configuration.

In the following sections we present several MediK constructs relevant to defining BPGs using their \mathbb{K} -rules. We first present the rule for sending and receiving messages.

```

1 rule
2 <instance>
3 <k> send instance(RecvId) , EventName:Id , ( Args )
4   => done ... </k> ...
5 </instance>
6 <instance>
7   <id> RecvId </id> ....
8   <inBuffer> ... (.List
9     => ListItem(
10       eventArgsPair(EventName | Args | Epoch + 1
11         )))
12   </inBuffer> ...
13 </instance>
14 <epoch> Epoch </epoch>

```

When the top of the k cell has `send`, the rule above (i) obtains the `id` of the receiver instance, the event name and the event arguments by matching the variables `RecvId`, `EventName` and `Args` against the current configuration (line 3), (ii) rewrites the top of the k cell (line 4) to `done`, marking the completion of execution for the construct, (iii) adds the event and associated arguments to the buffer of incoming events (lines 8-12) of the instance with `id RecvId` (line 7). (iv) The `epoch` decides when the machine can run, and is discussed in Section III-B2a.

To handle interaction with heterogeneous external sources, MediK models them as interfaces. An interface is a *FSM* that has its transition system defined externally. For example, certain measurements such as the heart rate are often obtained from sensors. The following code shows the process of obtaining external measurements in MediK.

```

1 interface HeartRateSensor { }
2
3 machine TreatmentMachine { ...
4   var hrSensor = createFromInterface(HeartRateSensor,
5     "heartRateSensor");
6   var heartRate = obtainFrom(hrSensor, "heartRate");
7 }

```

Since we don't have the transition system for the heart rate sensor, we declare it as an interface (line 1). Next, instead of using `new` to create an instance, we use a builtin MediK construct `createFromInterface`, which takes as arguments (a) the interface name (lines 4), (b) a unique identifier string used to identify the instance outside the MediK process. All other MediK machines can interact with external sensor using variable `hrSensor`. There is no need to make any distinction between external, and MediK-based machines. To deal with external interactions, input and output pipes are provided to the MediK process at launch. When the `send` construct is used on an external machine, MediK will write a JSON [44] message with the event data, the identifier from line 5, and a unique transaction id to the *write-end* of the output pipe. At the *read-end*, we need to write external code (in any programming language) to handle the JSON message. In the example above, this involves reading from the external heart rate sensor. To send data to MediK, a JSON message in a pre-specified format needs to be written to the *write-end* of the input pipe.

Next, we describe the rule for supporting tables in MediK. Once a measurement, such as the heart rate has been obtained from a sensor, we need to use a table, such as TABLE I

to check if the measurement is within a normal range. In MediK, we can write a function that does the required check, as shown in Fig. 4. In the code, if the `age` lies in any of the

```

1 fun isHeartRateNormal() {
2   days(age) in {
3     interval(days(0) , months(1)): return hr > 205;
4     interval(months(1), months(3)): return hr > 205;
5     // omitting other cases
6     default : return hr > 100;
7   }
8 }

```

Fig. 4: Checking abnormality using tables

intervals (closed on the left, open on the right) on lines 3-5, the corresponding statement to the right of the colon (`:`) is run. Otherwise line 6 is run. In MediK, the following rules are responsible for assigning semantics to the `in-interval` construct:

```

1 rule E in interval(L, U) => (E >= L) && (E < U)
2   [macro]
3 rule E in { interval(L, U): S:Stmt Cs:CaseDecl }
4   => if (E in interval(L, U)) {S} else {E in { Cs }}
5   [macro-rec]

```

Note the rules above are marked with the attributes `macro` (line 2) or `macro-rec` (line 5). This specifies that these constructs are not part of the language's semantics, but merely syntactic sugar. On line 1, we specify that `E in interval(L, R)` desugars to checking the expression `e` is between the lower and upper bound `L` and `U` respectively. Similarly we desugar each case statement to an `if-else` statement. In lines 3-5, we say that if the expression `E` is in `interval` with lower and upper bounds `L` and `U` respectively, then execute `S`, otherwise check `E` against the remaining cases `Cs`. Note the postfix `-rec` after `macro` specifies that the rule applies recursively, to desugar the remaining case statements.

a) *MediK Scheduling Semantics*: Since the \mathbb{K} -generated interpreter is single-threaded, MediK employs interleaving-semantics for concurrency, using a single `executor` thread shared between machine instances. A machine instance that is either at the start of an entry block, or has an event in the input buffer that it can handle is said to be *enabled*, i.e. one that can run once the `executor` becomes available. But, a naive strategy that non-deterministically chooses one *enabled* machine instance may lead to unfairness. Specifically, there may be situations where a machine instance is *enabled* but is never chosen for execution. Therefore, to ensure fairness, we use a scheduling strategy based on a monotonically increasing global counter called the *epoch*. We show this execution strategy in Fig. 5

Recall from Section III-A that a MediK program consists of a set of machines, of which one, prefixed with the keyword `init`, is the *initial* machine. Each machine has one state prefixed with `init`, referred to as the *initial* state. Let $P = \{\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_{n-1}\}$ be a program with n machines, where \mathcal{M}_0 is prefixed with `init`. MediK allows instances of a machine to be created dynamically at runtime. For machine $\mathcal{M}_i \in P$, let $\mathcal{I}_{\mathcal{M}_i, j-1}$ be its j -th instance.


```

1  $epoch \leftarrow 0$ 
2  $scheduled \leftarrow \{\mathcal{I}_{\mathcal{M}_0,0}^0\}$ 
3 while  $scheduled \neq \emptyset$  do
4    $\mathcal{I}_{\mathcal{M}_i,j}^\tau \leftarrow \text{choose}(scheduled) \text{ s.t.}$ 
      $\tau \leq epoch \wedge \text{enabled}(\mathcal{I}_{\mathcal{M}_i,j}^\tau)$ 
5    $scheduled \leftarrow scheduled \setminus \mathcal{I}_{\mathcal{M}_i,j}^\tau$ 
6    $\text{execute}(\mathcal{I}_{\mathcal{M}_i,j}^\tau, scheduled)$ 
7   if  $\nexists i', j', \tau' \text{ s.t. } (\mathcal{I}_{\mathcal{M}_{i'},j'}^{\tau'} \in scheduled)$ 
      $\wedge (\tau' \leq epoch) \wedge (\text{enabled}(\mathcal{I}_{\mathcal{M}_{i'},j'}^{\tau'}))$  then
8      $epoch \leftarrow epoch + 1$ 
9   end if
10 end while

```

Fig. 5: MediK Scheduling Semantics

Execution begins in *epoch* zero with the implicit (first) instance of the initial machine, denoted by $\mathcal{I}_{\mathcal{M}_0,0}^0$. We use $\mathcal{I}_{\mathcal{M}_i,j-1}^\tau$ to say that the j -th instance of machine \mathcal{M}_i is scheduled for execution in *epoch* τ . Recall that a state definition may have an entry block, containing code that is executed when the state is entered, or event handlers containing code that is executed when an event is dequeued from the input buffer. When execution begins, the entry block of the *initial state* of the *implicit instance* of the *initial machine* $\mathcal{I}_{\mathcal{M}_0,0}$ becomes scheduled (line 2) at *epoch* 0. On line 4, an instance $\mathcal{I}_{\mathcal{M}_i,j}^\tau$ is non-deterministically chosen from all machines that are both scheduled to run when $\tau \leq epoch$ and *enabled*. We use $\text{execute}(\mathcal{I}_{\mathcal{M}_i,j}^\tau, scheduled)$ on line 5 to denote this execution process. Execution of the entry or event handler block is atomic, i.e., a context-switch can only occur at the end of the block. Note that when a new instance of a machine is created using the keyword *new*, the *entry* block of the *initial state* of the *target* machine is executed synchronously before control returns to the source machine, and the instance is added to the multiset of *scheduled* machines. A context switch only occurs in three cases: *goto*, *sleep*, and *obtainFrom*, which we describe later.

During execution, if an instance $\mathcal{I}_{\mathcal{M}_i,j}$ sends an event to another instance $\mathcal{I}_{\mathcal{M}_{i'},j'}$, then the event is scheduled to be handled by $\mathcal{I}_{\mathcal{M}_{i'},j'}$ in or after the next epoch, i.e., $scheduled \leftarrow scheduled \cup \{\mathcal{I}_{\mathcal{M}_{i'},j'}^{epoch+1}\}$. Similarly, if a *goto* statement is encountered, the entry block of the target state is scheduled for execution at *epoch* + 1. If no other machine is both *scheduled* to run in the current *epoch*, and *enabled*, then the *epoch* advances by one (line 8).

b) Timer Semantics: Next, we discuss how MediK handles temporal aspects of BPGs. For instance, consider the Fluid resuscitation guideline BPG from Section 2. After administering fluids, the BPG recommends waiting for 15 minutes before evaluating their effectiveness. This waiting behavior in MediK is implemented using a `sleep(duration)` statement. Formalizing the execution semantics of such a statement in \mathbb{K} presents a challenge as \mathbb{K} does not provide builtin support for timers. Therefore, in MediK, `sleep(duration)` is described by the following rule:

```

1 rule <k> sleep(Duration: Int) ;
2   =>   jsonWrite( { "action"   : "sleep"
3                     , "duration" : Duration
4                     , "tid"      : TId }
5                     , ... )
6         ~> releaseExecutor
7         ~> waitForSleepResponse(TId) ...
8   </k>
9   <tidCount> TId => TId + Int 1 </tidCount>

```

`sleep` results in a JSON message being sent to a remote endpoint (lines 1-5) specified when the MediK process is launched. This mimics sending an event to an external *timer* machine, with the desired duration as the payload. At the remote endpoint, code must be provided (in any programming language) to parse the message, and respond with a JSON message indicating the expiration of the timer once the desired duration has passed. A unique transaction-id (lines 4, 7, 9), which the code at the endpoint is expected to provide in the response, uniquely identifies the machine instance being responded to. `sleep` causes a context-switch to occur on line 6, releasing the executor lock to process other *scheduled* machines.

When a message signifying the expiration of the timer is sent to the MediK process, along with the transaction id of source instance, the corresponding event signalling the completion of the sleep statement is placed at the *beginning* of the source machine instance's input buffer, and the instance is scheduled to resume execution in the next epoch. The following rule handles the external response:

```

1 rule
2   <k> waitForSleepResponse(TId) => . ... </k>
3   <inBuffer>
4     (ListItem(event($SleepDone | TId | Tau))
5     => .List) ...
6   </inBuffer>
7   <executorAvailable>
8     true => false
9   </executorAvailable>
10  <epoch> Epoch </epoch>
11    requires Tau <= Int Epoch

```

The `waitForSleepResponse(TId)` blocks execution until the external response indicating the expiration of the sleep timer is received in the input buffer (line 4). Once the response is received, the machine instance resumes execution when (a) the execution lock becomes available (indicated by `true` on line 8), and, (b) the epoch the instance was scheduled in (line 4) is less than or equal to the current epoch (lines 10-11).

An `obtainFrom` statement also results in a context switch. Just as in the case of `sleep`, a json message is sent to the remote endpoint, while the machine instance release the execution lock, and waits for a response. Once data for the requested field is available, it's communicated as an event to the MediK process, and the machine resumes execution.

IV. EVALUATION

A. Sepsis Management CDSS

To evaluate our approach, we collaborated with the Children's Hospital of Illinois at OSF St. Francis Medical Center to develop a MediK-based CDSS for screening and management of Pediatric Sepsis³.

³the entire CDSS for sepsis management is available at [45].

```

1 machine SepsisScreening receives .. {
2   init state Start {
3     on StartScreening do {
4       goto ObtainAge;
5     }
6   }
7   state ObtainAge {
8     entry {
9       send tablet, Instruct, ("get age");
10    } on ConfirmAgeEntered do {
11      goto ObtainWeight;
12    }
13  }
14  state ObtainWeight { ... }
15  state ObtainHighRiskConditions { ... }
16  state CalculateScore {
17    var hrAbnormal = !isInNormalRange("HR", ...);
18    var bucket1 = hrAbnormal || ...
19    var bucket3 = mentalStatusAbnormal || ...
20
21    var sepsisSuspected
22      = bucket1 && bucket2 && bucket3;
23
24    send tablet, SepsisDiagnosis
25      , (sepsisSuspected);
26  }
27 }

```

Fig. 6: Sepsis Screening in MediK

Recall from Fig. 1 the guideline for sepsis screening. In Fig. 6, we show MediK code corresponding to the sepsis screening guideline. When modeled in MediK, a flowchart in the guideline is represented using a MediK machine. *Nodes* in the flowchart are represented as *states* in a MediK machine, while flowchart *edges* as *state-transitions*. Note that we use *node* to refer to constructs in the flowchart, and *state* to refer to counterparts in MediK. Also, while it's desirable to represent each flowchart *node* as a state machine *state*, the task in the flowchart *node* may warrant using multiple state-machine *states*. For example, in Fig. 1, the step “Obtain Patient Age, Weight, and High Risk Conditions” is translated to states ObtainAge (lines 7-13), ObtainWeight (line 14), and ObtainHighRiskConditions (line 15) in Fig. 6. Within each of these states, the code permits communication with heterogeneous external agents for obtaining required parameters. For instance, on line 9, an Instruct event is sent to an external tablet machine with the payload "get age". The recipient process runs on a tablet held by the Healthcare Provider, and handles the event by prompting the provider to enter the patient's age. A ConfirmAgeEntered event, emitted once the age is obtained, enables the screening machine to proceed to the next step (lines 11-13). Once all appropriate measurements have been obtained, they are checked for abnormality (lines 18-26) using tables shown in Fig. 4 to arrive upon a diagnosis.

Recall from Section II that once a sepsis diagnosis has been arrived upon, one of the guideline suggested actions include administering fluids as shown in Fig. 2. In Fig. 7, we show the corresponding MediK code for administering fluids. The process starts when an external StartFluidTherapy event, corresponding to a button press by the HCP is received (line 6). The next steps include (a) obtaining any *risks* associated with administering fluids (lines 11-13), (b) suggesting an

appropriate *dose* to administer based on the risks, if any (lines 15-17), and, (c) waiting for the HCP to confirm that the suggested dose was administered (line 21). Once the dose is administered, the machine waits for the for 15 minutes as specified by the guideline (line 22), before prompting the HCP to evaluate the patient's responsiveness to the administered fluid dose (lines 27-38), and check for any signs of fluid

```

1 machine FluidTherapy
2   receives StartFluidTherapy, ... {
3
4
5   init state Start {
6     on StartFluidTherapy do {
7       goto ObtainRisks;
8     }
9   }
10
11  state ObtainRisks {
12    // Obtain fluid overload related risks
13  }
14
15  state SuggestFluidDosage {
16    // Suggest a dosage based on risks
17  }
18
19  state WaitForAdministerFluidConfirmation {
20    // Handler for Normal Saline Administration
21    on ConfirmNormalSalineAdministered do {
22      sleep(900);
23      goto EvaluateResponsiveness;
24    }
25  }
26
27  state EvaluateResponsiveness {
28    entry {
29      send tablet
30        , Instruct
31        , ("get responsiveness to fluids");
32    }
33
34    on FluidResponsivenessEntered(responsiveness) do {
35      isResponsiveToFluids = responsiveness;
36      goto ObtainFluidOverloadSigns;
37    }
38  }
39  state ObtainFluidOverloadSigns {
40    // Obtain signs of fluid overload
41  }
42
43  state AskNextStep {
44    entry {
45      var recommendation;
46      if (this.fluidOverload) {
47        recommendation = "handle fluid overload";
48      } else {
49        // obtain total saline dose
50        if ((totalSalineDose >
51            measurementBounds.salineDosageUpperBound) {
52          if (isResponseiveToFluids) {
53            recommendation = "maintainence fluids"
54          } else {
55            recommendation = "consider inotropic support";
56            broadcast ConsiderInotropicSupport;
57          }
58        } else {
59          recommendation = "repeat fluid bolus";
60        }
61      }
62      // Send recommendation to tablet
63      // Wait for HCP response
64    }
65  }
66 }

```

Fig. 7: Fluid Resuscitation in MediK

overload (lines 39-41). If the patient exhibits any signs of fluid overload, then a recommendation to handle the overload is made (line 47). Otherwise, the total dose of administered fluid is obtained from an external source (line 50). If the total dose is above the maximum allowed dose, then a recommendation based on the patient's responsiveness to administered fluids is made to either (a) reduce the fluid flow to maintenance levels (line 53), or, (b) switch to inotropic support to address circulatory issues is made (lines 52-58). If the total dose of administered fluids is less than the maximum allowed limit, then a recommendation to administer one more fluid bolus is made (lines 58-60).

Note that both the `SepsisScreening` and `FluidTherapy` machines structurally resemble their paper based counterparts in Fig. 1 and Fig. 2 respectively, making it easier for Healthcare Providers to comprehend and validate the code.

B. Formal Analysis using MediK

During execution of a MediK program, a machine may be considered *stuck* if an event at the head of its input buffer does not have an associated handler, rendering said machine non-responsive. For this reason, languages for modeling large concurrent systems, such as P [42] raise an exception for unhandled events. To mitigate such exceptions, we can enforce every machine to define event handlers for all possible events in all states, and use static analysis to detect possible violations. But, for MediK programs, we found that for complex CDSSs, such as the one for screening and management of sepsis: (a) it's tedious and error prone to define handlers for every event in every state, and, (b) it reduces the comprehensibility of the program, as many spurious event handlers that may never fire during execution have to be specified.

Thus, for MediK, we employ a weaker notion of responsiveness. We verify that every event that a state may possibly receive during execution must have a handler defined for it. This presents a challenge for reactive systems, or systems involve interactions with the external world, such as MediK-based CDSSs, as exploring the system's state space requires modeling the external components. In MediK, we address this by specifying external components as *ghost* machines - a technique also used by other state machine formalisms such as P [42]. For program analysis, *ghost* machines substitute external agents, permitting exploration of the state space. During execution, *ghosts* are discarded and replaced by actual external agents. Due to this, *ghost* machines may have statements to express non-determinism in processes. Consider, for instance, on a positive sepsis diagnosis, a HCP may chose to either administer fluids first, followed by antibiotics, or vice-versa. MediK supports such non-determinism using *either-or* statements as follows:

```
either {
  broadcast StartFluidTherapy;
  broadcast StartAntibioticTherapy;
} or {
  broadcast StartAntibioticTherapy;
  broadcast StartFluidTherapy;
}
```

When writing *ghosts*, values of measurements need to be abstract, to encompass all possible values that may be encountered during execution. For instance, when modeling entering a parameter such as the Heart Rate, we need to use an abstract value, representing all possible concrete values. To this end, we allow using an abstract value `#nondet` in *ghost* machines, with the following abstract semantics:

```
1 rule #nondet + _:Val => #nondet
2 rule _:Val <= #nondet => #nondet
3 rule #nondet && _ => #nondet
4 rule if (#nondet) Block => Block
5 rule if (#nondet) _ => .
```

The use of abstract encodings leads to a reduction of the state space. Recall from Section II that we needed to: (1) utilize patient's basic information such as age and weight to calculate normal ranges for clinical measurements such as blood pressure and heart rate, and, (2) calculate abnormality in clinical measurements using aforementioned ranges. For example, determining whether the patient's heart rate is abnormal is performed using the *in-interval* construct as show in Fig. 4. Recall that the *in-interval* construct is merely syntatic sugar for nested *if-else* statements. When using *ghost* machines for model checking, since the actual measurement is an abstract value, we know the final result of this abnormality checking operation is an abstract boolean value. Thus, instead of exploring each branch of *if-else* statements corresponding to *in-interval* constructs in Fig. 4, we replace the entire checking process with an final abstract boolean value. This reduces the state space but still allows us to explore all treatment options for both the normal and abnormal cases.

C. Model Checking the Sepsis CDSS

To verify responsiveness of the Sepsis CDSS, we implemented *ghost* machines for the external components using support for non-determinism and abstract values. We then added the following rule to the semantics, that takes a machine in an active state with an unhandled event at the head of the input buffer to a terminal *stuck* state.

```
1 rule
2 <k> handleEvents -> _ => stuck </k>
3 <activeState> ActiveState </activeState>
4 <class> MachineName </class>
5 <inBuffer>
6   ListItem(event(InputEvent | _ | _)) ...
7 </inBuffer> ...
8 </k>
```

We utilize the semantics-generated bounded model checker to search the state space to a depth of 300,000 for a *stuck* pattern, i.e., a machine that's no longer responsive. This depth we used was adequate for a complete run of the both the fluid and antibiotics machines simultaneously. The search command was executed on a machine with 64 GB of memory, and took roughly 90 minutes, and reported no such state was possible. To the best of our knowledge, this makes ours the first system for screening and management of sepsis with some formal safety guarantees.

V. RELATED WORKS

We broadly classify existing work into two categories: (a) languages/DSLs specifically developed to express BPGs in executable format, and, (b) languages/DSLs with a more general focus on expressing asynchronous event driven systems. We first focus on category (a). The Arden Syntax [46] is a widely used medium for expressing CIGs. Guidelines are described using Medical Logic Modules that contains information related to guideline's purpose, maintenance, and medical knowledge. But, Arden Syntax is focused on describing *simple*, modular, and independent guidelines (such as reminders), and not on guidelines with complex logic (such as treatment protocols) [47]. Arden Syntax's limitation in modeling complexity is addressed by GLIF [48]: a language that uses flowcharts to express guidelines. A multi-level approach is employed to manage complexity: at the top is the conceptual level, where only high-level details relevant for human-comprehension are present. In the middle is a computable-level, where details of guideline execution flow and patient data elements are specified. At the bottom is the implementable level, where institution-specific details and mappings into patient data are specified. Both Arden Syntax and GLIF eliminate the gap between the BPG, i.e. the specification, and the CIG, i.e. implementation as they're meant to be either directly used by clinicians (or in collaboration with computer scientists) to express BPGs in an executable medium. CIGs expressed in them are meant to be shared across hospitals, and are thus modular. However, neither formalism has *complete formal semantics*, or a *comprehensive suite* of formal analysis tools.

The need for formal analysis is identified by Asbru: a formalism with formally defined syntax and semantics [27]. In Asbru, a guideline is modeled as a plan that contains: (i) intentions that define aims, (ii) conditions that specify when the plan is applicable, (iii) effects that define expected behavior during execution, and, (iv) a body containing other subplans. Apart from an execution engine, the Asbru ecosystem also contains other tools, such as a model checker for verification [49]. However, the formal semantics of Asbru have been only partially defined, and is insufficient to implement tools for the language [26]. The importance of a complete formal-semantics is identified and addressed by PROforma [26], another formalism that uses plans to model guidelines. A PROforma plan is made of a sequence of tasks. The plan defines constraints on their enactment, and circumstances for termination (for example, exceptions) [26]. But, despite having complete formal semantics, PROforma's semantics is not executable. Therefore, an interpreter and analysis tools have to be implemented in an ad-hoc manner. Our work builds on these existing languages, and addresses their shortcomings by utilizing a *semantics-first* approach to build a DSL for expressing CIGs. This provides MediK with a *complete, executable* semantics, and a suite of *correct-by-construction* tools derived from it, such as an *interpreter, model checker* and *deductive verifier*.

Next we look at existing work for defining large concurrent

systems as State Machines. The closest project to this work, is probably the P language [42]. While P was considered for this project, it was given up for the lack of an executable semantics that would allow the language to quickly evolve to incorporate physician feedback. Moreover, until recently, P didn't even have a symbolic execution, or executable semantics based tools that can be derived automatically from the executable semantics, features that we plan to use in future work.

VI. FUTURE WORK

In this work, we introduced MediK, the first step towards building safe CDSSs. While MediK has been used to implement and analyze a real system, we're aware of many challenges that need addressing. Specifically (a) ghost machines may provide the ideal scenario for behavior of external agents, and may not take factors such as uncertainties into account, (b) the need to move beyond bounded model checking, and using deductive verification capabilities of K, (c) using symbolic execution to precisely trim unnecessary interleavings, and, (d) using semantics based compilation to extract inform, such as HCP-friendly diagrams from the code itself.

VII. CONCLUSION

Guideline-based Clinical Decision Support Systems (CDSSs) are now considered vital to the future of Medical Decision making in general, But, to find widespread adoption, guideline-based CDSSs must be held to the highest standards for safety-critical systems. While several advances have been made to CDSS over the years, several limitations have also been identified. This work fixed said limitations by introducing MediK - a new DSL for expressing BPGs that uses a semantics-first approach to build CDSS. MediK programs consist of *concurrently* executing *instances* of *State Machine*. MediK models external agents as machines with *transition systems* external to the program called *interfaces*, allowing for a *uniform* way of dealing with *heterogeneous external agents*. For program analysis, MediK allows modeling external agents via ghost machines that support *non-determinism*, enabling model-checking CDSSs for responsiveness. We collaborated with the Children's Hospital of Illinois at OSF St. Francis Medical Center to develop a system for screening and management of pediatric sepsis using MediK, and demonstrated it satisfies desired safety properties. To the best of our knowledge, our is the first system for sepsis management with any formal safety guarantees.

ACKNOWLEDGEMENT

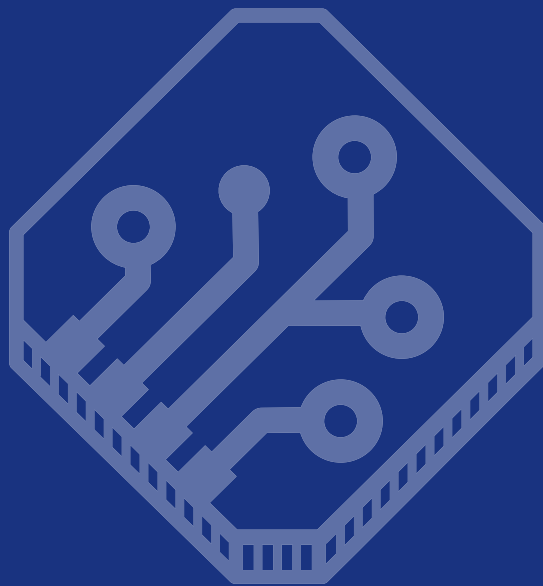
The authors would like to thank Dr. Johnathan A. Gehlbach and Dr. Paul M. Jeziorczak at the Children's Hospital of Illinois at OSF St. Francis Medical Center for their assistance with the pediatric sepsis CDSS. The authors extend their gratitude to Dr. Gehlbach for his assistance with evaluating the strengths and limitations of the MediK DSL.

REFERENCES

- [1] T. L. Rodziewicz, B. Houseman, and J. E. Hippskind, *Medical Error Reduction and Prevention*. StatPearls Publishing, Treasure Island (FL), 2022. [Online]. Available: <http://europepmc.org/books/NBK499956>
- [2] M. S. Donaldson, J. M. Corrigan, L. T. Kohn *et al.*, *To err is human: building a safer health system*. National Academies Press, 2000.
- [3] M. A. Makary and M. Daniel, "Medical error—the third leading cause of death in the us," *The BMJ*, vol. 353, 2016. [Online]. Available: <https://www.bmj.com/content/353/bmj.i2139>
- [4] C. Anel, S. L. Davidow, M. Hollander, and D. A. Moreno, "The economics of health care quality and medical errors," *Journal of health care finance*, vol. 39, no. 1, p. 39, 2012.
- [5] M. J. Field, K. N. Lohr *et al.*, *Clinical practice guidelines*. National Academies Press (US) Washington, DC, USA, 1990.
- [6] E. Steinberg, S. Greenfield, D. M. Wolman, M. Mancher, R. Graham *et al.*, *Clinical practice guidelines we can trust*. National Academies Press, 2011.
- [7] A. R. Panchal, J. A. Bartos, J. G. Cabañas, M. W. Donnino, I. R. Drennan, K. G. Hirsch, P. J. Kudenchuk, M. C. Kurz, E. J. Lavonas, P. T. Morley *et al.*, "Part 3: adult basic and advanced life support: 2020 american heart association guidelines for cardiopulmonary resuscitation and emergency cardiovascular care," *Circulation*, vol. 142, no. 16_Suppl_2, pp. S366–S468, 2020.
- [8] A. A. Topjian, T. T. Raymond, D. Atkins, M. Chan, J. P. Duff, B. L. Joyner Jr, J. J. Lasa, E. J. Lavonas, A. Levy, M. Mahgoub *et al.*, "Part 4: Pediatric basic and advanced life support: 2020 american heart association guidelines for cardiopulmonary resuscitation and emergency cardiovascular care," *Circulation*, vol. 142, no. 16_Suppl_2, pp. S469–S523, 2020.
- [9] J. P. Ornato, M. A. Peberdy, R. D. Reid, V. R. Feeser, H. S. Dhindsa, N. investigators *et al.*, "Impact of resuscitation system errors on survival from in-hospital cardiac arrest," *Resuscitation*, vol. 83, no. 1, pp. 63–69, 2012.
- [10] H. A. Wolfe, R. W. Morgan, B. Zhang, A. A. Topjian, E. L. Fink, R. A. Berg, V. M. Nadkarni, A. Nishisaki, J. Mensinger, R. M. Sutton *et al.*, "Deviations from aha guidelines during pediatric cardiopulmonary resuscitation are associated with decreased event survival," *Resuscitation*, vol. 149, pp. 89–99, 2020.
- [11] C. P. Crowley, J. D. Saliccioli, and E. Y. Kim, "The association between acls guideline deviations and outcomes from in-hospital cardiac arrest," *Resuscitation*, vol. 153, pp. 65–70, 2020.
- [12] K. Honarmand, C. Mephram, C. Ainsworth, and Z. Khalid, "Adherence to advanced cardiovascular life support (acls) guidelines during in-hospital cardiac arrest is associated with improved outcomes," *Resuscitation*, vol. 129, pp. 76–81, 2018.
- [13] M. D. McEvoy, L. C. Field, H. E. Moore, J. C. Smalley, P. J. Nietert, and S. H. Scarbrough, "The effect of adherence to acls protocols on survival of event in the setting of in-hospital cardiac arrest," *Resuscitation*, vol. 85, no. 1, pp. 82–87, 2014.
- [14] C. Rand, N. Powe, A. Wu, and M. Wilson, "Why don't physicians follow clinical practice guidelines," *Journal of the American Medical Association (JAMA)*, vol. 282, p. 14581465, 1999.
- [15] D. A. Davis and A. Taylor-Vaisey, "Translating guidelines into practice: a systematic review of theoretic concepts, practical experience and research evidence in the adoption of clinical practice guidelines," *Canadian Medical Association Journal (CMAJ)*, vol. 157, no. 4, pp. 408–416, 1997.
- [16] S. H. Woolf, R. Grol, A. Hutchinson, M. Eccles, and J. Grimshaw, "Potential benefits, limitations, and harms of clinical guidelines," *The BMJ*, vol. 318, no. 7182, pp. 527–530, 1999.
- [17] A. X. Garg, N. K. Adhikari, H. McDonald, M. P. Rosas-Arellano, P. J. Devereaux, J. Beyene, J. Sam, and R. B. Haynes, "Effects of computerized clinical decision support systems on practitioner performance and patient outcomes: a systematic review," *Journal of the American Medical Association (JAMA)*, vol. 293, no. 10, pp. 1223–1238, 2005.
- [18] K. Kawamoto, C. A. Houlihan, E. A. Balas, and D. F. Lobach, "Improving clinical practice using clinical decision support systems: a systematic review of trials to identify features critical to success," *The BMJ*, vol. 330, no. 7494, p. 765, 2005.
- [19] P. Bennett and N. R. Hardiker, "The use of computerized clinical decision support systems in emergency care: a substantive review of the literature," *Journal of the American Medical Informatics Association (JAMIA)*, vol. 24, no. 3, pp. 655–668, 12 2016.
- [20] N. Sahota, R. Lloyd, A. Ramakrishna, J. Mackay, J. Prorok, L. Weise-Kelly, T. Navarro-Ruan, N. Wilczynski, and b. Haynes, "Computerized clinical decision support systems for acute care management: A decision-maker-researcher partnership systematic review of effects on process of care and patient outcomes," *Implementation Science (IS)*, vol. 6, p. 91, 08 2011.
- [21] B. C. James, "Making it easy to do it right," *New England Journal of Medicine (NEJM)*, vol. 345, no. 13, pp. 991–993, 2001.
- [22] R. T. Sutton, D. Pincock, D. C. Baumgart, D. C. Sadowski, R. N. Fedorak, and K. I. Kroeker, "An overview of clinical decision support systems: benefits, risks, and strategies for success," *npj Digital Medicine*, vol. 3, no. 1, p. 17, 2020.
- [23] M. Peleg, "Computer-interpretable clinical guidelines: A methodological review," *Journal of Biomedical Informatics (JBI)*, vol. 46, no. 4, pp. 744–763, 2013.
- [24] P. A. De Clercq, J. A. Blom, H. H. Korsten, and A. Hasman, "Approaches for creating computer-interpretable guidelines that facilitate decision support," *Artificial Intelligence in Medicine (AIM)*, vol. 31, no. 1, pp. 1–27, 2004.
- [25] P. Kubben, M. Dumontier, and A. Dekker, *Fundamentals of clinical data science*. Springer, 2019.
- [26] D. R. Sutton and J. Fox, "The syntax and semantics of the pro forma guideline modeling language," *Journal of the American Medical Informatics Association (AMIA)*, vol. 10, no. 5, pp. 433–443, 2003.
- [27] Y. Shahar, S. Miksch, and P. Johnson, "An intention-based language for representing clinical guidelines," in *Proceedings of the AMIA Annual Fall Symposium*. American Medical Informatics Association, 1996, p. 592.
- [28] A. Rhodes, L. E. Evans, W. Alhazzani, M. M. Levy, M. Antonelli, R. Ferrer, A. Kumar, J. E. Sevransky, C. L. Sprung, M. E. Nunnally *et al.*, "Surviving sepsis campaign: international guidelines for management of sepsis and septic shock: 2016," *Intensive Care Medicine*, vol. 43, pp. 304–377, 2017.
- [29] M. Eisenberg, E. Freiman, A. Capraro, K. Madden, M. C. Monuteaux, J. Hudgins, and M. Harper, "Comparison of manual and automated sepsis screening tools in a pediatric emergency department," *Pediatrics*, vol. 147, no. 2, 2021.
- [30] S. L. Weiss, J. C. Fitzgerald, F. Balamuth, E. R. Alpern, J. Lavelle, M. Chilutti, R. Grundmeier, V. M. Nadkarni, and N. J. Thomas, "Delayed antimicrobial therapy increases mortality and organ dysfunction duration in pediatric sepsis," *Critical Care Medicine*, vol. 42, no. 11, p. 2409, 2014.
- [31] I. V. Evans, G. S. Phillips, E. R. Alpern, D. C. Angus, M. E. Friedrich, N. Kissoon, S. Lemeshow, M. M. Levy, M. M. Parker, K. M. Terry *et al.*, "Association between the new york sepsis care mandate and in-hospital mortality for pediatric sepsis," *Journal of American Medicine (JAMA)*, vol. 320, no. 4, pp. 358–367, 2018.
- [32] F. Balamuth, E. R. Alpern, M. K. Abbadessa, K. Hayes, A. Schast, J. Lavelle, J. C. Fitzgerald, S. L. Weiss, and J. J. Zorc, "Improving recognition of pediatric severe sepsis in the emergency department: contributions of a vital sign–based electronic alert and bedside clinician identification," *Annals of Emergency Medicine*, vol. 70, no. 6, pp. 759–768, 2017.
- [33] R. J. Sepanski, S. A. Godambe, C. D. Mangum, C. S. Bovat, A. L. Zaritsky, and S. H. Shah, "Designing a pediatric severe sepsis screening tool," *Frontiers in Pediatrics*, vol. 2, p. 56, 2014.
- [34] "Cpr and emergency cardiovascular care algorithms." [Online]. Available: <https://cpr.heart.org/en/resuscitation-science/cpr-and-ecc-guidelines/algorithms>
- [35] "Clinical practice algorithms." [Online]. Available: <https://www.mdanderson.org/for-physicians/clinical-tools-resources/clinical-practice-algorithms.html>
- [36] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of c," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 336–345.
- [37] D. Bogdănaş and G. Roşu, "K-Java: A complete semantics of Java," in *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, January 2015, pp. 445–456.
- [38] D. Park, A. Stefănescu, and G. Roşu, "Kjs: A complete formal semantics of javascript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 346–356.

- [39] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, “KEVM: A complete formal semantics of the Ethereum Virtual Machine,” in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217.
- [40] T. F. Șerbănuță, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roșu, “The K primer (version 3.3),” *Electronic Notes in Theoretical Computer Science*, vol. 304, pp. 57–80, 2014, proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).
- [41] G. Roșu and T. F. Șerbănuță, “An overview of the k semantic framework,” *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.
- [42] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, “P: Safe asynchronous event-driven programming,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 321–332. [Online]. Available: <https://doi.org/10.1145/2491956.2462184>
- [43] “The K semantics of MediK.” [Online]. Available: <https://github.com/fmcd-78/medik-semantics.git>
- [44] “Ecma-404 the json data interchange standard.” [Online]. Available: <https://www.json.org/json-en.html>
- [45] “Sepsis screening and management application.” [Online]. Available: <https://github.com/fmcd-78/psepsis.git>
- [46] G. Hripcsak, “Writing arden syntax medical logic modules,” *Computers in biology and medicine*, vol. 24, no. 5, pp. 331–363, 1994.
- [47] M. Peleg, A. A. Boxwala, E. Bernstam, S. Tu, R. A. Greenes, and E. H. Shortliffe, “Sharable representation of clinical guidelines in glif: relationship to the arden syntax,” *Journal of biomedical informatics*, vol. 34, no. 3, pp. 170–181, 2001.
- [48] A. A. Boxwala, M. Peleg, S. Tu, O. Ogunyemi, Q. T. Zeng, D. Wang, V. L. Patel, R. A. Greenes, and E. H. Shortliffe, “Glif3: a representation format for sharable computer-interpretable clinical practice guidelines,” *Journal of Biomedical Informatics (JBI)*, vol. 37, no. 3, pp. 147–161, 2004.
- [49] S. Bäumler, M. Balser, A. Dunets, W. Reif, and J. Schmitt, “Verification of medical guidelines by model checking – a case study,” in *Model Checking Software*, A. Valmari, Ed. Springer Berlin Heidelberg, 2006, pp. 219–233.

The Conference on Formal Methods in Computer-Aided Design (FMCAD) is an annual conference on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to researchers in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers formal aspects of computer-aided system design including verification, specification, synthesis, and testing.



ISBN 978-3-85448-060-0



www.tuwien.at/academicpress