

Ruzica Piskac / Michael W. Whalen (Eds.)

PROCEEDINGS OF THE 21ST CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2021



Academic Press



fmcad.²¹

Ruzica Piskac / Michael W. Whalen (Eds.)

PROCEEDINGS OF THE 21ST CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED
DESIGN – FMCAD 2021

Conference Series: Formal Methods in Computer-Aided Design

Volume 2

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 Wien, 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 2 edited by:

Ruzica Piskac, Yale University
51 Prospect Street, New Haven, CT 06511, USA | ruzica.piskac@yale.edu

Michael W. Whalen, Amazon Web Services, Inc.
323 N Washington Ave, Minneapolis, MN 55401, USA | mww@amazon.com

Ruzica Piskac / Michael W. Whalen (Eds.)

PROCEEDINGS OF THE 21ST CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2021

Cite as:

Piskac, R. & Whalen, M. W. (Eds.). (2021). *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design – FMCAD 2021*. TU Wien Academic Press. <https://doi.org/10.34727/2021/isbn.978-3-85448-046-4>

TU Wien Academic Press, 2021

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): 978-3-85448-046-4
ISSN (online): 2708-7824

Available online: <https://doi.org/10.34727/2021/isbn.978-3-85448-046-4>

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): Ruzica Piskac and Michael W. Whalen

Preface

These are the proceedings of the twenty-first International Conference on Formal Methods in Computer-Aided Design (FMCAD), which was held online from October 18 – October 22, 2021 due to the coronavirus. FMCAD was constituted in 1996 as a conference covering formal aspects of specification, verification, synthesis, testing, and security, and as a leading forum for researchers and practitioners in academia and industry alike. 2021 marks the 25th anniversary of that original meeting, and so we wish to celebrate the vision of those original organizers!

The program of FMCAD 2021 is comprised of four tutorials, three invited talks, a student forum, an industry night, a panel session on “25 years of FMCAD”, and the main program consisting of presentations of 30 accepted papers. The tutorial day featured four presentations:

- *Active Automata Learning: from L^* to $L^\#$* by Frits Vaandrager
- *Stainless Verification System Tutorial* by Viktor Kuncak
- *Reactive Synthesis Beyond Realizability* by Rayna Dimitrova
- *Formal Methods for the Security Analysis of Smart Contracts* by Matteo Maffei

and the main conference featured three invited talks:

- *From Viewstamped Replication to Blockchains* by Barbara Liskov
- *Algorithms for the People* by Seny Kamara
- *Engineering with Full-scale Formal Architecture: Morello, CHERI, Armv8-A, and RISC-V* by Peter Sewell

FMCAD’21 also hosted the ninth 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 2021 was organized by Mark Santolucito and featured short presentations of 11 accepted contributions. A detailed description of the Student Forum, listing all accepted contributions, is provided in the conference proceedings. FMCAD 2021 received 72 submissions out of which the committee decided to accept 30 for publication. Each submission received at least three reviews. The topics of the accepted papers include hardware and software verification, SAT, SMT, learning, synthesis, Neural-Network verification, and more. Out of the accepted papers, 23 are classified as regular papers (20 long and 3 short) and 7 are classified as tool/case study papers (5 long and 2 short).

Organizing this event would not have been possible without 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, which helped the authors to improve their submissions and guided the selection of the papers accepted for publication. We thank each and everyone of them for dedicating their time and providing their expertise. We thank William Hallahan (Yale University) for being the web master, Daniel Schoepe for being the Sponsorship Chair, and Mark Santolucito for organizing this year’s FMCAD Student Forum. We thank Georg Weissenbacher (TU Wien) 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): Amazon Web Services, Amazon Prime Video, Cadence, Centaur Technology, Galois, Intel, Mentor Graphics, Novi, and Synopsys.

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 2021 (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 a stimulating and enjoyable event.

October, 2021

Ruzica Piskac, Yale University
Michael W. Whalen, Amazon Inc. and the University of Minnesota

Organizing Committee

Program Co-Chairs

Ruzica Piskac
Michael W. Whalen

Yale University
Amazon Inc. and the University of Minnesota

Webmaster

William Hallahan

Yale University

Student Forum Chair

Mark Santolucito

Barnard College of Columbia University

Publication Chair

Georg Weissenbacher

TU Wien

Steering Committee

Clark Barrett
Armin Biere
Anna Slobodova
Georg Weissenbacher

Stanford University
Johannes Kepler University Linz
Centaur Technology
TU Wien

Program Committee

Erika Abraham	RWTH Aachen University
Jade Alglave	University College London
Pranav Ashar	Real Intent
Per Bjesse	Synopsys
Roderick Bloem	Graz University of Technology
Ivana Cerna	Masaryk University
Supratik Chakraborty	IIT Bombay
Sylvain Conchon	Université Paris-Sud
Leonardo de Moura	Microsoft
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Grigory Fediyukovich	Florida State University
Arie Gurfinkel	University of Waterloo
Liana Hadarean	Amazon Web Services
Ziyad Hanna	Cadence Design System
Fei He	Tsinghua University
Marijn Heule	Carnegie Mellon University
Warren A. Hunt, Jr.	The University of Texas at Austin
Alexander Ivrii	IBM
Dejan Jovanović	Amazon Web Services
Alan Jovic	University of Zagreb
Laura Kovacs	TU Wien
Ton Chanh Le	Stevens Institute of Technology
Rebekah Leslie-Hurd	Intel
Kuldeep S. Meel	National University of Singapore
Ruzica Piskac	Yale University
Elizabeth Polgreen	University of California, Berkeley
Andrew Reynolds	University of Iowa
Christoph Scholl	University of Freiburg
Natasha Sharygina	Università della Svizzera italiana (USI Lugano, Switzerland)
Anna Slobodova	Centaur Technology
Christoph Stickel	The MathWorks
Murali Talupur	Amazon Web Services, Inc.
Jean-Baptiste Tristan	Boston College
Yakir Vizel	The Technion
Thomas Wahl	Northeastern University
Georg Weissenbacher	TU Wien
Michael Whalen	Amazon Inc. and the University of Minnesota
Thomas Wies	New York University
Valentin Wüstholtz	ConsenSys
Lenore Zuck	University of Illinois in Chicago

Additional Reviewers

Asadi, Sepideh
Athanasίου, Konstantinos

Bansal, Suguman
Barnett, Lee
Bendík, Jaroslav
Blichá, Martin
Bustan, Doron

Cano, Filip
Chalupa, Marek
Cheang, Kevin
Chen, Hao
Chernigovskaia, Lidiia

Ebrahimi, Masoud

Fan, Hongyu
Fernandez, Matt
Fraer, Ranan

Georgiou, Pamina
Goel, Shilpi
Golia, Priyanka
Grundy, Jim

Hamza, Ameer
Hjort, Håkan
Hoereth, Stefan
Hozzová, Petra
Huang, Daniel
Hyvärinen, Antti

Jacoby, Reily
Jain, Himanshu
Jain, Mitesh
Jin, Hoon Sang
Jonas, Martin

Könighofer, Bettina
Kwan, Carl

Larrauri, Alberto
Le, Nham

Maderbacher, Benedikt
Majumdar, Rupak
Moosbrugger, Marcel
Mora, Federico

Nalbach, Jasper

Otoni, Rodrigo

Ramanathan, Vivek
Rane, Ashay
Reeves, Joseph
Rehak, Vojtech
Ročkai, Petr

Santolucito, Mark
Schoisswohl, Johannes
Seufert, Tobias
Shi, Yunong
Soos, Mate
Stankovic, Miroslav
Strejček, Jan
Strichman, Ofer
Sumners, Rob
Swords, Sol

Tassarotti, Joseph
Temel, Mertcan

Vediramana Krishnan, Hari Govind

Wolfowitz, Guy

Table of Contents

Tutorials

Reactive Synthesis Beyond Realizability	1
<i>Rayna Dimitrova</i>	
Stainless Verification System Tutorial	2
<i>Viktor Kuncak and Jad Hamza</i>	
Formal Methods for the Security Analysis of Smart Contracts	8
<i>Matteo Maffei</i>	
Active Automata Learning: from L^* to $L^\#$	9
<i>Frits Vaandrager</i>	

Invited Talks

From Viewstamped Replication to Blockchains	10
<i>Barbara Liskov</i>	
Algorithms for the People	11
<i>Seny Kamara</i>	
Engineering with Full-scale Formal Architecture: Morello, CHERI, Armv8-A, and RISC-V	12
<i>Peter Sewell</i>	

Student Forum

The FMCAD 2021 Student Forum	13
<i>Mark Santolucito</i>	

Hardware

CocoAlma: A Versatile Masking Verifier	14
<i>Vedad Hadžić and Roderick Bloem</i>	
End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers	24
<i>Dapeng Gao and Tom Melham</i>	
Hardware Security Leak Detection by Symbolic Simulation	34
<i>Neta Bar Kama and Roope Kaivola</i>	
Scaling Up Hardware Accelerator Verification using A-QED with Functional Decomposition	42
<i>Saranyu Chattopadhyay, Florian Lonsing, Luca Piccolboni, Deepraj Soni, Peng Wei, Xiaofan Zhang, Yuan Zhou, Luca Carloni, Deming Chen, Jason Cong, Ramesh Karri, Zhiru Zhang, Caroline Trippel, Clark Barrett and Subhasish Mitra</i>	
Sound and Automated Verification of Real-World RTL Multipliers	53
<i>Mertcan Temel and Warren Hunt</i>	

Model Checking and IC3

IC3 with Internal Signals	63
<i>Rohit Dureja, Arie Gurfinkel, Alexander Ivrii and Yakir Vizel</i>	
Single Clause Assumption without Activation Literals to Speed-up IC3	72
<i>Nils Froleys and Armin Biere</i>	
Logical Characterization of Coherent Uninterpreted Programs	77
<i>Hari Govind Vadiramana Krishnan, Sharon Shoham and Arie Gurfinkel</i>	
Data-driven Optimization of Inductive Generalization	86
<i>Nham Le, Xujie Si and Arie Gurfinkel</i>	
Model Checking AUTOSAR Components with CBMC	96
<i>Timothee Durand, Katalin Fazekas, Georg Weissenbacher and Jakob Zwirchmayr</i>	

Concurrency and Distributed Systems

Automating System Configuration	102
<i>Nestan Tsiskaridze, Maxwell Strange, Makai Mann, Kavya Sreedhar, Qiaoyi Liu, Mark Horowitz and Clark Barrett</i>	
Towards an Automatic Proof of Lamport's Paxos	112
<i>Aman Goel and Kareem A. Sakallah</i>	
Refinement-Based Verification of Device-to-Device Information Flow	123
<i>Ning Dong, Roberto Guanciale and Mads Dam</i>	
Celestial: A Smart Contracts Verification Framework	133
<i>Samvid Dharanikota, Suvam Mukherjee, Chandrika Bhardwaj, Aseem Rastogi and Akash Lal</i>	
The Civi Verifier	143
<i>Bernhard Kragl and Shaz Qadeer</i>	

Applied Verification and Synthesis

Synthesizing Pareto-Optimal Interpretations for Black-Box Models	153
<i>Hazem Torfah, Shetal Shah, Supratik Chakraborty, S. Akshay and Sanjit A. Seshia</i>	
Dynamic Partial Order Reduction for Spinloops	163
<i>Michalis Kokologiannakis, Xiaowei Ren and Viktor Vafeiadis</i>	
Robustness between Weak Memory Models	173
<i>Soham Chakraborty</i>	
Pruning and Slicing Neural Networks using Formal Verification	183
<i>Ori Lahav and Guy Katz</i>	
Towards Scalable Verification of Deep Reinforcement Learning	193
<i>Guy Amir, Michael Schapira and Guy Katz</i>	

SAT Solving

Exploiting Isomorphic Subgraphs in SAT	204
<i>Alexander Ivrii and Ofer Strichman</i>	
On Decomposition of Maximal Satisfiable Subsets	212
<i>Jaroslav Bendík</i>	
Designing Samplers is Easy: The Boon of Testers	222
<i>Priyanka Golia, Mate Soos, Sourav Chakraborty and Kuldeep S. Meel</i>	
SAT-Inspired Eliminations for Superposition	231
<i>Petar Vukmirović, Jasmin Blanchette and Marijn Heule</i>	
SAT Solving in the Serverless Cloud	241
<i>Alex Ozdemir, Haoze Wu and Clark Barrett</i>	

SMT and First-Order Logic

Induction with Recursive Definitions in Superposition	246
<i>Marton Hajdu, Petra Hozzová, Laura Kovacs and Andrei Voronkov</i>	
Fair and Adventurous Enumeration of Quantifier Instantiations	256
<i>Mikolas Janota, Haniel Barbosa, Pascal Fontaine and Andrew Reynolds</i>	
Mathematical Programming Modulo Strings	261
<i>Ankit Kumar and Panagiotis Manolios</i>	
Lookahead in Partitioning SMT	271
<i>Antti Hyvärinen, Matteo Marescotti and Natasha Sharygina</i>	
A Multithreaded Vampire with Shared Persistent Grounding	280
<i>Michael Rawson and Giles Reger</i>	

Reactive Synthesis Beyond Realizability

Rayna Dimitrova

CISPA Helmholtz Center for Information Security

Saarbrücken, Germany

dimitrova@cispa.de

Abstract—The automatic synthesis of reactive systems from high-level specifications is a highly attractive and increasingly viable alternative to manual system design, with applications in a number of domains such as robotic motion planning, control of autonomous systems, and development of communication protocols. The idea of asking the system designer to describe what the system should do instead of how exactly it does it, holds a great promise. However, providing the right formal specification of the desired behaviour of a system is a challenging task in itself. In practice it often happens that the system designer provides a specification that is unrealizable, that is, there is no implementation that satisfies it. Such situations typically arise because the desired behavior represents a trade-off between multiple conflicting requirements, or because crucial assumptions about the environment in which the system will execute are missing. Addressing such scenarios necessitates a shift towards synthesis algorithms that utilize quantitative measures of system correctness. In this tutorial I will discuss two recent advances in this research direction.

First, I will talk about the maximum realizability problem, where the input to the synthesis algorithm consists of a hard specification which must be satisfied by the synthesized system, and soft specifications which describe other desired, possibly prioritized properties, whose violation is acceptable. I will present a synthesis algorithm that maximizes a quantitative value associated with the soft specifications, while guaranteeing the satisfaction of the hard specification. In the second half of the tutorial I will present algorithms for synthesis in bounded environments, where a bound is associated with the sequences of input values produced by the environment. More concretely, these sequences consists of an initial prefix followed by a finite sequence repeated infinitely often, and satisfy the constraint that the sum of the lengths of the initial prefix and the loop does not exceed a given bound. I will also discuss the synthesis of approximate implementations from unrealizable specifications, which are guaranteed to satisfy the specification on at least a specified portion of the bounded-size input sequences. I will conclude by outlining some of the open avenues and challenges in quantitative synthesis from temporal logic specifications.

This tutorial is based on joint work with Mahsa Ghasemi and Ufuk Topcu published in [1], [2], and with Bernd Finkbeiner and Hazem Torfah published in [3].

REFERENCES

- [1] R. Dimitrova, M. Ghasemi, and U. Topcu, “Reactive synthesis with maximum realizability of linear temporal logic specifications,” *Acta Informatica*, vol. 57, no. 1-2, pp. 107–135, 2020.
- [2] —, “Maximum realizability for linear temporal logic specifications,” in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 11138. Springer, 2018, pp. 458–475.
- [3] R. Dimitrova, B. Finkbeiner, and H. Torfah, “Synthesizing approximate implementations for unrealizable specifications,” in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11561. Springer, 2019, pp. 241–258.

Stainless Verification System Tutorial

Viktor Kunčák

LARA Research Group

School of Computer and Communication Sciences

EPFL

Lausanne, Switzerland

viktor.kuncak@epfl.ch

Jad Hamza

LARA Research Group

School of Computer and Communication Sciences

EPFL

Lausanne, Switzerland

jad.hamza@epfl.ch

Abstract—Stainless (<https://stainless.epfl.ch>) is an open-source tool for verifying and finding errors in programs written in the Scala programming language. This tutorial will not assume any knowledge of Scala. It aims to get first-time users started with verification tasks by introducing the language, providing modelling and verification tips, and giving a glimpse of the tool's inner workings (encoding into functional programs, function unfolding, and using theories of satisfiability modulo theory solvers Z3 and CVC4).

Stainless (and its predecessor, Leon) has been developed primarily in the EPFL's Laboratory for Automated Reasoning and Analysis in the period from 2011-2021. Its core specification and implementation language are typed recursive higher-order functional programs (imperative programs are also supported by automated translation to their functional semantics). Stainless can verify that functions are correct for all inputs with respect to provided preconditions and postconditions, it can prove that functions terminate (with optionally provided termination measure functions), and it can provide counter-examples to safety properties. Stainless enables users to write code that is both executed and verified using the same source files. Users can compile programs using the Scala compiler and run them on the JVM. For programs that adhere to certain discipline, users can generate source code in a small fragment of C and then use standard C compilers.

Index Terms—verification, formal methods, proof, counter-example, model checking, Scala, functional programming, satisfiability modulo theories

I. INTRODUCTION

Stainless [1] is a tool for verifying and finding errors in programs written in a subset of the Scala [2] programming language. Stainless is open source (distributed under Apache license) and hosted on GitHub at:

<https://github.com/epfl-lara/stainless/>
<https://epfl-lara.github.io/stainless/>

Stainless (and its predecessor, Leon) have been developed primarily in the EPFL's Laboratory for Automated Reasoning and Analysis in the period from 2011-2021, see, in particular [1], [3] as well as [4]–[14]. The core specification and implementation language of Stainless are typed recursive higher-order functional Scala programs. It also supports certain imperative programs [4], [6]. Stainless can verify that functions are correct for all inputs with respect to provided preconditions and postconditions, it can prove that functions terminate (with

optionally provided termination measure functions), and it can also provide counter-examples to safety properties.

Stainless can be used to write programs that are directly executable and proven correct. In particular, because it uses Scala's syntax and type system, users can execute Stainless programs using the standard Scala compiler (version 2.12.13 at the time of writing). In addition, there are passes that eliminate non-executable (ghost) code from source to make sure that it does not result in run-time overhead after compilation. For programs that adhere to certain discipline the “genc” option of Stainless can be used to generate C source code that compiles with common compilers such as gcc.

A. Outline

In this tutorial, we show examples demonstrating how to use Stainless to develop verified models and programs. We will mostly use basic notation for functional programming, which we will introduce along the way. We will use Stainless version 0.9 or later.

In addition to basic introduction, we will suggest strategies for specifying programs and helping Stainless prove them correct. An example is using lemmas and proving them by induction expressed through terminating recursion.

To help users be more effective when using Stainless, we also outline key mechanisms that Stainless uses in proof and counterexample search: encoding into functional programs, function unfolding, and using rich theories of satisfiability modulo theory solvers Z3 and CVC4.

II. GETTING STARTED

Stainless is a command line application that runs on the Java virtual machine, version 1.8. We mostly test it on Ubuntu Linux. We provide releases for Linux and Mac. Others use it on Windows as well, where it may be simplest to use Windows Subsystem for Linux to get started. Download the release file from

<https://github.com/epfl-lara/stainless/releases/>

then unzip the file and put a link to `stainless` in your path.

The following is a simple program, call it `MaxBug.scala`, containing a function `max`. `Max` attempts to compute maximum of the two 32-bit integers by returning one of them, depending on the sign `d` of their difference.

```
object TestMax {
  def max(x: Int, y: Int): Int = {
    val d = x - y
    if (d > 0) x
    else y
  } ensuring(res =>
    x <= res && y <= res && (res == x || res == y))
}
```

We use `object` to group functions into modules. We define functions using `def` and provide their parameters (here: `x` and `y`) and their types, as well as the return type. We define local immutable values using `val` keyword. Scala infers the type of `d` as `Int`.

After the usual body, we introduced an `ensuring` statement. The first identifier, `res`, binds the return value of the function. After the arrow `=>` we state the property we would like the result to satisfy. In this case, the result should be greater than each argument and it should be equal to one of them.

Invoke `stainless MaxBug.scala` and you may get output containing some of the following.

```
MaxBug.scala:7:49: warning: => INVALID
  x <= res && y <= res && (res == x || res == y))
  ^
warning: Found counter-example:
warning:   y: Int -> -2147483648
warning:     x: Int -> 1
Verified: 0 / 3

stainless summary

MaxBug.scala:3:13: max Subtraction overflow invalid
MaxBug.scala:7:37: max postcondition          invalid
MaxBug.scala:7:49: max postcondition          invalid
.....
total: 3      valid: 0      (0 from cache) invalid: 3
```

Use `--timeout=5` to set time out to 5 seconds. and `--no-colors` to request clean ASCII output with parsable line numbers in reports.

Why did Stainless report a counterexample? Indeed, executing `max` with the two provided values computes using signed 32-bit arithmetic the value `-11` for `d`, so the function returns `y` as the result `res` so `y <= res` is false. We can repair this example in at least two ways:

- Use `if (x <= y)` instead of the value `d`.
- Use `BigInt` instead of `Int`, thus adopting unbounded integers instead signed 32-bit ones.

If you run your program several times, you may notice that Stainless reports that a valid verification condition was persistently cached (inside `.stainless-cache`). You can turn off caching with `--vc-cache=false`.

You may find the `--watch` option useful when modifying a file several times, which makes Stainless run verification whenever the source file is changed.

By default, Stainless uses a version of `z3` (4.7.1) which is packaged inside Stainless (`--solvers=nativez3`). This allows Stainless to interact with `z3` through Java calls. You may also use an externally built version of `z3` (for instance, `z3 4.8.12` is shipped with the release) by specifying `--solvers=smt-z3`. In that case, Stainless will communicate with `z3` using SMT-LIB files, which might be slower than Java calls, but has two

benefits. First, you get to use the newest release of `z3`. Second, `smt-z3` is more likely to respect timeouts than `nativez3`.

You can also use CVC4 as the solver if you download and put `cvc4` executable on your path. You can use both with `--solvers=smt-cvc4,smt-z3`. Use `--debug=smt` to preserve the generated SMT-LIB files and look for them in the `smt-sessions` directory.

III. VERIFIED FUNCTIONAL PROGRAMMING

We will now implement a simple function that computes differences of successive elements of a list. Let us start our file with `import stainless.collection._` so we can use the immutable `List` library of Stainless. You can find the sources of this and other library files at following URL:

<https://github.com/epfl-lara/stainless/blob/master/frontend/library/stainless/collection/List.scala>

Let's try to write a function `diffs` that takes a list of elements, for example `x1, x2, x3, x4` and keeps the first element and then follows it by the list of their differences. In this case we would like to obtain `x1, x2 - x1, x3 - x2, x4 - x3`. For empty and one-element list the output equals input. Let us write this as the default implementation. We can also state the example of four-element list as a symbolic test case. To state it, we use another function with a dummy body and a postcondition that invokes `diffs`.

```
import stainless.collection._
object Diffs {
  def diffs(l: List[BigInt]): List[BigInt] = {
    1 match {
      case Nil() => l
      case _ :: Nil() => l
      // missing cases
    }
  }
  def test(x1: BigInt, x2: BigInt,
           x3: BigInt, x4: BigInt): Unit = {
    } ensuring(_ =>
      diffs(List(x1, x2, x3, x4)) ==
        List(x1, x2 - x1, x3 - x2, x4 - x3))
  }
}
```

After developing a function that meets this partial specification, we can see whether it meets a stronger specification. For example, we can define the inverse function `undiff` that takes `y0, y1, ..., yn` and computes `y0, y0 + y1, ..., ∑i=0n yi`. Being masters of functional programming, we recognize that this is just a prefix sum of a list, so we define it by

```
def undiff(l: List[BigInt]): List[BigInt] =
  l.scanLeft(BigInt(0))(_ + _).tail
```

where `scanLeft` is defined in our `List` library. Now we can add as the `ensuring` condition of `diffs` the condition `ensuring (res => (undiff(res) == l))`. It so happens that Stainless proves this condition automatically using its algorithm. As an off-line exercise, try to prove this result with pen and paper. This might give you a sense on how Stainless is able to prove this property.

The algorithm of Stainless initially treats called functions as unknown (uninterpreted) mathematical functions. It then

iteratively expands each call by defining the function to be equal to one unfolding of its body and also inserts the `ensuring` clause as an assumption.

IV. AMORTIZED QUEUE

We have found Stainless to work very well for verification of purely functional data structures. Let us examine the case of an amortized queue such as the one from [15, Section 5.2, Page 42]. We will start by writing down an *abstract class*. In this class we define methods with dummy bodies denoted by `???` but with `ensuring` clauses that specify the desired behavior of operations. To specify the behavior we use `toList` function, which is also left unspecified in the abstract class.

```
import stainless.collection._
import stainless.lang._
abstract class Queue[A] {
  def enqueue(a: A) = (??? : Queue[A])
    .ensuring(res =>
      res.toList == this.toList ++ List(a))

  def dequeue: Option[(A, Queue[A])] =
    (??? : Option[(A, Queue[A])])
    .ensuring(res => res match {
      case None() =>
        this.toList == Nil[A]()
      case Some((a, q)) =>
        this.toList == a :: q.toList
    })

  def toList: List[A]
}
```

When we extend the abstract class, Scala requires us to define `toList`, whereas Stainless ensures that our implementation meets the specifications in the abstract class. We can implement an inefficient queue using a single list.

```
case class SimpleQueue[A](l: List[A])
  extends Queue[A] {
  def enqueue(a: A) = SimpleQueue(l ++ List(a))

  def dequeue = l match {
    case Nil() => None()
    case Cons(x, xs) => Some((x, SimpleQueue(xs)))
  }

  def toList = l
}
```

Stainless successfully verifies that the properties required by a queue are satisfied by this implementation. Even if correct, this implementation is inefficient because `enqueue` takes linear time in the current number of queue elements. We will thus try to develop and prove correct the implementation like one from [15, Section 5.2, Page 42] that uses two lists and that has constant time amortized complexity.

```
case class AmortizedQueue[A](front: List[A],
                             rear: List[A])
  extends Queue[A] {
  def toList = front ++ rear.reverse
}
```

The `toList`, which we use only for specification, gives us a hint on how to implement `enqueue` efficiently. For `dequeue` we will need a `reverse` operation on lists, which we can implement in linear time. Despite its complexity, our version

of `dequeue` will be verified automatically. As for `enqueue`, its implementation is simple, yet its proof turns out to require some well known property of lists that we need to tell Stainless to invoke explicitly!

```
def enqueue(a: A): Queue[A] = {
  val res: Queue[A] = // to fill

  // You can state using assertions things you know are true,
  // to see if Stainless is able to prove them:
  assert(res.toList == front ++ (a :: rear).reverse)

  // Alternatively, you can use an equation style reasoning.
  // Here Stainless should timeout from the second to the third
  // step, because some steps are missing.
  (
    res.toList ==:| trivial |:
    front ++ (a :: rear).reverse ==:| trivial |:
    // Add missing steps here to arrive to the result.
    // For complicated steps, you need to invoke lemmas
    // instead of writing 'trivial'.
    this.toList ++ List(a)
  ).qed

  res
}
```

V. PROPERTIES AND PROOFS

How do we state properties in Stainless? We write a property $\forall x : T. F(x)$ as a function `lemmaF` defined by:

```
def lemmaF(x: T): Unit = {
  ()
} ensuring (_ => F(x))
```

When we wish to instantiate the property taking x to be some specific value v , we insert a function invocation `lemmaF(v)` into the part of the code where we need this property. Suppose that proving property $\forall x : T. F(x)$ is not automatic. Then verification of `lemmaF` itself will fail, as stated. If $F(x)$, for example, follows from $G(x, x + 1)$ that is established in `lemmaG(x, y)`, then we can state and prove `lemmaF` as:

```
def lemmaF(x: T): Unit = {
  lemmaG(x, x+1)
} ensuring (_ => F(x))
```

Thus, we can adopt the following strategies for libraries of lemmas:

- introduce a function for a lemma
- use a function parameter for each universally quantified variable
- write lemma statement in the `ensuring` clause
- use the body of the function to encode a high-level proof, with function invocations corresponding to applying previously proven lemmas.

Purely universal statements can return `Unit` type. For existential statements, we can often state their constructive Skolemized form and return a witness for the existential quantifier from the lemma.

It can be helpful to examine some proofs of properties in the `List` library. Remarkably, we can even make recursive invocations of functions in their bodies. Which mathematical reasoning principle do such proofs correspond to?

VI. DIGITS

For built-in types such as `Int` and `Long`, the SMT solvers will successfully reason about their bitwidth representation. What if we wish to reason about the bits of arbitrarily large numbers? As a simple example, let us define simple addition as a recursive function on lists of bits.

```
import stainless.annotation._
import stainless.lang._
import stainless.collection._
object AddBitwise {
  type Digits = List[Boolean]
  val zero = Nil[Boolean]()

  def add(x: Digits, y: Digits, carry: Boolean):
    Digits = {
    require(x.length == y.length)
    (x,y) match {
      case (Nil(), Nil()) =>
        if (carry) true::zero else zero
      case (Cons(x1,xs), Cons(y1,ys)) => {
        val z = x1 ^ y1 ^ carry
        val carry1 = (x1 && y1) ||
                     (x1 && carry) ||
                     (y1 && carry)
        z :: add(xs, ys, carry1)
      }
    }
  }
}
```

How can we state that such addition is commutative? How can we prove it in Stainless? As an off-line exercise, think about how we can prove that this corresponds to actual addition on integers (`BigInt`).

VII. TERMINATION

The following recursive function searches for an element in a sorted array, but it has a bug. You may run Stainless on this file to spot it. Fix the issue, and add a `decreases` clause at the beginning of the function to ensure that Stainless can prove the function terminating.

```
import stainless.lang._

object BinarySearch1 {

  def search(arr: Array[Int], x: Int, lo: Int, hi:
    Int): Boolean = {
    if (lo <= hi) {
      val i = (lo + hi) / 2
      val y = arr(i)
      if (x == y) true
      else if (x < y) search(arr, x, lo, i-1)
      else search(arr, x, i+1, hi)
    } else {
      false
    }
  }
}
```

In Stainless, all functions are required to have a measure (either inferred automatically, or written in a `decreases` clause by the user). The system in its current design would be unsound (we would be able to prove false postconditions or assertions) if we allowed non-terminating functions.

VIII. IMPERATIVE FEATURES

Stainless supports some imperative features, such as local mutable variables, while loops, return statements, and more (see <https://epfl-lara.github.io/stainless/imperative.html>). Stainless transforms these constructs into functional programs.

Using a while loop and a return statement, rewrite the `findIndexOpt` function:

```
def findIndexOpt(ar: Array[Int], v: Int):
  Option[Int] = {
    ...
  }
```

that finds an index of element `v` in a sorted array `ar`. Prove that, when your function returns `Some(i)`, then `ar(i) == v`. To prove that array indices are within bounds, you will need a loop invariant, for which the syntax is:

```
(while(...) {
  decreases(...)
  ...
}).invariant(...)
```

Does Stainless help you if you make an overflow mistake when computing the middle of an interval using bounded arithmetic?

Note that while loops require `decreases` clauses as well (when the measure cannot be inferred automatically), because they are translated into recursive functions by Stainless. To see how the while loop and the return statement are transformed, you may run the command below on your file. Stainless has a pipeline containing several phases, and `ReturnElimination` is the one that removes while loops and return statements. The `--debug-objects` option tells Stainless to only display the `findIndexOpt` function in the debug output.

```
stainless --debug=trees
--debug-objects=findIndexOpt
--debug-phases=ReturnElimination FindIndex.scala
```

As a harder exercise, identify and prove a stronger postcondition of `findIndexOpt`: what can we state in the postcondition for the case when the function returns `None`? What assumptions and loop invariants do we need to be able to prove this postcondition?

IX. DESIGN PRINCIPLES

A number of verification systems have been developed in the past decades. Stainless tries to borrow many of the features that others and us have found useful in other systems. At the same time, it is driven by a somewhat unique combination of principles, whose understanding may help set the expectations from the tool.

A. Searching for Both Proofs and Counterexamples

From the beginning [13], the system was designed to search for both counterexamples and proofs in a unified iterative loop. Thanks to this design, on many programs Stainless behaves like a combination of a bounded model checker and a k-inductive prover such as [16]: we can often expect a definite answer, whether the program verifies or has a counterexample.

B. Recursive programs as foundation, not transition systems.

Operational semantics tells us that we can translate functional (and many other) programs into transition systems. This has even been used in verification tools with success [1]. Nonetheless, we believe that it carries significant overhead, especially for proofs. Thus, like in ACL2 [17], [18] our intermediate representation is based on recursive functions [13] and we hope to leverage high-level structure to make verification more feasible, much like Liquid Haskell [19] which needs to be complemented with symbolic execution to also generate counterexamples [20]. Consequently, iterative unfolding of our recursive functions in Stainless gives a different sequence of approximations than the one we would obtain by representing programs using control-flow graphs and explicit stacks [21].

C. Top-down verification for each function.

Stainless verifies each desired function one by one. When verifying a function f , it does not check which other parts of code invoke f . In particular, it will, in its current design, not infer preconditions for a function automatically. Preconditions need to be explicitly specified using a `require` clause at function entry. On the other hand, when Stainless examines the body of f and finds a function g , then it will examine not only the specification of g , but also its body. If g is recursive, this process will continue, with a check for counterexample and check for unsatisfiability performed at each step. This process treats functions more transparently than some modular verifiers. The process is also breadth-first, instead of having the form of directed rewriting as in some other systems. The effectiveness of this process is explained in part by the fact that it results in a decision procedure for certain classes of functions [14], [22], [23]. Furthermore, we continue to be surprised by how well this simple strategy works in practice, even if we have no theoretical reason to know that it will succeed.

D. Scala subset as the input language.

Stainless uses Scala as a language that has substantial user base, regularly ranked higher than Haskell and LISP in Stack Overflow developer surveys [24], which is relevant for maintaining the correspondence between what executes and that is verified. As a functional language, Scala contains an expressive purely functional fragment which can be used for specification and modelling. The users of Stainless thus largely avoid the need to learn a separate specification language, because functional programs are a great specification vehicle. At the same time, the system supports polymorphism and subtyping with a type system that eliminates many nonsensical programs before they waste user’s time inside the program verifier’s loop. That said, Stainless purposely avoids by design certain Scala 2 features, such as null references and complex initialization. Other features, such as machine integers, are modelled precisely: it is certainly necessary in practice to have machine integers of various width available (for example, 32-bit Int and 64-bit Long), but it is also helpful to use unbounded BigInt data types, especially for specifications, and

these different types should not be confused. Stainless provides the user a choice and maps these data types and operations on them to the appropriate types and theories inside SMT solvers [8]. Subtyping is currently implemented via a translation into a language with disjoint types [3]; its use requires additional encoding and may slow down verification. Imperative features are supported as a choice of either unshared mutable state [6] or using a model [4] that, at user level, is similar to dynamic frames [25] of Dafny [26].

E. Embracing SMT solver theories, avoiding quantifiers.

Instead of using axioms to encode program semantics and data types, Stainless leverages algebraic data types, sets, and arrays. Stainless thus currently emits quantifier-free queries to solvers (either Z3 or CVC4). The hope with this choice is that SMT solvers will remain predictable for both proofs and counterexamples. In contrast, the use of quantifiers may lead to more automation and sometimes excellent performance for proofs, but quickly leads outside of the space where the solvers can reliably report counterexamples.

F. Executability of programs and specifications.

In Stainless we aim to write programs that can be compiled using the standard Scala compiler. Specification constructs in Stainless are defined in a Scala library and they have dummy execution semantics. In some cases, even such dummy semantics may result in overhead, so we have developed passes that eliminate some of the specification code altogether. In addition, Stainless has a subset that can be used to generate C code suitable for embedded systems, an enhanced version of such functionality developed for Leon [27].

Acknowledgements. Research on Stainless has been funded in part by (i) the Swiss Science Foundation grants 200021_132176, 200020_138204, 200020_146649, 200021_144503, 200020_159949, and 200021_175676. (ii) European Research Council (ERC) Starting Grant PE6-306484-IMPRO, (iii) The Swiss State Secretariat for Education, Research and Innovation, Swiss Space Office grant “Embedded Flight Software Verification–ESOVER” and (iv) the envelope budget for the LARA group from the EPFL School of Computer and Communication Sciences.

Stainless and Inox were created from parts of Leon code by Nicolas Voirol. In addition to Nicolas and the two authors of this tutorial, contributors to Stainless and Inox include: Roman Ruetschi, Georg Stefan Schmid, Marco Antognini, Ravichandhran Madhavan, Etienne Kneuss, Lars Hupel, Emmanouil Koukoutos, Philippe Suter, Roman Edelmann, Utkarsh Upadhyay, Ivan Kuraj, Sandro Stucki, Ruzica Piskac, Tihomir Gvero, Czipó Bence, Sumith Kulal, Lucien Iseli, Regis Blanc, Iulian Dragos, Dragana Milovančević, Antoine Brunner, Mirco Dotta, Yann Bolliger, Rodrigo Raya, Samuel Gruetter, Mikael Mayer, Guillaume Massé. Romain Jufer worked with Jad Hamza on a fork for smart contract verification and Solidity code generation, Romain Edelmann and Rodrigo Raya developed an interactive proof assistant concept

based on Inox. Regis Blanc developed a Scala library for input and output of SMT-LIB files. ScalaZ3 interface to the Z3 dynamically linked library additionally received contributions from Ali Sinan Köksal and Thorsten Tarrach. Contributors to Stainless Bolts case studies include additionally Samuel Chassot and Clément Burgelin. We thank users of Stainless from Ateleris GmbH including Simon Felix, Filip Schramka, and Ivo Nussbaumer. We also thank MSc students at EPFL taking the Formal Verification course, completing interesting case studies and identifying bugs in the system.

REFERENCES

- [1] J. Hamza, N. Voirol, and V. Kunčák, “System FR: Formalized foundations for the Stainless verifier,” *Proc. ACM Program. Lang.*, no. OOPSLA, November 2019.
- [2] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 4th ed. Artima Inc., 2008.
- [3] N. C. Y. Voirol, “Verified functional programming,” Ph.D. dissertation, EPFL, thesis number 9479, 2019. [Online]. Available: <http://doi.org/10.5075/epfl-thesis-9479>
- [4] G. Schmid and V. Kunčák, “Proving and disproving programs with shared mutable data,” 2021.
- [5] R. Madhavan, S. Kulal, and V. Kuncak, “Contract-based resource verification for higher-order functions with memoization,” in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.
- [6] R. W. Blanc, “Verification by reduction to functional programs,” Ph.D. dissertation, EPFL, thesis number 7636, 2017. [Online]. Available: <http://doi.org/10.5075/epfl-thesis-9479>
- [7] N. Voirol, E. Kneuss, and V. Kuncak, “Counter-example complete verification for higher-order functions,” in *Scala Symposium*, 2015.
- [8] R. Blanc and V. Kuncak, “Sound reasoning about integral data types with a reusable SMT solver interface,” in *Scala Symposium*, 2015.
- [9] V. Kuncak, “Developing verified software using Leon (invited contribution),” in *NASA Formal Methods (NFM)*, 2015.
- [10] E. Koukoutos and V. Kuncak, “Checking data structure properties orders of magnitude faster,” in *Runtime Verification (RV)*, 2014.
- [11] R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter, “An overview of the Leon verification system: Verification by translation to recursive functions,” in *Scala Workshop*, 2013.
- [12] A. Köksal, V. Kuncak, and P. Suter, “Constraints as control,” in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.
- [13] P. Suter, A. S. Köksal, and V. Kuncak, “Satisfiability modulo recursive programs,” in *Static Analysis Symposium (SAS)*, 2011.
- [14] P. Suter, M. Dotta, and V. Kuncak, “Decision procedures for algebraic data types with abstractions,” in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2010.
- [15] C. Okasaki, *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [16] A. Champion, A. Mebsout, C. Stickse, and C. Tinelli, “The kind 2 model checker,” in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9780. Springer, 2016, pp. 510–517.
- [17] J. S. Moore, “Milestones from the pure lisp theorem prover to ACL2,” *Formal Aspects Comput.*, vol. 31, no. 6, pp. 699–732, 2019.
- [18] R. S. Boyer and J. S. Moore, “Proving theorems about LISP functions,” in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Stanford, CA, USA, August 20-23, 1973*, N. J. Nilsson, Ed. William Kaufmann, 1973, pp. 486–493. [Online]. Available: <http://ijcai.org/Proceedings/73/Papers/053.pdf>
- [19] N. Vazou, “Liquid haskell: Haskell as a theorem prover,” Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, SAN DIEGO, 2016.
- [20] W. T. Hallahan, A. Xue, M. T. Bland, R. Jhala, and R. Piskac, “Lazy counterfactual symbolic execution,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 411–424. [Online]. Available: <https://doi.org/10.1145/3314221.3314618>
- [21] L. Lamport, “The pluscal algorithm language,” in *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, ser. Lecture Notes in Computer Science, M. Leucker and C. Morgan, Eds., vol. 5684. Springer, 2009, pp. 36–60.
- [22] V. Sofronie-Stokkermans, “Locality results for certain extensions of theories with bridging functions,” in *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, ser. Lecture Notes in Computer Science, R. A. Schmidt, Ed., vol. 5663. Springer, 2009, pp. 67–83.
- [23] T. Pham, A. Gacek, and M. W. Whalen, “Reasoning about algebraic data types with abstractions,” *J. Autom. Reason.*, vol. 57, no. 4, pp. 281–318, 2016.
- [24] S. Overflow, “Annual developer survey,” 2021. [Online]. Available: <https://insights.stackoverflow.com/survey/>
- [25] I. T. Kassios, “Dynamic frames: Support for framing, dependencies and sharing without restrictions,” in *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006. Proceedings*, ser. Lecture Notes in Computer Science, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 268–283.
- [26] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 348–370.
- [27] M. Antognini, “Extending safe C support in Leon,” Master’s thesis, EPFL, 2017. [Online]. Available: <https://infoscience.epfl.ch/record/227942/>

Formal Methods for the Security Analysis of Smart Contracts

Mattei Maffei
TU Wien
Vienna, Austria
matteo.maffei@tuwien.ac.at

Abstract—Smart contracts consist of distributed programs built over a blockchain and they are emerging as a disruptive paradigm to perform distributed computations in a secure and efficient way. Given their nature, however, program flaws may lead to dramatic financial losses and can be hard to fix. This motivates the need for formal methods that can provide smart contract developers with correctness and security guarantees, ideally automating the verification task.

This tutorial introduces the semantic foundations of smart contracts and reviews the state-of-the-art in the field, focusing in particular on the automated, sound, static analysis of Ethereum smart contracts. We will highlight the strengths and drawbacks of different methods, suggesting open challenges that can stimulate new research strands. Finally, we will overview eThor, an automated static analysis tool that we recently developed based on rigorous semantic foundations.

Active Automata Learning: from L^* to $L^\#$

Frits Vaandrager
Radboud University
Nijmegen, The Netherlands
F.Vaandrager@cs.ru.nl

Abstract—In this tutorial on active automata learning algorithms, I will start with the famous L^* algorithm proposed by Dana Angluin in 1987, and explain how this algorithm approximates the Nerode congruence by means of refinement. Next, I will present a brief overview of the various improvements of the L^* algorithm that have been proposed over the years. Finally, I will introduce $L^\#$, a new and simple approach to active automata learning. Instead of focusing on equivalence of observations, like the L^* algorithm and its descendants, $L^\#$ takes a different perspective: it tries to establish *apartness*, a constructive form of inequality.

From Viewstamped Replication to Blockchains

Barbara Liskov

MIT Computer Science & Artificial Intelligence Lab

Cambridge, MA, USA

liskov@csail.mit.edu

Abstract—This talk will discuss two replication protocols. The first, Viewstamped Replication, was developed in the 1980s when research on replication protocols was concerned primarily with systems that survived crash failures, e.g., individual replicas could fail only by crashing. Viewstamped replication is similar to Paxos; it was the earliest practical replication algorithm that provided the ability to execute general operations (as opposed to just reads and writes).

In the 1990s, researchers became interested in systems that could survive Byzantine failures, in which replicas fail arbitrarily. Replicated systems that survive Byzantine failures are substantially more complex, requiring both more replicas and more phases of communication, than those that survive only crash failures. The talk will present PBFT, the first practical replication technique that handles Byzantine failures. PBFT is now of great interest to researchers working on blockchains.

Algorithms for the People

Seny Kamara

Brown University

Providence, Rhode Island, USA

seny@brown.edu

Abstract—Algorithms have transformed every aspect of society, including communication, transportation, commerce, finance, and health. The revolution enabled by computing has been extraordinarily valuable. The largest tech companies generate a trillion dollars a year and employ 1 million people. But technology does not affect everyone in the same way. In this talk, we will examine how new technologies affect marginalized communities and think about what technology and academic research would look like if its goal was to serve the disenfranchised.

Engineering with Full-scale Formal Architecture: Morello, CHERI, Armv8-A, and RISC-V

Peter Sewell
University of Cambridge
Cambridge, UK
Peter.Sewell@cl.cam.ac.uk

Abstract—Architecture specifications define the fundamental interface between hardware and software. Historically, mainstream architecture specifications have been informal prose-and-pseudocode documents. This talk will describe our work to establish and use mechanised semantics for full-scale instruction-set architectures (ISAs): the mainstream Armv8-A architecture, the emerging RISC-V architecture, the CHERI-MIPS and CHERI-RISC-V research architectures that use hardware capabilities for improved security, and Arm’s prototype Morello architecture – an industrial demonstrator incorporating the CHERI ideas.

We use a variety of tools, especially our Sail ISA definition language and Isla symbolic evaluation engine, to build semantic definitions that are readable, executable as test oracles, support reasoning within the Coq, HOL4, and Isabelle proof assistants, support SMT-based symbolic evaluation, support model-based test generation, and can be integrated with operational and axiomatic concurrency models. These models are all complete enough to boot operating systems and hypervisors, covering the full sequential ISA (though not other SoC components, such as the Arm Generic Interrupt Controller). They range from 5000 to 60000 lines of specification.

For CHERI-MIPS and CHERI-RISC-V, we have used Sail models (and previously L3 models) as the golden reference during design, working with our systems and computer architecture colleagues in the CHERI team to use lightweight formal specification routinely in documentation, testing, and test generation. We have stated and proved (in Isabelle) some of the fundamental intended security properties of the full CHERI-MIPS ISA.

For Armv8-A, building on Arm’s internal shift to an executable model in their ASL language, we have the complete sequential ISA semantics automatically translated from the Arm ASL to Sail, and for RISC-V, we have hand-written what is now the officially adopted model. For their concurrent semantics, the “user” semantics, partly as a result of our collaborations with Arm and within the RISC-V concurrency task group, have become simplified and well-defined, with multiple models proved equivalent, and we are currently working on the “system”

This work was partially supported by the UK Government Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694), ERC AdG 789108 ELVER, EPSRC programme grant EP/K008528/1 REMS, Arm iCASE awards, EPSRC IAA KTF funding, the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust. Approved for public release; distribution is unlimited. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 (“CTSRD”), FA8750-11-C-0249 (“MRC2”), HR0011-18-C-0016 (“ECATS”), and FA8650-18-C-7809 (“CIFV”), as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

semantics. Our symbolic execution tool for Sail specifications, Isla, supports axiomatic concurrency models over the full ISA.

Morello, supported by the UKRI Digital Security by Design programme, offers a path to hardware enforcement of fine-grained memory safety and/or secure encapsulation in the production Armv8-A architecture, potentially excluding or mitigating a large fraction of today’s security vulnerabilities for existing C/C++ code with little modification. During the ISA design process, we have proved (in Isabelle) fundamental security properties for the complete Morello ISA definition, and generated tests from the definition which were used during hardware development and for QEMU bring-up.

All these tools and models are (or will soon be) available under open-source licences, providing well-validated models for others to use and build on.

This is joint work by many people, including especially, *for Sail and Isla*: Alasdair Armstrong, Brian Campbell, Kathryn E. Gray, Mark Wassell, Jon French, Neel Krishnaswami; *for Morello verification and ASL-to-Sail translation*: Thomas Bauereiss, Thomas Sewell, Brian Campbell, Alasdair Armstrong, Alastair Reid; *for Morello and CHERI-MIPS test generation*: Brian Campbell; *for CHERI-MIPS verification*: Kyndylan Nienhuis; *for RISC-V and CHERI-RISC-V specifications*: Robert M. Norton, Prashanth Mundkur, Jessica Clark; *for MIPS and CHERI-MIPS specifications*: Alexandre Joannou, Anthony Fox, Michael Roe, Matthew Naylor; and *for Concurrency semantics*: Christopher Pulte, Shaked Flur, Will Deacon, Ben Simner, Luc Maranget, Susmit Sarkar, Jean Pichon-Pharabod, Ohad Kammar, Jeehoon Kang, Sung-Hwan Lee, Chung-Kil Hur. All this is in collaboration with the rest of the CHERI team and others in Arm (especially Richard Grisenthwaite, Graeme Barnes, and the Morello team) and in the RISC-V community, with the CHERI team jointly led by Robert N. M. Watson, Simon W. Moore, Peter Sewell, Peter G. Neumann, and Ian Stark.

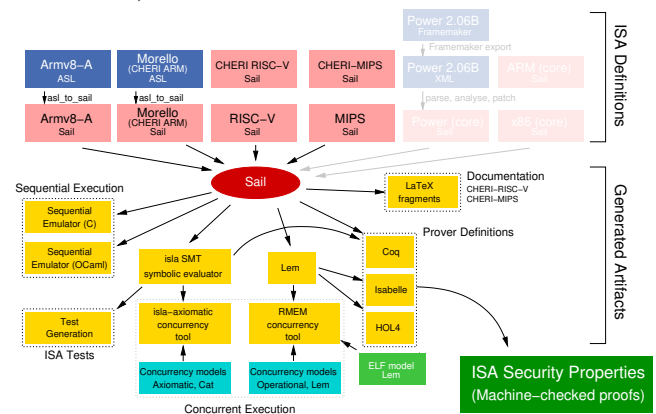



Fig. 1. Sail models and infrastructure (grayed-out models are partial ISA models in an older version of Sail)

The FMCAD 2021 Student Forum

Mark Santolucito 

Barnard College, Columbia University

New York City, USA

msantolu@barnard.edu

Abstract—The Student Forum at the International Conference on Formal Methods in Computer-Aided Design (FMCAD) gives undergraduate and graduate students the opportunity to engage with to the Formal Methods community by presenting their working and receiving feedback. The Student Forum was held in a hybrid format, with some students participating in limited in-person events in New Haven, Connecticut, USA.

The Graduate Student Forum was first introduced in 2013 to the FMCAD conference series. The goal of the Forum is to enable graduate students to attend the conference, even if they do not have a paper accepted at the main conference track. Students were attracted with an opportunity to present their on-going work to a broader scientific audience and receive valuable feedback about the research they are currently pursuing.

FMCAD 2021 hosted the ninth edition of the Student Forum. There was an open call for papers from both undergraduate and graduate students working broadly in the area of Formal Methods. In the call, students were asked to submit a 2-page summary of their current research and on-going work. We received a number of high quality submissions to the Student Forum and accepted a total of 10 submissions. Reviews were based on the overall quality and novelty of work, the potential for impact of the work on the field of Formal Methods, as well as the potential positive impact on the student to have the opportunity to participate in the forum.

This year, the Student Forum allowed for the submission of joint research where two student researchers collaborated and contributed equally in the eyes of their advisors. The topics covered by the accepted submissions ranged across the field of Formal Methods, including foundational advancements as well as a variety of application domains. The accepted submissions are listed below with their respective student authors:

- Wonhyuk Choi: *Can Reactive Synthesis and Syntax-Guided Synthesis Be Friends?*
- Shmuel Berman: *Programming-By-Example by Programming-By-Example: Synthesis of Looping Programs*
- Ameer Hamza: *Automated Alignment for Equivalence Checking*
- Amitash Nanda: *NeuCASL: From Logic Design to System Simulation of Neuromorphic Engines*
- Guy Amir: *Verifying Deep Reinforcement-Learning Systems*
- Ori Lahav: *Neural Network Simplification using Formal Verification*

- Y. Cyrus Liu: *Source-Level Bitwise Branching for Temporal Verification*
- Maxwell Levatch: *Using Z3 to Validate Executions of a Program Partitioner*
- Priyanka Golia: *Boolean Functional Synthesis and its Applications*
- John Hui and Robert Krook: *Toward Sparse Synchronous Computing on Embedded Systems*


This edition of the FMCAD Student Forum follows a series of previous successful iterations of the forum [1]–[8].


We would like to thank the organizers of FMCAD, as well as the entire program committee of FMCAD, who have made the FMCAD student forum possible. Additionally, we are grateful to the student authors and their research mentors who have contributed their excellent work to the program.

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.
- [4] H. Hojjat, “The FMCAD 2016 graduate student forum,” in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, p. 8. [Online]. Available: <https://doi.org/10.1109/FMCAD.2016.7886654>
- [5] K. Heljanko, “The FMCAD 2017 graduate student forum,” 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. 10. [Online]. Available: <https://doi.org/10.23919/FMCAD.2017.8102234>
- [6] D. Jovanovic and A. Reynolds, “The FMCAD 2018 graduate student forum,” in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, p. 1. [Online]. Available: <https://doi.org/10.23919/FMCAD.2018.8602995>
- [7] G. Fedyukovich, “The FMCAD 2019 student forum,” in *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, C. W. Barrett and J. Yang, Eds. IEEE, 2019, p. 1. [Online]. Available: <https://doi.org/10.23919/FMCAD.2019.8894257>
- [8] P. Schrammel, “The FMCAD 2020 student forum,” in *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, p. 1. [Online]. Available: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_6

COCOALMA: A Versatile Masking Verifier

Vedad Hadžić 
Graz University of Technology

Roderick Bloem 
Graz University of Technology

Abstract—Masking techniques are an effective countermeasure against power side-channel attacks. Unfortunately, correctly masking a hardware circuit is difficult, and mistakes may lead to functionally correct circuits with insufficient protection. We present COCOALMA, a tool that formally verifies the side-channel resistance of stateful hardware circuits. Although COCOALMA was initially used to verify programs running on CPUs, we extended it to verify the security of several industrial masked hardware implementations. We give an overview of the tool’s structure, implementation details, optimizations that make it faster and more scalable than its predecessor REBECCA, and changes that enable verifying the probing security of any stateful hardware circuit. Finally, we evaluate COCOALMA with masked implementations of the PRINCE and AES ciphers.

Index Terms—Side-channels, Hardware masking, Formal verification

I. INTRODUCTION

Integrated circuits that process sensitive data are susceptible to passive *side-channel attacks* like differential power analysis. Naturally, attackers are interested in the secret keys of symmetric ciphers because that would break the confidentiality of the processed data [22], [23], [26], [21]. Classical power analysis attacks exploit the correlation of the circuit’s power consumption to bits of the secret key. Ultimately, the key is reconstructed using statistic analysis techniques in a series of key guesses [22], [27].

Masking is an algorithmic countermeasure against power analysis attacks. It relies on splitting all secrets and intermediate computations into multiple signals. The circuit is rewritten so that attackers can only reconstruct the original value if they can observe all the shares simultaneously. Masking techniques achieve this by introducing randomness into the circuit and destroying the correlation between the power-trace and the original data. Several masking schemes describe how to make circuits secure against side-channel attacks. Among them, *domain-oriented masking* [15] and *threshold implementations* [9] are well studied and widely adopted. The security of masked hardware circuits is expressed using the *hardware probing model* [2], [18], [4], where an attacker can read the values of d wires. Traditionally, engineers validate masked hardware implementations empirically by creating power traces and computing the correlations over many executions. Recently, however, we see several formal masking verification methods that can substantially reduce the costs of validating power side-channel resistance of software and hardware [2], [1], [11].

This work was supported by the *Austrian Research Promotion Agency* (FFG) through the FERMION project (grant number 867542).

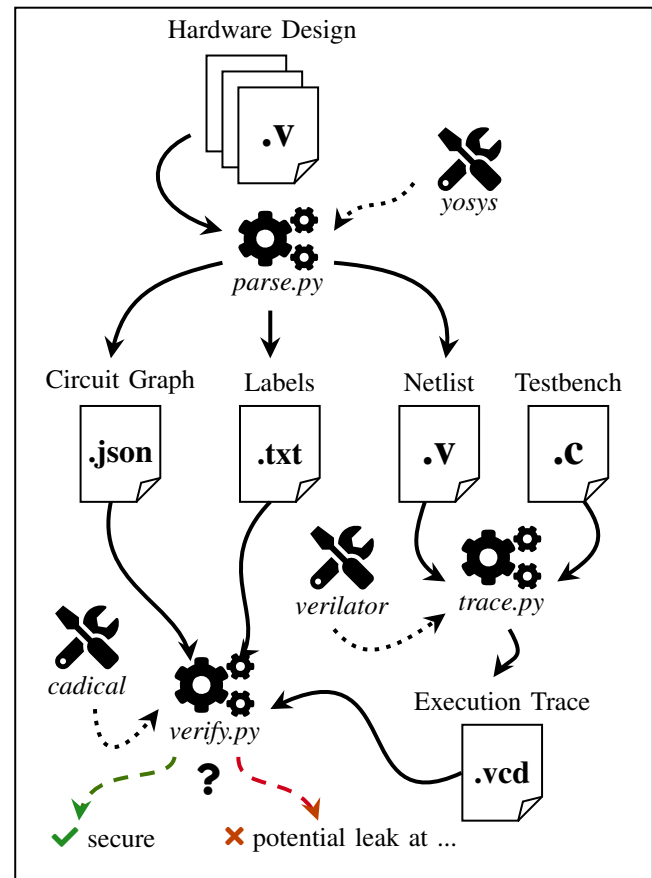


Figure 1. The workflow of COCOALMA showing the *parsing*, *tracing*, and *verification* phases, as well as their artifacts. At the end of the verification phase, COCOALMA either acknowledges that the analyzed design is secure or shows that a secret is leaked at a given location in the circuit.

COCOALMA is an open-source masking verifier¹ that assisted the hardening of a RISC-V processor² so it could safely execute masked software [13]. It considers the exact description of the hardware that runs the software and accounts for hardware leakage effects such as glitches. Figure 1 shows the workflow of COCOALMA. Starting with a hardware design written in Verilog, COCOALMA uses Yosys [31] to synthesize a flat gate-level Verilog netlist. Additionally, the parsing phase extracts a circuit graph of the synthesized design and creates a labeling template where the user can specify the contents of each register and input port of the circuit after the reset.

¹<https://github.com/IAIK/coco-alma>

²<https://github.com/IAIK/coco-ibex>

COCOALMA uses a testbench provided by the user to simulate the netlist with Verilator [28], resulting in a *value change dump* showing how the internal signals changed throughout the execution. For the analysis of software running on RISC-V processors, COCOALMA additionally requires the RISC-V toolchain to compile programs and add them to the testbench before starting the simulation. The resulting execution trace is used to determine the value and glitching properties of each wire in the design. Afterward, the time-constrained probing model, initial state, simulation trace, and glitching information are encoded as a SAT problem and solved with CaDiCaL [3]. If the problem is unsatisfiable, no possible observation would leak any of the secrets. Otherwise, COCOALMA gives a precise description of leakage location, the secret bits that are leaked, and a variety of other debugging information.

Although COCOALMA was first used for analyzing software running on CPUs [13], its roots in the older verification tool REBECCA [4] can be leveraged towards stateful hardware verification of masked cipher implementations. Luckily, all the principles used in COCOALMA also apply to hardware masking verification with minor tweaks. In this paper, we document the inner workings of COCOALMA, its features, and show the extensions necessary for applying it to cryptographic accelerator modules. We present the following details about COCOALMA’s implementation:

- In Section II, we define the supported probing models, emphasizing the newly supported *hardware probing model*, which allows us to prove the security of stateful hardware circuits. We also discuss the support for *random number generators*.
- In Section III-A, we give a breakdown of the *correlation set* methodology and show its encoding into a SAT formula in Section III-B. Here we give a precise description of the encoding, which is missing in the original publication [13], and more efficient than the encoding used in REBECCA [4]. Finally, in Section III-C, we describe details of several optimizations that reduce the size of the encoding and the number of probing locations. Here, the *hardware probing model* requires special considerations.
- In Section IV, we motivate and describe the execution-dependent correlation set simplifications. Additionally, we present the *stable signal detection* algorithm computing the *stability* of each control signal in Section IV-A. This optimization allows us to simplify the correlation sets even in the presence of glitches.
- In Section V, we demonstrate COCOALMA’s capabilities by verifying the probing security of state-of-the-art masked implementations of the PRINCE [6], [12], [20] and AES [30], [7], [17], [15] ciphers as they are popular in the semiconductor industry. Additionally, we go over the debugging tools provided with COCOALMA, which allow a designer to locate the source of the leakage and see how leakage propagates through the circuit.

II. SECURITY MODELS

Masked implementations split all intermediate data signals x into $d+1$ uniformly random pieces x_i , with $x = x_0 \oplus \dots \oplus x_d$. In practice, for $i \neq d$, the signal shares x_i are sampled from a random number generator, whereas x_d is chosen as $x \oplus x_0 \oplus \dots \oplus x_{d-1}$ to fit the equality. This countermeasure tries to prevent an attacker, who can observe intermediate computations through side-channels, from learning anything about the processed data. When investigating whether a masked implementation is actually side-channel resistant, several security models describe the capabilities of an attacker and the real-world effects they can observe. COCOALMA implements three different probing models that consider different attacker capabilities and system behavior. More specifically, this work extends COCOALMA to support continuous probing as part of the *hardware probing model*.

Software probing model. The original probing model defined by Ishai et al. [18] considers the stable state of computations, ignoring hardware side-effects such as glitches and transitions. Their seminal paper says that an attacker in this probing model can choose d intermediate values that they can observe. The attacker can then interactively query the execution of the system several times with different inputs and starting states. The inputs of the computation are declared either (a) *public*, which means that learning them does not benefit the attacker, (b) fixed uniformly random values called *masks*, or (c) parts of a secret called *shares*. The attacker’s goal is to learn all the shares of a secret and use them to reconstruct the secret value they are not supposed to know. Proving that an implementation is d -probing secure requires showing that no attacker adhering to this probing model can learn the secrets, irrespective of their strategy.

Time-constrained probing model.³ When COCOALMA was first presented [13], its primary goal was verifying the masking of software programs running on an accurate description of the underlying hardware. Naturally, this required an adequate probing model that translates software probing into the hardware domain. The *time-constrained probing model* uses the gate-level description of the hardware and an execution trace generated by simulating the hardware running the software, instead of a purely algorithmic description. The goals of the attacker are the same as in the *software probing model*. However, this model is more realistic, as the attacker can probe d observation tuples (g, t) , where g is a logic gate or register and t is a cycle in the execution trace. This gives an attacker access to all the intermediate values of gate g in cycle t , including all the values caused by hardware effects such as glitches and register transition leakage. The two parameters g and t are not coupled, meaning that the attacker can also probe the same gate in multiple clock cycles or even probe d different gates in the same clock cycle. Although this model limits each probe to observing only one clock cycle, instead of running throughout the computation, its inclusion of hardware effects significantly enhances the capabilities of an attacker.

³Barthe et al. [2] and Moos et al. [24] call this the *robust probing model*.

Due to the different signal timings in hardware, an attacker observing gate $g = a \odot b$ in this model would also observe the signals a and b in addition to g . Registers are synchronous elements triggered by a clock, making them the only hardware elements exempt from this phenomenon. Another effect that increases the attacker's capabilities is transition leakage, which causes the power consumption to correlate with the linear combination $g^{t-1} \oplus g^t$ of the old signal value in cycle $t - 1$ and the new signal value in cycle t . Transition leakage applies to all hardware elements equally, including registers.

Hardware probing model. This paper extends the tool COCOALMA with a model where probes are not bound to one clock cycle like in the *time-constrained probing model*. The attacker's goals remain the same as before, only that in this more rigorous model, the probes record continuously throughout the whole computation. More precisely, instead of choosing a clock cycle for each observed location, the attacker observes all values, including those caused by glitches and transitions, that pass through a wire. In a sense, this is a more powerful rephrasing of the original probing model of Ishai et al. [18], as they also did not limit the duration of the probes for stateful circuits. As this model significantly increases the capabilities of an attacker, hardware designers employ random number generators to create fresh uniformly random masks in each clock cycle, intending to break any correlations that might otherwise be observed. These mask-generating circuits are usually not part of the masked hardware designs and are only used as black-boxes that provide random inputs to the masked circuit. We incorporate this in COCOALMA, allowing designers to label input ports of a circuit as *random*. The values read from these ports behave similarly to fixed *masks*, only that they represent a new mask in each clock cycle, which is then considered during verification. The semantics of *public* and *share* signals remains the same, and we even allow fixed *masks*, just like in the other probing models.

III. VERIFICATION METHOD

COCOALMA tries to verify the side-channel resistance of a masked implementation in one of the given security models. A correctly masked implementation computes the values of arbitrary logic functions without exposing the value of the secret to an attacker through intermediate computations. Therefore, a masked implementation must ensure that intermediate signals do not correlate with *secrets*; that is, the value of an intermediate signal should be statistically independent of all secrets. COCOALMA checks whether these properties hold by tracking the correlations of each logic operation throughout the computation [4], [13]. For instance, if a circuit were to compute the expression $f = a \wedge b$, then f correlates positively with a , b , and the constant \perp because they have the same value in three out of four cases. For the same reason, f correlates negatively with the linear combination $a \oplus b$ because they only have the same value in one of four cases, *i.e.*, when both a and b are \perp . An exact algorithm that computes these correlations would solve the #SAT problem [14], meaning that computing

Table I
PROPAGATION RULES FOR STABLE AND TRANSIENT CORRELATION SETS

Gate type of f		Stable set S_f^t	Transient set T_f^t
Constant	\perp or \top	$\{\perp\}$	$\{\perp\}$
Input Port	p^t	$\{p^t\}$	$\{p^t\}$
Negation	$\neg a$	S_a^t	T_a^t
Register	$\leftarrow_R a$	S_a^{t-1}	S_a^{t-1}
Linear	$a \oplus b$	$S_a^t \otimes S_b^t$	$\langle T_a^t \rangle \otimes \langle T_b^t \rangle$
Non-linear	$a \wedge b$ $a \vee b$	$\langle S_a^t \rangle \otimes \langle S_b^t \rangle$	$\langle T_a^t \rangle \otimes \langle T_b^t \rangle$
Multiplexer	$c ? a : b$	$\langle S_c^t \rangle \otimes (S_a^t \cup S_b^t)$	$\langle T_c^t \rangle \otimes \langle T_a^t \rangle \otimes \langle T_b^t \rangle$

correlations is at least #P-Complete [29], which is harder than NP by definition. Because of the structure of secrets and the uniform randomness of secret shares and masks, it is sufficient to track the correlations to linear combinations of the inputs [4]. Furthermore, the correlations yield a sound over-approximation that reduces the complexity of the problem and is also used in COCOALMA. In the following sections, we describe this over-approximation and its implementation, but refer to the soundness proofs in the original publication [4].

A. Correlation Sets

Instead of painstakingly computing the exact correlation factor for each linear combination of inputs, COCOALMA over-approximates the correlations. In particular, COCOALMA only considers whether the correlation factor is non-zero, and ignores its exact value. All linear combinations a gate correlates to are grouped together and tracked as so-called *correlation sets*. The exact correlations are approximated using propagation rules that determine the correlation set of $f = a \odot b$ by considering the correlation sets of a and b , as well as the used logic operation \odot . Using the previous example $f = a \wedge b$, we have shown that the correlation set contains all linear combinations of a and b , *i.e.*, $\{\perp, a, b, a \oplus b\}$. In contrast, $f = a \oplus b$ only correlates with itself, *i.e.*, the set $\{a \oplus b\}$, because the value of $a \oplus b$ coincides with \perp , a , and b in exactly half of the cases, yielding a correlation factor of zero. Consequently, knowing f would not reveal any information about a and b . In general, we cannot compute the correlation set of the output of a logical operation precisely from the correlation sets of its inputs, so COCOALMA over-approximates these sets.

Table I presents the propagation rules COCOALMA uses to compute the correlation sets of a gate using its inputs. The propagation rules define two kinds of correlation sets necessary for the verification: (a) *stable* sets S_f^t that define the normal behavior of a gate f , and (b) *transient* sets T_f^t that define the behavior of f in the presence of glitches and transition leakage effects. Both types of correlation sets are defined for each clock cycle t , as gates change their value over time. Although the hardware probing model only talks about these transient correlation sets, the stable correlation sets are necessary for synchronizing elements such as registers. For simpler exposition and encoding, Table I shows the computation of correlation sets using the operators \otimes and

$\langle \cdot \rangle$. Here, \otimes is the element-wise exclusive-or between two correlation sets, i.e., $X \otimes Y = \{x \oplus y \mid x \in X, y \in Y\}$. The operator $\langle \cdot \rangle$ adds a correlation with \perp to a correlation set, i.e., $\langle X \rangle = X \cup \{\perp\}$.

The presented propagation rules are based on COCOALMA's original publication [13], [4] but were adapted for stateful hardware verification with continuously recording probes. Naturally, constants only correlate to \perp , and negations only change the sign of the correlation but do not impact the correlations themselves. As discussed previously, linear gates only correlate to the linear combination of the inputs, so the correlation set is computed as the element-wise exclusive-or of the inputs' correlation sets. For non-linear gates, the correlation set is computed similarly, only that in this case, a bias is introduced in each input's correlation set. Using the introduced notation, the correlation set of gate $f = a \wedge b$, where a and b are inputs, is computed as

$$\langle \{a\} \rangle \otimes \langle \{b\} \rangle = \{\perp, a\} \otimes \{\perp, b\} = \{\perp, a, b, a \oplus b\}. \quad (1)$$

For transient correlations, linear gates behave like non-linear gates. Glitches induced by different signal timings can force a gate to forward a constant or either of the inputs, in addition to the correct correlations. A multiplexer correlates to both of its data inputs a and b , as well as their linear combinations with the selector c , i.e., $a \oplus c$ and $b \oplus c$. For the transient correlation set, COCOALMA assumes that all three input signals can be combined non-linearly.

When verifying masked software running on a processor, the input pins of the hardware design are not relevant, as they are part of the micro-architecture and not visible to the programmer. Secret shares, masks, and public values are all stored in both the RAM and the ROM, and for the verification process, we label their locations and simulate the design to execute a program [13]. Verifying masked hardware is different, as there are no such memory blocks, and the registers get cleared with a reset signal. Computation-relevant data, such as plaintexts, keys, and masks, is provided by the environment through the input ports of the circuit. Therefore we extend COCOALMA with support for input ports and introduce an appropriate propagation rule, which states that an input port only correlates to its value in cycle t . In our implementation, *public* values, *shares*, and *masks* have the same value throughout the execution of the circuit. However, input ports labeled as *random* are provided by an external *random number generator* and change their value in each cycle, and therefore, the correlation set also changes each cycle. In addition, to the support for input ports, we also optimized the propagation rules for registers. Since the probes in the *hardware probing model* record data continuously, we do not need to account for transition leakage because all values passing through a wire are recorded anyway.

Computing correlation sets from other correlation sets can result in over-approximations that include non-existent correlations. For example, representing the exclusive-or function $f = a \oplus b$ as $f = (a \wedge \neg b) \vee (\neg a \wedge b)$ would result in the spurious correlation set $\{\perp, a, b, a \oplus b\}$, when in reality f only

correlates with $\{a \oplus b\}$. This means that a hardware designer applying this over-approximative method must be aware of false leakage reports and debug them properly. Oftentimes, as illustrated in this toy example, the over-approximative error can be fixed by either re-writing the circuit or removing the problematic correlation term from the correlation set.

However, despite being imprecise, this over-approximation is easy to encode and retains some useful information. For example, function $f = (a \oplus b) \wedge c$ is correctly claimed to correlate with $\{\perp, c, a \oplus b, a \oplus b \oplus c\}$, even though the correlation set of f was computed using the correlation sets of $g = a \oplus b$ and c . This result reflects the intuition that we cannot “remove” masking from a signal by combining it with another value, i.e., the correlation set does not contain values where a appears without b .

B. SAT Encoding

The upper bound for the size of the correlation sets is exponential in the number of inputs, so COCOALMA cannot store or enumerate them explicitly and instead relies on an implicit encoding method that utilizes a SAT solver. While the used encoding is similar to the one presented by Bloem et al. [4], it was significantly optimized and streamlined in COCOALMA to simplify the implementation of all the propagation rules in Table I. As mentioned previously, the user needs to label each input port $p \in \mathcal{I}$ as either a *share* $s \in \mathcal{K}^i$ of the i -th secret, a fixed random *mask* $m \in \mathcal{M}$, a *random port* with a new value $r \in \mathcal{R}^t$ in each clock cycle t , or a public value that is ignored. For simpler notation, we do not implicitly associate correlation sets or propositional variables with clock cycles or gates in the circuit, and instead specify them with \mathcal{C}_- and \mathcal{P}_- , where the subscript is used to differentiate them. In our SAT encoding, a correlation set \mathcal{C}_x is represented by a set of propositional variables $\mathcal{P}_x = \{x_p \mid p \in \mathcal{I}\}$, such that every valid assignment to the propositional variables \mathcal{P}_x corresponds to an element in the correlation set \mathcal{C}_x . Additionally, just like \mathcal{I} , \mathcal{P}_x can be further split as $\mathcal{P}_x = \bigcup_i \mathcal{K}_x^i \cup \mathcal{M}_x \cup \bigcup_t \mathcal{R}_x^t$. Example 1 gives an intuition of the introduced variable sets and correlation set encoding.

Example 1: Let $\mathcal{I} = \{s_0, s_1, m\}$ be the labeled input ports given by the user, where $s = s_0 \oplus s_1$ is a secret with shares $\mathcal{K}^0 = \{s_0, s_1\}$, and fixed uniformly random masks $\mathcal{M} = \{m\}$. Let $\mathcal{C}_x = \{\perp, s_1, s_0 \oplus m, s_0 \oplus s_1 \oplus m\}$ be a correlation set. Then $\mathcal{P}_x = \{x_{s_0}, x_{s_1}, x_m\}$ are the propositional variables used for encoding \mathcal{C}_x , where $\mathcal{K}_x^0 = \{x_{s_0}, x_{s_1}\}$, and $\mathcal{M}_x = \{x_m\}$, and there are no random ports. The propositional variables in \mathcal{P}_x are constrained in such a way that the only satisfying assignments for the propositional tuple (x_{s_0}, x_{s_1}, x_m) are (\perp, \perp, \perp) , (\perp, \top, \perp) , (\top, \perp, \top) , and (\top, \top, \top) . These assignments represent the elements of \mathcal{C}_x , where x_p indicates whether the port p appears in the current term of \mathcal{C}_x .

COCOALMA maps the correlation terms in \mathcal{C}_x to satisfying assignments to the propositional variables \mathcal{P}_x by translating the propagation rules from Table I into satisfiability constraints. However, in order to simplify the exposition, we only

demonstrate how we encode the correlation set operations $\langle \cdot \rangle$, \cup , and \otimes , as well as the creation of a correlation set with only one element. All of the propagation rules from Table I can be obtained by applying different combinations of these individual encodings, e.g., the transient rule for linear gates is obtained by combining the encodings of $\langle \cdot \rangle$ and \otimes .

First off, the correlation set of an input port only contains the port itself. Therefore, we restrict all of its propositional variables that correspond to other ports to be \perp , whereas the propositional variable representing the port itself must be set to \top . More precisely, for a port p in clock cycle t , the propositional variables \mathcal{P}_x are constrained with

$$x_{pt} \wedge \bigwedge_{x_a \in \mathcal{P}_x, a \neq p^t} \neg x_a, \quad (2)$$

where only *random* input ports are different in each clock cycle and $p = p^t$ in all other cases.

Extending a correlation set \mathcal{C}_x with the \perp element, written as $\langle \mathcal{C}_x \rangle$, is required for the propagation rules of linear and non-linear operations. When translating this into constraints for propositional variables \mathcal{P}_x , COCOALMA introduces a new set of variables \mathcal{P}'_x and a fresh propositional variable q . The SAT solver can pick the value of q freely. Depending on the choice, all propositional variables \mathcal{P}'_x are forced to equal their corresponding variables in \mathcal{P}_x or forced to be \perp . We write this constraint as

$$\bigwedge_{x_a \in \mathcal{P}_x, x'_a \in \mathcal{P}'_x} x'_a \leftrightarrow (q \wedge x_a). \quad (3)$$

All satisfying assignments of \mathcal{P}'_x correspond to elements of the correlation set $\langle \mathcal{C}_x \rangle$. Each time the propagation rules in Table I use the $\langle \cdot \rangle$ operator, we introduce the variables \mathcal{P}'_x and q and apply the given constraint.

Encoding the propagation rule for multiplexers requires a similar constraint when representing the union of two correlation sets. Given the correlation set $\mathcal{C}_z = \mathcal{C}_x \cup \mathcal{C}_y$, we introduce corresponding propositional variables \mathcal{P}_z and a fresh propositional variable q . We subsequently constrain the introduced propositional variables with

$$\bigwedge_{z_a \in \mathcal{P}_z, x_a \in \mathcal{P}_x, y_a \in \mathcal{P}_y} z_a \leftrightarrow ((q \wedge x_a) \vee (\neg q \wedge y_a)), \quad (4)$$

where whenever $q = \top$ an element of \mathcal{C}_x is encoded, and otherwise an element of \mathcal{C}_y . This encoding ensures that \mathcal{C}_z contains all elements of \mathcal{C}_x and \mathcal{C}_y , even if they are duplicates.

Finally, COCOALMA encodes the element-wise exclusive-or of two correlation sets $\mathcal{C}_z = \mathcal{C}_x \otimes \mathcal{C}_y$ using their corresponding propositional variables and a straightforward equivalence encoding

$$\bigwedge_{z_a \in \mathcal{P}_z, x_a \in \mathcal{P}_x, y_a \in \mathcal{P}_y} z_a \leftrightarrow (x_a \oplus y_a). \quad (5)$$

Unlike the encoding of unions, no additional fresh propositional variables are necessary as there is no choice involved.

The constraints (2)-(5) only show how each of the propagation rules shown in Table I can be translated into SAT.

COCOALMA needs an additional encoding for the conditions under which information leakage occurs. With correlation sets, we check whether there is an element of the correlation set where all shares of a secret are present, without being hidden by uniformly random values, such as fixed masks, random input ports, or shares of other secrets. Looking back at Example 1, we see that each time both shares s_0 and s_1 appear in a correlation term, they are masked by mask m . This means that the correlation set does not leak information about $s = s_0 \oplus s_1$. When checking this leakage property using the SAT encoding, we require two constraints.

First, we enforce that for each secret, either all shares are active, or all shares are inactive. Furthermore, we say that at least one secret must be active in order to have a leak. We encode this property by introducing one fresh propositional variable k_i for each secret and constraining them with

$$\left(\bigvee_i k_i \right) \wedge \bigwedge_i \bigwedge_{x_s \in \mathcal{K}_x^i} k_i \leftrightarrow x_s. \quad (6)$$

The first conjunct guarantees that at least one of the secrets is present in the correlation term. The rest of the expression ensures that either all shares of a secret are active in a correlation term, or none of them are, which is necessary since shares of incomplete secrets are uniformly random.

Second, we enforce that no masks appear in the correlation term, so the secrets are not *hidden* by uniformly random values, as discussed in Example 1. We represent this in the SAT encoding as

$$\left(\bigwedge_{x_m \in \mathcal{M}_x} \neg x_m \right) \wedge \left(\bigwedge_t \bigwedge_{x_r \in \mathcal{R}_x^t} \neg x_r \right), \quad (7)$$

which ensures that a satisfying solution must assign all the variables representing masks and random values with \perp .

Constraints (6) and (7) go hand in hand, and both are required when testing whether a given correlation set leaks information about the secrets. When checking the security of a circuit in one of the supported security models, COCOALMA determines the observations an attacker can make, where each observation is made up of multiple correlation sets. For the *software probing model*, COCOALMA takes all the d -tuples \mathcal{O} of probing locations (g, t) and tests the non-linear combination of their stable correlation sets

$$\bigotimes_{(g,t) \in \mathcal{O}} \langle S_g^t \rangle, \quad (8)$$

where g is the chosen gate, and t is the chosen clock cycle. The same applies to the *time-constrained probing model*, where COCOALMA checks the transient correlation sets T_g^t instead. In contrast, for the full *hardware probing model*, the probing locations \mathcal{O} are a d -tuple of gates g instead, and concern all the clock cycles t for the given gates. Therefore, COCOALMA must check the correlation set

$$\bigotimes_{g \in \mathcal{O}} \bigotimes_t \langle T_g^t \rangle, \quad (9)$$

which significantly increases the observations an attacker can make. For example, using a register to store one share of a secret early in the computation and store the other share later in the computation would still allow an attacker to reconstruct the secret. Naturally, longer executions of a circuit get progressively harder to verify.

C. Encoding Optimizations

Although the shown SAT encoding is sufficient for showing whether the circuit leaks information about the processed secrets, the size of the produced constraints and formulas is unnecessarily large. In this section, we present some of the optimizations that dramatically reduce the effort of showing that a masked hardware circuit is secure.

Variable elimination. The sets of propositional variables \mathcal{P}_x often include variables constrained through unit clauses, so their assignment is predetermined and equal in all satisfying solutions. Constraint (2) is an example of such a situation. Building constraints for such variables is unnecessary, and they can be removed entirely, substantially reducing the size of formula given to the SAT solver. In practice, COCOALMA implements this by storing \mathcal{P}_x as a dictionary of propositional variables, as well as a set of variables trivially set to \top . All variables from \mathcal{P}_x that are not present are known to have the value \perp . Consequently, whenever creating any of the shown constraints (3)–(7), we first check for trivial simplifications using the properties of logic operators. Although this optimization might seem superficial, it single-handedly reduces the number of variables and clauses by anywhere between 90% and 98% for the probing verification problems we have investigated so far. Notably, this optimization does not reduce the complexity of the queries given to the SAT solver, as solvers usually detect unit clauses anyway, but instead significantly reduces the memory consumption. Without this optimizations, verifying the probing security of longer executions would not be possible because the formula would not fit into memory.

Covering sets. Due to the nature of the propagation rules from Table I, some correlation sets are supersets of others. Take the propagation rules for non-linear gates as an example. For gate $f = a \wedge b$, the stable correlation set is computed as $S_f^t = \langle S_a^t \rangle \otimes \langle S_b^t \rangle = \{\perp\} \cup S_a^t \cup S_b^t \cup (S_a^t \otimes S_b^t)$, which implies that $S_a^t \subseteq S_f^t$ and $S_b^t \subseteq S_f^t$. Consequently, it is sufficient to perform the security checks for S_f^t , ignoring both S_a^t and S_b^t because their elements are already *covered*. For element-wise exclusive-or operations like $C_z = C_x \otimes C_y$, the resulting set C_z covers C_x whenever $\perp \in C_y$, and C_y whenever $\perp \in C_x$. It turns out that in the *software probing model*, we only need to check gates that are inputs to XOR gates, selectors of a multiplexer, inputs to a register, and circuit outputs. In the *time-constrained probing model*, we only check register inputs and circuit outputs because in that model linear gates behave non-linearly due to glitches. In the full *hardware probing model*, the covering properties are slightly more complex, and we check all gates that have at least one clock cycle where another gate does not cover them.

Table II
SIMPLIFICATION RULES FOR STABLE CORRELATION SETS

Gate type	f	Stable set \mathcal{C}_f	f	Stable set \mathcal{C}_f
Linear	$a \oplus \perp$	\mathcal{C}_a	$a \oplus \top$	\mathcal{C}_a
Non-linear	$a \wedge \perp$	–	$a \wedge \top$	\mathcal{C}_a
	$a \vee \perp$	\mathcal{C}_a	$a \vee \top$	–
Multiplexer	$\perp ? a : b$	\mathcal{C}_b	$\top ? a : b$	\mathcal{C}_a
	$c ? \perp : b$	$\langle \mathcal{C}_c \rangle \otimes \langle \mathcal{C}_b \rangle$	$c ? \top : b$	$\langle \mathcal{C}_c \rangle \otimes \langle \mathcal{C}_b \rangle$
	$c ? a : \perp$	$\langle \mathcal{C}_c \rangle \otimes \langle \mathcal{C}_a \rangle$	$c ? a : \top$	$\langle \mathcal{C}_c \rangle \otimes \langle \mathcal{C}_a \rangle$

IV. SIMULATIONS

Although the method presented in Section III is sufficient to check the security of a masked implementation in the supported probing models, it does not consider how the control signals change over time. As mentioned in the introduction, COCOALMA uses simulations to obtain information about the exact values of control signals and subsequently uses them to simplify the correlation sets accordingly.

In the hardware probing model, all values marked as *sensitive*, i.e., secret shares, mask registers and random input ports, are assumed to be uniformly random. This is a requirement for the execution environment, in this case the testbench, which performs the secret sharing steps and includes a random number generator that drives the random input ports in each clock cycle. In any reasonable probing model, the attacker can only control the values of un-shared plaintext values, and we assume they can request an unlimited number of encryptions for the DPA attack. If the attacker were able to mess with the random number generator of the environment, they would be able to break any conceivable masking scheme, so this is out-of-scope in the hardware probing model.

Other input signals, such as control signals, which marked as *public* are assumed to be independent of the secrets and masks processed in the hardware circuit, so their values can be taken directly from a circuit simulation. Since their values are known, COCOALMA uses them to perform simplifications while applying the propagation rules. Consider the gate $f = a \wedge b$, where a is a public value and b has a correlation set \mathcal{C}_b . Because COCOALMA knows the value of a , f is simplified accordingly. If $a = \perp$, then we know that $f = \perp$ independently of b , meaning that f is also a public value and does not need a correlation set. Similarly, if $a = \top$, we know that $f = b$, and we can reuse the correlation set as $\mathcal{C}_f = \mathcal{C}_b$. Table II defines analogous simplifications for all propagation rules with multiple inputs when the constant signal is stable. Using the simulated execution of the circuit and the labeling provided by the user, each gate g at each clock cycle t is classified as either being a control signal or having a correlation set, but never both. Empty entries in Table II indicate that the gate does not have a correlation set and is instead declared a control signal.

A. Signal Stability

Unlike with stable correlation sets, applying simplifications based on the simulation trace is not straightforward for transient correlation sets, where COCOALMA must also consider

Table III
SIGNAL STABILITY COMPUTATIONS

Gate type of f	Computation of $st(f)$ in current clock cycle
Constant	\perp or \top
Input Port	$\neg cr(p)$
Negation	$st(a)$
Register	$\neg cr'(a) \wedge (vl'(a) \leftrightarrow vl'(f))$
Linear	$st(a) \wedge st(b)$
Non-linear	$st(a) \wedge \neg vl(a) \vee st(b) \wedge \neg vl(b) \vee st(a) \wedge st(b)$ $st(a) \wedge vl(a) \vee st(b) \wedge vl(b) \vee st(a) \wedge st(b)$
Multiplexer	$st(c) \wedge (vl(c) \wedge st(a) \vee \neg vl(c) \wedge st(b)) \vee$ $\vee st(a) \wedge st(b) \wedge (vl(a) \leftrightarrow vl(b))$

glitches. Glitches are hardware phenomena that behave like temporary faults while switching values. A gate $f = a \odot b$ will pass on a 's value if its signal arrives at f before the new signal of b . After both signals arrived, the fault is corrected, and f becomes the value it is supposed to have. Ultimately, the signal must be stable at the end of a clock cycle, when the clock triggers the registers and synchronizes the computation.

However, there are certain conditions when a gate cannot experience a glitch, e.g., when the values a and b come directly out of a register and do not change from the previous clock cycle. In that particular case, even though the signal timings are different, the value transmitted through the wires did not change the entire time, and no glitching is possible. As a result, even the signal produced by f would be stable and glitch-free. This property recursively propagates throughout the whole circuit and allows us to determine which values can be used for the simplifications shown in Table II, even for transient correlation sets.

COCOALMA uses the concrete values of a simulation trace to determine the glitching behavior of public values such as control signals. Assume the same situation as before, with $f = a \wedge b$, where a is a public value and b might correlate with masks or shares, and thus, has a correlation set \mathcal{C}_b . Knowing whether f can forward b is crucial, as it might lead to an information leak in a later part of the circuit. If $a = \perp$ and its signal is stable, meaning it cannot produce glitches, then f is a public value with $f = \perp$. Therefore, a being a stable public signal set to \perp effectively stops the propagation of a correlation set from b to f . In the rest of this section, we outline a recursive method for determining whether a signal is stable in a given clock cycle.

In the following exposition, we introduce three predicates that help define the algorithm computing the signal stability. We use the $st(x)$ predicate to say that the signal x is stable. The predicate $cr(x)$ is true whenever the signal x is associated with a transient correlation set. Finally, predicate $vl(x)$ represents the value of signal x taken from the execution trace. All three predicates also have a version that applies to the previous clock cycle: $st'(x)$, $cr'(x)$, and $vl'(x)$. The rules computing the stability of any given signal f are shown in Table III. All values of the predicates are computed directly, and none of them are given to the SAT solver.

First, all input ports are held stable by the environment. That is, another circuit that controls the input ports must keep

Table IV
VERIFICATION RESULTS FOR TWO VERSIONS OF PRINCE-TI

Algorithm	#Sec.	#Rand.	#Rnds.	#Cyc.	SW	TC	HW
PRINCE-TI	192	48	1	3	✓ 0.72 s	✗ 1.97 s	✗ 2.43 s
PRINCE-TI	192	192	1	3	✓ 3.37 s	✓ 7.21 s	✓ 11.57 s
PRINCE-TI	192	192	2	5	✓ 187.8 s	✓ 150.6 s	✓ 236.9 s
PRINCE-TI	192	192	3	7	✓ 0.77 h	✓ 3.80 h	✓ 17.92 h
AES-DOM	256	46	1	21	✓ 195.3 s	✓ 1.82 h	✓ 2.89 h

their signals stable and avoid glitches. Since public signals and signals with correlation sets are mutually exclusive in COCOALMA, an input port is only considered stable when it does not have a correlation set. Similarly, the output of a register is stable if the register does not change its value from the previous cycle and does not have a correlation set associated with its input. If the value did change, we consider the signal unstable because it can cause glitches in gates connected to it during the clock-cycle transition. Linear gates such as XOR are only stable if both of their inputs are stable. If one of the inputs produces a glitch, then an XOR would forward it to all gates it is connected to since the other signal cannot stop it.

Non-linear gates such as AND (OR) can remain stable even if one of their inputs produces glitches. If at least one of the inputs of an AND (OR) gate is stable at \perp (\top), then no change or glitch in the other input can make it unstable. Otherwise, the output of an AND (OR) gate is only stable if both of its inputs are also stable. The conditions under which a multiplexer is stable are similar. For instance, if selector c is stable with the value \top (\perp), then the output of the multiplexer is stable if and only if the selected input a (b) is stable. In contrast, if selector c is not stable, the output is only stable if the inputs a and b are stable and have equivalent values.

V. CASE STUDIES

In this section, we investigate the probing security of the masked hardware implementations PRINCE-TI [6] and AES-DOM [16]. In particular, we analyze the complexity of verifying round-reduced versions in all three of the supported probing models. Additionally, we demonstrate how COCOALMA's debugging functionalities allow us to identify potential issues and fix them accordingly. All experimental results shown in Table IV were captured on a notebook with the Intel Core i7-8550U 1.8GHz CPU and 16 GiB of RAM.

A. Verifying PRINCE-TI

PRINCE is a state-of-the-art lightweight block cipher. It is designed with hardware implementations in mind, so that ideally, the entire encryption process can be done in one clock cycle [5] when no masking is applied. PRINCE takes as input a 64-bit plaintext block and encrypts it with a 128-bit key. The encryption process consists of two phases with six rounds each. In the first phase, the first round adds the round key onto the data block, whereas the other five rounds apply a 4-bit S-Box, an affine transformation, and then mix the round key into the data block. After the first phase, the

data block is transformed using the 4-bit S-Box, another affine transformation, and the inverse 4-bit S-Box, before starting the second phase. In the second phase, each round applies the inverse operations performed in the rounds of the first phase, meaning that the first five rounds add the round key, apply the inverse affine transformation followed by the inverse 4-bit S-Box. The last round of the second phase only adds the round key to the data block.

Unlike the unmasked version of PRINCE, the threshold implementation PRINCE-TI [6] cannot be completed in one clock cycle. This restriction is due to the re-sharing phase present in threshold implementations, which requires additional synchronization to prevent leakage caused by glitches. For first-order probing security, the implementation splits all the plaintext and key bits into two shares and treats them as secrets. PRINCE-TI uses random inputs to re-share the outputs of its sixteen 4-Bit S-Boxes, where each S-Box requires twelve random bits. In the official implementation, this process is optimized in such a way that four S-Boxes share the same randomness, so the re-sharing only requires a total of 48 random bits.

The first row of Table IV shows the results produced by COCOALMA, where 192 (*i.e.*, 128 key bits and 64 plaintext bits) pairs of ports are labeled as shares of secrets, and 48 ports are labeled as coming from a random number generator. The first round of the cipher needs three clock cycles to complete since we first need to load the inputs into internal registers and start the encryption. Within one second, COCOALMA has proven that the implementation is secure in the *software probing model* (SW), indicated with (✓) in Table IV. However, COCOALMA claims it found a leak (✗) in the *time-constrained probing model* (TC) in the third clock cycle and provides us with debugging information.

B. Debugging Information

After finding a leak in a hardware circuit, COCOALMA attempts to simplify the leaking correlation. For example, COCOALMA could report that the output of a gate correlates with the linear combination of many secrets. This information, while correct, is often not useful for a designer because looking through the implementation and tracking the data dependencies of so many secret bits is extremely cumbersome. Therefore, COCOALMA attempts to minimize the number of secrets in the leaking correlation term. In particular, we go through all secret bits and greedily assume that the leaking correlation term does not contain them but still leaks information. If the SAT solver returns UNSAT, we know that the investigated secret must appear in the correlation term. At the end of this procedure, COCOALMA has produced a minimized example of a leaking correlation term.

Next, COCOALMA provides a *leakage graph*, which allows the designer to visualize the structure of the leaking part of the circuit. In particular, the leakage graph highlights the leaking gates and only includes gates that influence the leak. We perform this graph minimization by starting at the leaking gates and computing their *cone of influence*.

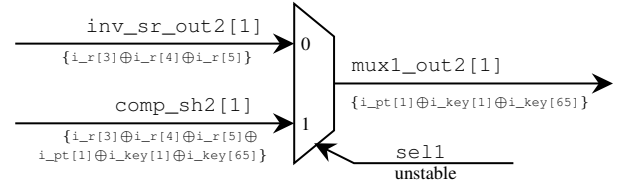


Figure 2. The PRINCE-TI leakage found with COCOALMA. Signal names are shown on top of lines, whereas the problematic correlation term or signal stability is shown below.

Finally, COCOALMA produces a *leakage trace* where the correlation terms of all relevant correlation sets are displayed. In particular, we take the model produced by the SAT solver and show the ports $p \in \mathcal{I}$ whose corresponding propositional variables in \mathcal{P}_x are assigned to \top , indicating they are part of the correlation term. The designer can combine this information with the leakage graph to deduce the cause of the leak.

C. Debugging PRINCE-TI

In the particular case of PRINCE-TI, we have identified the leak at multiplexer `mux1_out2[1]`, as shown in Figure 2. Here, the control signal `sel1` determines whether the output is the inverse of the shift rows operation `inv_sr_out2[1]`, or the compression operation `comp_sh2[1]`. Here, a glitch on the control signal `sel1` causes the multiplexer to forward both inputs in the third clock cycle. Unfortunately, `inv_sr_out2[1]` correlates to the uniformly random value $r = i_r[3] \oplus i_r[4] \oplus i_r[5]$, whereas `comp_sh2[1]` correlates with $r \oplus i_{pt}[1] \oplus i_{key}[1] \oplus i_{key}[65]$. Observing these two values allows an attacker to compute $i_{pt}[1] \oplus i_{key}[1] \oplus i_{key}[65]$, breaking the security guarantees promised by masking schemes.

Although the leakage is observable at `mux1_out2[1]`, its root cause is somewhere else. Under closer inspection of the *leakage trace* and *leakage graph*, we see that the shift rows operation, in combination with glitches, causes a forwarding of the random bits used to re-share the thirteenth S-Box, making them observable at `inv_sr_out2[1]`. Since the same random bits are used to re-share the first S-Box, which eventually leads to `comp_sh2[1]`, the random bits cancel out at the multiplexer. Ultimately, the reuse of random bits causes a leak in the presence of glitches. We fix this by increasing the size of the random input `i_r` from 48 to 192 bits, and avoiding the reuse of random inputs for the re-sharing of S-Box outputs. The second and third row of Table IV show the verification results for the fixed version of PRINCE-TI, where we were able to verify up to two rounds of the cipher in under four minutes.

D. Verifying AES-DOM

Rijndael, better known as the *Advanced Encryption Standard* (AES), is an extremely popular, secure, and widely adopted block cipher [8]. The 128-bit version of AES takes as input a 128-bit plaintext and encrypts it through ten rounds using a 128-bit key. First, the cipher adds the initial secret key

to the plaintext to create the cipher’s state and then expands the key into ten individual round keys. The first nine rounds apply the S-Box to each state byte, re-order the bytes, apply a linear transformation to 32-bit chunks, and mix the state with the round key. The last round does not apply the linear transformation as it does not contribute to security.

AES is not intended for masked implementations because it has a highly non-linear S-Box that is applied sixteen times per round. In order to minimize the used design area, masked AES implementations opt for only one S-Box module that is sequentially fed new bytes each clock cycle [25], [16].

We have analyzed the probing security of the DOM-protected [16] implementation of AES by Gross et al. in all three security models. The open-source implementation of AES-DOM⁴ is written in VHDL and not in Verilog, so it is not directly compatible with our verification flow. However, due to the modularity of COCOALMA, we can produce a netlist with another synthesis flow, e.g., GHDL⁵, and extend it with a compatibility wrapper in Verilog so we can use Verilator for the *tracing* step of the original verification flow depicted in Figure 1. Although this is convenient, it is not strictly required, and COCOALMA also supports execution traces produced by other simulators in VCD format.

Executing the first round of the cipher requires one cycle of setup and twenty computation cycles. Notably, because of the parallelism in hardware designs, AES-DOM computes the linear operations of the first round just-in-time for their use as S-Box inputs in the second round. Therefore, the first 21 cycles only include the key addition, sixteen S-Box applications, and the byte re-ordering. The implementation processes 256 secrets, that is, 128 key bits and 128 plaintext bits. In each clock cycle, the AES-DOM consumes 46 uniformly random bits, yielding a total of 966 random bits for the first round of the cipher. The last column of Table IV shows the verification results for the first round of AES-DOM. The verification was successful in all three probing models, and since the AES-DOM implementation is more complex than PRINCE-TI, it naturally takes longer to verify. COCOALMA only takes about three hours to verify that the implementation of AES-DOM is secure in the hardware probing model.

VI. RELATED WORK

The formal verification of power analysis countermeasures is a well-established research field [1], [2], [4], [13], [10], [11], [19]. The community has been investigating two fundamentally different principles. On the one hand, there are approximative methods like those used in REBECCA [4], *maskVerif* [2], and COCOALMA. In contrast to REBECCA and COCOALMA, *maskVerif* opts for a language-based verification approach, tracks the symbolic representation of probing locations, and simulates the observations an attacker can make using uniformly random values. On the other hand, model counting methods inspect the truth table of a given

function and check whether the correlation strength is zero for all secret values. Tools such as *QMV*erif [10] and *QMS*infer [11] apply these methods to overcome the shortcomings of heuristics used in faster approximative methods. Similarly, probability-distribution tracking approaches such as SILVER [19] (implicitly) rely on model counting to determine the distribution type for any possible observation an attacker can make.

To our knowledge, *maskVerif* and SILVER were not used for stateful hardware verification. The authors of *QMV*erif and *QMS*infer claim they support stateful hardware verification, but the tools are not open-source, so we could not replicate their results.

VII. FUTURE WORK

The current version of COCOALMA is a significant improvement over its predecessor REBECCA [4]. However, there are still open questions that could yield performance improvements or usability improvements.

The model of glitches used in COCOALMA seems too conservative, but we have no empirical evidence to the contrary. In particular, we assume that glitches are unpredictable and can forward any combination of the new and old signal values, even constants. This assumption might be too strict, and some combinations would not be observable in a power trace. Similarly, we assume the worst-case interaction between transition and glitch leakage, which might also be unnecessarily cautious. Eliminating these overly paranoid precautions would single-handedly reduce the verification complexity. Another avenue for increasing the scalability would be to consider implementation modules separately and tie the individual proofs together using composability notions [2].

VIII. CONCLUSION

Although COCOALMA was originally designed for verifying software in the *time-constrained probing model*, it can also verify stateful hardware circuits in the *hardware probing model*. COCOALMA improves upon REBECCA in terms of scope and verification capabilities. It supports more security models, includes an elegant correlation-set encoding, supports circuit simulation, and uses it throughout the verification. The native support for stateful verification allows a tighter integration into the design flow, and as demonstrated with PRINCE-TI and AES-DOM, COCOALMA can be applied to industry-scale designs. We have successfully identified a leakage location in PRINCE-TI, which cannot be found by only analyzing the PRINCE-TI S-Box, as it requires the full context of the cipher’s implementation. Through the debugging support provided by COCOALMA, we found the cause of the information leakage and fixed it by adding more random inputs. Furthermore, we have also demonstrated the modularity and adaptability of COCOALMA by verifying an AES-DOM design that uses an entirely different synthesis flow in another HDL language.

Overall, we think COCOALMA is an excellent addition to any synthesis flow and can be used for the early detection of mistakes.


⁴<https://github.com/hgrosz/aes-dom>


⁵<https://github.com/ghdl/ghdl-yosys-plugin>

REFERENCES

- [1] Arribas, V., Nikova, S., Rijmen, V.: VerMI: Verification tool for masked implementations. In: ICECS 2018. pp. 381–384 (2018)
- [2] Barthe, G., Belaïd, S., Cassiers, G., Fouque, P., Grégoire, B., Standaert, F.: maskverif: Automated verification of higher-order masking in presence of physical defaults. In: ESORICS 2019. pp. 300–318 (2019)
- [3] Biere, A.: CaDiCaL at the SAT Race 2019. In: SAT Race 2019. pp. 8–9. University of Helsinki (2019)
- [4] Bloem, R., Groß, H., Iusupov, R., Könighofer, B., Mangard, S., Winter, J.: Formal verification of masked hardware implementations in the presence of glitches. In: EUROCRYPT 2018 (2018)
- [5] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In: ASIACRYPT (2012)
- [6] Bozilov, D., Knezevic, M., Nikov, V.: Optimized threshold implementations: Securing cryptographic accelerators for low-energy and low-latency applications. IACR (2018)
- [7] Chellam, M.B., Natarajan, R.: AES hardware accelerator on FPGA with improved throughput and resource efficiency. AJSE (2018)
- [8] Daemen, J., Rijmen, V.: Aes proposal: Rijndael (1999)
- [9] Dhooghe, S., Nikova, S., Rijmen, V.: Threshold implementations in the robust probing model. In: TIS@CCS 2019. pp. 30–37 (2019)
- [10] Gao, P., Xie, H., Zhang, J., Song, F., Chen, T.: Quantitative verification of masked arithmetic programs against side-channel attacks. In: TACAS (2019)
- [11] Gao, P., Zhang, J., Song, F., Wang, C.: Verifying and quantifying side-channel resistance of masked software implementations. ACM Trans. Softw. Eng. Methodol. pp. 16:1–16:32 (2019)
- [12] Ghosh, S., Zhao, L., Misoczki, R., Sastry, M.R.: Ultra-lightweight cryptography accelerator system. Tech. rep., Intel Corporation (2018), patent number: US20180183573A1
- [13] Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: Coco: Co-design and co-verification of masked software implementations on cpus. Tech. rep., IACR Cryptology ePrint Archive report 2020/1294 (2020)
- [14] Gomes, C.P., Sabharwal, A., Selman, B.: Model counting. In: Handbook of satisfiability (2009)
- [15] Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. CCS (2016)
- [16] Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In: TIS@ CCS. p. 3 (2016)
- [17] Groß, H., Mangard, S., Korak, T.: An efficient side-channel protected aes implementation with arbitrary protection order. In: RSA (2017)
- [18] Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: CRYPTO 2003. pp. 463–481 (2003)
- [19] Knichel, D., Sasdrich, P., Moradi, A.: SILVER - statistical independence and leakage verification. In: ASIACRYPT (2020)
- [20] Kruse, J., Schinianakis, D.: A high-throughput, low area implementation of PRINCE algorithm for industrial IoT. Tech. rep., Bell Labs, Nokia (2017), <https://www.bell-labs.com/institute/publications/itd-17-57329p/>
- [21] Mangard, S.: A simple power-analysis (SPA) attack on implementations of the AES key expansion. In: ICISC (2002)
- [22] Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)
- [23] Messerges, T.S., Dabbish, E.A.: Investigations of power analysis attacks on smartcards. In: USENIX Smartcard (1999)
- [24] Moos, T., Moradi, A., Schneider, T., Standaert, F.: Glitch-resistant masking revisited or why proofs in the robust probing model are needed. TCHES pp. 256–292 (2019)
- [25] Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of aes. In: EUROCRYPT 2011 (2011)
- [26] Örs, S.B., Gürkaynak, F.K., Oswald, E., Preneel, B.: Power-analysis attack on an ASIC AES implementation. In: ITCC (2004)
- [27] Quisquater, J., Samyde, D.: Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In: E-smart 2001. pp. 200–210 (2001)
- [28] Snyder, W.: Verilator, <https://www.veripool.org/wiki/verilator>, <https://www.veripool.org/wiki/verilator>. Retrieved on July 10th, 2020
- [29] Valiant, L.: The complexity of computing the permanent. Theoretical Computer Science (1979)
- [30] Wang, Y., Ha, Y.: FPGA-based 40.9-gbits/s masked aes with area optimization for storage area network. IEEE TCAS II (2013)
- [31] Wolf, C., Glaser, J.: Yosys — a free verilog synthesis suite. Proceedings of Austrochip (2013)

End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers

Dapeng Gao 
University of Oxford

Tom Melham 
University of Oxford

Abstract—Capability Hardware Enhanced RISC Instructions (CHERI) extend conventional ISAs with *capabilities* that can enable fine-grained memory protection and scalable software compartmentalisation. CHERI-RISC-V is an extended version of the RISC-V ISA with support for CHERI, and Flute is an open-source 64-bit RISC-V processor with a five-stage, in-order pipeline. This case study presents the formal verification of CHERI-Flute, a modified version of Flute that implements CHERI-RISC-V, against the Sail CHERI-RISC-V specification. To the best of our knowledge, this is the first extensive formal verification of a CHERI-enabled processor.

We first translated relevant portions of the Sail CHERI-RISC-V specification to SystemVerilog Assertions. Then we formulated and proved four classes of end-to-end correctness properties about CHERI-Flute, covering the CHERI instructions and certain liveness properties about the entire processor. None of these results are routine—they all rely on novel proof engineering methodologies that extract microarchitectural invariants to serve as lemmas for the end-to-end proofs.

This work exposed several previously-unknown bugs in CHERI-Flute, most of which occur in the implementation of sophisticated combinational logic for certain CHERI instructions.

I. INTRODUCTION

Despite decades of hardening and mitigation efforts—such as stack protection, garbage collection, and virtualisation—memory safety issues remain a common and dangerous source of security vulnerabilities. A 2019 report by Microsoft [1] states that ‘70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues’. The root cause of this phenomenon is the pervasive use of an unsafe memory model for interpreting the C programming language [2]. This model can be traced back to the PDP-11 and presumes that memory is simply a linear array of individually addressable bytes. This has induced a number of deeply ingrained assumptions about pointer behaviour that go beyond what is guaranteed by the C specification and rely only on ‘implementation-defined behaviour’.

The Capability Hardware Enhanced RISC Instructions (CHERI) project offers an alternative model that provides better memory safety [3]. Its main features include a new machine representation of C pointers called *capabilities*, and extensions to existing instruction set architectures (ISA) that enable the secure manipulation of capabilities. For intuitive understanding, capabilities can be regarded as traditional pointers with extra properties that make them more like object references in a memory-managed language, such as Java. On one hand, this model continues to support limited arithmetic operations on

capabilities that, for example, allow a loop to iterate through an array by repeatedly incrementing a capability. On the other hand, it makes it impossible to construct arbitrary capabilities that can be dereferenced—a significant departure from the usual ‘unsafe’ understanding of the C programming language.

Well-developed ISAs that integrate capabilities include CHERI-RISC-V and CHERI-MIPS [4], which are extended from RISC-V and MIPS. Rigorous engineering techniques have been used extensively in their development [5]. Specifically, Sail [6] specifications of these CHERI ISAs exist that give a precise and executable definition to each instruction.

This case study explores the formal verification of an open source implementation of CHERI-RISC-V. Flute is a 64-bit RISC-V processor with a five-stage, in-order pipeline [7] released by Bluespec Inc. in late 2018. Researchers at Cambridge University have extended Flute with support for CHERI-RISC-V [8], and this extended implementation, named CHERI-Flute, was our verification target.

A. Contributions

We have verified several classes of properties for CHERI-Flute using the JasperGold formal verification environment [9]. The scope of our verification comprises the correct execution of all 80-plus CHERI instructions as well as certain liveness properties for the processor as a whole. Our proof does not cover the existing RISC-V instructions, which do not involve capabilities. Formal verification methodologies for these instructions are well-established and so they are not of central interest in this case study.

To the best of our knowledge, this is the first extensive formal verification of a CHERI processor implementation. Our aim in this paper is to make the methodology accessible for future verification projects on novel architectures, including ones that target capability hardware. All our verification code is available open-source [10].

We have deliberately taken an end-to-end approach. That is, properties are proved for the entire core, as opposed to individual components such as the individual execution units. In CHERI-Flute, the hardware that deals with capabilities is novel, complex, and distributed across the pipeline stages. Our end-to-end approach avoids the necessity to isolate this hardware and characterise its environment.

Our verification results all rely on novel proof engineering methodologies that extract microarchitectural invariants to serve as lemmas for the end-to-end proofs. Some of these

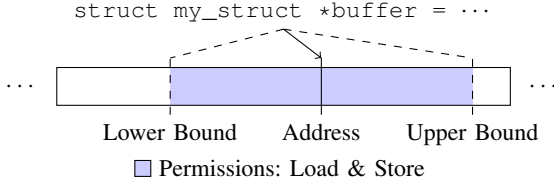


Fig. 1. A typical pointer represented by a capability

invariants are of interest in themselves. For example, one of them shows that the core can never create a malformed capability—an important consistency invariant.

This case study exposed several previously-unknown bugs in the implementation of CHERI-Flute, which have all been reported to and confirmed by the designers [11], [12], [13]. Most of these bugs occur in the implementation of sophisticated bit manipulation logic for CHERI-related instructions, demonstrating the effectiveness of formal verification in catching subtle bugs in a novel processor design. In some cases, we have been able to provide verified bugfixes to the designers.

II. BACKGROUND TO CAPABILITY ARCHITECTURE

CHERI extends ISAs with a new hardware representation for pointers and new instructions for manipulating them. See [4] for its full specification and [14] for a high-level summary of the large research effort surrounding CHERI.

Instead of using 32- or 64-bit integers to represent pointers, CHERI uses a richer representation called *capabilities* that can be stored in *capability registers* in the core or in *capability-sized* and *capability-aligned* words in the memory. The program counter, which usually holds integer addresses, is replaced by the *program counter capability* (pcc).

A capability, illustrated in Fig. 1, contains additional information compared to a traditional pointer, most notably including the following.

Validity Tag. A 1-bit tag that indicates whether the capability is valid. Such a tag is associated with ‘each location that can hold a capability—whether a capability register or a capability-sized, capability-aligned word of memory’ and it ‘tracks capability validity for the value stored at that location’ [4]. When a location that can hold a capability is *untagged*, its contents are simply data and hence do not grant any privilege.

Permissions. A bitmask that controls what the capability can be used for, such as loading or storing from the memory, or setting pcc to execute code.

Bounds. A capability with a set of permissions is not by default authorised to exercise them at all addresses. Instead, the capability also encodes a range of addresses within which it may exercise its permissions.

CHERI instructions operate on capabilities in accordance to security principles such as *privilege minimisation*, *monotonicity*, and *provenance*; these are enforced by checking the Validity Tag, Permissions, Bounds, and other information attached to capabilities [4]. For example, only a valid capability, with

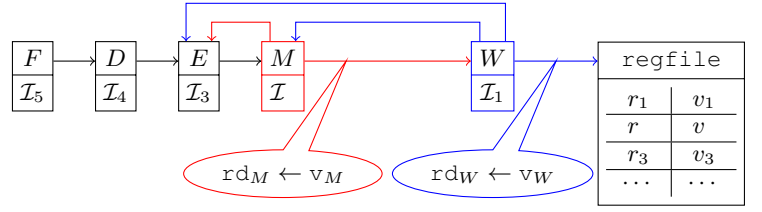


Fig. 2. Pipeline of Flute, including forwarding paths

permission to load, and whose address is within its bounds, can be used to load from that memory address. Otherwise, the processor traps and potentially causes the program to crash. The checks performed by each CHERI instruction are known as its *guard conditions*, and the correctness of their hardware implementation is crucial to the security protections provided by CHERI.

III. BASICS OF CHERI-RISC-V

CHERI-RISC-V extends the RISC-V ISA with support for CHERI [4]. This case study treats its 64-bit variant.

A. Compression of Capabilities

When stored in memory, capabilities are represented in a compressed format [4], [15]. A compressed capability in 64-bit CHERI-RISC-V takes 128 bits (plus an out-of-band validity tag bit)—twice as many bits as a traditional pointer. In the capability registers of the core, however, they are represented in a decompressed format that occupies even more bits. Decompression and compression are done transparently when they are moved between memory and the core.

Capability compression is lossy. That is, there exist decompressed capabilities that do not correspond to any compressed capability. These decompressed capabilities are termed *unrepresentable*. Such a capability poses a significant problem if it appears in the core, since there is no well-defined way to store it to the memory—as that would require compressing the capability first. Part of our verification is to show that unrepresentable capabilities can never be created by the processor.

B. Sail CHERI-RISC-V Instruction Specification

The definition of each CHERI instruction in the Sail CHERI-RISC-V specification [16] roughly takes the form of Algorithm 1. An instruction can retire either *unsuccessfully*, due to violations of one of its guard conditions, or *successfully*, after modifying the architectural state of the processor. As will be seen in Section V-A, the distinction between successful and unsuccessful retirement is central to the way we specify instruction correctness in this work.

IV. FLUTE AND CHERI-FLUTE

Flute [7] is a 64-bit RISC-V processor with a five-stage, in-order pipeline designed for low- to medium-end applications. The processor is designed in Bluespec SystemVerilog (BSV) and has been synthesised and tested on Xilinx FPGAs.

Flute has the basic pipelined microarchitecture commonly found in computer architecture textbooks [17], featuring a

Algorithm 1: Typical CHERI instruction specification

```
if  $\neg$ guard condition 1 then retire FAIL(TagViolation);
else if  $\neg$ guard condition 2 then retire
  FAIL(PermitLoadViolation);
...
else if  $\neg$ guard condition 12 then retire
  FAIL(LengthViolation);
else
  modify architectural state;
  retire SUCCESS;
end
```

Fetch (F), a Decode (D), an Execute (E), a Memory (M), and a Write-back (W) stage. It also comes with forwarding mechanisms to make the pipeline more efficient. The register file (`regfile`) consists of 32 general-purpose registers r_0, \dots, r_{31} , where r_0 is hardwired to zero.

Fig. 2 illustrates the pipeline of Flute with its stages occupied by instructions $\mathcal{I}_1, \dots, \mathcal{I}_5$. Outgoing paths from stage M and W , including forwarding paths, are highlighted in red and blue respectively. These paths carry information about pending updates to the register file: the pending update in stage W writes the value v_W into register `rdW`, and the pending update in stage M writes the value v_M into register `rdM`.

To articulate properties, we define two subscripted register files: `regfileM`, which contains the contents of `regfile` after committing the pending update in stage W , and `regfileE`, which contains the contents of `regfile` after committing the pending updates in both stages W and M , in that order. The subscripted versions are essentially what the register file appears to be to stages M and E after forwarded values are taken into account. Hence their subscripts.

A. CHERI-Flute

CHERI-Flute [18] extends Flute with support for CHERI-RISC-V. We sketch here the main relevant changes.

First, the registers are widened to become hybrid registers that can be used as both integer and capability registers. Second, most of the computation supporting the CHERI instructions—calculating bounds, incrementing addresses, and so on—is implemented within the ALU located in stage E . Finally, circuitry is added to stage M that partially checks whether any CHERI instruction passing through it violates the instruction’s guard conditions. The rest of the checks are performed earlier by the ALU. While these checks could in principle all be placed in the ALU, this would cause unacceptably long delays in stage E for certain instructions. Hence they are spread across stages E and M instead.

V. FORMULATING CORRECTNESS

Our formal verification flow is driven by JasperGold. The design is first compiled into SystemVerilog using the open-source `bmc` compiler and then imported into JasperGold. This pre-compilation is necessary because JasperGold cannot read the Bluespec SystemVerilog source of CHERI-Flute directly.

The specification for correctness, which in our case is the Sail CHERI-RISC-V specification, also needs to be mapped into properties—written as SystemVerilog Assertions (SVA)—about the compiled SystemVerilog design. Tooling does not exist to achieve this automatically, so for this case study we manually translated those portions of the Sail specification necessary for the verification effort into SVA. This yielded more than 1000 lines of data structures and functions of SystemVerilog and almost 100 correctness properties in SVA. As these properties are about a compiled design, a certain amount of ‘reverse engineering’ was needed to identify the relevant signal names.

A. The Instruction Specification Framework

A RISC-V processor is simple enough to formulate correctness of its instructions in the classical, direct way that will be familiar from many examples in the literature.

Let α be an abstraction function that maps each microarchitectural state of CHERI-Flute to a CHERI-RISC-V architectural state. Write $s \xrightarrow{\mathcal{I}} s'$ to mean that a CHERI-Flute processor retires instruction \mathcal{I} and thereby transitions from microarchitectural state s to microarchitectural state s' . Similarly, write $S \xrightarrow{\mathcal{I}} S'$ to mean that, according to the CHERI-RISC-V specification, executing instruction \mathcal{I} alters the architectural state S to architectural state S' . Note that both transition relations are deterministic.

Now for the implementation of an instruction \mathcal{I} to conform to specification, we require that

$$\forall s s'. s \xrightarrow{\mathcal{I}} s' \implies \alpha(s) \xrightarrow{\mathcal{I}} \alpha(s') \quad (1)$$

where s ranges over the reachable microarchitectural states of CHERI-Flute. The reachability of s is, of course, crucial; this is further discussed in Section VI-B.

Now the formulation Prop. (1) faces a significant practical challenge. A CHERI instruction can be retired either successfully or unsuccessfully—and, in the latter case, there are sometimes more than a dozen ways in which it can fail. So formulating correctness as in Prop. (1) will require a full specification of what the processor’s behaviour, and the resulting architectural state, should be for each kind of failure. This would be ideal, but also greatly increases the effort of formulating the required properties.

We therefore formulate a weaker notion of correctness that greatly simplifies the properties, albeit at the cost of a less comprehensive verification. Define two checkmarked relations as follows. For any instruction \mathcal{I} and microarchitectural states s and s' , the relation $s \xrightarrow{\mathcal{I}\checkmark} s'$ holds iff $s \xrightarrow{\mathcal{I}} s'$ and instruction \mathcal{I} is retired successfully. And for any instruction \mathcal{I} and architectural states S and S' , the relation $S \xrightarrow{\mathcal{I}\checkmark} S'$ holds iff $S \xrightarrow{\mathcal{I}} S'$ and all instruction \mathcal{I} ’s guard conditions are met.

Now, consider the property expressed by the proposition

$$\forall s s'. s \xrightarrow{\mathcal{I}\checkmark} s' \implies \alpha(s) \xrightarrow{\mathcal{I}\checkmark} \alpha(s') \quad (2)$$

which says that any *successful* retirement of instruction \mathcal{I} occurs in compliance with the specification. Proving the stronger

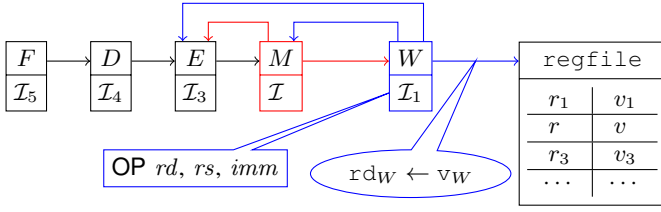


Fig. 3. Microarchitectural state with register-only instruction

condition Prop. (1) shows the processor complies with the full specification indicated by Algorithm 1, which has numerous branches leading to different types of failures. Prop. (2) is a weaker condition but greatly simplifies the properties.

This simplified property cannot detect a faulty processor with incorrect *unsuccessful* retirement. That is, a processor that correctly prevents a certain CHERI instruction that violates its guard conditions from being retired at the end of the pipeline, but which nonetheless produces an incorrect processor state according to the CHERI RISC-V specification. The property will, however, still detect processors with incorrect *successful* retirement. That is, processors that produce the wrong architectural state upon a CHERI instruction being retired the end of the pipeline, or processors that retire a CHERI instruction at the end of the pipeline that violates its guard conditions. This ensures that none of the security guarantees offered by CHERI is compromised. To see this, suppose for contradiction that Prop. (2) is true for some faulty processor which *incorrectly retires successfully* some instruction \mathcal{I} , i.e., there exist s and s' such that the relation $s \xrightarrow{\mathcal{I}} s'$ holds but some of instruction \mathcal{I} 's guard conditions are not met. Consequently, by Prop. (2), the relation $\alpha(s) \xrightarrow{\mathcal{I}} \alpha(s')$ also holds. But this implies that all of instruction \mathcal{I} 's guard conditions are met, which contradicts the assumption. Section IX discusses ways to relatively easily obtain properties that reflect the stronger specification.

B. Expressing Specifications as Properties

For mechanised formal verification in JasperGold, it is of course necessary to articulate the intent of the abstract correctness condition described by Prop. (2) as a group of SystemVerilog expressions. In practice, this means

- (i) characterising the microarchitectural states s and s' for which $s \xrightarrow{\mathcal{I}} s'$ holds, and
- (ii) defining the mapping α for at least microarchitectural states s and s' where $s \xrightarrow{\mathcal{I}} s'$ does hold.

Note that expressing (i) means characterising when the instruction \mathcal{I} has retired successfully. One of the contributions of our methodology is to observe that this can be tied to the detection of certain microarchitectural states. Note also that (ii) is much simpler than having also to define the architectural states resulting from every kind of unsuccessful retirement.

In practice, we have developed these properties in separate groups for each of three distinct classes of instructions that share common structure. The sections that follow explain these. In the actual proof code, a systematic scheme of

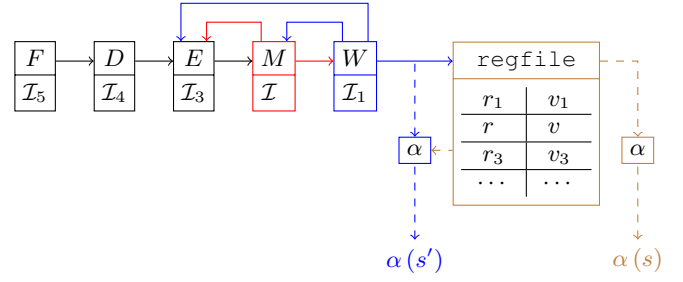


Fig. 4. Microarchitectural state with state abstractions

‘property templates’ is employed to make it easy to create and manage almost 100 properties without having to maintain multiple copies of boilerplate code. It also allowed us to quickly implement and validate proof engineering ideas for a large batch of properties, improving research efficiency.

C. Register-Only CHERI Instructions

A register-only CHERI instruction computes a function of its operands and writes a result into a given register, causing a trap if any of its guard conditions is not met.

Recall from Section V-B that two expressions are needed to formulate the required correctness properties. To express (i), consider Fig. 3, which shows the microarchitectural state when some register-only instruction \mathcal{I}_1 is in stage W . Denote this state by s and the state right after instruction \mathcal{I}_1 is retired by s' . Since stage W is at the end of the pipeline, any instruction reaching stage W is retired at the end of the current cycle. Moreover, any instruction reaching stage W can no longer cause traps, so it is bound to be retired successfully. Conversely, if a register-only instruction is retired successfully, then it must have been in stage W just before its retirement. So $s \xrightarrow{\mathcal{I}_1} s'$ and (i) can be expressed simply by checking whether the given instruction is in stage W .

To express (ii), consider Fig. 4, which illustrates the microarchitectural state of CHERI-Flute in some state s that is about to successfully retire instruction \mathcal{I}_1 and enter state s' , i.e., $s \xrightarrow{\mathcal{I}_1} s'$. Hence $\alpha(s)$ and $\alpha(s')$ must give the architectural states right before and after instruction \mathcal{I}_1 is retired. Then observe that

- $\alpha(s)$ can be obtained directly from the current register file, `pcc`, etc., and
- $\alpha(s')$ can be obtained by combining the current register file, `pcc`, etc. with the pending updates contained in the output of stage W ,

so (ii) can be expressed as a function of state s .

Given formulations of expressions (i) and (ii), the SVA property for a register-only instruction with register addresses rd and rs , and immediate data imm will say that if stage W contains an instruction with opcode `OP`, then

- $rd_W = rd$,
- $v_W = result_{OP}(regfile[rs], imm)$, and
- $guard_{OP}(regfile[rs], imm)$.

Where $result_{OP}$ and $guard_{OP}$ are SystemVerilog functions translated from the Sail specification of the instruction with opcode OP that compute its write-back result and guard conditions respectively.

D. Branching CHERI Instructions

A branching CHERI instruction redirects the control flow and (optionally) saves the return address in a given register. Of course, it also has guard conditions to ensure that the updated pcc has the right Bounds and Permissions. This creates an opportunity to decompose what a branching instruction does into two operations: checking its guard conditions and (optionally) saving the return address, and (conditionally or unconditionally) redirecting the control flow.

The first of these is just what a register-only instruction does, so we can simply reuse the property template developed in Section V-C. So the rest of this section is devoted to formulating the correctness properties about the second operation.

First, it is necessary to briefly explain how the control flow is managed in CHERI-Flute. Initially, stage F fetches an instruction from $fetch_addr$ and predicts the address of the next instruction using the branch predictor. This predicted address ($pred_addr$) is *by default* used as the next $fetch_addr$, and it is also passed along the pipeline with the *currently* fetched instruction until it reaches stage E , where the ALU computes the correct address of the next instruction ($next_addr$). The processor then compares the computed $next_addr$ with the $pred_addr$ it received. If the two addresses do not match, then a branch misprediction has occurred, and stage F has been fetching the wrong instructions and passing them along the pipeline. To rectify this, $fetch_addr$ is set to $next_addr$, and all pipeline stages prior to stage E are flushed. Otherwise, if the branch prediction has been correct, no flushing is needed and $fetch_addr$ is updated in the default way.

Fig. 5 shows the microarchitectural state when some branching instruction \mathcal{I}_3 is in stage E . To formulate the correctness properties about control flow redirection, the framework developed in Section V-A is slightly generalised. Specifically, if a branching instruction \mathcal{I} is in stage E and a branch misprediction has occurred, then instruction \mathcal{I} is now considered ‘about to be retired successfully’ insofar as control flow redirection is concerned, and it is now considered to have been ‘retired successfully’ after $fetch_addr$ is set to $next_addr$. This gives the expression (i) discussed in Section V-B. As for expression (ii), the architectural states of the processor right before and after some branching instruction is retired successfully are taken from the values of $fetch_addr$ before and after that instruction is retired successfully, respectively.

E. Memory CHERI Instructions

A memory CHERI instruction loads from or stores to the memory using the capability (directly or indirectly) specified by its operands, causing a trap if any of its guard conditions is not met. What a memory instruction does can be decomposed

into two operations: checking its guard conditions, and loading from or storing to the memory.

The correctness properties about the first operation can be formulated simply by reusing the property template developed in Section V-C. Hence this section focuses on formulating the correctness properties about the second operation.

CHERI-Flute is connected to the memory hierarchy through an interface consisting of several input and output ports, which must be properly used in order for the memory to function correctly. As with register-only instructions, a memory instruction \mathcal{I} is about to be retired successfully when it is in stage W , after having sent and fulfilled its request to the memory in stage M . Thus, the correctness property should assert that before \mathcal{I} is retired successfully, when it was in stage M , the memory interface had been properly used to fulfil what the specification requires of it. In our proof, SVA sequences are used to precisely specify the exact sequence of events that must have taken place when instruction \mathcal{I} was in stage M .

Fig. 6 and Fig. 7 show how a memory *load* instruction \mathcal{I}_2 is moved from stage M to stage W and becomes ready to be retired successfully. The correctness property checks that

- a new memory request was not sent before the previous request had been fulfilled,
- a memory exception did not occur,
- the value returned from the memory when \mathcal{I}_2 was in stage M was decompressed correctly (if it was a capability) and used in the pending update to the register file, and
- the content of the pending update remains stable as \mathcal{I}_2 is moved from stage M to stage W .

The correctness properties about memory *store* instructions are highly similar and thus omitted here.

F. Processor Liveness

All correctness properties discussed so far are safety properties. Our verification also tackled the important issue of processor liveness—demonstrating that the processor does not freeze so that the pipeline never progresses.

Of course, there are challenges when dealing with liveness. First, it is usually very difficult to prove liveness properties in practice, and there is no such thing as a bounded proof for liveness that can at least give some confidence. Second, even if a liveness property is proved, there is still no guarantee about *when* the desirable event will occur, which is not ideal when performance is critical. Third, a necessary condition for a processor to exhibit liveness is the correct behaviour of the external components connected to it. For example, if the memory never fulfils a load request, then the processor might wait indefinitely for a response, stalling the pipeline. This can be ruled out by assuming certain fairness constraints about the external components, but these can of course potentially be violated unless they are themselves verified.

There is a conventional workaround to the first two problems. Instead of proving the liveness property that ‘the pipeline eventually progresses’, we derive a *safety* property that ‘the pipeline progresses within n cycles’ parametrised by n and search for the smallest n (if it exists) for which the safety

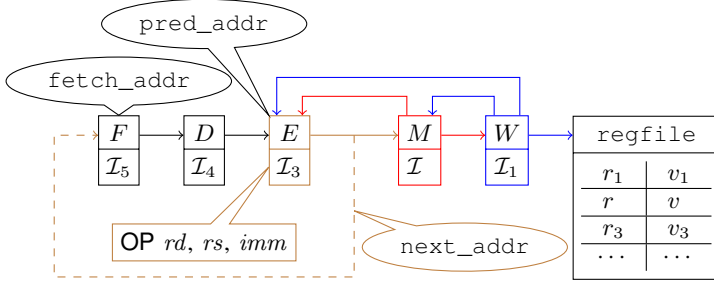


Fig. 5. Microarchitectural state with branching instruction

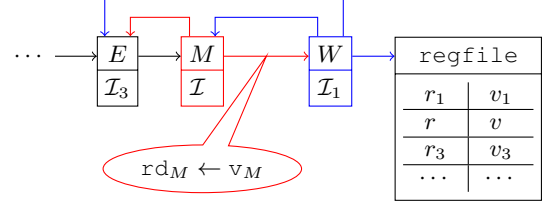


Fig. 6. Microarchitectural state with load instruction in stage M

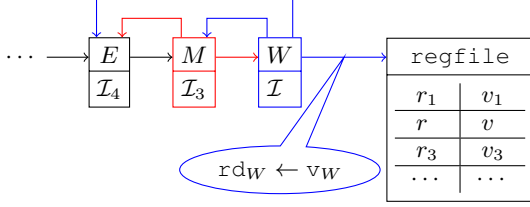


Fig. 7. Microarchitectural state with load instruction in stage W

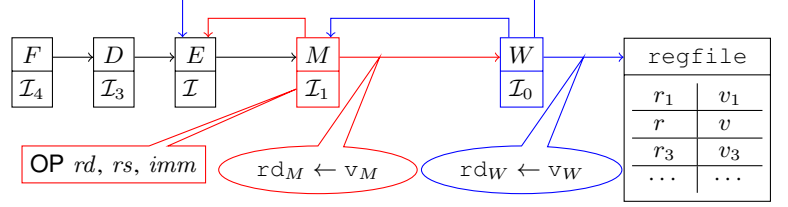


Fig. 8. Microarchitectural state with register-only instruction in stage M

property can be proved. This not only averts the difficulty of proving liveness properties but also generates a concrete bound on when the pipeline progresses.

The derived safety property we proved for CHERI-Flute says that if an instruction enters stage *E*, then within nine cycles, either a new instruction enters stage *E*, or the processor enters one of three special states, triggered by particular instructions, that requires it to wait for certain external signals.

This property shows that as long as the processor does not enter one of the special states, new instructions will enter stage *E* periodically, so the pipeline never freezes. The number ‘nine’ is the smallest number for which this property can be proved, and the focus on stage *E* is because certain RISC-V instructions are retired in stage *E*—i.e. they are never moved into stages *M* or *W*. Asserting this property on any stage *prior to* stage *E* always attracts a counterexample where an instruction is repeatedly issued but never reaches beyond stage *E*, effectively stalling the subsequent stages.

Of course, the proof of this property relies on several fairness constraints. Most notably, it is assumed that the memory always fulfils a request within *two* cycles. The number ‘two’ here is arbitrarily chosen, and it is reasonable to conjecture that a different number can be used without making any substantial difference other than perhaps affecting the number ‘nine’ in the derived safety property.

VI. PROOF ENGINEERING

Not all our correctness properties can be proved in a push-button manner. Specifically, those properties about register-only CHERI instructions as well as those about the register-only components of branching and memory CHERI instructions *cannot* be proved straightforwardly. Instead, proof convergence on these properties relies on proof engineering methodologies that are explained in this section.

A. Decomposing the Pipeline

This methodology is called ‘decomposing the pipeline’ because it enables one to prove some property about a desired instruction when it is in a *later* stage of the pipeline by first proving some lemmas about the instruction when it was in *earlier* stages of the pipeline.

1) *The First Lemma*: The correctness property shown in Section V-C for any register-only instruction cannot be proved directly in JasperGold. Instead, we prove a structurally identical version of the property that is ‘pushed back’ one stage in the pipeline, referencing regfile_M instead of regfile , rd_M and v_M instead of rd_W and v_W , and using a suitably adjusted $\text{guard}_{\text{OP}}^M$ function, as we sketch below.

If this version of the property can be proved, then it can be used as a lemma to successfully prove the original correctness property through *k*-induction [19]. The lemma is a property of a register-only instruction in stage *M* instead of stage *W*. Observe that the write-back result of any register-only instruction is computed by the ALU in stage *E*. Therefore, for any register-only instruction \mathcal{I}_1 in stage *M* with opcode *OP* as illustrated in Fig. 8, its write-back result must already be available in v_M . This means that we can assert

$$\text{v}_M = \text{result}_{\text{OP}}(\text{regfile}_M[\text{rs}], \text{imm})$$

in the lemma, where the subscripted regfile_M is used to take into account any forwarded value v_W from stage *W*.

Now recall from Section IV-A that checks for guard conditions are spread across stages *E* and *M*. Thus, when instruction \mathcal{I}_1 reaches stage *M*, only the checks in stage *E* have been performed, whereas the checks in stage *M* are still underway. Therefore, it is incorrect to assert that

$$\text{guard}_{\text{OP}}(\text{regfile}_M[\text{rs}], \text{imm})$$

in the lemma. Rather, the lemma only asserts that the subset of instruction \mathcal{I}_1 's guard conditions that are checked in stage E have been met. This subset is given by $guard_{OP}^M$.

Given the lemma, the original correctness property can be proved by k -induction. But without it, k -induction is unable to converge because for any value of k , the SAT-solver can always find a trace that violates the inductive hypothesis. Such a trace would begin at an *unreachable* microarchitectural state where the desired instruction is in stage M . It would then *stall* the pipeline during the next $(k - 1)$ steps, only moving the desired instruction to stage W at the $(k + 1)$ -th step, where the inductive hypothesis fails to hold. The pipeline can stall for arbitrarily many cycles in such traces due to the absence of the very fairness constraints that enable the proof of the liveness properties in Section V-F. However, it is unnecessary to add fairness constraints here. Instead, we use the given lemma to prevent the SAT-solver from exploring such unreachable states. And since stage M is immediately prior to stage W , $k = 1$ is sufficient for the proof to converge.

2) *The Second Lemma*: To actually prove the lemma just explained, the same methodology is simply reapplied. That is, a *second* lemma is used to narrow the space of states in which the desired instruction is in stage E so as to exclude traces that violate the first lemma.

Fortunately, this second lemma is relatively easy to discover, since the only state information contained in stage E is the decoded content of the current instruction in stage E . Thus, the second lemma simply needs to assert that any instruction in stage E is properly decoded, which enables the proof of the first lemma by 1-induction.

Now this second lemma can, in turn, be proved by 1-induction if a similar *third* lemma is proved about stage D . And so on. This chain of lemmas stops, of course, at stage F where the last lemma can be proved directly. In practice, however, since CHERI-Flute's design of stages F and D is relatively simple, we took advantage of one of JasperGold's black-box proof engines to automatically complete the proof.

B. Developing Microarchitectural Invariants

CHERI instructions compute relatively sophisticated functions of their operands. In the Sail specification, these are given by total functions on all decompressed capabilities, including the unrepresentable ones mentioned in Section III-A. But since unrepresentable capabilities pose a significant problem if they appear in the processor, CHERI-Flute is designed so they can never be created by the hardware in the first place. CHERI-Flute is then excused from conformance with the specification for unrepresentable capabilities.

This, of course, leads to the generation of unreachable counterexamples in model checking, so our verification includes a global consistency invariant over the entire processor, showing that only representable capabilities are present. Formulating and proving this invariant was challenging because there are many internal registers in CHERI-Flute's microarchitecture that can influence the architecturally visible registers. A weak invariant that does not cover these internal registers cannot be

proved by k -induction since the SAT-solver can always find an unreachable state in which one of these registers contains an unrepresentable capability, which then 'pollutes' one of the architecturally visible registers within the next few cycles.

This challenge was overcome using State-Space Tunnelling, a JasperGold feature that allows the user to prune unreachable portions of the state space when performing k -induction proofs. Essentially, it allows us to specify some k and let the SAT-solver generate a trace of length k that violates the invariant. The user then examines this trace to identify any internal register that causes the violation, and manually strengthens the invariant to include it.

This process repeats until, for some sufficiently large k , no violating trace can be found, at which point proof convergence for the invariant is achieved. In the end, the invariant in our proof was sufficiently strong to be proved by 1-induction.

VII. RESULTS AND EVALUATION

In this case study, the implementations of all 80-plus CHERI instructions (except a very few not yet implemented) have been subject to formal verification in JasperGold against the correctness properties in Section V through the proof engineering methodologies in Section VI.¹ While the implementations of most instructions were found to satisfy the correctness properties, several were found to be buggy.

The bugs found roughly fell into two categories. The first category are simple coding mistakes: the designer failed to notice details of the specification, or the specification changed after the design was created. These bugs are usually detectable with a moderate amount of scrutiny or simulation testing. The second category are algorithmic errors, typically caused by subtle mistakes in complex pieces of logic. These are much more difficult to uncover, even with the most intensive code review or simulation testing.

- In the `incOffsetFat` function, a bit vector is truncated but subsequent code still uses the old non-truncated value. This can potentially lead to the creation of unrepresentable capabilities for certain inputs.
- Several CSR registers are not initialised to the null capability when the processor is reset.

These two bugs have been confirmed and fixed by the designers [11], [13]. The following have also been confirmed by the designers and fixes are pending:

- The `getTop` function incorrectly truncates the returned value.
- `AUIPCC` incorrectly clears the validity tag of the returned capability for certain inputs.
- `CUnseal` fails to check a permission bit.
- `CCSeal` incorrectly causes the processor to trap for certain inputs.

One final bug illustrates an especially productive collaboration between verification and design: in the `setAddress`

¹On a 24-core AMD EPYC 7F72 processor, with 256 GB of RAM, the proofs are completed within two hours through parallelisation.

function, the validity tag of the returned capability is cleared incorrectly in a corner case.

This function was originally developed by trial and error using the BlueCheck automated test generation framework [20] and as well as TestRIG, a framework for testing RISC-V processors with random instruction generation [21]. But neither method detected this corner case. The designers’ initial patch for the function was buggy because it mishandles another corner case, which was yet again detected by formal verification. Consequently, we redesigned the function from scratch and formally verified its correctness against the specification before it was submitted to and accepted by the designers [12].

A. Bug or Feature?

Two issues belong to an interesting category sometimes encountered in formal verification: a trace violates the specification, but it is unclear whether the hardware should be changed to match the specification or *vice versa*.

The first was that specification requires the `CSetOffset` and `CIncOffset` instructions perform a standard ‘representability check’ to determine if the capabilities they return are representable. But in CHERI-Flute the `CSetOffset` instruction performs a slightly different, non-standard check optimised for that particular instruction, although the `CIncOffset` instruction uses the standard check.

So the behaviour of the `CSetOffset` instruction violates the specification, but in a beneficial way. It is therefore up to the designers to decide whether the specification should be changed to incorporate this optimised representability check.

The second was that, when trying to prove the global consistency invariant, we found counterexample traces where memory corruption causes injects corrupted capabilities into the core. Since memory bit-flips do occur in actual hardware, we suggested that the core should perform sanity checks on any capability retrieved from the memory, clearing its validity tag if it is found to be corrupted.

In the end, the designers decided not to add the sanity checks because it may cause even more unexpected behaviour when memory corruption occurs, making the situation more complex to debug. So to make the proof of the global consistency invariant converge, we added an assumption that the memory never returns a corrupted capability.

VIII. RELATED WORK

The correctness of processor cores and their implementation of instructions has been a focus of verification research for decades, going at least back to the pioneering work of Hunt on verifying the FM8501 [22] and FM8502 processors [23]. To verify more complicated, pipelined designs, Burch and Dill devised the flushing abstraction [24], a member an extensive family of formulations of correctness that has expanded to cover even out-of-order designs. Aagaard et al. [25] present a useful framework for classifying these different approaches.

From about the mid 1990s, verification was increasingly adopted in industry to verify critical components of large-scale designs. Notable experiments include Kaivola et al.’s

verification of the Pentium 4 floating-point divider [26], Jacobi et al.’s fully automated verification of fused-multiply-add floating-point units [27], Kaivola’s methodology for large-scale formal verification of control-intensive circuits [28], and Slobodova’s verification of AES hardware support [29]. A landmark achievement in this direction was Kaivola et al.’s work on replacing testing with formal verification for validating the core execution cluster of the Core i7 design [30].

The starting point of our work was Reid et al.’s end-to-end verification of Arm processors [31]. But our approach to verifying properties differs significantly from this work. While the Arm verification uses bounded model checking, we obtained much stronger unbounded proofs of all correctness properties by extracting microarchitectural invariants. Of course, the relative simplicity of RISC-V helped make this possible, but it was also enabled by the complexity management methodologies we explain in this paper.

A landmark in the verification of complex cores is the work by Goel et al. [32] on verifying x86 instructions. This was done using the ACL2 theorem prover in concert with a number of tightly integrated support tools, and achieved an end-to-end verification that encompasses decoding, translation into microcode, traps to microcode ROM, and execution.

There has been related work on verifying processors using Symbolic Quick Error Detection (SQED) and its variants [33], [34], [35]. These methodologies use bounded model checking to find sequence-dependent bugs that violate a self-consistency property, but they are not intended for checking single-instruction bugs where an instruction always produces the wrong result for certain inputs [33]. In contrast, our methodology checks for both types of bugs. Indeed, most, if not all of the bugs we found were single-instruction bugs that could not be uncovered by checking for self-consistency. Instead, a more traditional approach using a formal specification was required.

IX. CONCLUSIONS AND PROSPECTS

There are several ways in which the present work can be improved and extended.

For this project, we manually translated the Sail specification of CHERI-RISC-V into SVA. It would obviously be preferable to have an automatic translation, and we are investigating some options for this. Apart from the usual benefits of automation, automatic translation could eliminate the pragmatic need to weaken the specification as described in Section V-A. As Sail has been adopted by the RISC-V Foundation for its golden formal model, a flow from Sail to SVA seems highly desirable in any case.

Further work can also be done to address the drawbacks of the liveness properties described in Section V-F. For example, it would be ideal to remove the proof’s reliance on fairness constraints that contain arbitrarily chosen numbers. Also, the work can be made more complete by proving liveness properties about pipeline stages subsequent to stage *E*.

Attempts could be made to verify more complex CHERI-RISC-V processors, such as Toooba [36], where the main challenge will be to formulate correctness properties about

an *out-of-order* microarchitecture. We note, however, that the SystemVerilog functions translated from the Sail specification during the present work can be completely reused when formulating the new correctness properties.

Finally, we mention that in 2019, the UK announced its *Digital Security by Design* programme with £190 million of funding for a set of research projects [37] to ‘radically update the foundation of our insecure digital computing infrastructure, by demonstrating that mainstream processor technology ... can be updated to include new security technologies based on the CHERI Architecture’ [38]. A cornerstone of the programme is Morello [39], a CHERI-enabled prototype developed by Arm and scheduled for release in late 2021. We hope that this early RISC-V case study provides at least some insights that might eventually apply in the formal verification of Morello.

X. ACKNOWLEDGEMENTS

We are grateful to members of the CHERI group at Cambridge. Alasdair Armstrong, Alexandre Joannou, Simon Moore, Peter Rugg, Peter Sewell, Robert Watson, and Jonathan Woodruff all kindly provided assistance or comments on this work. Thanks also go to Ziyad Hanna at Cadence and to Joe Stoy at Bluespec, who thoughtfully answered our questions about Bluespec SystemVerilog. This work was funded in part by the UKRI programme on Digital Security by Design (Ref. EP/V000225/1, SCoRCH [40]).

REFERENCES

- [1] M. Miller. (2019, February) Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. Presented at the BlueHat IL. [Online]. Available: <https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf>
- [2] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, “Beyond the PDP-11: Architectural support for a memory-safe C abstract machine,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, March 2015, pp. 117–130.
- [3] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, “The CHERI capability model: Revisiting RISC in an age of risk,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, June 2014, pp. 457–468.
- [4] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, “Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 8),” University of Cambridge Computer Laboratory, Technical Report UCAM-CL-TR-951, October 2020.
- [5] K. Nienhuis, A. Joannou, T. Bauereiss, A. 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)*. IEEE, May 2020, pp. 1003–1020.
- [6] A. Armstrong, T. Bauereiss, B. Campbell, S. Flur, K. E. Gray, P. Mundkur, R. M. Norton, C. Pulte, A. Reid, P. Sewell, I. Stark, and M. Wassell, “Detailed models of instruction set architectures: From pseudocode to formal semantics,” in *Proceedings of the 25th Automated Reasoning Workshop: Bridging the Gap between Theory and Practice: ARW 2018*. University of Cambridge, April 2018, pp. 23–24.
- [7] Bluespec announces flute. [Online]. Available: <https://bluespec.com/2018/12/13/bluespec-announces-flute/>
- [8] CHERI-RISC-V. [Online]. Available: <https://www.cl.cam.ac.uk/research/security/ctsrds/cheri/cheri-risc-v.html>
- [9] Jaspergold formal verification platform. [Online]. Available: https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html
- [10] Dpgao/FMCAD2021. [Online]. Available: <https://github.com/dpgao/FMCAD2021>
- [11] Use tmpAddr in place of pointer in incOffsetFat following Dapeng Gao’s ... CTSRD-CHERI/Cheri-Cap-Lib@508da81. [Online]. Available: <https://github.com/CTSRD-CHERI/cheri-cap-lib/commit/508da818baa0d3d103c563c148b6ef2dc4aba057>
- [12] Use Dapeng’s algorithm (adapted from his verified verilog) for ... CTSRD-CHERI/Cheri-Cap-Lib@6e8df02. [Online]. Available: <https://github.com/CTSRD-CHERI/cheri-cap-lib/commit/6e8df025326565f146c2fd1d3e2d7f8ebec61af>
- [13] Fix some CSRs with missing reset logic spotted by Dapeng Gao ... CTSRD-CHERI/Flute@36de38d. [Online]. Available: <https://github.com/CTSRD-CHERI/Flute/commit/36de38d87740baed6ebbc242a872206d9f0b032>
- [14] “An Introduction to CHERI,” Computer Laboratory, Technical Report UCAM-CL-TR-941, Sep. 2019.
- [15] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore, “CHERI Concentrate: Practical Compressed Capabilities,” *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, Oct. 2019.
- [16] CTSRD-CHERI/sail-cheri-riscv: CHERI-RISC-V model written in Sail. [Online]. Available: <https://github.com/CTSRD-CHERI/sail-cheri-riscv>
- [17] R. E. Bryant and D. R. O’Hallaron, *Computer Systems: A Programmer’s Perspective*, third edition ed. Pearson, 2016.
- [18] CTSRD-CHERI/Flute: RISC-V CPU, simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance. [Online]. Available: <https://github.com/CTSRD-CHERI/Flute>
- [19] M. Sheeran, S. Singh, and G. Stålmarck, “Checking Safety Properties Using Induction and a SAT-Solver,” in *Formal Methods in Computer-Aided Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, vol. 1954, pp. 127–144.
- [20] CTSRD-CHERI/bluecheck: A generic test bench written in Bluespec. [Online]. Available: <https://github.com/CTSRD-CHERI/bluecheck>
- [21] CTSRD-CHERI/TestRIG: Testing processors with Random Instruction Generation. [Online]. Available: <https://github.com/CTSRD-CHERI/TestRIG>
- [22] W. A. Hunt, *FM8501: A Verified Microprocessor*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, vol. 795.
- [23] —, “Microprocessor design verification,” *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 429–460, Dec. 1989.
- [24] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessor control,” in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, vol. 818, pp. 68–80.
- [25] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones, “A Framework for Microprocessor Correctness Statements,” in *Correct Hardware Design and Verification Methods*. Springer Berlin Heidelberg, 2001, vol. 2144, pp. 433–448.
- [26] R. Kaivola and K. Kohatsu, “Proof engineering in the large: Formal verification of pentium 4 floating-point divider,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, no. 3, pp. 323–334, May 2003.
- [27] C. Jacobi, Kai Weber, V. Paruthi, and J. Baumgartner, “Automatic Formal Verification of Fused-Multiply-Add FPU,” in *Design, Automation and Test in Europe*. Munich, Germany: IEEE, 2005, pp. 1298–1303.
- [28] R. Kaivola, “Formal verification of Pentium components with symbolic simulation and inductive invariants,” in *Computer Aided Verification*. Springer Berlin Heidelberg, 2005, vol. 3576, pp. 170–184.
- [29] A. Slobodova, “Formal Verification of Hardware Support for Advanced Encryption Standard,” in *2008 Formal Methods in Computer-Aided Design*. IEEE, Nov. 2008, pp. 1–4.

- [30] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel core i7 processor execution engine validation," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2009, vol. 5643, pp. 414–429.
- [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 ARM Processors with ISA-Formal," in *Computer Aided Verification*. Springer International Publishing, 2016, vol. 9780, pp. 42–58.
- [32] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Verifying x86 instruction implementations," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*. ACM, 2020, pp. 47–60.
- [33] D. Lin, E. Singh, C. Barrett, and S. Mitra, "A structured approach to post-silicon validation and debug using symbolic quick error detection," in *2015 IEEE International Test Conference (ITC)*. IEEE, pp. 1–10. [Online]. Available: <http://ieeexplore.ieee.org/document/7342397/>
- [34] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. Fadiheh, D. Stoffel, W. Kunz, C. Barrett, W. Ecker, and S. Mitra, "Symbolic QED Pre-silicon Verification for Automotive Microcontroller Cores: Industrial Case Study," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1000–1005. [Online]. Available: <https://ieeexplore.ieee.org/document/8715271/>
- [35] 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)*. IEEE, pp. 1–8. [Online]. Available: <https://ieeexplore.ieee.org/document/8942096/>
- [36] CTSRD-CHERI/Toooba: RISC-V Core; superscalar, out-of-order, multi-core capable; based on RISCY-OOO from MIT. [Online]. Available: <https://github.com/CTSRD-CHERI/Toooba>
- [37] Digital security by design challenge – UKRI. [Online]. Available: <https://www.ukri.org/our-work/our-main-funds/industrial-strategy-challenge-fund/artificial-intelligence-and-data-economy/digital-security-by-design-challenge/>
- [38] Department of Computer Science and Technology – CHERI: The Digital Security by Design (DSbD) Initiative. [Online]. Available: <https://www.cl.cam.ac.uk/research/security/ctsrds/cheri/dsbd.html>
- [39] Department of Computer Science and Technology – CHERI: The Arm Morello Board. [Online]. Available: <https://www.cl.cam.ac.uk/research/security/ctsrds/cheri/cheri-morello.html>
- [40] SCorCH: Secure code for capability hardware. [Online]. Available: <https://scorch-project.github.io>

Hardware Security Leak Detection by Symbolic Simulation

Neta Bar Kama
Core and Client Development Group
Intel Corporation
Haifa, Israel
neta.bar.kama@intel.com

Roope Kaivola
Core and Client Development Group
Intel Corporation
Portland, OR, USA
roope.k.kaivola@intel.com

Abstract—Aiming to expose security risks in hardware designs, we describe a novel usage of symbolic simulation that led to discoveries of previously unknown potential local data leakages on an Intel Core processor design. Symbolic simulation is an established formal verification method, the main vehicle for verification of arithmetic data-paths in Intel Core processor designs for twenty years. It extends traditional simulation by allowing symbolic variables in the stimulus, covering the circuit behavior for all possible values simultaneously. A special trait of symbolic simulation is that every variable has a name. In the security context, named values allow us to know the exact origin of data and identify data leakages by determining whether values are expected to be read by an operation or present a risk. Leveraging the existing formal verification infrastructure and observing an operation's data dependencies we could identify local leaks without the need to have a complete functional specification for the operation.

Index Terms—Security, Data Leakage, Formal Verification, Symbolic Simulation

I. INTRODUCTION

Comprehensive formal verification of execution engines has been standard practice in virtually all Intel® Core™ processor development projects in the last two decades, and extensive infrastructure has been built to support these efforts. The technical basis of this work is symbolic simulation, a technology extending usual digital circuit simulation with symbolic values, representing sets of concrete values in a single simulation.

In the aftermath of the Spectre and Meltdown vulnerabilities, security has become a greater focus area for validation. In this paper we discuss a novel approach leveraging the existing formal infrastructure for Intel Core processor Execution clusters (EXE) to analyze potential data leakages, security violations where privileged data could be made visible to non-privileged parties. The approach is based on the special feature of symbolic simulation that stimulus values have names that can be used to uniquely relate a value to a specific signal and time.

Intel provides these materials as-is, with no express or implied warranties. Intel processors might contain design defects or errors known as errata, which might cause the product to deviate from published specifications. No product or component can be absolutely secure. Intel, Intel Core, Intel Atom, Pentium and Intel logo are trademarks of Intel Corporation. Other names and brands might be claimed as the property of others.

Below we first discuss the concept of symbolic simulation and its use in EXE formal verification, and the security challenges in EXE. Then, we will describe the principles of our solution analyzing potential data leakages using symbolic simulation, practical considerations in the implementation of the solution over a live Intel Core processor development project, and the results of our experiments. With a moderate engineering effort, we were able to extend the existing formal environment with extra checkers detecting potential data leakages. On the one hand, this allowed us to verify the absence of data leaks for large classes of micro-operations, and on the other to identify several previously undiscovered local data leakage issues, where micro-operations unintentionally wrote back data that had been left behind in the internal state of the cluster by a previous micro-operation.

The closest counterpart to our work in the scientific literature or commercial tools is taint analysis [1], [2], [3], [4]. Like our approach, taint analysis tracks the propagation of values from one signal to another. However, taint analysis works by attaching extra information, the 'taint', to simulation values to track their progress, and requires extra engineering either in the simulator or in post-simulation analysis. In our approach values are tracked using the symbolic variable names already present in the symbolic simulation for the verification, and we only needed to implement a thin analysis layer on top of the existing collateral. Second, taint analysis generally assumes a static classification of signals to 'secret' and 'non-secret' and analyzes possible paths leaking secret values to non-secret signals. This does not adequately reflect the common design pattern of pipelined designs, like the EXE cluster, where the same signals are used to carry both secret and non-secret data at different times, and the notion of a 'secret' is relative to a micro-operation. To our knowledge, our work is among the first published explorations of the application of symbolic simulation into security verification of hardware designs (cf. [2], [5]).

II. SYMBOLIC SIMULATION IN EXE VERIFICATION

A. Symbolic Circuit Simulation

Digital circuit simulation is a standard tool in the arsenal of every working circuit design and validation engineer. Symbolic simulation extends this technology with the ability to carry out

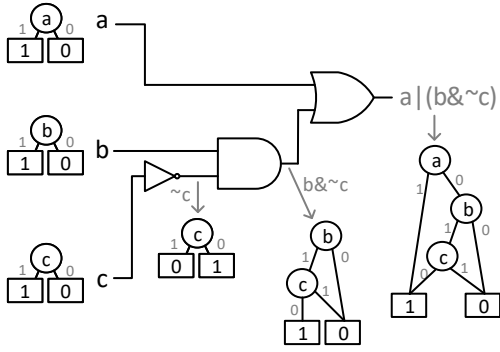


Fig. 1. Symbolic expressions in simulation

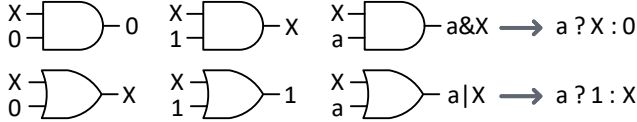


Fig. 2. Logic with the undefined value X

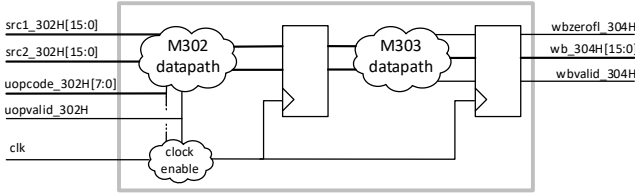


Fig. 3. Simplified ALU

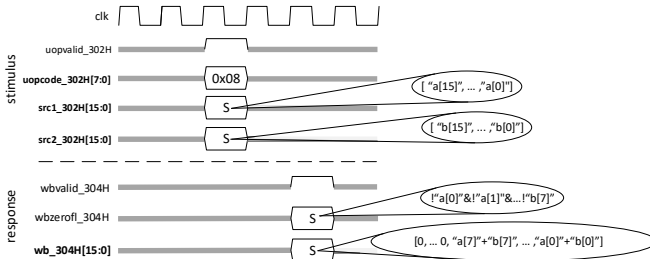


Fig. 4. Symbolic trace

a simulation using symbolic representations of sets of values in a single simulation trace [6], [7].

In a symbolic simulator the input stimulus may contain symbolic variables in addition to the traditional concrete values 0, 1, X or Z. These symbolic variables are effectively names of values, denoting sets of possible actual concrete values. In the simulation, these symbolic values propagate alongside the constant values, and in each logic gate, they may be combined with each other or one of the constants to result in either a logical expression on the symbolic variables, represented by an expression graph, or a constant. See Figure 1 for an example.

In a bit level symbolic simulator a single symbolic variable corresponds to the set of Boolean values containing both 0 and 1. If stimulus to a symbolic simulation refers to the variables

a , b and c , the internal signals might carry values like $a \wedge b$ or $a \vee (b \wedge \neg c)$. Usual logic rules apply: if the inputs to an AND-gate are a and 1, the output will be a , if the input to a NOT-gate is b , the output will be $\neg b$, and if the inputs to an AND-gate are a and b , the output is the logical expression $a \wedge b$. In symbolic simulation, a specific symbolic variable is associated with a specific signal and time in the stimulus. Associating a variable with a signal at a time does not fix the value, but instead gives a name that can be used to refer to the value.

In symbolic simulation, the constant value X is used to denote a universal undefined or unknown value, which propagates according to rules depicted in Figure 2. The value X denotes lack of information: we do not know whether the value is 0 or 1. The propagation rules reflect this intuition. Symbolic simulation uses X's as an abstraction mechanism: unlike symbolic variables, X's are an over-approximation of Boolean circuit behavior. Both symbolic variables and X's allow us to verify a property over a single symbolic trace, and conclude that it is valid over every possible trace instantiating the X's and the symbolic variables with 0's or 1's.

Figure 3 depicts a simplified pipelined ALU circuit with a 16-bit wide two-cycle data-path from sources to write-back. Figure 4 depicts a typical symbolic trace that might be used in the verification of this ALU, focusing on a single instance of an eight-bit wide bitwise OR micro-operation. The control signals are driven with concrete values corresponding to the operation, and the source data is driven with symbolic variables $a[15], \dots, a[0]$ and $b[15], \dots, b[0]$ in the one cycle in which the operation is issued. In all other cycles these signals are driven with the undefined value X (gray waveform). In the simulation, the values of the write-back data and zero flag two cycles later are then expressions on the symbolic variables associated with the source data.

A single symbolic simulation trace corresponds to a set of ordinary simulation traces, covering behaviors of the simulated circuit for all the possible instantiations of the symbolic variables with concrete values. The ability to cover all behaviors forms the basis of using symbolic simulation as a formal verification method. In this role symbolic simulation excels in verification of deep targeted properties of fixed length pipelines, typically of the transactional form *stimulus A at time t is followed by response B at time t + n*. It has a unique ability to carve out the circuit logic relevant to the progression of a pipeline while ignoring the rest of the circuit and other transactions in flight. As the approach is conceptually simple and concrete, it gives the human verifier a fine-grained visibility into the progress of the computation during a verification task, enabling precise analysis and mitigation of computational complexity bottlenecks. Because of these advantages, symbolic simulation can routinely handle circuits that are magnitudes above the capacity of more traditional formal property verification approaches, as well as circuits where the pipelines are too enmeshed to be amenable to equivalence-based verification methods.

B. Execution Cluster

Intel Core processor architecture has evolved gradually over the years. Typically, a new design project maintains functional backwards compatibility with earlier designs while providing improvements along different axes: new instructions and capabilities, improved performance or power, or design adjustments to meet side conditions set by a new manufacturing process. A design project routinely inherits components from earlier designs.

At high level, a single core consists of a set of major design components called *clusters*. The front-end cluster fetches and decodes architectural instructions, translates them to micro-operations and computes branch predictions. The out-of-order cluster receives streams of micro-operations from the front end, keeps track of dependencies between them, schedules ready-to-execute micro-operations for execution, takes care of branch misprediction and event recovery, retires completed instructions, and updates architectural state. The execution cluster carries out data computations for all micro-operations implemented by the design, performs memory address calculations, and determines and signals branch mispredictions. The memory cluster handles memory accesses, may contain first level caches and interfaces with a system-on-chip layer outside the core, including for example a graphics processing unit and a memory controller. The SystemVerilog source code of a cluster usually contains several hundred thousand lines of code. While not a physical entity like the above, microcode is also a major design component, the complexity of which is comparable to that of the clusters.

In this paper we focus on security validation of the execution cluster (EXE) on an Intel Core processor design. The EXE cluster consists of six main units: the integer execution unit (IEU) contains logic for plain integer and miscellaneous other operations, the single instruction multiple data (SIMD) integer unit (SIU) contains logic for packed integer operations, the floating-point unit (FPU) implements plain and packed floating-point operations such as DIV, MUL, ADD, etc., the address generation unit (AGU) performs address calculations and access checks for memory accesses, the jump execution unit (JEU) implements jump operations and determines and signals branch mispredictions, and the memory interface unit (MIU) receives load data from and passes store data to memory cluster, maintains store forwarding buffers, performs various datatype conversions, and takes care of data bypassing. In a typical contemporary Intel Core processor design, the EXE cluster implements over 5000 distinct micro-operations and supports multi-threading.

At an abstract level, the EXE cluster is a pipelined machine, receiving as input streams of micro-operations (micro-ops, uops) through a set of schedule ports. Each micro-operation receives its source data either through the cluster interface or through a bypass from a previous operation, and produces its result through a write-back port after an operation-dependent latency. The cluster has state components, which a micro-operation may read or update synchronously.

C. EXE Formal Verification

Formal verification of arithmetic data-paths has been a focus area at Intel ever since the Pentium® FDIV bug in 1994. The primary vehicle for this work is symbolic simulation, incorporated in Intel's in-house Forte verification toolset under the name of Symbolic Trajectory Evaluation (STE) [7]. Initially a research initiative during the Pentium Pro design cycle, Formal Verification has been carried out as a routine part of Intel processor development projects since Pentium 4 in 1999. All Intel Core processor EXE data-paths since 2005, as well as most Intel Atom® processor and Gen Graphics arithmetic engines have been formally verified using symbolic simulation [8], [9].

In concrete terms, EXE formal verification is carried out through a shared verification system called Cluster Verification Environment (CVE), a large software artifact that creates a standard, uniform methodology for writing specifications and carrying out verification tasks [8]. Underlying CVE is the Forte/reFlect toolset, consisting of the high performance simulator STE wrapped in a full-fledged functional programming language [7]. All verification takes place at the level of the full cluster, not the underlying individual units.

In verification of the EXE cluster, every micro-operation and every port on which the micro-operation can execute correspond to a separate symbolic simulation task. This simulation starts from a totally unconstrained initial state and focuses on one instance of the micro-operation under verification. The control signals that are relevant to the micro-operation are restricted according to the micro-operation, and the source data signals are driven with symbolic variables, as in the simplified example in Figure 4. Additionally, some internal and external control signals of the circuit are driven with symbolic variables and may be restricted using control invariants that are used to capture reachable state restrictions. Due to the unconstrained initial state of the simulation, such reachable state restrictions are not automatically accounted for in the verification and need to be manually formulated and separately verified. All other signals in the simulation are driven with the undefined value *X*. Altogether, in this setup the single instance of the micro-operation under verification in the single symbolic trace covers all possible invocations of the micro-operation in any legal trace of the circuit.

Effectively, in the verification setup for a single micro-operation the control signals are set to fix the data-path controls to match a single instance of that micro-operation, and symbolic variables on the data are used to exhaustively simulate the data-path instance. The simulation is then connected to an abstract functional reference model for the micro-operation through source and write-back mappings, and the output of the design and the reference model compared. These design-dependent mappings extract the intended source and result values for the micro-operation at the relevant times relative to the instance we are verifying.

For a large majority of micro-operations in the EXE cluster, the data-path can be exhaustively symbolically simulated in

one pass at the full cluster level. For certain complex operations like floating-point addition, careful case splits on the data space are needed to contain symbolic expression growth in the simulation, and for most complex operations like floating point divide or fused multiply add, a sequential decomposition strategy is applied.

III. EXE SECURITY VERIFICATION

A. EXE and Data Security

Traditionally EXE validation has focused on the functional correctness of the micro-operations, including the validation of control logic required for non-interference from other operations simultaneously in flight. Since the Spectre and Meltdown vulnerabilities, security validation has become a greater focus area. In both exploits, a rogue process can theoretically gain access to privileged data by observing the side effects of speculative, although ultimately unsuccessful access to a memory location containing the secret. A key ingredient of these exploits is that secret data temporarily propagates and influences execution flows in the micro-architectural level, although the results of the computations on the secret data are appropriately squashed before they become architecturally visible. In the classic functional correctness sense this is not a problem, as the secret data is never directly exposed. However, in the exploits a rogue process tracks the ways in which the secret data has influenced the execution flows, especially through timing analysis, in an effort to statistically deduce the secret with a high probability. This means that we need to secure the propagation of secret data also at the micro-architectural level. As it is difficult to foresee all the ways in which the secrets' influences on execution could be exploited, the best strategy is to try to limit the propagation of secrets in the system as best as we can, and try to block any leakages at a local level as early as possible.

Looking at the EXE cluster from the security and data leakage perspective, the first thing to note is that in the larger context some micro-operations may be privileged, and some may not, some data may be secret, and some may not, but EXE has no awareness of that. All it sees are micro-operations and data. Privileged and less privileged operations are interleaved out-of-order in the same thread and between threads. The mixture of secret and non-secret makes it harder to formulate a property *Thou shalt not leak secrets*, as we don't have a good measure of what counts as a secret. However, each micro-operation has a well-defined notion of the data it is expected to process: which buses at which times relative to the operation carry its source and result data. Relative to an operation, we can then over-approximate all other data as secret. This leads to the following fundamental security property for EXE:

For every micro-operation executing in EXE, its result data should be exclusively a function of its source data.

By 'result data' we mean the main write-back data bus, flags, faults, and all auxiliary outputs together. This security property can be formalized more accurately as:

For every micro-operation u , there is a function $spec(u)$ such that for every trace T of the circuit and every point t of T , if uop u is issued at point t of T and we write src for the source data of u and wb for the write-back data of u relative to the point t of T , then $wb = spec(u)(src)$.

For many micro-operations, this security property follows automatically from functional correctness. If the specification for the operation is fully defined for all possible source values, and we have verified that the implementation fully agrees with the specification, there is simply no logical possibility for the result data not to be purely a function of the source data. However, many operations have partially undefined results, where some result components are unspecified either for all or some source values. For example, some floating-point micro-operations do not fully support all possible source values, reverting to microcode flows for rare or hard-to-implement cases, leaving the result data undefined. Similarly, certain helper operations that are used only in specific microcode flows in contexts where some parts of the result are never used may leave these result components undefined. Designs take advantage of the undefined spaces, as they allow an implementation to be optimized without a need to maintain identical behavior in the undefined space. These undefined spaces provide an opportunity for a micro-operation to write back values that are derived from some other data than its sources, including possibly secret data that has been or is being processed by other micro-operations.

The most common scenario of data leakage in undefined spaces is when secret data processed by an earlier micro-operation lingers in some internal flops of EXE and is passed to the write-back bus as a later micro-operation's undefined result. In a fully pipelined machine where all clocks toggle all the time, this scenario cannot happen, as secret data stays in any pipe-stage for exactly the one cycle when it is being processed before being overwritten by the next wave of values. However, such always-toggling designs are a thing of the past. Qualified clocks are ubiquitous, and their use increases and becomes more fine-grained by every design generation because of power considerations. In many data-paths the clocks toggle at most once for each operation. This means that any secret data processed by an operation remains in internal flops in every pipe-stage, until the next operation executing in the same data-path clears it. In this context the security property above can be viewed as setting a security perimeter around EXE. Secret data can linger on inside the cluster but cannot be exported through the write-back bus by any micro-operation.

The general concept of the analysis of data leakages through undefined behavior is directly relevant for the prevention of Meltdown-type vulnerabilities, although the areas primarily contributing to Meltdown are outside our focus area in EXE. An essential part of Meltdown is transient execution after a faulting load micro-operation from an out-of-bounds memory location containing secret data [10]. While the problematic load micro-operation produces a fault due to an access check violation, it may, under certain circumstances, nevertheless

have read the secret value from the memory location and passed the value on to a subsequent flow that exposes the secret. The specification for a load micro-operation is likely to be of the form *if the load does not generate a fault, the writeback data will be the value held by the memory location pointed to by the sources, otherwise the writeback data is a don't-care*. Note that the naive specification, without the faulting condition and the don't-care space, is very unlikely to hold for any real implementation, as a load can fault for a variety of reasons, many of which prevent the routing of the memory data to the writeback. This undefined space in the specification allows the secret to be exposed, or conversely, as pointed out by Canella et al: “...merely replacing the data of a faulting instruction with a dummy value suffices to block Meltdown-type leakage in silicon...” [10, p 252].

B. EXE Security Analysis with Symbolic Simulation

Considering the fundamental security property formulated above, an extremely useful feature of symbolic simulation is that every symbolic variable can be uniquely related to the signal and time it was associated with in the stimulus. Each 1 in stimulus looks exactly like any other 1, each 0 like any other 0, but every symbolic variable carries immediately in its name the notion of which signal and time it originated from. The uniqueness of names and the setup of EXE verification allows us to re-phrase the security property as:

For every micro-operation executing in EXE, the symbolic expressions for its result data should only refer to symbolic variables associated with its source data, and should not allow the undefined value X.

This property is relative to the symbolic simulation task for the micro-operation, as outlined in Section II-C. The symbolic re-formulation of the security property guarantees the original version since the single symbolic simulation for the micro-operation is an over-approximation of every possible invocation of the micro-operation in any trace. This means that we can simply read the function $spec(u)$ required by the original definition, mapping source data to the result, from the symbolic expressions for the result data.

Another way of viewing the matter is that the symbolic expressions on the write-back signals fully capture all dependencies of the write-back on any signals in their fan-in cone. The constant values in the simulation do not matter in this respect. Since the symbolic simulation for the micro-operation over-approximates every possible invocation of it in any trace, every constant value in the symbolic simulation is also present in all these invocations. Consequently, the propagation of such constants in the simulation to the write-back cannot disclose anything about the internal state of the circuit that would not be universally true. As a technical restriction, in our work all case splits and decompositions used to alleviate verification complexity are on data and not on control signals and will not turn any symbolic variables on control signals to constants.

Notice that the symbolic formulation of the security property is not a property about the value of the result data itself.

Instead, it is a property about the symbolic expression used to represent the value of the result data in the simulation, and the symbolic names that occur in that expression. Because it talks about names, not values, it is not something that could be coded in methods that describe properties of signal values, such as SystemVerilog Assertions.

When we run a micro-operation that has a fully specified result data, we naturally verify that it writes exactly the data we expect it to and nothing else, as otherwise the verification would fail. However, when there is an undefined space in the output, the situation is trickier because we don't know what value to expect. The use of named variables allows us to verify that the result data is a function of the source data without the need to say what that function $spec(u)$ is, i.e. **without needing to specify the expected result value**. This is very efficient when we are looking at the undefined space, where typically there is no good definition of what the result should be.

C. Implementation

Next, we describe in detail how this idea was implemented. In high level, named variables allow us to:

- A) Sample the output of a DUT to get a list of named variables that have propagated to it and occur in the symbolic expression it holds. In the example in Figure 4, bit [0] of the write-back data carries the expression $a[0] + b[0]$, referring to the variables $[a[0], b[0]]$. We call this list the *dependency list* of the expression.
- B) Identify suspicious names in the dependency list. The CVE infrastructure has a known naming convention, so the variable name allows us to distinguish the data that we would expect to propagate from suspicious data. In the example in Figure 4, the names $a[0]$ and $b[0]$ are expected, since they are the named variables driven to the sources of the operating uop.

The security analysis has two outcomes. First, we can detect security vulnerabilities where they exist. Second, the absence of detected vulnerabilities for the vast majority of micro-operations provides strong evidence that no secrets can be leaked to the interface of the cluster through those operations.

Data propagation in the circuit is often gated by specific operations that exclusively enable the data flow. If that enabling is too short, and there is no mechanism that clears the data after the operation, it can hang there. Stale data becomes a security risk when another operation can read this data. In early stages of verification environment development for a new project, the validation focuses on pure data-path verification in a sterile environment, and as a simplification, disables power gating and lets clocks toggle freely. At this stage all data flows uninterrupted, and we cannot guarantee there are no leakages coming from stale data on a power-gated bus. Security verification analysis becomes effective and meaningful only when we enable all power optimizations in the formal environment. At the time we started this security initiative, this pre-condition was met in almost all areas of the design we were working on.

Formal verification of arithmetic data-paths in the EXE cluster is fully covered in CVE using symbolic simulation. We have specifications for all existing micro-operations and the infrastructure to run a full regression to collect any information needed for the extra layer of security check. This provided a solid base for our analysis, and an efficient process that led to interesting results in a short time. The process can be divided into three stages.

1) *Identify operations that have an undefined result.*

As an example, in the simplified ALU in Figure 3 the write-back bus is 16 bits wide, but a shorter operation like the eight-bit OR only uses bits [7:0] for the result. The upper bits [15:8] could be left undefined, which might provide an opportunity for data leakage. For any micro-operation, CVE provides two different mechanisms for undefined results:

- Each uop in CVE has a defined data type signature, which specifies useful static information about the shape of the sources and result of the uop, such as data size, data type (integer, floating-point), signed/unsigned etc. The source or write-back data can be of NULL type, meaning it is not used by the uop. For NULL write-back, the checkers will not sample the write-back bus at all in a simulation.
- A uop may have a defined write-back datatype, but its specification may explicitly encode a don't-care space. For example, the data output of a divide operation could be defined as a don't-care when the divisor is zero. In this case the checkers will sample the output in a simulation but will ignore the value for the functional correctness check. In the eight-bit OR example, we could sample the full 16 bit write-back bus, but not necessarily check the upper eight bits, leaving them explicitly undefined.

For both methods the existing CVE data structures allowed us to easily identify the set of uops that produce undefined results, creating a clear goal for the main security analysis. The first step in enabling the security check was to switch from the first method to the second one for all uops, to make sure we always sample the write-back bus: identify the uops using the first method, convert the NULL data signatures to a meaningful type, and incorporate the explicit don't-care space into the functional specification.

2) *Sample results and detect unexpected variables.*

This stage is the heart of the process, using the existing symbolic simulation capability in the two steps above: A) Sample the output and extract the list of variables in the symbolic expression, and B) Identify suspicious variable names in the list. The ingredients of this stage are:

- Every variable in the dependency list has a name.
- Expected variables are the named variables associated with the source signals in the aligned source pipe-stage of the current operating uop, as discussed

above. As sampled by the operating uop, they are considered safe.

- All X values on the outputs are flagged, since the unnamed undefined value X cannot tell where it came from and is therefore inherently suspicious.
- By convention, a driven variable that is not part of expected source data for a uop uses a name that is a combination of the signal and the time at which it was driven, for example: "SignalName@24".

Given the values in the write-back bus, we check for X 's and query the variable dependency list for suspicious names. In the eight-bit OR example of Figure 4, there are no X values, and the dependency list includes only 'good' names such as $a[7]$ or $b[0]$.

This check is fully automated, as the classification of variable names to good vs suspicious ones can be done mechanically based on existing information about the intended uop source interfaces and variable naming conventions.

3) *Trace the suspicious variables.*

The presence of the undefined value X or a suspicious name in the dependency list does not yet automatically mean that what we see is real data leakage. By methodology, symbolic simulation uses a maximally uninitialized start state for the simulation, with all signals having the value X , and uses stimulus that drives X 's on most inputs to the circuit, overapproximating the real legal behaviors of the circuit. We need to trace the suspicious variable or X , see how it propagated to the write-back, and understand whether the path to the write-back is possible in the real operating environment of the circuit. This stage is like the debug process of any simulation, tracing the origin of a value in the circuit. We use a schematic viewer that shows symbolic values and trace the ones that we find interesting. In some cases, to better analyze a behavior, we strengthen the simulation to drive a variable at an internal signal that used to hold an unnamed X that may propagate to the write-back.

Consider for example the simplified ALU of Figure 3 and assume that the circuit is augmented with power gating logic that turns off clocks for the high eight bits [15:8] of the data-path for operations that only operate on the low eight bits [7:0] of data. If we now simulate an eight-bit OR operation on the circuit as in Figure 5, we might observe X values in bits [15:8] of the write-back as in Figure 6, instead of the 'good' result of Figure 4. Tracing back the X values on the write-back, we would find an internal flop with the output X and a clock that does not toggle, as in Figure 7. In the circuit, this flop will hold any value the previous operation has left there, presenting a leakage risk. To check whether this data really propagates to the output, we want to track a concrete named variable. To do this, we drive unique named variables "Src1[15]@23" ... "Src1[8]@23" to the internal flop as in Figure 8, and observe these variables in the write-back, as in Figure 9. Once we understand the leakage mechanism, we can

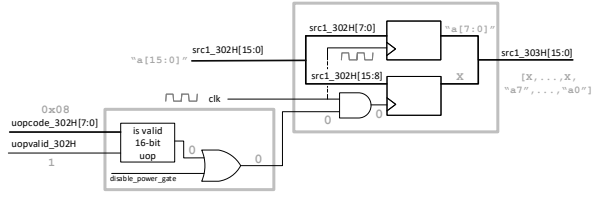


Fig. 5. Clock gating for eight-bit operations

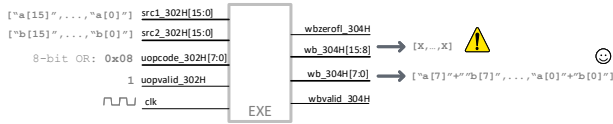


Fig. 6. Sampled X on the write-back bus

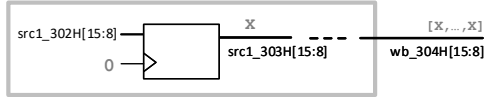


Fig. 7. Trace back the X to a gated clock



Fig. 8. Replace the X with a named variable

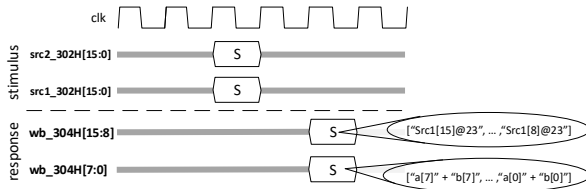


Fig. 9. Symbolic waveform with data leak

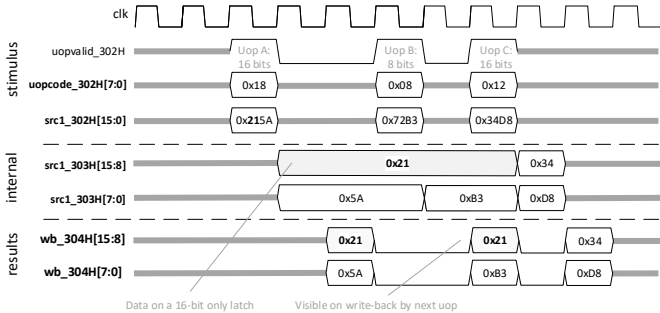


Fig. 10. Concrete waveform with data leak

then manually generate a concrete example exhibiting both an earlier uop leaving behind stale data, and a later uop that leaks the stale data to the write-back bus, as in Figure 10. In this example, the high eight data bits of a 16-bit uop A remain in the internal state until they are overwritten by the next 16-bit uop C, and are exposed by the 8-bit uop B in the meanwhile.

IV. RESULTS

The flow of security verification was implemented as an automated extra check on top of the traditional data-path symbolic simulation. The process leveraged the existing capabilities of CVE that already supported all EXE uops. This gave us the ability to run a full regression and get first results quickly.

We chose to focus on the write-back data interface buses and concentrated on the about 2000 uops for which these buses are relevant, out of about 5000 legal uops for the cluster in total. Among these uops we first identified the ones that have fully or partially unspecified write-back data. Our analysis showed that 89.4% of the uops were completely specified, and 10.6% had unspecified write-back data. We then further analyzed the uops with unspecified write-back data by symbolic dependency analysis and found that 97.8% of uops were either completely specified or exhibited no unexpected data at write-back, whereas 2.2% of the uops had an undefined result space and failed the dependency analysis.

For the 97.8% of the uops that passed our analysis, we provided strong evidence that there is no risk of data leakage, as our analysis took place in the formal framework covering all possible behaviors. Note also that the dependency analysis allowed us to reduce the ratio of suspicious uops from 10.6% to 2.2%. As a restriction in scope, we did not look at data leakages in the bypass network, although the method would be equally applicable there.

The first real local EXE potential data leakage was discovered in less than a month. In a total effort of about two months of work, we discovered several different potential leakage mechanisms, all previously unknown. The failures were analyzed and grouped to RTL bugs with a common cause. Examples of potential leakage mechanisms include:

- 1) Uop A computed information intended to be written to the write-back data bus. It went through a latch that was toggling only while uop A was operating, for one cycle, and shut down right after uop A had completed. Therefore, the output of that latch was not cleared, and the data was stuck there on an internal bus. Analyzing uop B that was not expected to produce data (undefined write-back), we could see that uop A's data was propagating freely all the way to the write-back bus.
- 2) The data-path of a certain unit contained a MUX prior to the write-back bus with separate selects for specific uops and default logic shared by many uops. A particular uop C with undefined write-back executing in the unit read stale data left behind by any previous uop using the default logic.

- 3) Most uops that write only part of the write-back bus, for example 32 bits out of 128, have a clear definition of the unused bits, and we sample them along with the computed result of the lower part in regular data-path verification. In one exception, the upper part for a specific uop D was left unspecified. Tracing back the write-back, we reached an internal source bus shared by several operations, with a clock toggling just once per uop, causing the data to hang. Usually, the next uop would clear the bus. Uop D did not, leaving the upper bits of the source data left behind by the previous uop.

These bugs were all reproduced in normal simulation. They did not cause a functional failure: the results are never checked since they fall into the don't-care space of the specification. However, it was clear that the value written to the write-back is exactly the value left behind by a previous uop.

After the detection of these kinds of potential data leaks, there are several options for actions to fix them. The straightforward solution is to modify the currently undefined uop to have a defined value, e.g. write zeroes to the write-back data. This will be the easiest to verify because it will become again a strongly defined data-path verification task. It will also be the strongest solution, as it truly closes the leak. Another solution is to clear the stale data left by the earlier uop, for example by opening the gating clock for an extra cycle. Both options close the leak at the EXE boundary but require changing the design and could cost power or area.

If it is not possible to fix the design, another option is in the microcode level, making sure the undefined operation is not used in any way it could be exploited. Effectively here one establishes a security perimeter with a larger scope than EXE to see that the compromised data is contained before it becomes visible through a vulnerability at a higher level. This method is less optimal than the ones above, as the analysis scope is larger, outside the scope of existing formal tools, and relies more on finding parallels with known vulnerabilities, while new ways of exploiting information leaked out of the cluster may emerge. Also, micro-code implementation is dynamic, and it is possible that changes to the usage model that is safe today may make it unsafe tomorrow.

The potential local data leakages discovered by our analysis were addressed during the design project and as a result do not lead to a security violation at a user visible level in the final product.

V. SUMMARY

Symbolic simulation's special trait — the usage of named variables — makes it a productive method to analyze data leakage risks. The scope of this work was huge for any formal analysis: a whole cluster, thousands of operations, and hundreds of thousands of flops in the circuit. Out of those, without having any prior knowledge where to look for the risks, we hit the relatively few instances that mattered in a short time. We found real issues, in a live project, issues that were not detected by any other method.

In this paper we described how we leveraged the existing environment of CVE that already supports the thousands of specifications in EXE cluster, holds information about data types and has a clear naming convention. This made the process efficient and demonstrated the importance of the complete verification environment covering EXE data-path. It is also important to clarify that the general concept we describe here is not dependent on it. Security verification by symbolic simulation can be implemented in various designs, where we do not have such infrastructure to rely on. Symbolic simulation is the key in analyzing data leakage risks of this kind, not the formal environment in itself.

In future design projects, with the increasing demand for security validation, we hope to explore where we can further develop this usage of symbolic simulation.

ACKNOWLEDGEMENTS

We would like to thank Arkady Neyshadt for his security analysis, Gilad Holzstein, Robert Jones, Alex Levin, Yoav Moratt and Nir Shildan for discussions on security, Annette Upton for detailed feedback on the paper, and David Turner, Yaniv Dana and Alon Flaisher for the opportunity to carry out this work.

REFERENCES

- [1] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pp. 288–28815, 2019.
- [2] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, pp. 317–331, 2010.
- [3] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial SoC design," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–2, 2014.
- [4] "Cadence JasperGold Security Path Verification (SPV) App," 2021.
- [5] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 337–342, 2016.
- [6] 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.
- [7] C.-J. Seger, R. Jones, J. O'Leary, T. Melham, M. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, 2005.
- [8] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel Core i7 processor execution engine validation," in *Computer Aided Verification* (A. Bouajjani and O. Maler, eds.), pp. 414–429, Springer Berlin Heidelberg, 2009.
- [9] A. Gupta, M. V. A. KiranKumar, and R. Ghughal, "Formally verifying graphics FPU," in *FM 2014: Formal Methods* (C. Jones, P. Pihlajasaari, and J. Sun, eds.), pp. 673–687, Springer International Publishing, 2014.
- [10] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 249–266, USENIX Association, Aug. 2019.

Scaling Up Hardware Accelerator Verification using A-QED with Functional Decomposition

Saranyu Chattopadhyay^{*}, Florian Lonsing^{id*}, Luca Piccolboni^{id†}, Deepraj Soni[¶], Peng Wei[§], Xiaofan Zhang^{||}, Yuan Zhou[‡], Luca Carloni[†], Deming Chen^{id||}, Jason Cong^{id§}, Ramesh Karri[¶], Zhiru Zhang[‡], Caroline Trippel^{*}, Clark Barrett^{id*}, Subhasish Mitra^{*}

^{*}Stanford University, [†]Columbia University, [‡]Cornell University, [§]University of California, Los Angeles, [¶]New York University, ^{||}University of Illinois, Urbana-Champaign

Abstract—Hardware accelerators (HAs) are essential building blocks for fast and energy-efficient computing systems. *Accelerator Quick Error Detection (A-QED)* is a recent formal technique which uses Bounded Model Checking for pre-silicon verification of HAs. A-QED checks an HA for *self-consistency*, i.e., whether identical inputs within a sequence of operations always produce the same output. Under modest assumptions, A-QED is both sound and complete. However, as is well-known, large design sizes significantly limit the scalability of formal verification, including A-QED. We overcome this scalability challenge through a new decomposition technique for A-QED, called *A-QED with Decomposition (A-QED²)*. A-QED² systematically decomposes an HA into smaller, functional sub-modules, called *sub-accelerators*, which are then verified independently using A-QED. We prove completeness of A-QED²; in particular, if the full HA under verification contains a bug, then A-QED² ensures detection of that bug during A-QED verification of the corresponding sub-accelerators. Results on over 100 (buggy) versions of a wide variety of HAs with millions of logic gates demonstrate the effectiveness and practicality of A-QED².

I. INTRODUCTION

Hardware accelerators (HAs) are critical building blocks of energy-efficient System-on-Chip (SoC) platforms [1]–[3]. Unlike general-purpose processors, HAs implement a set of domain-specific functions (e.g., encryption, 3D Rendering, deep learning inference), referred to as *actions* in this paper, for improved energy and throughput. Today’s SoCs integrate dozens of diverse HAs (e.g., 40+ HAs in Apple’s A12 mobile SoC [4]).

Unfortunately, the energy and throughput improvements enabled by HAs come at the cost of increased design complexity. Ensuring that a given SoC will behave correctly and reliably requires verifying each and every constituent HA. Furthermore, HAs must achieve short design-to-deployment timelines in order to meet the needs of a wide variety of evolving applications [5]. Using conventional formal verification techniques to verify HAs faces several key challenges. Manually crafting extensive design-specific formal properties or full abstract functional specifications can be time-consuming and error-prone [6], [7]. Moreover, scaling verification to large HAs (with millions of logic gates) is difficult or even infeasible using off-the-shelf formal tools.

A recent formal verification technique targeting HAs, *Accelerator-Quick Error Detection (A-QED)* [8], overcomes the first challenge above. A-QED is readily applicable for a

popular class of HAs: *loosely-coupled accelerators (LCAs)* [9], [10] (i.e., HAs that are not integrated as part of a central processing unit (CPU), but via an SoC’s network-on-chip or a bus) that are also *non-interfering*. Non-interfering HAs produce the same result for a given action independent of their context within a sequence of actions (not to be confused with combinational circuits). In other words, the state of the accelerator does not affect future computations, and each computation is independent from previous computations. In contrast, computations of *interfering* HAs depend on state that is the result of previous computations. A-QED uses Bounded Model Checking (BMC) [11] to symbolically check sequences of actions for *self-consistency*. Specifically, it checks for *functional consistency (FC)*, the property that identical inputs within a sequence of operations always produce the same outputs. It was shown that FC checks, together with *response bound (RB)* checks and *single-action correctness (SAC)* checks, provide a thorough verification technique for non-interfering LCAs [8]. However, despite its success in discovering bugs in moderately-sized HA designs, A-QED suffers from the scalability challenges of formal tools. For example, A-QED (backed by off-the-shelf formal verification tools) times out after 12 hours when run on NVDLA, NVIDIA’s deep-learning HA [12] with approximately 16 million logic gates.

In this paper, we present a new verification approach called *A-QED with Decomposition (A-QED²)* to address the scalability challenge. First, we introduce a new, more general formal model of HA execution, which captures both interfering and non-interfering LCAs. We then show how A-QED² can *decompose* a large LCA into smaller *sub-accelerators* in such a way that both FC and RB checks can be directly applied to the sub-accelerators. Unlike conventional verification approaches based on decomposition, no new properties need to be devised to apply FC and RB to the decomposed sub-accelerators. Existing decomposition approaches can be leveraged to additionally check SAC of the sub-accelerators. A-QED² is complementary to verification approaches that rely on design abstraction, which can be used to further improve scalability and to simplify the effort required for SAC checks on decomposed sub-accelerators.

This paper presents both a formal foundation of A-QED² and an empirical evaluation that demonstrates its bug-finding capabilities in practice. We prove that A-QED’s completeness guarantees [8] continue to hold for A-QED²—if the full HA

under verification contains a bug, then A-QED² will detect that bug. Furthermore, we apply A-QED² to a wide variety of non-interfering LCAs (although our theoretical proofs apply to interfering LCAs as well): 109 different (buggy) versions of large open-source HAs of up to 200 million logic gates (including industrial HAs). Our empirical results focus on designs which are described in a high-level language (e.g., C/C++) and then translated to Register-Transfer-Level (RTL) designs (e.g., Verilog) using High-Level Synthesis (HLS) flows, where appropriate optimizations like pipelining and parallelism are instantiated. Such HLS-based HA design flows are becoming increasingly common in industry. However, A-QED² is not restricted to these specific HA design styles. Our empirical results show:

- 1) Off-the-shelf formal tools cannot handle large HAs with millions of logic gates, even when the HAs are expressed as high-level C/C++ designs. In our experiments, A-QED verification of many such HAs times out after 12 hours or runs out of memory.
- 2) A-QED² is broadly applicable to a wide variety of HAs and detects all bugs detected by conventional simulation-based verification. For very large HAs with several million (up to over 200 million) logic gates, A-QED² detects bugs in less than 30 minutes in the worst case and in a few seconds in most cases.
- 3) A-QED² is thorough – it detected all bugs that were detected by conventional (simulation-based) verification techniques. At the same time, A-QED² improves verification effort significantly compared to simulation-based verification – $\sim 5X$ improvement on average, with $\sim 9X$ improvement (one person month with A-QED² vs. 9 person months with conventional verification flows) for the large, industrial designs.

The rest of this paper is organized as follows. Sec. II presents related work. Sec. III presents a formal model of the accelerators targeted by A-QED² and our decomposition technique. Sec. IV details the A-QED² algorithms. Results are presented in Sec. V, and Sec. VI concludes.

II. RELATED WORK

Conventional formal HA verification, e.g., [13]–[16], requires a specification, typically in the form of manually written, design-specific properties. These are then combined with a formal model of the design and handed to a formal tool, which attempts to prove the properties or find counter-examples. For the verification of latency-insensitive designs, an approach was developed to automatically derive and check properties from the RTL synthesized in HLS flows [17]. However, these derived properties are targeted at specific types of bugs.

Large design sizes have always been a challenge for formal techniques, and various approaches to this problem have been proposed. Among techniques to improve scalability are abstraction [18] and compositional reasoning (cf. [19]). The former removes details of the design, gaining scalability at the cost of possible false errors. Finding a scalable abstraction that does not generate false errors can be difficult and may be

impossible in some cases. The latter uses *assume-guarantee* reasoning (e.g., [20]–[25]) and can be applied to decompose a large HA into smaller sub-modules. Importantly, the property p of the HA to be verified must also be decomposed into properties of the sub-modules. The properties of the sub-modules are verified individually under certain assumptions about the behavior of the other sub-modules. If all the properties of the sub-modules hold under the respective assumptions, then it can be concluded that p holds. However, finding the right properties for this decomposition can be very challenging.

Unlike for general compositional reasoning, the two main components of A-QED² (FC and RB) do not require decomposing properties. FC, in particular, leverages a universal *self-consistency* property. Self-consistency expresses the property that a design is expected to produce the same outputs whenever it is provided with the same inputs [26]. In A-QED², self-consistency is checked independently for each sub-module (sub-accelerator in our case). Importantly, these aspects of A-QED² do not require complex assumptions about the behavior of the other sub-modules.

It is challenging to establish general *completeness guarantees* for conventional formal verification techniques [27]–[31], since completeness depends on the set of properties being checked. Designer-guided approaches [32], [33] require manual effort. Automatic generation of properties is usually incomplete and depends on abstract design descriptions [34] or models [35], or analysis of simulation traces [36], which may be difficult. In contrast, we have general completeness results for A-QED².

A-QED² builds on A-QED [8] and leverages BMC [11], [37]. Similar approaches based on self-consistency have been successfully applied to other classes of hardware designs, such as processor verification (as *symbolic quick error detection (SQED)* [38]–[43]), as well as to hardware security [44]–[49].

III. FORMAL MODEL AND THEORETICAL RESULTS

In this section, we introduce a formal model for HAs, define functional consistency (FC), single-action correctness (SAC), and responsiveness for the model, and show how these properties provide correctness guarantees. We then define a notion of functional composition for our model and show how the above properties can be applied in a compositional way.

Our formal model differs from the one in previous work [8] in several important ways. It allows multiple inputs to be provided simultaneously by explicitly modeling the notion of *input batches*. The HAs we consider are *batch-mode accelerators* as they process input batches and produce output batches. Modeling batches is useful because it more closely matches the interfaces of real HAs. Moreover, input batches enable *intra-batch checks* for FC checking, as we describe below. With intra-batch checks, only one input batch is used for FC checking. Intra-batch checks are more restricted than general FC checks. However, they are easier to set up and run in practice, and they are highly effective at finding bugs, as we demonstrate empirically.

Our model also explicitly separates control states and memory states. Control states represent control-flow information

such as, e.g., program counters in HLS models of HAs. Memory states represent all other state-holding elements, e.g., program variables.

In our model we distinguish starting and ending control states in which inputs are provided and the computed outputs are ready, respectively. This makes the formulation simpler and is also a better match for HLS designs written in a high-level language, which is our main target in the experimental evaluation. Further, our model enables us to formulate the notion of *strong FC*, which leads to a complete approach to bug-finding with only two input batches.

In previous work [8], a ready-valid protocol was used to model input/output transactions in RTL designs. In contrast, our focus is on HLS designs. Finally, we distinguish so-called *relevant states*, which are parts of the state space that can affect output values. This makes it possible to model interfering as well as non-interfering HAs. In our experiments we focus on non-interfering HAs.

Before presenting formal definitions, we illustrate terminology informally with an example of a non-interfering batch-mode HA as shown in Listing 1 (a slightly modified excerpt of an HA implementing AES encryption [50]).

Function `fun` of the HA has two sub-accelerators in lines 8-10 and 13-14 which are identified and verified by A-QED². Each sub-accelerator applies a certain operation to all inputs in an input batch of HA. In general, the *batch size* of an HA is the number of inputs in each batch, which is 256 for this HA. The first sub-accelerator ACC_1 processes an input batch provided via `data` and stores its output batch in `buf`. The second sub-accelerator ACC_2 takes its input batch from `buf`, where it also stores the output batch it produces. The control state of the HA is only implicitly represented by the program counter when executing function `fun`. Variables `key` and `local_key` are global and determine the relevant state of the HA on which the result of the encryption operation depends. The HA is non-interfering because `key` and `local_key` are left unchanged by ACC_1 and ACC_2 . Constants `BS`, `UF`, and `US` are used in HLS to configure the generated RTL.

Listing 1: HA Example (AES Encryption)

```

1  #define BS ((1) << 12) // BUF_SIZE
2  #define UF 2 // UNROLL_FACTOR
3  #define US BS/UF // UNROLL_SIZE
4
5  void fun(int data[BS], int buf[UF][US], int key[2]){
6      int j, k;
7      //==ACC1 START==
8      for(j=0; j<UF; j++)
9          for(k=0; k<BS/UF; k++)
10             buf[j][k] = *(data + i*BS + j*US + k)^key[0];
11     //==ACC1 END==
12     //==ACC2 START==
13     for(j=0; j<UF; j++){
14         aes256_encrypt(local_key[j], buf[j]);
15     //==ACC2 END==
16 }

```

Definition 1. A batch-mode hardware accelerator (HA) is a finite state transition system [51], [52] $Acc := (b, A, D, O, S, s_{c,I}, s_{c,F}, S_m, I, T)$, where

- $b \in \mathbb{N}$ with $b \geq 1$ is the batch size,

- A is a finite set of actions,
- D is a finite set of data values,
- O is a finite set of outputs,
- $S = S_C \times S_M$ is the set of states consisting of control states S_C and memory states $S_M = S_{In} \times S_{Out} \times S_R \times S_N$, where
 - $S_{In} = (A \times D)^b$ are the input states,
 - $S_{Out} = O^b$ are the output states,
 - S_R are the relevant states, and
 - S_N are the non-relevant states,
- $s_{c,I} \in S_C$ is the unique initial control state, which defines the set $S_I = \{s_{c,I}\} \times S_M$ of initial states,
- $s_{c,F} \in S_C$ is the unique final control state, which defines the set $S_F = \{s_{c,F}\} \times S_M$ of final states,
- $S_{m,I}$ is the set of allowable initial memory states, which defines the set $S_{CI} = \{s_{c,I}\} \times S_{m,I}$ of concrete initial states,
- and $T : S \rightarrow S$ is the state transition function.

When referring to different HAs, e.g., Acc_0 and Acc_1 , we use subscript notation to identify their components, e.g., $Acc_0 := (b_0, A_0, D_0, O_0, S_0, s_{c,I,0}, s_{c,F,0}, S_{m,I,0}, T_0)$.

We use $\mathbf{v} = \langle v_1, \dots, v_{|\mathbf{v}|} \rangle$ to denote a sequence with elements denoted v_i and length $|\mathbf{v}|$. We concatenate sequences (and for simplicity of notation, single elements with sequences) using \cdot , e.g., $\mathbf{v} = v_1 \cdot \mathbf{v}'$, where $\mathbf{v}' = \langle v_2, \dots, v_{|\mathbf{v}|} \rangle$. We will sometimes identify a sequence \mathbf{v} with the corresponding tuple, and we write $v \in \mathbf{v}$ to denote that v appears in \mathbf{v} . We denote the i -th element of a tuple \mathbf{t} as $t(i)$.

An HA Acc operates on a set I^b of *input batches*, where b is the *batch size* and $I = A \times D$. An input batch $in \in I^b$ has b *batch elements*, each consisting of a pair (a, d) containing an action $a \in A$ to be executed and data $d \in D$ (the data on which action a operates).

A state $s \in S$ of Acc with $s = (s_c, s_m)$ consists of a control state $s_c \in S_C$ and a memory state $s_m \in S_M$. The control state s_c represents control-flow-related state (e.g., the program counter in an execution of a high-level model of Acc). In a run of Acc , the control state starts at a distinguished initial state $s_{c,I}$ and ends at a distinguished final state $s_{c,F}$.

The memory state represents all other state-holding elements of Acc (including, e.g., global variables, local variables, function parameters, and memory elements). The memory state $s_m = (s_{in}, s_{out}, s_r, s_n)$ is divided into four parts. The first part, $s_{in} \in S_{In}$, contains the input to Acc . More precisely, in a run of Acc , the value of s_{in} in the initial state is considered the input for that run. Similarly, at the end of a run of Acc , $s_{out} \in S_{Out}$ contains the outputs for that run (i.e., the values computed by Acc based on the inputs present at the start of the run).

The relevant state s_r represents those state elements (other than s_{in}) that can influence the values of the outputs. Any part of the state that can affect the output value in at least one execution should be included in the relevant state. As an example of when this is needed, consider an encryption HA with actions for setting the encryption key and for encrypting data. The internal state that stores the key is part of the relevant state because it affects the way the output is computed from the

input. The non-relevant state s_n is everything else. We write $ctrl(s)$, $mem(s)$, $inp(s)$, $out(s)$, $rel(s)$, and $nrel(s)$ to denote the components s_c , s_m , s_{in} , s_{out} , s_r , and s_n , respectively. We overload the latter four operators to apply to memory states as well, and we lift the notation to sequences of states.

The set S_I of initial states contains all states resulting from combining a memory state in S_M with the unique initial control state $s_{c,I}$. The concrete initial states, S_{CI} , are a subset of S_I , and essentially represent the reset state(s) of the HA. They play a role in defining the *reachable* states (see Definition 3, below). The set S_F of final states contains all states resulting from combining a memory state in S_M with the unique final control state $s_{c,F}$. Finally, the transition function T defines the successor state for any given state in S .

Given an input batch $in \in I^b$, the HA produces an *output batch* $o \in O^b$ as follows. Let $s_0 \in S_I$ be an initial state with $inp(s_0) = in$, and let $s = T(s_0) = \langle s_1, \dots, s_k \rangle$ denote the sequence of $|s| = k$ *successor states* generated by the transition function T , where $s_i = T(s_{i-1})$ for $1 \leq i \leq k$, such that $s_k \in S_F$ is a final state (and no earlier states in s are final states). We also assume, without loss of generality, that $ctrl(s_i) \neq s_{c,I}$ for $i > 0$. The final state s_k holds the output batch $out(s_k) = o$ with $o \in O^b$ that is produced for the input batch $inp(s_0) = in$. Given a sequence s , we write $initsym(s)$ and $final(s)$ to denote the subsequence of s containing all initial and final states that occur in s , respectively.

Given a sequence of input batches, an HA generates a sequence of output batches based on concatenating executions for each input batch.

Definition 2. Let in be a sequence of inputs with $n = |in|$, and let $s_0 \in S_I$. Then, $StateSeq(in, s_0)$ denotes the sequence of successor states of s_0 that result from executing in , which is defined as follows.

- Let s'_0 be the result of replacing $inp(s_0)$ with in_1 in s_0 . Let $s' = s'_0 \cdot T(s'_0)$.
- If $|in| = 1$ then $StateSeq(in, s_0) = s'$
- If $|in| > 1$, then
 - let $s_f = final(s')$ (which is unique),
 - let $s_i = (s_{c,I}, mem(s_f))$,
 - let $s'' = StateSeq(\langle in_2, \dots, in_n \rangle, s_i)$.
 - Then, $StateSeq(in, s_0) = s' \cdot s''$.

In Definition 2, the state s_i from which each subsequent input batch is executed is obtained from the final state s_f produced from executing the previous input batch. Given an HA Acc , we write $StateSeq(Acc, in, s_0)$ to explicitly refer to the successor states of s_0 generated by Acc . If Acc is clear from the context, we omit it.

Definition 3. A state $s \in S$ is *reachable* if $s \in S_{CI}$ or if there exists a concrete initial state $s_0 \in S_{CI}$ and sequence in of input batches such that $s \in StateSeq(in, s_0)$. A *relevant state* s_r is *reachable* if $s_r = rel(s)$ for some *reachable state* s .

Note that the initial states S_I are not necessarily all reachable.

Next, we define an abstract specification for an HA function. Note that we use this to define correctness, but one of the

features of A-QED is that the specification is not needed for the main verification technique.

Definition 4 (Abstract Specification). For an HA Acc , let $Spec : I \times S_R \rightarrow O$ be an abstract specification function.

Definition 4 states that the value of an output computed by an HA is completely determined by the corresponding input and the relevant part of the memory state when the HA was started. Note that the inclusion of the relevant memory state makes the definition general enough to model interfering HAs. To model non-interfering HAs, we can either make the output dependent on only the input batch, or require that the relevant state does not change in state transitions.

Based on the abstract specification, we define the *functional correctness* of an HA in terms of the output batches that are produced for given input batches as follows.

Definition 5 (Functional Correctness). An HA Acc is functionally correct with respect to an abstract specification $Spec$ if, for all concrete initial states $s_0 \in S_{CI}$ and all sequences in of input batches, if

- $in = \langle in_1, \dots, in_n \rangle$,
- $s = StateSeq(in, s_0)$,
- $s_I = initsym(s) = \langle s_{I,1}, \dots, s_{I,n} \rangle$,
- $o = out(final(s)) = \langle o_1, \dots, o_n \rangle$,

then $\forall j \in [1 \dots b]. o_n(j) = Spec(in_n(j), rel(s_{I,n}))$.

A bug is simply a failure of functional correctness.

As mentioned above, even without a formal specification, we can apply the core technique of A-QED. To do so, we leverage the concept of *functional consistency*, the notion that under modest assumptions, two identical inputs will always produce the same outputs.

Definition 6 (Functional Consistency (FC)). An HA Acc is functionally consistent if, for all concrete initial states $s_0 \in S_{CI}$ and for all sequences in of input batches, if

- $in = \langle in_1, \dots, in_n \rangle$, $s = StateSeq(in, s_0)$,
- $s_I = initsym(s) = \langle s_{I,1}, \dots, s_{I,n} \rangle$,
- $o = out(final(s)) = \langle o_1, \dots, o_n \rangle$,

then $\forall i \in [1, n], j, j' \in [1, b]$.

$$in_i(j) = in_n(j') \wedge rel(s_{I,i}) = rel(s_{I,n}) \rightarrow o_i(j) = o_n(j').$$

Definition 6 illustrates the need for the *relevant* designation for memory states. It essentially says that two inputs, even if started at different times and in different batch positions, should produce the same output, as long as the relevant part of the memory is the same when the two inputs are sent in. The following lemma is straightforward (see the online appendix [53] for proofs of this and other results).

Lemma 1 (Soundness of FC). If an HA is functionally correct, then it is functionally consistent.

Checking FC requires running BMC over multiple iterations of the HA and may be computationally prohibitive for large designs or for large values of n . Often, it is possible to verify a stronger property, which only requires checking consistency across two runs of the HA.

Definition 7 (Strong FC). *An HA Acc is strongly functionally consistent if, for all reachable initial states s_0, s'_0 and input batches in, in' , if*

- $s = \text{StateSeq}(\langle in \rangle, s_0)$, $s' = \text{StateSeq}(\langle in' \rangle, s'_0)$,
- $s_F = \text{final}(s) = \langle s_F \rangle$, $s'_F = \text{final}(s') = \langle s'_F \rangle$,
- $o = \text{out}(s_F) = \langle o \rangle$, $o' = \text{out}(s'_F) = \langle o' \rangle$,

then $\forall j, j' \in [1, b]$.

$$in(j) = in'(j') \wedge rel(s_0) = rel(s'_0) \rightarrow o(j) = o'(j').$$

The main difference between FC and strong FC is that the initial states s_0 and s'_0 can be any reachable states. In contrast to that, the initial state $s_0 \in S_{CI}$ in the definition of FC is a concrete one. It is easy to see that strong FC implies FC, but the reverse is not true in general. This is because it may not be possible for two reachable initial states s_0 and s'_0 chosen in a strong FC check to both appear in a single sequence of states resulting from executing a sequence of input batches starting in a concrete initial state. Similar to previous work on A-QED for non-batch-mode HAs [8], FC checking relies on sequences of input batches to reach all reachable states from a concrete initial state. For strong FC checking, on the other hand, two individual input batches are sufficient because the two initial states s_0 and s'_0 can be arbitrarily chosen from the reachable states. Like FC, strong FC is a sound approach.

Lemma 2 (Soundness of Strong FC). *If an HA is functionally correct then it is strongly functionally consistent.*

A challenge with using strong FC is that it requires starting with reachable initial states. However, we found that in practice (cf., Section V), it is seldom necessary to add any constraints on the initial states. This may seem surprising given the well-known problem of spurious counterexamples that arises when using formal to prove functional correctness without properly constraining initial states. There are at least two reasons for this. First, many HAs have less dependence on internal state (none for non-interfering HAs) than other kinds of designs. But second, and more importantly, FC is a much more forgiving property than design-specific correctness. Many designs are functionally consistent, even when run from unreachable states. In fact, we believe that this is a natural outcome of good design and that designing for FC is a sweet spot in the trade-off between design for verification and other design goals. If designers take care to ensure FC, even from unreachable states, then strong FC is both sound and easy to formulate.

Even simpler versions of the checks above can be obtained by making them *intra-batch* checks. An HA is *intra-batch functionally consistent* if it is functionally consistent when $i = n = 1$. That is, intra-batch FC checks are based on sending a single input batch to the HA. Consequently, it is not necessary to identify and compare the relevant parts of the initial states (cf. Definition 6) as there is precisely one initial state being used. Similarly, an HA is *intra-batch strongly functionally consistent* if it is strongly functionally consistent when $s_0 = s'_0$ and $in = in'$. Again, only one input batch is sent to the HA and the relevant parts of the initial states are thus always equal. As we will show in Section V, intra-batch

checks can be a very effective approach for cheaply finding bugs. Intra-batch checks are applicable only to batch-mode HAs; i.e., they are not applicable in the context of A-QED targeted at HAs processing sequences of single inputs [8] rather than input batches.

While functional consistency alone can find many bugs, it becomes a complete technique (i.e., it finds all bugs) by combining it with *single-action checks*.

Definition 8 (Single-Action Correctness (SAC)). *An HA Acc is single-action correct (SAC) with respect to an abstract specification Spec if, for every batch element (a, d) and for every reachable relevant state s_r , there exists some reachable initial state s , such that $inp(s)(j) = (a, d)$ for some j , $rel(s) = s_r$, and $out(\text{final}(\mathbf{T}(s)))(j) = \text{Spec}((a, d), s_r)$.*

Essentially, SAC requires that for each action a , data d , and reachable relevant state s_r , we have checked that the result is computed correctly when starting from some reachable initial state s whose relevant state matches s_r . For every batch element (a, d) and s_r , it is sufficient to run a single check where we can choose (a, d) to be at any arbitrary position j in the batch $inp(s)$. Checking SAC *does* require using the specification explicitly, but these kinds of checks typically already exist in unit or regression tests. SAC may even be possible to verify using simulation. As we show in Section V, many bugs can be discovered without checking SAC at all.

When formalizing single-action checks, we again advocate using an over-approximation for reachability and encourage the design of HAs with simple over-approximations for the set of reachable relevant states. For the encryption example we gave above, the set of reachable relevant states is just the set of valid keys, which should be easy to specify.

In earlier work, using a slightly different HA model, we showed that SAC and functional consistency ensure correctness only when the HA is *strongly connected* (SC), that is, when there exists a sequence of state transitions from every reachable state to every other reachable state. The same is true here.

Lemma 3 (Completeness of SAC + FC + SC). *If an HA is strongly connected and single-action correct and has a bug, then it is not functionally consistent.*

However, strong functional consistency leads to an even stronger result.

Lemma 4 (Completeness of SAC + Strong FC). *If an HA is single-action correct and has a bug, then it is not strongly functionally consistent.*

Finally, to address timeliness of results in addition to correctness, we define a notion of *responsiveness* for our model.

Definition 9 (Responsiveness). *An HA is responsive with respect to bound n if, for all concrete initial states $s_0 \in S_{CI}$, sequences in of input batches, and input batches in , if*

- $s = \text{StateSeq}(in, s_0) = \langle s_0, \dots, s_m \rangle$ and
- $s' = \text{StateSeq}(in \cdot in, s_0) = \langle s_0, \dots, s_{m+l} \rangle$,

then $l \leq n$.

A. Decomposition for FC Checking

We now show how FC of a decomposed design can be derived from FC of its parts. We first give conditions under which two HAs can be composed.

Definition 10 (Functionally Composable). *Acc₁ and Acc₂ are functionally composable if: (i) $b_1 = b_2$; (ii) $O_1 = A_2 \times D_2$; (iii) $S_{C,1} \cap S_{C,2} = \emptyset$; (iv) $S_{R,1} = S_{R,2}$; and (v) $S_{N,1} = S_{Out,2} \times S'_{N,1}$ and $S_{N,2} = S_{In,1} \times S'_{N,2}$ for some $S'_{N,1}$.*

Note in particular that composability requires that the outputs of Acc_1 match the inputs of Acc_2 . We also require that the two HAs have isomorphic memory states, which is ensured by including $S_{Out,2}$ in the non-relevant states of Acc_1 and $S_{In,1}$ in the non-relevant states of Acc_2 . In order to map a memory state of Acc_1 to the corresponding memory state in Acc_2 , we define a mapping function $\alpha : S_{M,1} \rightarrow S_{M,2}$ as follows: $\alpha(s_m) = (out(s_m), nrel(s_m)(1), rel(s_m), (inp(s_m), nrel(s_m)(2)))$. We next define functional composition.

Definition 11 (Functional Composition, Sub-Accelerators). *Given functionally composable HAs Acc_1 and Acc_2 , we define the functional composition $Acc_0 = Acc_2 \circ Acc_1$ (Acc_1 and Acc_2 are called sub-accelerators of Acc_0) as follows: $b_0 = b_1$, $A_0 = A_1$, $D_0 = D_1$, $O_0 = O_2$, $S_{C,0} = S_{C,1} \cup S_{C,2}$, $S_{M,0} = S_{M,1}$, $s_{c,I,0} = s_{c,I,1}$, $s_{c,F,0} = s_{c,F,2}$, $S_{m,I,0} = S_{m,I,1}$. The transition function is defined as follows. $T_0(s_c, s_m) =$*

- (i) if $s_c \in S_{C,1}$ and $s_c \neq s_{c,F,1}$ then $T_1(s_c, s_m)$;
- (ii) if $s_c \in S_{C,2}$ then $T_2(s_c, \alpha(s_m))$; and
- (iii) if $s_c = s_{c,F,1}$ then $(s_{c,I,2}, \alpha(s_m))$.

Definition 11 essentially states that an execution of $Acc_0 = Acc_2 \circ Acc_1$ is obtained by first running Acc_1 to completion, then passing the outputs of Acc_1 to the inputs of Acc_2 , and then running Acc_2 to completion. As a variant of Definition 11, it is also possible to define functional composition where the sub-accelerators operate in parallel. This way, the sub-accelerators process non-overlapping parts of a given input batch and produce the respective non-overlapping parts of the output batch.

We now introduce a compositional version of FC.

Definition 12 (Strong FC for Decomposition (FCD)). *An HA Acc is strongly functionally consistent for decomposition (strongly FCD) if it is strongly functionally consistent and, in addition to $o(j) = o'(j')$, the property $rel(s_F) = rel(s'_F)$ holds in the conclusion of the implication in Definition 7.*

Note that strong FCD is stronger than strong FC. In order to stitch together results on sub-accelerators, we need to establish that not only the output but also the relevant memory state is the same after processing identical inputs. The following is clear from the definition.

Corollary 1. *If an HA Acc is strongly FCD, then Acc is strongly FC.*

We now show that composition preserves strong FCD and then state our main result.

Lemma 5 (Functional Composition and Strong FCD). *Let $Acc_0 = Acc_2 \circ Acc_1$. If both Acc_1 and Acc_2 are strongly FCD then Acc_0 is strongly FCD.*

Theorem 1 (Completeness of A-QED²). *Let Acc_0, Acc_1 , and Acc_2 be HAs such that $Acc_0 = Acc_2 \circ Acc_1$ and Acc_0 is single-action correct. If Acc_1 and Acc_2 are strongly FCD then Acc_0 is functionally correct.*

Theorem 1 states that A-QED² is complete. That is, by contraposition, if an HA Acc_0 has a bug, i.e., it is not functionally correct, then either Acc_1 or Acc_2 is not strongly FCD, and thus the bug can be detected by A-QED².

Note that there is no corresponding soundness result. This is because it is possible to decompose a functionally consistent HA into functionally inconsistent sub-accelerators. However, as shown in Section V, this appears to be rare in practice, and here again we reiterate our position on design for verification and advocate that also sub-accelerators should be designed with functional consistency in mind.

Functional composition can easily be generalized to more than two sub-accelerators. Moreover, it can be applied recursively to further decompose sub-accelerators. If functional decomposition based on Definition 11 is not applicable to further decompose a sub-accelerator, then such a sub-accelerator can be decomposed using existing formal decomposition approaches, though these require significant manual effort. Our approach identifies conditions under which simple, automatable decomposition of FC checking is possible.

IV. A-QED² FUNCTIONAL DECOMPOSITION IN PRACTICE

We now present our implementation of A-QED², which builds on the theoretical framework of the previous section. We combine functional decomposition with checks for FC (dFC), SAC (dSAC), and responsiveness (dRB).

A. Decomposition for FC: dFC

dFC takes as input a non-interfering LCA design Acc (satisfying Definitions 1 and 2) together with designer-provided annotations (explained in this section). dFC decomposes Acc into sub-accelerators (following Definition 11). FC checks are run on the sub-accelerators and any counterexamples are reported. Note that the way in which Acc is actually decomposed into sub-accelerators has no influence on the completeness of A-QED² (Theorem 1). That said, FC checks may scale better for certain decompositions. While failing FC checks expose consistency issues at the sub-accelerator level, it is possible that they do not cause incorrect behaviors at the full Acc level. However, we did not observe any instances of this in our experiments.

Our dFC implementation relies on identifying *batch operations* in a given Acc . A batch operation operates on a vector of inputs, applying some action to each input in order to produce a vector of outputs. The input to a batch operation could be an intermediate output batch of another sub-accelerator or an input batch to Acc itself. A batch operation produces either an

intermediate output batch which is subsequently processed by another sub-accelerator or an output batch of *Acc* itself.

We assume that *Acc* is expressed in a high-level language, specifically as a C/C++ program¹ that implements sequential computation of *Acc* outputs from *Acc* inputs.² Batch operations in the C/C++ program are identified by finding contiguous C/C++ statements called *functional blocks* that implement those batch operations. Each functional block represents a sub-accelerator.

We have developed a set of annotations by which the designer can help identify these functional blocks. Examples of such annotations are given in Listing 2 (extends Listing 1). It has two functional blocks corresponding to batch operations: lines 15-17 and 32-33.

Annotations are defined by particular keywords that are prefixed by “%” (and denoted in blue) in Listing 2. These annotations describe the compute and memory access patterns of the functional block as it transforms an input batch into an output batch. In practice, hardware designers already use similar annotations frequently, e.g., to express parallelization opportunities for HLS to generate efficient hardware. As a result, we expect manageable effort in creating such annotations to support dFC. The HLS research community is actively developing new techniques to automatically explore the HA design space and derive optimal design points together with appropriate parallelization and pipelining [54]–[56]. With tight integration of A-QED² with HLS, we expect that it will be possible to generate dFC annotations with low effort.

Listing 2: C/C++ Annotation Example (AES Encryption)

```

1  #define BS ((1) << 12) // BUF_SIZE
2  #define UF 2 // UNROLL_FACTOR
3  #define US BS/UF // UNROLL_SIZE
4
5  void fun(int data[BS], int buf[UF][US], int key[2]){
6      int j, k;
7
8      %IN_SIZE 16 // variables per input batch element
9      %IN_BATCH_SIZE BS/IN_SIZE // input batch size
10     %BATCH_MEM_IN data // input batch source
11     %IN_ALLOC_RULE in(x) addr range =
12         [i*BS + x*IN_SIZE :
13          i*BS + (x + 1)*IN_SIZE] // BATCH_MEM_IN layout
14     // ===ACC1 START===
15     for(j=0; j<UF; j++){
16         for(k=0; k<BS/UF; k++){
17             buf[j][k] = *(data + i*BS + j*US + k)^key[0];
18         }
19     }
20     // ===ACC1 END===
21     %OUT_SIZE 16 // variables per output batch element
22     %OUT_BATCH_SIZE BS/OUT_SIZE // output batch size
23     %BATCH_MEM_OUT buf // output batch source
24     %IN_ALLOC_RULE out(x) addr range =
25         [x/US][(x%US)*OUT_SIZE :
26          ((x + 1)%US)*OUT_SIZE] // BATCH_MEM_OUT layout
27
28     %IN_SIZE 16
29     %IN_BATCH_SIZE BS/IN_SIZE
30     %BATCH_MEM_IN buf
31     %IN_ALLOC_RULE in(x) addr range =
32         [(x%US)*IN_SIZE : ((x+1)%US)*IN_SIZE][x/US]
33 }

```

¹HAs expressed in Verilog or SystemC can be converted into C/C++, and then our dFC implementation can be applied. We do this in Sec. V.

²Existing HLS tools (e.g., Xilinx Vivado HLS, Mentor Catapult HLS) can then optimize *Acc*, incorporate appropriate pipelining and parallelism, and produce Verilog for subsequent logic synthesis and physical design steps. Such HLS-based HA design flows are becoming increasingly common.

```

31 // ===ACC START===
32 for(j=0; j<UF; j++){
33     aes256_encrypt(local_key[j], buf[j]);
34 }
35 // ===ACC END===
36 %OUT_SIZE 16
37 %OUT_BATCH_SIZE BS/OUT_SIZE
38 %BATCH_MEM_OUT buf
39 %OUT_ALLOC_RULE out(x) addr range =
40     [(x%US)*OUT_SIZE : ((x+1)%US)*OUT_SIZE][x/US]

```

From the annotations, we create sub-accelerators. For example, the annotations in Listing 2 generate two sub-accelerators: *Acc*₁ corresponding to the functional block in Lines 15-17 with annotations in Lines 8-13 and 19-24, and *Acc*₂ corresponding to the functional block in Lines 32-33 with annotations in Lines 26-30 and 35-39. For each sub-accelerator, we create an A-QED² module for FC checking.³ It generates symbolic inputs for the sub-accelerator and symbolically executes the corresponding functional block in order to produce symbolic expressions for the outputs. For strong FC checks (Definitions 6 and 7), the relevant states (Definition 1) must additionally be identified and explicitly constrained to be consistent across sub-accelerator calls processing two input batches. Identifying the relevant states is not necessary for intra-batch FC checks (discussed in the context of Lemma 2). For example, in sub-accelerator *Acc*₁ in Listing 2, *key*[0] is a relevant state element (distinct from the batch input *data*). Between two calls of *Acc*₁ during a strong FC check, *key*[0] must be consistent. In our implementation, we ignore reachability and allow all checks to start from fully symbolic initial states. This does not lead to spurious counterexamples in our experiments.

B. Decomposition for RB: dRB

The sub-accelerators for A-QED²’s RB checks (Definition 9) can be (and often are) different from those for FC because RB involves a much simpler check: *some* output is produced within the response bound *n*. We expect *n* to be provided by the designer for the top-level accelerator. We then use the same bound *n* for each sub-accelerator. The rationale is that if a sub-accelerator fails an RB check, then the full accelerator would also fail the same RB check.

For dRB, we generate a static single assignment (SSA) representation of the design. We then apply a *sliding window algorithm* to dynamically generate sub-accelerators. Lines of code in the SSA that fall within a certain *window W* form the sub-accelerator. Due to SSA form, the inputs of this sub-accelerator are variables that are never updated or assigned in *W* while the outputs are the variables which update variables outside *W*. The current size of *W* is given by the number of LOCs that fit in *W*, and it changes dynamically during a run of the algorithm to incorporate the largest sub-accelerator that will fit the BMC tool. Once the sub-accelerator is verified, *W* slides by δ LOCs (δ is a parameter) and adjusts its boundary to get the next largest sub-accelerator that can be verified. We synthesize that sub-accelerator using HLS (since some responsiveness bugs only manifest after HLS) and then run RB checks using BMC. The initial states of each generated

³See the online appendix [53] for details.

sub-accelerator are left unconstrained (i.e., fully symbolic) in order to analyze all possible behaviors. The specific size of W and its position in the SSA code change dynamically as dRB proceeds. dRB terminates when W reaches the end of the SSA code or if at any time an RB check fails.

C. Decomposition for SAC: dSAC

As mentioned above, and as will be shown in the next section, many bugs can be detected using only dFC and dRB. The advantage of this is that both of these checks can be run without any functional specification. dSAC completes the story, but at the cost of requiring specifications. We use standard functional decomposition techniques (essentially, writing preconditions, invariants, and postconditions) to decompose SAC checks. One feature of dSAC is that only a single input in a batch needs to be checked—all other inputs in the batch can be set to constants (we use zero in our experiments). This makes both writing the properties and checking them much simpler. The non-input part of the initial state for each check is again kept fully symbolic for simplicity. If a sub-accelerator is too big, we further decompose it using finer-grained functional blocks.

V. EXPERIMENTAL RESULTS

We demonstrate the practicality and effectiveness of A-QED² for 109 (buggy) versions of several non-interfering LCAs,⁴ including open-source industrial designs [12]. We selected these designs for the following reasons:

- They cover a wide variety of HAs (neural nets, image processing, natural language processing, security). Most are too large for existing off-the-shelf formal tools.
- They have been thoroughly verified (painstakingly) using state-of-the-art simulation-based verification techniques. Thus, we can quantify the thoroughness of A-QED².
- With access to buggy versions, we did not have to artificially inject bugs. Bugs we encountered include incorrect initialization, incorrect memory accesses, incorrect array indexing, and unresponsiveness in HLS-generated designs.

Many of the designs were already available in sequential C or C++. We converted Verilog and SystemC designs into sequential C. To facilitate dFC, we manually inserted annotations (like those in Listing 2). For A-QED FC, we used CBMC for all designs originally represented in sequential C or C++. For designs in Verilog and SystemC, we used Cadence JasperGold (SystemC designs converted to Verilog via HLS). For A-QED² FC and SAC checks, we used CBMC version 5.10 [66]. For A-QED and A-QED² RB checks, we used Cadence JasperGold version 2016.09p002 on Verilog designs generated by the HLS tools used by the designers. Lastly, we used Frama-C [67] to check for initialization and out-of-bounds bugs on the entire C/C++ designs. We ran all our experiments on Intel Xeon E5-2640 v3 with 128GB of DRAM.

Tables I, II, and III summarize our results. We present comparisons between A-QED² (dFC, dRB, dSAC) and A-QED

(FC, RB, SAC). Table I also compares A-QED² intra-batch FC vs. A-QED² strong FC (cf. details in the online appendix [53]).

Observation 1: HAs from various domains (including industry) show that non-interfering LCAs are highly common.

Observation 2: The vast majority of the studied HAs are too big for existing off-the-shelf formal verification tools, for both A-QED and conventional formal property verification.

Observation 3: Table I shows that A-QED² intra-batch FC checks detected bugs inside sub-accelerators (with batch sizes > 1) very quickly—under a minute for almost all of the designs, and just over a minute for `nv_large`. For most batch-mode sub-accelerators—except two for each of the following four designs (amounting to eight sub-accelerators in total): `grayscale64`, `grayscale32`, `mean128`, and `mean32`—intra-batch dFC checks were easily completed using off-the-shelf formal tools. Strong FC checks incur more complexity. Hence, the formal tool timed out after 12 hours for 62 sub-accelerators when running strong FC checks, distributed across multiple designs. Empirically, we found that intra-batch FC checks detected all bugs that were detected by strong FC checks.

Observation 4: A-QED² RB and A-QED² SAC are also highly effective in detecting bugs inside sub-accelerators. For the first 11 designs (`AES` to `gsm`) in Table II, we do not expect unresponsiveness bugs (confirmed by simulations). Hence, A-QED² RB checks ran for 12 hours (for increasingly longer input sequences) without detecting unresponsiveness. For designs with RB bugs, A-QED² RB checks on sub-accelerators were able to detect those in less than 11 minutes on average. For A-QED² dSAC, we observed that a significant fraction (26 out of 46 bugs (56%)) of these bugs were also detected by A-QED² FC checks. Thus, FC alone is effective at catching a wide variety of bugs.

Observation 5: A-QED² detected all bugs that were detected by conventional (simulation-based) verification techniques. Further, all counterexamples produced from verifying sub-accelerators corresponded to real accelerator-level bugs. Compared with traditional simulation-based verification, we report a $\sim 5X$ improvement in verification effort on the average, with a $\sim 9X$ improvement for the large, industrial NVDLA designs. The overhead of inserting our annotations for dFC can be small compared to what designers already insert to optimize the design. For ISmartDNN, for example, the total number of annotations is 304, which is 2.8% of the total lines of code of the design. In the code of the HLS designs we considered, pragmas amount to 11% on average. We also observe a $\sim 60X$ improvement in average verification runtime compared to conventional simulations.⁵

VI. CONCLUSION

Our theoretical and experimental results demonstrate that A-QED² is an effective and practical approach for verification

⁴See the online appendix [53] for design details and the software artifact [65].

⁵The conventional verification effort for NVDLA was based on start and end commit dates in its `nv_small` Github repository. The conventional verification runtime for NVDLA, ISmartDNN, and dnn HAs were obtained by running the available simulation tests on our platform. The remaining runtime and effort information were provided by the designers.

Design (#Gates) (#Versions)			A-QED FC	A-QED ² dFC: Intra-batch FC			A-QED ² dFC: Strong FC		
94 versions in table, 15 in caption [†]			Avg. RT (min)	Avg. RT (min)	#Bugs	#Sub-Acc.(T/P/C/B)	Avg. Runtime (min)	#Bugs	#Sub-Acc.(T/P/C/B)
AES [50]	(382k)	(4)	OOM	0.97	4	8 / 7 / 7 / 4	timeout	0	8 / 7 / 2 / 0
ISmartDNN [57]	(42M)	(3)	timeout	0.10	2	38 / 5 / 5 / 2	0.18	2	38 / 5 / 2 / 2
grayscale128 [33]	(351k)	(5)	timeout	0.03	3	3 / 3 / 2 / 2	0.07	3	3 / 3 / 2 / 2
grayscale64 [33]	(194k)	(5)	timeout	0.02	3	3 / 3 / 2 / 2	0.02	3	3 / 3 / 2 / 2
grayscale32 [33]	(106k)	(5)	8.20	<0.01	5	3 / 3 / 3 / 3	0.30	5	3 / 3 / 3 / 3
mean128 [33]	(202k)	(5)	timeout	0.35	3	3 / 3 / 2 / 2	0.17	3	3 / 3 / 2 / 2
mean64 [33]	(104k)	(5)	timeout	0.38	3	3 / 3 / 2 / 2	0.13	3	3 / 3 / 2 / 2
mean32 [33]	(54k)	(5)	5.53	0.17	5	3 / 3 / 3 / 3	0.33	5	3 / 3 / 3 / 3
dnn [58]	(2M)	(11)	timeout	0.03	5	34 / 14 / 14 / 5	0.13	5	34 / 14 / 8 / 5
nv_large [12]	(16M)	(23)	timeout	1.17	11	89 / 46 / 46 / 11	2.93	9	89 / 46 / 21 / 9
nv_small [12]	(1M)	(23)	timeout	0.07	11	89 / 46 / 46 / 11	1.03	11	89 / 46 / 26 / 11

TABLE I: **Avg. RunTimes** of FC checks for A-QED and A-QED². For A-QED², sub-accelerator counts are provided, including the Total count that resulted from dFC decomposition, the count with batch sizes greater than one (i.e., Parallel), the count (with batch sizes greater than one) for which FC checks were successful on 1 and 2 batches for intra-batch FC and strong FC respectively, and the count for which Bugs were detected by FC checks. For A-QED FC, experiments could not complete FC check for a single batch in 12 hours (**timeout**) or exhibited out-of-memory (**OOM**) errors before timeout. **Average** runtimes result from dividing the time to detect all bugs by the number of bugs. [†]**keypair** [59], **gsm** [60], **HLSCNN** [61], **FlexNLP** [62], **Dataflow** [63], and **Opticalflow** [64] all time out for A-QED FC and do not contain any sub-accelerators with batch size greater than one. One OOB bug was detected in **gsm** and one initialization bug in **keypair**.

Design (#Gates) (#Versions)			A-QED RB	A-QED ² dRB		
Total Versions = 109			Avg. RT (min)	Avg. RT (min)	#Bugs	#Sub-Acc. (T/C/B)
AES [50]	(382k)	(4)	timeout			13 / 13 / 0
ISmartDNN [57]	(42M)	(3)	timeout	No RB bug detected up to input sequence length 11 and 24 depending on the design		
grayscale128 [33]	(351k)	(5)	timeout			
grayscale64 [33]	(194k)	(5)	timeout			
grayscale32 [33]	(106k)	(5)	timeout			
mean128 [33]	(202k)	(5)	timeout			
mean64 [33]	(104k)	(5)	timeout			
mean32 [33]	(54k)	(5)	timeout			
dnn [58]	(2M)	(11)	timeout			
keypair [59]	(>200M)	(1)	timeout			
gsm [60]	(8.8k)	(1)	timeout			
nv_large [12]	(16M)	(23)	timeout	No RB bugs expected		
nv_small [12]	(1M)	(23)	timeout			
HLSCNN [61]	(323k)	(2)	timeout	2.33	1	25 / 25 / 1
FlexNLP [62]	(567k)	(9)	timeout	10.77	9	15 / 15 / 9
Dataflow [63]	(296k)	(1)	0.45	0.25	1	9 / 9 / 1
Opticalflow [64]	(555k)	(1)	timeout	0.17	1	3 / 3 / 1

TABLE II: RB checks for A-QED and A-QED². For A-QED², sub-accelerator counts produced by dFC are provided, as in Table I. A-QED² RB checks are performed on all sub-accelerators regardless of batch size, so **P** is omitted compared to Table I. For A-QED RB, RB checks did not complete even for a input sequence length of 1 within 12 hours (**timeout**). Sub-accelerators for which RB checks for at least input sequence length of 1 was completed were considered Complete. For the first 11 designs, from **AES** to **gsm**, no bugs related to unresponsiveness were detected by traditional simulation-based verification. Results are omitted for **nv_large** and **nv_small**; responsiveness related bugs generally result from parallelism and pipelining, both of which were lost in our manual translation of NVDLA from Verilog to sequential C code.

of large non-interfering LCAs. A-QED² exploits A-QED principles to decompose a given HA design into sub-accelerators such that A-QED can be naturally applied to the sub-accelerators. A-QED² is especially attractive for HLS-based HA design flows. A-QED² creates several promising research directions:

- Extension of our A-QED² experiments to include interfering LCAs (already covered by our theoretical results).
- Automation of dFC annotations via HLS techniques.
- dFC approaches beyond our current implementation.

Design (#Gates) (#Versions)			A-QED ² dSAC			
Total Versions = 109			Avg. RT (min)	#Bugs	Bug overlap with dFC	#Sub-Acc. (T/C/B)
AES [50]	(382k)	(4)	0.12	0	0	8 / 8 / 0
ISmartDNN [57]	(42M)	(3)	0.22	3	2	38 / 38 / 3
grayscale128 [33]	(351k)	(5)	0.04	2	2	3 / 2 / 2
grayscale64 [33]	(194k)	(5)	0.01	2	2	3 / 2 / 2
grayscale32 [33]	(106k)	(5)	<0.01	2	2	3 / 3 / 2
mean128 [33]	(202k)	(5)	0.21	2	2	3 / 2 / 2
mean64 [33]	(104k)	(5)	<0.01	2	2	3 / 2 / 2
mean32 [33]	(54k)	(5)	<0.01	2	2	3 / 3 / 2
dnn [58]	(2M)	(11)	0.01	6	0	34 / 14 / 6
keypair [59]	(>200M)	(1)	timeout	0	0	14 / 14 / 0
gsm [60]	(8.8k)	(1)	timeout	0	0	5 / 5 / 0
nv_large [12]	(16M)	(23)	0.84	12	6	89 / 89 / 12
nv_small [12]	(1M)	(23)	0.11	12	6	89 / 50 / 12
HLSCNN [61]	(323k)	(2)	0.45	1	0	25 / 11 / 1
FlexNLP [62]	(567k)	(9)	timeout	0	0	21 / 21 / 0
Dataflow [63]	(296k)	(1)	timeout	0	0	8 / 8 / 0
Opticalflow [64]	(555k)	(1)	timeout	0	0	14 / 14 / 0

TABLE III: SAC checks for A-QED². Sub-accelerator counts produced by dSAC are provided, as in Table I. A-QED² SAC checks were performed on all sub-accelerators regardless of batch size, so **P** is omitted compared to Table I.

- Further A-QED² scalability using abstraction.
- Extension of A-QED² beyond sequential (C/C++) code to include concurrent programs.
- Effectiveness of A-QED² for RTL designs (without converting them to sequential C/C++).
- Applicability of A-QED² beyond functional bugs (e.g., to detect security vulnerabilities in HAs).
- Comparison of A-QED² and conventional decomposition.
- Identifying conditions under which A-QED² is sound.

ACKNOWLEDGMENT


This work was supported by the DARPA POSH program (grant FA8650-18-2-7854), NSF (grant A#:1764000), and the Stanford SystemX Alliance. We thank Prof. David Brooks, Thierry Tambe and Prof. Gu-Yeon Wei from Harvard University, and Kartik Prabhu and Prof. Priyanka Raina from Stanford University for their design contributions in our experiments.

REFERENCES

- [1] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-rich architectures: Opportunities and progresses," in *Proc. DAC*. IEEE, 2014, pp. 1–6.
- [2] L. P. Carloni, "The Case for Embedded Scalable Platforms," in *Proc. DAC*. IEEE, 2016, pp. 1–6.
- [3] W. J. Dally, Y. Turakhia, and S. Han, "Domain-Specific Hardware Accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.
- [4] M. Hill and V. J. Reddi, "Accelerator-level Parallelism," *CoRR*, vol. abs/1907.02064, 2019, <https://arxiv.org/abs/1907.02064>.
- [5] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The Design Process for Google's Training Chips: TPUv2 and TPUv3," *IEEE Micro*, 2021.
- [6] H. D. Foster, "Trends in functional verification: a 2014 industry study," in *Proc. DAC*. ACM, 2015, pp. 48:1–48:6.
- [7] B. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification," *ACM Trans. Design Autom. Electr. Syst.*, vol. 24, no. 1, pp. 10:1–10:24, 2019.
- [8] E. Singh, F. Lonsing, S. Chattopadhyay, M. Strange, P. Wei, X. Zhang, Y. Zhou, D. Chen, J. Cong, P. Raina, Z. Zhang, C. W. Barrett, and S. Mitra, "A-QED Verification of Hardware Accelerators," in *Proc. DAC*. IEEE, 2020, pp. 1–6.
- [9] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "An Analysis of Accelerator Coupling in Heterogeneous Architectures," in *Proc. DAC*. ACM, 2015, pp. 202:1–202:6.
- [10] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," in *Proc. DAC*. ACM, 2012, pp. 843–849.
- [11] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.
- [12] NVIDIA, "NVIDIA Deep Learning Accelerator," <http://nvidia.org/primer.html>, 2021, [Online]. Accessed: August 2021.
- [13] K. A. Campbell, D. Lin, L. He, L. Yang, S. T. Gurumani, K. Rupnow, S. Mitra, and D. Chen, "Hybrid Quick Error Detection: Validation and Debug of SoCs Through High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 7, pp. 1345–1358, 2019.
- [14] Y. Chi, Y. Choi, J. Cong, and J. Wang, "Rapid Cycle-Accurate Simulator for High-Level Synthesis," in *Proc. FPGA*. ACM, 2019, pp. 178–183.
- [15] IEEE, "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2017*, pp. 1–472, 2017.
- [16] Y. Choi, Y. Chi, J. Wang, and J. Cong, "FLASH: Fast, Parallel, and Accurate Simulator for HLS," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4828–4841, 2020.
- [17] S. Dai, A. Klinefelter, H. Ren, R. Venkatesan, B. Keller, N. R. Pinckney, and B. Khailany, "Verifying High-Level Latency-Insensitive Designs with Formal Model Checking," *CoRR*, vol. abs/2102.06326, 2021. [Online]. Available: <https://arxiv.org/abs/2102.06326>
- [18] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.
- [19] D. Giannakopoulou, K. S. Namjoshi, and C. S. Pasareanu, "Compositional Reasoning," in *Handbook of Model Checking*. Springer, 2018, pp. 345–383.
- [20] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh, "Assume-Guarantee Verification of Source Code with Design-Level Assumptions," in *Proc. ICSE*. IEEE Computer Society, 2004, pp. 211–220.
- [21] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu, "Learning Assumptions for Compositional Verification," in *Proc. TACAS*, ser. LNCS, vol. 2619. Springer, 2003, pp. 331–346.
- [22] A. Gupta, K. L. McMillan, and Z. Fu, "Automated assumption generation for compositional verification," *Formal Methods in System Design*, vol. 32, no. 3, pp. 285–301, 2008.
- [23] R. Jhala and K. L. McMillan, "Microarchitecture Verification by Compositional Model Checking," in *Proc. CAV*, ser. LNCS, vol. 2102. Springer, 2001, pp. 396–410.
- [24] C. Y. Cho, V. D'Silva, and D. Song, "BLITZ: Compositional bounded model checking for real-world programs," in *Proc. ASE*. IEEE, 2013, pp. 136–146.
- [25] H. Koo and P. Mishra, "Functional test generation using design and property decomposition techniques," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 4, pp. 32:1–32:33, 2009.
- [26] R. B. Jones, C. H. Seger, and D. L. Dill, "Self-Consistency Checking," in *Proc. FMCAD*, ser. LNCS, vol. 1166. Springer, 1996, pp. 159–171.
- [27] S. Katz, O. Grumberg, and D. Geist, "'Have I written enough Properties?'" - A Method of Comparison between Specification and Implementation," in *Proc. CHARME*, ser. LNCS, vol. 1703. Springer, 1999, pp. 280–297.
- [28] K. Claessen, "A Coverage Analysis for Safety Property Lists," in *Proc. FMCAD*. IEEE, 2007, pp. 139–145.
- [29] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage Metrics for Temporal Logic Model Checking," in *Proc. TACAS*, ser. LNCS, vol. 2031. Springer, 2001, pp. 528–542.
- [30] D. Große, U. Kühne, and R. Drechsler, "Estimating functional coverage in bounded model checking," in *Proc. DATE*. EDA Consortium, San Jose, CA, USA, 2007, pp. 1176–1181.
- [31] H. Chockler, D. Kroening, and M. Purandare, "Coverage in interpolation-based model checking," in *Proc. DAC*. ACM, 2010, pp. 182–187.
- [32] 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.
- [33] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni, "KAIRIS: Incremental Verification in High-Level Synthesis through Latency-Insensitive Design," in *Proc. FMCAD*. IEEE, 2019, pp. 105–109.
- [34] U. Kühne, S. Beyer, J. Bormann, and J. Barstow, "Automated formal verification of processors based on architectural models," in *Proc. FMCAD*. IEEE, 2010, pp. 129–136.
- [35] M. Soeken, U. Kühne, M. Freibothe, G. Fey, and R. Drechsler, "Automatic property generation for the formal verification of bus bridges," in *Proc. DDECS*. IEEE, 2011, pp. 417–422.
- [36] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke, "Advanced verification by automatic property generation," *IET Comput. Digit. Tech.*, vol. 3, no. 4, pp. 338–353, 2009.
- [37] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proc. TACAS*, ser. LNCS, vol. 1579. Springer, 1999, pp. 193–207.
- [38] D. Lin, E. Singh, C. Barrett, and S. Mitra, "A structured approach to post-silicon validation and debug using symbolic quick error detection," in *Proc. ITC*. IEEE, 2015, pp. 1–10.
- [39] E. Singh, D. Lin, C. Barrett, and S. Mitra, "Logic Bug Detection and Localization Using Symbolic Quick Error Detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018.
- [40] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. R. Fadiheh, D. Stoffel, W. Kunz, C. W. Barrett, W. Ecker, and S. Mitra, "Symbolic QED Pre-Silicon Verification for Automotive Microcontroller Cores: Industrial Case Study," in *Proc. DATE*. IEEE, 2019, pp. 1000–1005.
- [41] F. Lonsing, K. Ganesan, M. Mann, S. S. Nuthakki, E. Singh, M. Srouji, Y. Yang, S. Mitra, and C. W. Barrett, "Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED: Invited Paper," in *Proc. ICCAD*. ACM, 2019, pp. 1–8.
- [42] 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 *Proc. DATE*. IEEE, 2018, pp. 55–60.
- [43] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. W. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz, "Gap-free Processor Verification by S²-QED and Property Generation," in *Proc. DATE*. IEEE, 2020, pp. 526–531.
- [44] M. R. Fadiheh, D. Stoffel, C. W. Barrett, S. Mitra, and W. Kunz, "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking," in *Proc. DATE*. IEEE, 2019, pp. 994–999.
- [45] 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 *Proc. DAC*. IEEE, 2020, pp. 1–6.
- [46] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *Proc. CSFW-17*. IEEE, 2004, pp. 100–114.
- [47] G. Barthe, J. M. Crespo, and C. Kunz, "Relational Verification Using Product Programs," in *Proc. FM*, ser. LNCS, vol. 6664. Springer, 2011, pp. 200–214.

- [48] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying Constant-Time Implementations,” in *Proc. USENIX*. USENIX Association, 2016, pp. 53–70.
- [49] W. Yang, Y. Vizel, P. Subramanyan, A. Gupta, and S. Malik, “Lazy Self-composition for Security Verification,” in *Proc. CAV*, ser. LNCS, vol. 10982. Springer, 2018, pp. 136–156.
- [50] J. Cong, P. Wei, C. H. Yu, and P. Zhou, “Bandwidth optimization through on-chip memory restructuring for HLS,” in *Proc. DAC*. IEEE, 2017, pp. 1–6.
- [51] R. M. Keller, “Formal Verification of Parallel Programs,” *Commun. ACM*, vol. 19, no. 7, pp. 371–384, 1976.
- [52] —, “A Fundamental Theorem of Asynchronous Parallel Computation,” in *Parallel Processing, Proc. Sagamore Computer Conference*, ser. LNCS, vol. 24. Springer, 1974, pp. 102–112.
- [53] S. Chattopadhyay, F. Lonsing, L. Piccolboni, D. Soni, P. Wei, X. Zhang, Y. Zhou, L. Carloni, D. Chen, J. Cong, R. Karri, Z. Zhang, C. Trippel, C. Barrett, and S. Mitra, “Scaling Up Hardware Accelerator Verification using A-QED with Functional Decomposition,” *CoRR*, vol. abs/2108.06081, 2021, FMCAD 2021 proceedings version with appendix. [Online]. Available: <https://arxiv.org/abs/2108.06081>
- [54] S. Wang, Y. Liang, and W. Zhang, “FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs,” in *Proc. DAC*. ACM, 2017, pp. 27:1–27:6.
- [55] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, “COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications,” in *Proc. ICCAD*. IEEE, 2017, pp. 430–437.
- [56] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, “Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators,” in *Proc. DAC*. ACM, 2016, pp. 136:1–136:6.
- [57] X. Zhang, H. Lu, C. Hao, J. Li, B. Cheng, Y. Li, K. Rupnow, J. Xiong, T. S. Huang, H. Shi, W. Hwu, and D. Chen, “SkyNet: a Hardware-Efficient Method for Object Detection and Tracking on Embedded Systems,” in *Proc. MLSys*. mlsys.org, 2020.
- [58] M. Giordano, K. Prabhu, K. Koul, R. M. Radway, A. Gural, R. Doshi, Z. F. Khan, J. W. Kustin, T. Liu, G. B. Lopes, V. Turbiner, W.-S. Khwa, Y.-D. Chih, M.-F. Chang, G. Lallement, B. Murmann, S. Mitra, and P. Raina, “CHIMERA: A 0.92 TOPS, 2.2 TOPS/W Edge AI Accelerator with 2 MByte On-Chip Foundry Resistive RAM for Efficient Training and Inference,” in *Proc. VLSI*. IEEE, 2021, pp. 1–2.
- [59] K. Basu, D. Soni, M. Nabeel, and R. Karri, “NIST Post-Quantum Cryptography- A Hardware Evaluation Study,” IACR Cryptology ePrint Archive, Report 2019/047, 2019, <https://eprint.iacr.org/2019/047>.
- [60] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, “Chstone: A benchmark program suite for practical c-based high-level synthesis,” in *Proc. ISCAS*. IEEE, 2008, pp. 1192–1195.
- [61] P. N. Whatmough, S. K. Lee, M. Donato, H. Hsueh, S. L. Xi, U. Gupta, L. Pentecost, G. G. Ko, D. M. Brooks, and G. Wei, “A 16nm 25mm² SoC with a 54.5x Flexibility-Efficiency Range from Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators,” in *Proc. VLSI*. IEEE, 2019, p. 34.
- [62] T. Tambe, E. Yang, G. G. Ko, Y. Chai, C. Hooper, M. Donato, P. N. Whatmough, A. M. Rush, D. Brooks, and G. Wei, “A 25mm² SoC for IoT Devices with 18ms Noise-Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET,” in *Proc. ISSCC*. IEEE, 2021, pp. 158–160.
- [63] Y. Chi, Y. Choi, J. Cong, and J. Wang, “Rapid Cycle-Accurate Simulator for High-Level Synthesis,” in *Proc. FPGA*. ACM, 2019, pp. 178–183.
- [64] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. K. Srivastava, H. Jin, J. Featherston, Y. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, “Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs,” in *Proc. FPGA*. ACM, 2018, pp. 269–278.
- [65] “A-QED² Software Artifact,” 2021. [Online]. Available: <https://github.com/upscale-project/aqed-decomp-FMCAD2021/>
- [66] D. Kroening and M. Tautschnig, “CBMC - C bounded model checker - (competition contribution),” in *Proc. TACAS*, ser. LNCS, vol. 8413. Springer, 2014, pp. 389–391.
- [67] “Frama-C,” <https://frama-c.com/>, 2021, [Online]. Accessed: August 2021.

Sound and Automated Verification of Real-World RTL Multipliers

Mertcan Temel 
Electrical and Computer Engineering
University of Texas at Austin
 Austin, TX, USA
 mert@utexas.edu

Warren A. Hunt, Jr.
Computer Science
University of Texas at Austin
 Austin, TX, USA
 hunt@cs.utexas.edu

Abstract—We have developed an algorithm, S-C-Rewriting, that can automatically and very efficiently verify arithmetic modules with embedded multipliers. These include ALUs, dot-product, multiply-accumulate designs that may use Booth encoding, Wallace-trees, and various vector adders. Outputs of the target multiplier designs might be truncated, right-shifted, or a combination of both. We evaluate the performance of other state-of-the-art tools on verification problems beyond isolated multipliers and we show that our method applies to a broader range of design techniques encountered in real-world modules. Our verification software is verified using the ACL2 theorem prover, and we can soundly verify 1024x1024-bit isolated multipliers and similarly large dot-product designs in minutes. We can also generate counterexamples in case of a design bug. Our tool and benchmarks are available online.

Index Terms—Formal Verification, Integer Multipliers, Hardware Verification, Arithmetic Circuits, ACL2, Term-rewriting

I. INTRODUCTION

Integer multipliers are fundamental building blocks for general-purpose (e.g., CPUs and GPUs), image, communications, and cryptographic processors. Multipliers are used to implement dot-product, division, square-root, and floating-point operations; in turn, these operations find their way into graphics, cryptography, and signal processing systems. In some cases, such as cryptographic processors, integer multipliers might be used to multiply numbers as large as 1024 bits.

Given the ubiquity of multipliers, it is crucial to have a sound verification method for designs that include multipliers. However, the formal verification process of multipliers is still a challenge, especially for the most common design approaches such as Wallace tree and Booth encoding. Decision-procedure-based tools such as BDDs, SAT solvers do not scale [1], [2]. In recent years, multiplier verification efforts have shifted towards using computer algebra methods [2]–[6] and they have yielded more promising results. However, these studies focused heavily on isolated multiplier designs, and they do not perform well (if at all) for multipliers with truncated output (e.g., a 32x32-bit multiplier with a 32-bit output). Studies that explore the verification problem of embedded multipliers (e.g., multiply-accumulate, dot-product) have been limited, and they do not support designs with Wallace tree and Booth encoding [1]. Additionally, only one computer-algebra-based

tool [3] provides a system to check the correctness of the proof itself, leaving open the possibility that these tools might claim a design to be correct when the design is actually flawed.

In our previous work [7], we proposed a method to verify integer multipliers efficiently and automatically. Using the ACL2 theorem proving system, we developed a provably correct verification mechanism based on term-rewriting. This method has been shown to quickly verify a wide range of integer multiplier designs (e.g., 1024x1024-bit multipliers with simple partial products have been verified in less than 10 minutes). However, our focus concerned only untruncated isolated multiplier designs. Moreover, we did not discuss how the algorithm performs with buggy designs.

We have expanded our method and we have been able to:

- improve proof-time performance by a factor of 2 or more;
- verify designs beyond untruncated isolated multipliers;
- and quickly generate counterexamples.

Additionally, we retain the same level of proof automation and keep our tool provably correct.

In this paper, we aim to explore the verification problem of multipliers on more complex designs than explored in previous verification studies and deliver our solutions. We provide examples of complex multiplier architectures with optimizations that can be encountered in real-world designs. We discuss how existing state-of-the-art verification tools perform on such modules. Finally, we present our improved method and show that we can verify these complex designs very efficiently. For example, we can verify 64x64-bit isolated multipliers or similar designs within seconds and 1024x1024-bit isolated multipliers or similar dot-product designs in 5 minutes, no matter which design algorithm is used.

This paper is structured as follows. Sec. II summarizes the most common design algorithms for isolated and embedded multipliers. We show why it is important to develop a verification method for embedded and truncated multipliers and why it is not enough to have a verification tool only for isolated multipliers. In Sec. III, we summarize the related work from the most recent and/or prominent studies. Sec. IV recapitulates our term rewriting algorithm from our previous work and introduces some of its recently discovered limitations. Sec. V discusses our new improvements so that we can verify more designs with better efficiency and generate counterexamples

for buggy modules. Sec. VI describes how our lemmas are implemented and applied. Finally, we show our experiment results in Sec. VII and compare our performance with other state-of-the-art multiplier verification tools.

II. MULTIPLIER ARCHITECTURES

There are various algorithms to design RTL multipliers and integrate them in other arithmetic modules such as a multiply-accumulate (MAC). The difficulty of verifying these modules depends on the design algorithm. Some algorithms bring out clean and regularly structured modules, and some and most commonly used algorithms produce complex structures. This section elaborates on the verification problem by summarizing common algorithms to design multipliers and how they are implemented in other arithmetic circuits.

A. Isolated Multipliers

An isolated multiplier is a circuit with two bit-vector inputs and one bit-vector output. The output vector represents an integer equivalent to the multiplication of the input vectors, which can be signed or unsigned integers. Isolated multipliers are often implemented in two stages: partial product generation and partial product summation.

Partial products can be generated by multiplying (i.e., logical AND) each input bit with each other as in primary school multiplication. For signed numbers, the input numbers need to be sign-extended, in which case the Baugh-Wooley [8] sign extension technique can be used to lower the implementation area. Booth encoding [9] (particularly radix-4) is a more common and efficient way to generate partial products. Booth encoding incorporates more than two input bits at a time when generating partial products. This can provide more parallelism and fewer partial products. However, Booth encoding makes a circuit's structure and logic more complex, making it more difficult to reason about the circuit.

There are numerous methods to sum partial products in hardware. Unlike primary school multiplication, hardware algorithms do not sum partial products one column at a time, from right to left. Summations are performed more locally with unit adders such as half and full adders. An array multiplier is a simple example that is built with such unit adders following a shift-and-add methodology. Array multipliers have a regular structure, which makes it straightforward to verify them. However, they can have a large gate delay (i.e., propagation delay). On the other hand, Wallace-tree-like multipliers [10], such as Dadda tree [11], provide more parallelism. These summation tree algorithms sum partial products with less propagation delay and only slight changes in the implementation area. Designers can also utilize low gate-delay vector adders, such as Brent-Kung [12], Ladner-Fischer [13], and conditional sum, as a final stage adder to get the multiplication result. This can make Wallace-tree-like algorithms with complex final stage adders more preferable for hardware applications, but their irregular structures make the verification problem difficult, especially when paired with Booth encoding.

We should also note that an isolated multiplier implementation may not always return the full multiplication result. Instead, the result might be truncated, right-shifted, or a combination of both. For example, when two 32-bit numbers are multiplied, a lossless multiplier would output a 64-bit number. On the other hand, if the design only calculates the lower, say, 32-bits of the result, we say that the result is truncated. Similarly, when, say, only the upper 32-bits of the result are returned from the multiplier, we say that the result is right shifted. If only the middle portion of the result is returned, which may happen in fixed-point arithmetic, we say that the result is right shifted and truncated. Some designs implement rounding or saturation when a certain portion of the result is discarded when truncating and/or shifting.

B. Simple Arithmetic Modules with Embedded Multipliers

Integer multipliers can be implemented in various arithmetic modules such as MAC, dot-product, and floating-point arithmetic units. This section summarizes how a MAC module can be implemented in hardware.

A simple MAC computes $a * b + c$, where a , b and c are bit-vectors. When designing a MAC module, one may implement an isolated multiplier that computes $a * b$ and a vector adder that adds c to the multiplier's output. To verify such a MAC module, one can decompose the design, use different tools to verify the isolated multiplier and the final adder separately, and compose the proofs to show that the overall MAC module is correct. However, this design methodology uses two vector adders consecutively (one vector adder as part of the isolated multiplier and one for adding c). Vector adders can make up a large portion of the gate delay (and/or area) in such circuits, and this design technique can increase the gate delay considerably, making this approach a poor design choice.

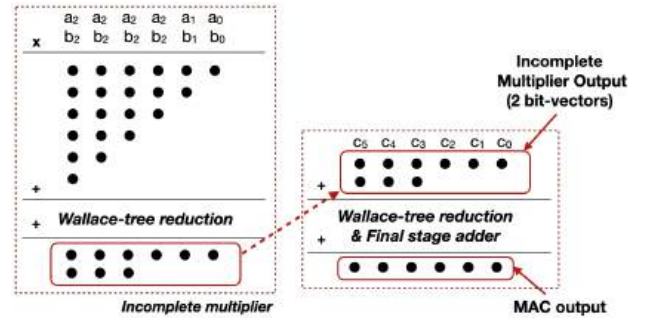


Fig. 1. An efficient way to compute MAC result

Fig. 1 shows an alternative approach that uses only one vector adder. This MAC module does *not* implement a complete isolated multiplier. Instead, it uses an *incomplete* multiplier. We define incomplete multipliers as modules that multiply two bit-vectors but do not use a final stage adder to return the complete multiplication result; instead, they return the two bit-vectors generated after the Wallace-tree reduction (summing these two vectors would give the multiplication result). This output form is also referred to as *redundant*

form. After the incomplete multiplication, the two bit-vector outputs are summed together with the addend (c) using another Wallace tree and a vector adder. This can be a preferable design approach as it provides better gate-delay performance. However, it removes the boundaries between multiplication and summation, which complicates the job of a verification engineer. Further complicating verification, an alternative design technique may sum c with the initial partial products with a single Wallace-tree and vector adder, which can remove the boundaries even further. In such cases, we cannot simply decompose the design and use a multiplier verification tool that works only with isolated multipliers.

We can see similar design methodologies in other modules. For example, a dot product design may use multiple *incomplete* multiplier modules and sum all the output vector pairs together in another summation tree using a Wallace-tree and a final stage adder. This method would prevent the increase in area and gate delay by using only one final stage adder in the overall design. Similarly, a floating-point module implementing FMA (fused multiply-add) may use an incomplete integer multiplier.

C. Multi-purpose Multipliers

Some processing units may implement multipliers for various arithmetic operations with different operand sizes. For example, x86 chips have many integer multiplication instructions such as PMADDWD (multi-lane multiply and add together, in other words, dot-product), PMULHW (multi-lane multiply and store upper half of the result), and PMULLW (multi-lane multiply and store lower half). Multiplier circuits can occupy a large implementation area, and it is common for such instructions to share resources and reuse multiplier modules.

We have created an example arithmetic circuit that shows how multiplier modules can be reused for different operations. We call this arithmetic unit *integrated multipliers* whose schematic diagram is shown in Fig. 2. This design multiplexes various multipliers and adders to perform 4-point 32-bit dot-product, 1-lane 64-bit multiply-accumulate, or 4-lane 32-bit multiply-accumulate with options to return lower or upper significant halves of the result. This module also includes an accumulator register that can be used, for example, to perform an 8-point 32-bit dot-product in two clock cycles, or 12-point 32-bit dot-product in three clock cycles, and so on. The mode of operation is determined by the control signal *mode*.

This module implements four identical 32x32-bit *incomplete* multipliers whose inputs are two 32-bit numbers with an additional sign bit and whose outputs are two bit-vectors. Depending on the mode of operation, the outputs of these multipliers are summed with another summation tree, and the final result is calculated with vector adders. The datapaths for 32-bit MAC and dot-product operations are as described in the previous section (Sec II-B). This module also supports 64-bit operands, in which case the outputs of the 32x32-bit incomplete multipliers are appropriately shifted, sign-extended, and summed to calculate the 64x64-bit multiplication result. We call such operations *merged multiplication*, where multiple

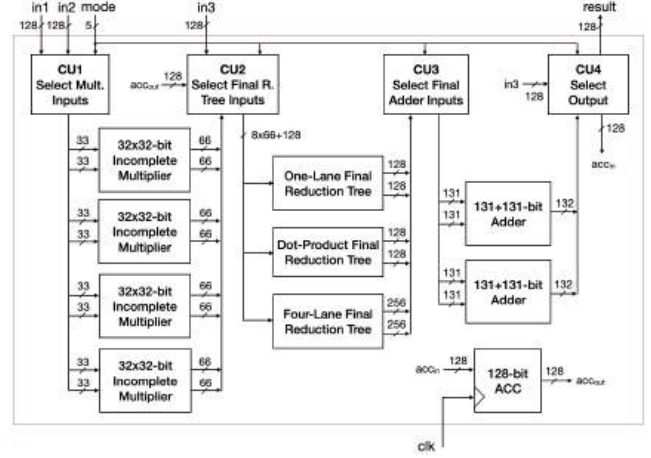


Fig. 2. The circuit diagram of integrated multipliers, our example arithmetic unit.

smaller multipliers are used to implement a larger multiplier. The module can also add a number to the 64x64-bit multiplication result and make this a 64-bit MAC operation.

We can verify this design for each possible mode of operation. For example, we can set the *mode* signal to perform dot product and check if the result matches the mode's specification. Industrial designs are often much more intricate than this module; however, it is often possible to reason about one arithmetic operation at a time. Then, the verification problem becomes as complex as verifying a single arithmetic operation.

III. RELATED WORK

The verification problem of multipliers continues to have a great deal of research interest, and researchers offer new techniques every year. This section covers the most recent and prominent studies that attempt to solve this problem, particularly for RTL designs with Booth encoding and Wallace-tree-like structures.

A. BDDs, BMDs, SAT and SMT Solvers

Automated and well-studied generic tools and methods such as BDDs, SAT, and SMT Solvers can theoretically be used to verify multiplier designs. However, it has been shown that these methods do not scale for designs larger than 12x12-bit multipliers [1], [2]. SAT solvers may scale better when generating counterexamples for buggy designs. Some success has been achieved with BMDs but only for regularly structured multipliers [14]. On the other hand, these automated tools may be used to verify some multiplier design components, such as the final stage adder [3].

B. Computer Algebra Methods

In computer algebra-based methods, multiplier circuits are modeled with a set of polynomials. Basic logical gates of a circuit are represented in terms of algebraic expressions (e.g., $\forall x, y \in \{0, 1\} \ x \vee y = x + y - xy$) as well as the multiplication result (see Example 1 for a 2x2-bit unsigned multiplier specification). The algebraic representation on its

own does not scale when verifying multipliers. Researchers implement various heuristics and optimizations that are specific to multiplier designs to achieve efficient and practical results. A notable optimization is identifying the logic from adder modules implemented in target multiplier designs [3], [4], [15]–[17].

Example 1. $4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0a_0$

Computer algebra methods have made a lot of progress towards the multiplier verification problem. However, these studies have focused mainly on isolated multipliers with untruncated outputs and the same operand sizes ($n \times n$ -bit multipliers with $2n$ -bit outputs). This makes it more difficult to utilize them for real-world designs where truncation, shifting, and integration with other arithmetic operations are common (See Sec. II).

Ciesielski et al. [1] showed that their method could be used for other multiplier-centric arithmetic operations, such as MAC; however, they showed that they only verified multiplier modules with regular structures. The benchmarks and their verification tool are not provided. We do not know of any publicly available tool that can scale and automatically verify designs such as MAC and dot-product. The underlying theory used by the computer-algebra methods may support verification of such arithmetic circuits. However, some optimizations that make these tools efficient may or may not be directly applicable to modules beyond isolated multipliers.

Verifying multipliers whose output is truncated or shifted is difficult for the computer algebra approach. Su et al. [18] discussed why computer algebra techniques are inefficient when verifying truncated arithmetic circuits. They stated that intermediate expressions, which are manageable in untruncated modules, can grow exponentially in truncated designs. They suggested a method to reconstruct a truncated multiplier into a complete multiplier by adding missing elements before verification. They did not discuss the soundness of their approach, their experiments were only on simple multipliers, and the benchmarks and the tool are not provided. Kaufmann et al. [3] suggested using modular arithmetic and defined a specification in the ring $\mathbb{Z}_{2^n}[X]$ where n is the multiplier output size. They showed that this approach works on a simple multiplier model, but our experiments with RTL designs resulted in time-out. We are not aware of any computer algebra studies that can verify truncated and/or shifted RTL multipliers.

C. Industrial Methods

Verification efforts of commercial multipliers often involve a great deal of manual work. A common method is to create a simple reference design that is structurally close (isomorphic) to the original and then repeatedly equivalence-check a litany of ever-increasingly complex designs [19]. Some engineers verify reference designs using mechanized proof systems [20]. Another common analysis method is to decompose a design into smaller parts, reason about these parts separately, and then compose these proofs into a top-level theorem [21]–[23]. Finding a workable decomposition and combining individual

proofs of multiplier fragments can be a cumbersome task. Such methods help formal verification engineers verify various multiplication operations such as multiply-accumulate and dot-product; however, this usually entails extensive manual effort. Moreover, these proofs are often design-specific, and even a slight change in the design might cause a previous proof procedure to fail.

IV. S-C-REWRITING ALGORITHM

In our previous work [7], we introduced a verified term-rewriting algorithm that can verify a wide range of isolated multiplier designs more quickly than the other state-of-the-art tools. In this section, we summarize this term-rewriting algorithm and discuss its recently discovered limitations.

We use the ACL2 theorem prover to verify and run our multiplier verification tool. ACL2 is an interactive and automated theorem proving system, and a programming language that is used by both industry and academia [24]. For a target multiplier design, we try to prove conjectures of the form given in Listing 1. `defthm` is a commonly used utility by ACL2 users, and it asks the ACL2 system to check conjectures. On the left hand side, we specify symbolic simulation of a multiplier design representation. We use the SVL semantics [25] to simulate designs, which are automatically translated from Verilog (our verification tool can be used with other simulators as well). The right hand side has the multiplier specification; in this example, the target multiplier module returns a 128-bit number equivalent to the multiplication of two 64-bit signed numbers.

Listing 1. A correctness conjecture for a signed 64x64-bit isolated multiplier

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
    (equal (simulate :inputs (a b)
                    :design <signed_64x64_mult>)
      (truncate 128
        (* (signext 64 a)
           (signext 64 b))))))
```

We prove such conjectures by rewriting both sides of the equality to fixed final forms. We define two functions s (short for *sum*) and c (short for *carry*) as given in Def. 1. The target representations for the first few output bits of some modules (half, full, vector adders, and multipliers) are given in Table I. Our goal is to rewrite all such modules/operations to this form. We call this s - c representation or s - c form.

Definition 1. Functions s and c are defined as follows.

$$\begin{aligned} \forall x \in \mathbb{Z} \quad s(x) &= \text{mod}_2(x) \\ \forall x \in \mathbb{Z} \quad c(x) &= \left\lfloor \frac{x}{2} \right\rfloor \end{aligned}$$

While verifying multiplier designs, we wish not to work with the logical definition of adder modules but instead work with their s - c representations. The SVL semantics allow hierarchical reasoning such that if we previously prove that symbolic simulation of an adder module can be replaced with this s - c form, then the SVL system can use this form (as

TABLE 1
TARGETED FINAL FORMS FOR SOME MODULES/FUNCTIONS

Function	out_2	$out_1 / c_o \ t$	$out_0 / s_o \ t$
Half-adder	-	$c(a + b)$	$s(a + b)$
Full-adder	-	$c(a + b + c_{in})$	$s(a + b + c_{in})$
Bit-vector addition $a + b$	$s(a_2 + b_2 + c(a_1 + b_1 + c(a_0 + b_0)))$	$s(a_1 + b_1 + c(a_0 + b_0))$	$s(a_0 + b_0)$
Bit-vector multiplication $a * b$	$s(a_0 b_2 + a_1 b_1 + a_2 b_0 + c(a_1 b_0 + a_0 b_1 + c(a_0 b_0)))$	$s(a_1 b_0 + a_0 b_1 + c(a_0 b_0))$	$s(a_0 b_0)$

opposed to the adder's logical definition) while expanding the definition of multiplier designs. Therefore, we first prove that each distinct adder module can be represented with the s - c form. We use a term-rewriting algorithm to carry out the proofs for adder modules [7]. Since verifying adders is straightforward [3], we omit this rewrite algorithm here for brevity. After the adder proofs, we start verifying the target multiplier design. As we expand the definition of the multiplier, our program replaces each instance of its adder modules automatically with their s - c representation.

Using the s - c form for adders instead of their logical definitions can bring about simpler expressions representing the output bits of a multiplier. An example of such an expression is given in Example 2 for a Wallace-tree multiplier with simple partial products.

Example 2. The 4th LSB of a Wallace-tree multiplier output when its adders are represented in the s - c form:

$$s(s(s(a_3 b_0 + a_2 b_1 + a_1 b_2) + a_0 b_3 + c(a_2 b_0 + a_1 b_1 + a_0 b_2)) + c(s(a_2 b_0 + a_1 b_1 + a_0 b_2) + c(a_1 b_0 + a_0 b_1)))$$

We rewrite such terms to make them syntactically equivalent to our target final form. To do that, we define a set of lemmas of the form $lhs = rhs$ such that terms that match lhs are replaced with rhs with appropriate term bindings. All lemmas are proved using ACL2 and we omit the proofs here.

We investigated such terms from multiplier designs and realized that we could rewrite and simplify nested calls of s with Lemma 1. Rewriting with this lemma when applicable can simplify the term from Example 2 to the form given in Example 3.

Lemma 1. $\forall x, y \in \mathbb{Z} \ s(s(x) + y) = s(x + y)$

Example 3. Example 2 simplified with Lemma 1:

$$s(a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 + c(a_2 b_0 + a_1 b_1 + a_0 b_2) + c(s(a_2 b_0 + a_1 b_1 + a_0 b_2) + c(a_1 b_0 + a_0 b_1)))$$

Now, we observe more than one instance of c on the same summation level. We rewrite and simplify them by a set of lemmas. Lemmas 2-5 are applied to the term as rewrite rules,

where the function d is defined as $\forall x \in \mathbb{Z} \ d(x) = \frac{x}{2}$. Then, we get the term in Example 4. This is syntactically equivalent to our target form for the 4th output bit, and we can conclude that the multiplier is correct for this output bit.

Lemma 2. $\forall x, y \in \mathbb{Z} \ c(x) + c(y) = d(x + y - s(x) - s(y))$

Lemma 3. $\forall x, y \in \mathbb{Z} \ c(x) + d(y) = d(x + y - s(x))$

Lemma 4. $\forall x, y \in \mathbb{Z} \ d(x) + d(y) = d(x + y)$

Lemma 5. $\forall x \in \mathbb{Z} \ d(-s(x) + x) = c(x)$

Example 4. Example 3 rewritten with Lemma 2-5:

$$s(a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 + c(a_2 b_0 + a_1 b_1 + a_0 b_2 + c(a_1 b_0 + a_0 b_1)))$$

As Booth encoding can incorporate multiple input bits when generating partial products, we can see operators for logical gates (e.g., logical OR, XOR) when verifying Booth encoded multipliers. We use a few more simple lemmas to simplify terms from Booth encoding and we derive the same final form. These lemmas, along with examples, are provided in our previous work [7], and we omit them here for brevity. These extra lemmas are triggered automatically when Booth encoding is present, and they do not affect other proofs when simple partial products are used.

Once we are done rewriting the left-hand side in Listing 1, we rewrite the right hand side (specification) to the same form through proved rewrite rules from our library. When we see that the two sides are syntactically equivalent, we conclude that the multiplier is correct.

Note that our target representation has a separate term for each output bit whereas the computer algebra methods specify all output bits with a single expression (see Example 1). This makes it easier for our method to verify designs whose output may be manipulated on bit level such as by truncating, shifting, and bit-masking.

Example 5. The first instance of $a_2 b_0$ in Example 2 is replaced by $a_2 b_1$ to simulate a bug. Then, the rewriting algorithm returned:

$$s(a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 + d(-s(a_2 b_1 + a_1 b_1 + a_0 b_2) - s(a_2 b_0 + a_1 b_1 + a_0 b_2) + c(a_1 b_0 + a_0 b_1)) + s(a_2 b_0 + a_1 b_1 + a_0 b_2) + a_2 b_1 + a_1 b_1 + a_0 b_2 + c(a_1 b_0 + a_0 b_1)))$$

In our previous work, we did not investigate what happens when the design has a bug and whether or not the algorithm can work beyond isolated multipliers. If our program cannot verify a multiplier for some reason, it returns a term rewritten with our lemmas. For example, when we introduce a simple bug to the term in Example 2, the described rewriting algorithm will return the term given in Example 5. The resulting term is larger than the initial term, and the gap can grow even larger for big designs. When a proof attempt fails, either due

to a bug in the design or some problem with our verification method, resulting terms are often very large and users do not receive a useful feedback from the program.

A proof attempt might fail even when the target design is correct. We have found such an instance and we could not verify some Booth encoded *merged* multipliers (See Sec. II) larger than 16x16-bit multiplication. Since the resulting terms are so large, we could not understand if there was a missing lemma that could help finish the proofs. We encountered similar issues with some dot-product and MAC designs, and we were likewise unable to verify them.

V. IMPROVEMENTS TO S-C-REWRITING

We have developed and experimented with various alternatives to the existing S-C-Rewriting algorithm. Our goal is to verify designs beyond isolated multipliers and return small terms if a proof attempt fails due to a design bug or a problem in the verification system. We have found a rewriting scheme that meets these goals. Instead of rewriting c terms with Lemmas 2-5, we use only the new Lemma 6. Similar to Lemma 1, this lemma extracts the arguments of inner s calls but it also creates a byproduct $-c(x)$.

Lemma 6. $\forall x, y \in \mathbb{Z} \ c(s(x) + y) = c(x + y) - c(x)$

When the given designs are correct, this lemma helps simplify multiplier designs without needing Lemmas 2-5. We have also seen that when this lemma is used, proofs are actually much faster for Booth encoded designs as well as array multipliers by an order of magnitude (see Sec. VII).

For cases where a proof-attempt fails, we apply another lemma (Lemma 7) to cancel out common terms shared between the specification and the design. After all our lemmas are applied and the design is simplified, the rewriter compares if the simplified design is syntactically equivalent to the specification for each output bit. If they are not, then we rewrite the term that represents the equivalence of these two sides with Lemma 7.

Lemma 7. $\forall x, y \in \{0, 1\} \ (x = y) \iff (s(x + y) = 0)$

Lemma 6 and Lemma 7 help the program return a much smaller term if a proof attempt fails. Assume that we are rewriting a term that checks the equivalence of the term from Example 2 to its specification (Example 4). When we introduce the same bug from Example 5 to this term, our new rewrite method will return the term in Example 6.

Example 6. When the same bug from Example 5 is rewritten with the improved rewriting algorithm:

$$\begin{aligned} & s(c(a_0b_2 + a_1b_1 + a_2b_0) \\ & \quad + c(a_0b_2 + a_1b_1 + a_2b_1)) \\ & = 0 \end{aligned}$$

As seen in this example, the returned term is considerably smaller than what we would get from the older algorithm (Example 5). We have observed the same behavior with larger multipliers so much so that the returned term can sometimes

give a hint as to where the bug exists within the design. Moreover, since these terms are often small, we use the FGL [26] or the GL [27], [28] utilities in ACL2 to send such returned terms to an external SAT Solver. We have seen through our experiments (Sec. VII) that SAT Solver can return a counterexample very quickly from simplified terms.

As noted in Sec. IV, proof attempts may fail even when the design is correct. This was the case with our initial term rewriting strategy for some Booth encoded merged multipliers and some MAC and dot-product modules. Since the returned terms are smaller with the modified term-rewriting, we could find the source of the problem and determine the missing lemmas needed to verify these designs. We found out that we simply need to rewrite some c and s instances in terms of logical operators (see Lemmas 8-11) when certain syntactic conditions on their arguments are met. Those conditions are: the arguments x , y and z (if available) need to be instances of the logical AND (\wedge) function only, and the operands in y and z (if available) need to be a subset of the operands of x . For example, we can apply Lemmas 8-9 if $x = a \wedge b \wedge c \wedge d$, $y = a \wedge c$, and $z = b \wedge c$ but we cannot apply it if $z = b \wedge e$. The resulting terms from these rewrites are simplified the same way as Booth encoding logic. We have these strict syntactical conditions so that the rewriting system is more deterministic and there is minimal effect on the verification procedures for other designs. We leave these lemmas enabled in our program, and they help automatically verify the previously failed designs, such as merged multipliers.

Lemma 8. $\forall x, y, z \in \{0, 1\} \ c(x + y + z) = x \wedge y \vee x \wedge z \vee y \wedge z$

Lemma 9. $\forall x, y, z \in \{0, 1\} \ s(x + y + z) = x \oplus y \oplus z$

Lemma 10. $\forall x, y \in \{0, 1\} \ c(x + y) = x \wedge y$

Lemma 11. $\forall x, y \in \{0, 1\} \ s(x + y) = x \oplus y$

Additionally, we tested this method with another simulation tool, SVTV [24], to show that our method does not have to be used with the SVL system. The SVTV system sources designs from Verilog and flattens them before (symbolic) simulation. We found a way to mark the adder modules before flattening to easily rewrite them in the s - c form. We omit the details here for brevity, and the readers may refer to our online tutorials for details (<http://mtmel.com/fmcad21>).

VI. IMPLEMENTATION

All of our rewriting system consists of lemmas of the form $lhs = rhs$. When patterns found in conjectures match lhs , they should be replaced by rhs . Since conjectures for multiplier designs may yield very large terms, we implement a scalable mechanism to find such patterns and apply our lemmas.

We use a verified rewriter [29] that follows an inside-out rewriting strategy [30], [31]. Example 7 shows how a rewrite rule can modify a term from inside out. We can prove the associativity of summation (see the upper-left corner) using the existing libraries and the built-in axioms in ACL2. The

defthm event saves the proved lemma as a rewrite rule. When this rewrite rule is in the system, we can apply it to terms whenever the left hand side pattern finds a match. Assume that this is the only enabled rule, and we would like to prove another conjecture which contains the term shown on the upper-right corner. Since the rewriter performs inside-out rewriting, it will start with the innermost term to search for matching patterns. The first match occurs for the following bindings: a to $x3$, b to $x4$, and c to $x5$. With these term bindings, the term is replaced using the right hand side of the rewrite rule, and we obtain the term in the lower-left corner. The rule can find another match on this new term. After similarly rewriting this term, we obtain the term in the lower-right corner.

Example 7. A target term is rewritten with a rewrite rule.

Rewrite Rule	Target Term
<pre>(defthm sum-assoc (equal (+ (+ a b) c) (+ a (+ b c))))</pre>	<pre>(+ (+ x1 x2) (+ (+ x3 x4) x5))</pre>
After the First Rewrite	After the Second Rewrite
<pre>(+ (+ x1 x2) (+ x3 (+ x4 x5)))</pre>	<pre>(+ x1 (+ x2 (+ x3 (+ x4 x5))))</pre>

Even though the rewriter dives into every subterm, it keeps track of already processed terms and it does not attempt to rewrite them again. For example, assume that $x4$ in the target term from Example 7 is not a variable but it is a very large term that is already rewritten. After the first rewrite, $x4$ will have moved within the term. Since the applied rule has a fixed pattern on the left and right hand sides, the rewriter knows to not process $x4$ again. On the other hand, if there was an applicable rule, the new subterm $(+ x4 x5)$ could be rewritten.

Our overall rewriting system follows this basic rewriting strategy with many more lemmas that work together harmoniously. Fig. 3 shows a flow diagram when the rewriter processes a conjecture for multiplier designs. Assume that we are using the SVL system for simulation, and the user has already created rewrite rules for adder modules to represent them in the s - c form. When the user states a conjecture for the target multiplier design (see Listing 1) and submits it to ACL2, the rewriter dives into the innermost terms to search for applicable rules. The first subterm that it rewrites is the symbolic simulation instance for the target multiplier design.

The SVL system simulates designs by executing all the functional blocks (e.g., Verilog assignments and submodules) and one by one calculating the values for all internal wires and registers. As the rewriter is symbolically simulating an SVL design, derived expressions for internal wires and registers are tested against rewrite rules. If the rewriter encounters an

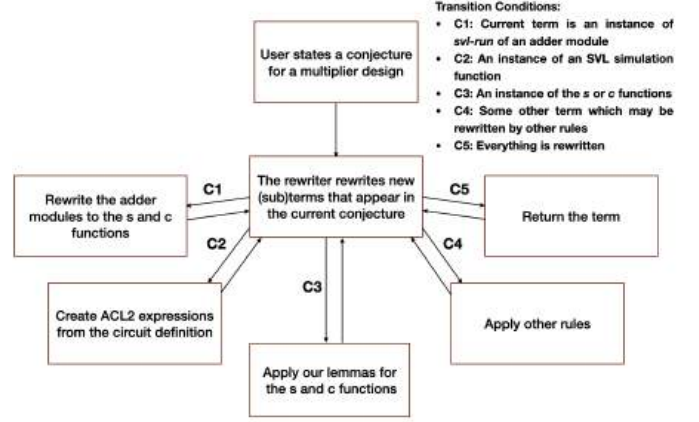


Fig. 3. Steps taken by the rewriter when rewriting a conjecture for a multiplier design

instantiation of an adder module, then it is replaced by the s and c functions using the rules created by the user. If the rewriter encounters some other module or an assignment, then regular ACL2 expressions representing their functionality are created from their logical definitions.

When new instances of the s and c are created after the adder modules are rewritten, our lemmas for these functions are triggered and our simplification algorithm is applied. For example, when the new term is an instance of c and one of its arguments is an instance of s , then Lemma 6 will be applied. If the arguments of the new s and c instances contain some Boolean expressions, then our lemmas for Booth encoding [7] are applied.

As the symbolic simulation of the circuit finishes, we get a term that is completely rewritten with our algorithm. After that, the system rewrites the right hand side (specification) to the s - c form with other rewrite rules in our library, compares the two sides syntactically, and exits. If the final term is \top , then we can conclude that the multiplier is correct. Otherwise, we can investigate this term and/or send it to a SAT solver so as to generate counterexamples or attempt to finish the proofs.

Note that our lemmas described in Sec. IV, Sec. V, and our previous work [7] do not trigger an expensive rewriting chain upon application. They each have an almost constant time complexity. The slowest component of the rewriting algorithm is lexicographical sorting of the terms in column summations, which are expected to be very small sets as compared to the overall size of the given design. Since our lemmas are applied as the circuit's definition is expanded and we never perform a global search, we observe an almost linear time complexity with respect to the design size as shown in the next section.

VII. EXPERIMENTS

We verified various multiplier designs using our tool and applicable tools from related work. We ran our experiments on an Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz computer with 32GB system memory. We used three RTL multiplier

TABLE II
PROOF-TIME RESULTS IN SECONDS (ROUNDED) FOR VARIOUS
UNTRUNCATED, SIGNED ISOLATED MULTIPLIER DESIGNS

Size	Architecture	RS [4]	AMu [3]	Prev [7]	This work
64x64	sp-cwt-ks	39	42	1	.5
	sp-ar-rc	3	2	1	.5
	sp-dt-bk	5	2	1	.5
	b4-wt-hc	154	28	1	1
	b2-wt-hc	123	77	4	1
	b4-dt-ks	17	28	1	1
	b4-dt-csel	19	5	4	1
	b4-os-bk	15	5	6	1
	b4-wt-csu	21	5	5	2
	b4-bdt-hc	131	6	5	2
	b4-rbat-ks	19	7	5	2
	b4-ar-vcska	17	5	12	2
	b4-4:2-lf	30	5	8	3
	b4-7:3-bcla	44	TO	12	6
	b4-wt-cla	22	14	21	12
128x128	sp-cwt-ks	1001	TO	3	2
	sp-ar-rc	96	10	20	2
	b4-wt-hc	TO	803	13	4
	b4-dt-ks	773	785	8	4
256x256	sp-cwt-ks	TO	TO	16	7
	sp-ar-rc	2416	176	556	11
	b4-wt-hc	TO	TO	62	15
	b4-dt-ks	TO	TO	47	15
512x512	sp-wt-lf	TO	1577	76	44
	sp-dt-bk	TO	1562	64	40
	b4-wt-hc	TO	TO	418	65
	b4-dt-ks	TO	TO	282	71
1024x1024	sp-wt-lf	TO	14005	345	240
	sp-dt-bk	TO	13247	397	220
	b4-wt-hc	TO	TO	MO	288
	b4-dt-ks	TO	TO	MO	300

MO: Out of memory (32GB) TO: Time-out (5400 secs./90 mins. for 64x64 and 128x128 multipliers, 16200 secs./270 mins. for the rest)

generators [32]–[34] to generate isolated multipliers, MAC, and dot-product designs. The benchmarks and our tool are available online (<http://mtmel.com/fmcad21>).

We verified various architectures with different configurations. For partial product generation algorithms, the designs use either simple partial products (*sp*), Booth encoding radix-4 (*b4*) or radix-2 (*b2*). Summation tree reduction algorithms include counter-based Wallace (*cwt*), array (*ar*), Dadda (*dt*), traditional Wallace (*wt*), overturned-stairs (*os*), balanced delay (*bdt*), redundant binary addition (*rbat*), 4-to-2 compressor (4:2), 7-to-3 compressor (7:3) trees, and merged multipliers with Dadda tree (*mdt*). For final stage addition, these multipliers implement Kogge-Stone (*ks*), ripple-carry (*rc*), Brent-Kung (*bk*), Han-Carlson (*hc*), Ladner-Fischer (*lf*), carry-select (*csel*), conditional sum (*csu*), variable-length carry-skip (*vc-ska*), block carry-lookahead (*bcla*) and regular carry-lookahead (*cla*) adders.

As far as we are aware, there are only two other publicly available tools from two different research groups that can verify these complex architectures for isolated multipliers. These are computer-algebra-based tools RevSCA2 [4] (shortened as RS) and AMulet 2.0 [3], [35] (shortened as AMu). The tools from other studies are not publicly available and/or they do

TABLE III
PROOF-TIME RESULTS IN SECONDS FOR SOME MULTIPLIER DESIGNS IN
VARIOUS CONFIGURATIONS

Function & I/O Size	Architecture	AMu [3]	Prev [7]	This work
16x16 = 16	usp-dt-hc	TO	.1	.04
16x16 = 16	ssp-dt-hc	NS	.1	.04
16x16 = 16	ub4-dt-hc	TO	.1	.06
16x16 = 16	sb4-dt-hc	NS	.1	.05
20x40 = 60	ub2-wt-rp	NS	.3	.1
20x40 = 60	sb2-wt-rp	NS	.3	.1
33x17 = 40	ub4-wt-hc	NS	.2	.1
33x17 = 40	sb4-wt-hc	NS	.2	.1
64x64 = 64	ub4-dt-hc	TO	1	.5
64x64 = 64	sb4-dt-hc	NS	1	.4
64x64 = 64 (r. shifted)	ub4-dt-hc	NS	2	1
64x64 = 64 (r. shifted)	sb4-dt-hc	NS	2	1
64x64 = 128	ub4-mdt-ks	45	F	1
64x64 = 128	sb4-mdt-ks	44	F	1
64x64 = 128	ub2-mdt-lf	61	F	1
64x64 = 128	sb2-mdt-lf	59	4	1
2(32x32)+32 = 66	sb4-dt-hc	NS	F	1
2(32x32)+32 = 66	sb4-os-bcla	NS	F	1
2(32x32)+32 = 66	sb4-bdt-csu	NS	F	1
2(32x32)+32 = 66	sb4-ar-csel	NS	F	1
2(32x32)+32 = 66	sb4-4:2-rp	NS	F	2
2(32x32)+32 = 66	sb4-7:3-bk	NS	F	3
64x64+128 = 128	ub4-dt-ks	NS	2	1
64x64+128 = 128	sb4-dt-ks	NS	2	1
64x64+128 = 129	sb4-dt-hc	NS	F	2

TO: Time-out (5400 secs) NS: Configuration is not supported by the tool.
F: Failed proof-attempt. The tool returns a large rewritten term.

not provide competitive results for the designs in question. RevSCA2 does not produce certificates and it is not verified. AMulet provides certificates to check the validity proofs by external tools; we include the certification time in our results (they can be around 3 times faster without certification). The verification tools from our previous and current work are verified using ACL2; thus, no additional check is required.

Table II delivers the proof-time results in seconds for signed and untruncated isolated multipliers. Our previous work scales substantially better than (RS [4]) and (AMu [3]) but the performance is not as strong for Booth encoded designs. Our improved rewriting algorithm is much faster than our previous work and others, and it can verify even very large Booth encoded multipliers in at most 5 minutes.

Table III delivers proof-time results for various architectures and configurations. This includes truncated or right shifted outputs, merged multipliers, multipliers with different operand sizes, two-point dot-product designs with accumulate, and truncated or untruncated MAC modules. The designs in this table are produced with two different generators [32], [33]. AMulet has a hard-coded specification and does not support many of these configurations. Users can determine the design specifications for our previous work, but our older tool cannot prove some merged multipliers, dot-product, and MAC designs. On the other hand, our new method could verify all of them very quickly.

Table IV shows how the proof-time performance of our tool

TABLE IV
OUR TOOL’S PROOF-TIME RESULTS IN SECONDS FOR SIGNED MAC AND DOT-PRODUCT DESIGNS

Size	Dot-product length				
	N=1	N=2	N=4	N=8	N=16
N(32x32)	0.2	0.5	1.0	2.0	4.5
N(32x32)+64	0.2	0.5	0.9	1.9	4.2
N(64x64)	0.9	1.9	3.8	8.2	19
N(64x64)+128	0.9	1.8	3.7	7.7	17
N(128x128)	3.5	7.8	18	35	81
N(128x128)+256	3.5	7.6	15	33	76
N(256x256)	15	32	67	151	356
N(256x256)+512	14	30	64	144	340

All designs use Booth radix-4 encoding, Dadda tree and Ladner-Fischer adder.

TABLE V
OUR TOOL’S PROOF-TIME RESULTS IN SECONDS FOR OUR EXAMPLE MODULE, INTEGRATED MULTIPLIERS, DESCRIBED IN SEC. II-C

Mode	SVL		SVTV	
	Signed	Unsigned	Signed	Unsigned
1-lane MAC	1.0	0.9	2.8	2.9
4-lane MAC (lower half)	1.0	0.9	2.8	2.8
4-lane MAC (upper half)	1.0	1.0	3.0	2.9
4-point dot-product	1.8	1.2	4.4	3.4
8-point dot-product (seq.)	4.9	2.9	14.5	10.1

scales on dot-product designs with different sizes. Even though it is not shown here, allocated system memory scales similarly. Finally, Table V shows the proof-time results for our example module integrated multipliers (see Sec. II-C) for both the SVL and SVTV simulation systems.

In addition to the designs reported here, we have also verified some private industrial designs at Centaur Technology with a similar performance. These designs include multiply-accumulate, dot-product, multiplication of signed and unsigned numbers, truncation, right-shifting, rounding, and saturation. Our program is not designed to handle branches implemented for saturation. Therefore, after our program simplified the saturated designs, we sent the resulting terms to a SAT Solver (*glucose* [36]) with the FGL utility [26], [37], and we have seen that proofs finished successfully in a few seconds.

We have also tried our tool on buggy designs and used a SAT solver (*glucose* [36]) to create counterexamples from simplified terms. We randomly inserted (one or more) bugs into various 64x64-bit, 128x128-bit, and 256x256-bit designs and experimented with 20 different scenarios. Our tool rewrote each multiplier design and returned simplified terms within the same amount of time as given in Table II. It took the SAT solver between 0.1 to 10 seconds to return a counterexample from rewritten terms. Our previous tool could not be used in this workflow because it returns massive terms when proof-attempts fail (see Sec. IV). Using the SAT solver with the original conjecture (in other words, without rewriting with our tool) could give a counterexample in some cases after a few minutes, but it timed out (60 minutes) in the majority of cases. Additionally, our tool can tell exactly which output bits are mismatching the specification. With our new method,

we see that our term-rewriting strategy can be very practical and efficient for debugging flawed designs.

VIII. CONCLUSION

We have presented a term-rewriting method that can be used to verify digital circuit designs with embedded integer multipliers. Our tool is efficient, automated, and provably correct. We have shown that we can verify isolated multipliers as large as 1024x1024-bit in less than 5 minutes. Our system allows the user to modify the specification per the target design. Therefore, we can verify multipliers with unusual operand sizes, whose output may be truncated, right-shifted, rounded or saturated. In addition, we can verify other multiplier-centric arithmetic operations such as dot-product and multiply-accumulate. Our library and tutorials are distributed with the ACL2 system, and this content is available online for public use (<http://mtemel.com/fmcad21>).

This work has been a continuation of our earlier study [7]. With the improvements detailed in this paper, we can verify Booth encoded designs with a much better proof-time efficiency, along with MAC, dot-product, and merged multiplier designs. In addition, we can now generate counterexamples for buggy designs. Moreover, we provide a more comprehensive summary of various multiplier design techniques and discuss why they might be challenging for verification tools.

We use the ACL2 programming language and interactive theorem prover to run and verify our multiplier verification tool, and we use the SVL semantics as our preferred method to simulate Verilog designs. However, our term rewriting algorithm does not require any specific feature from a particular a theorem prover or anything unique to the SVL system. Using a term rewriter and a simulator with hierarchical reasoning can be enough to implement our algorithm on any platform.

We have exploited design hierarchy when implementing our algorithm, whereas the other state-of-the-art tools [3], [4] work on flattened designs. We should note that these tools more or less depend on the original design having clear boundaries for adder modules for their good proof-time performance in the majority of cases. Our choice to use a symbolic simulation system that allows hierarchical reasoning reduces engineering costs and simplifies our program. This way, we do not need to implement any detection algorithm for adder logic. If necessary, using our term-rewriting algorithm for flattened designs might be possible by implementing some preprocessing techniques to reconstruct the design hierarchy. On the other hand, incorporating hierarchical reasoning into computer algebra methods may help improve their performance.


We continue to exercise and improve our method with ever more complex designs such as floating-point multiplication. We have laid a groundwork to permit verification procedures with improved automation and efficiency. The convenience that comes with our fast and automatic verification process can contribute to building reliable hardware systems that include embedded integer multipliers of varying sizes, including but not limited to general-purpose processing units, image processors, digital signal processors, and secure cryptoprocessors.

REFERENCES

- [1] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6. [Online]. Available: <https://doi.org/10.1145/2744769.2744925>
- [2] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal Verification of Integer Multipliers by Combining Gröbner Basis with Logic Reduction," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Research Publishing Services, 2016, pp. 1048–1053.
- [3] D. Kaufmann, A. Biere, and M. Kauers, "Verifying Large Multipliers by Combining SAT and Computer Algebra," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, Oct 2019, pp. 28–36.
- [4] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 185:1–185:6.
- [5] M. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
- [6] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: Clean your Polynomials before Backward Rewriting to verify Million-gate Multipliers," *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2018.
- [7] M. Temel, A. Slobodova, and W. A. Hunt, "Automated and scalable verification of integer multipliers," in *Computer Aided Verification*. Cham: Springer International Publishing, 2020, pp. 485–507. [Online]. Available: http://doi.org/10.1007/978-3-030-53288-8_5F23
- [8] C. R. Baugh and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. C-22, pp. 1045–1047, 1973.
- [9] A. D. Booth, "A Signed Binary Multiplication Technique," vol. 4, no. 2. Oxford University Press (OUP), 1951, pp. 236–240.
- [10] C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Trans. Electronic Computers*, vol. 13, pp. 14–17, 1964.
- [11] L. Dadda, "Some Schemes for Parallel Multipliers," 1965.
- [12] Brent and Kung, "A Regular Layout for Parallel Adders," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, mar 1982.
- [13] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, oct 1980.
- [14] R. E. Bryant and Y.-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," in *DAC 1994*, 1994.
- [15] M. A. Basith, T. Ahmad, A. Rossi, and M. Ciesielski, "Algebraic approach to arithmetic design verification," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, p. 67–71.
- [16] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G.-M. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 473–486.
- [17] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 544–549.
- [18] T. Su, C. Yu, A. Yasin, and M. Ciesielski, "Formal verification of truncated multipliers using algebraic approach and re-synthesis," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017, pp. 415–420.
- [19] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner, "Automatic formal verification of fused-multiply-add fpus," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, ser. DATE '05. USA: IEEE Computer Society, 2005, p. 1298–1303.
- [20] D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, 2019.
- [21] W. A. Hunt, S. Swords, J. Davis, and A. Slobodova, "Use of Formal Verification at Centaur Technology," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, pp. 65–88.
- [22] A. Slobodova, J. Davis, S. Swords, and W. A. Hunt, "A Flexible Formal Verification Framework for Industrial Scale Validation," in *Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. Cambridge, UK: IEEE/ACM, July 2011, pp. 89–97.
- [23] R. Kaivola and N. Narasimhan, "Formal Verification of the Pentium ® 4 Floating-Point Multiplier," in *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002)*, 4-8 March 2002, Paris, France, 2002, pp. 20–27.
- [24] W. A. Hunt, M. Kaufmann, Moore, J. S., 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, p. 20150399, sep 2017.
- [25] M. Temel, "ACL2 SVL Documentation," 2019. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2_SVL
- [26] S. Swords, "New rewriter features in FGL," *Electronic Proceedings in Theoretical Computer Science*, vol. 327, p. 32–46, Sep 2020. [Online]. Available: <http://dx.doi.org/10.4204/EPTCS.327.3>
- [27] S. Swords and J. Davis, "Bit-blasting ACL2 theorems," *Electronic Proceedings in Theoretical Computer Science*, vol. 70, p. 84–102, Oct 2011. [Online]. Available: <http://dx.doi.org/10.4204/EPTCS.70.7>
- [28] S. Swords, "Term-level reasoning in support of bit-blasting," *Electronic Proceedings in Theoretical Computer Science*, vol. 249, p. 95–111, May 2017. [Online]. Available: <http://dx.doi.org/10.4204/EPTCS.249.7>
- [29] M. Temel, "RP-Rewriter: An optimized rewriter for large terms in ACL2," vol. 327. Open Publishing Association, Sep 2020, p. 61–74. [Online]. Available: <http://dx.doi.org/10.4204/EPTCS.327.5>
- [30] H. R. Chamarthi, "Rewriting in ACL2," 2021. [Online]. Available: <http://www.ccs.neu.edu/home/harshrc/courses/cs2800-fall2010/f10-lec26.pdf>
- [31] M. Temel, "Automated, efficient, and sound verification of integer multipliers," Ph.D. dissertation, The University of Texas at Austin, 2021.
- [32] —, "Multgen: a fast multiplier generator," 2021. [Online]. Available: <https://github.com/temelmertcan/multgen>
- [33] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi, "Arithmetic module generator (AMG)," 2006. [Online]. Available: <https://www.ecsis.riec.tohoku.ac.jp/topics/amg/>
- [34] A. Mahzoon, D. Große, and R. Drechsler, "SCA multiplier generator GenMul," 2019. [Online]. Available: <http://www.sca-verification.org>
- [35] D. Kaufmann and A. Biere, "AMulet 2.0 for verifying multiplier circuits," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021*, ser. Lecture Notes in Computer Science, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 357–364.
- [36] N. Sörensson and N. Een, "Minisat v1.13-a sat solver with conflict-clause minimization," *International Conference on Theory and Applications of Satisfiability Testing*, 01 2005.
- [37] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Balancing automation and control for formal verification of microprocessors," 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. 26–45. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_2

IC3 with Internal Signals

Rohit Dureja 
IBM

Arie Gurfinkel 
University of Waterloo

Alexander Ivrii
IBM

Yakir Vizel 
The Technion

Abstract—IC3 is a highly-effective algorithm for formal hardware verification. It cleverly uses a SAT solver to compute an inductive invariant, an over-approximation of reachable states, of a hardware design. The invariant is computed in CNF as a conjunction of lemmas. This CNF representation over state variables, although efficient, leads to an obvious deficiency: IC3 is not effective for designs that do not have a concise CNF invariant over state variables. We show how to remedy this deficiency by extending traditional IC3 to learn invariants not only in terms of state variables, but also in terms of internal signals of the design. Our proposed method can learn significantly more compact invariants than IC3, while maintaining a highly-efficient CNF representation. We evaluate our technique on several industrial sequential equivalence checking (SEC) problems from IBM, SEC problems derived from designs in the Hardware Model Checking Competition (HWMCC) and SEC problems from academia. In addition, we evaluate it on HWMCC benchmarks. IC3 with internal signals is efficient for SEC and outperforms traditional IC3 on an important class of benchmarks.

I. INTRODUCTION

IC3 [1], [2] is a powerful algorithm for formal hardware verification, and is the primary model-checking engine in various state-of-the-art formal verification tools. IC3, and its several variants [3], is especially useful for establishing system safety (i.e., discovering an inductive invariant). Whenever IC3 succeeds in proving safety, it finds an inductive invariant justifying the property. Traditionally, such an invariant is a conjunction of lemmas represented in CNF, each lemma is a disjunction of literals, and each literal is either a state variable or its negation. Conversely, IC3 does not succeed in proving a property when it is unable to find such an inductive invariant within the specified verification-resource limits. This can happen for one of two reasons: (i) a small inductive invariant exists but IC3 is unable to find it, or (ii) a small inductive invariant does not exist. It is difficult to determine which of these two cases is responsible for IC3 failing to prove a property. Most research on improving IC3 (e.g., [4]–[6]) focuses on quickly finding the inductive invariant. However, finding the inductive invariant quickly can only help if a (reasonably) small invariant exists in the first place.

A known Achilles heel of IC3 are model-checking problems for which any inductive invariant (over state variables) is necessarily exponential in size. For example, let x_1, \dots, x_n be state variables, and suppose that the set of reachable states is characterized by $\{x_1, \dots, x_n \mid x_1 \oplus \dots \oplus x_n = 1\}$, while the set of bad states is characterized by $\{x_1, \dots, x_n \mid x_1 \oplus \dots \oplus x_n = 0\}$. In this case the (only) inductive invariant is exponential in size and contains 2^{n-1} clauses that correspond to representing $x_1 \oplus \dots \oplus x_n = 1$ in CNF. With $n = 3$, the inductive

invariant contains four clauses: $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$. A possible work-around is to extend the design with additional signals that are necessary to concisely represent an invariant. In this example, IC3 extended with a lemma over $z = x_1 \oplus \dots \oplus x_n$, can find a tiny inductive invariant consisting of only a single unit-clause lemma: $(z = 1)$.

This leads to the question of which additional signals to consider. A possible solution is to consider variables that represent logic gates in the transition relation of the system model. We refer to these as *internal nets* or *innards*. Prior work [7] uses innards to extend ternary valued simulation of counterexamples to induction in IC3, which enables a succinct description of the set of states that IC3 must eventually block. In this paper, we propose an approach based on learning lemmas directly over innards that improves the performance of IC3 in establishing safety by finding more concise inductive invariants. Our method of learning lemmas over internal nets can be viewed as a form of inductive generalization. A lemma is first generalized as usual, and then literals corresponding to latches are replaced by internal nets. Specifically, whenever IC3 learns a lemma C over state variables, it also tries to learn an additional lemma C_2 over state variables and internal signals. To this end, we first extend C to a lemma C_1 that is logically equivalent to C but contains the literals of C and (certain) internal nets. We obtain C_2 by inductively generalizing C_1 , while guiding the inductive generalization to remove state variables. It is guaranteed that C_2 is stronger than C . Therefore, C_2 blocks the same states (and maybe more) as C . We then add lemma C_2 to IC3’s inductive trace, so that it can be used for predecessor queries and convergence checks. A major advantage of our approach is that it can be easily integrated with any existing mature IC3 implementation.

Our work is motivated by a challenging set of microprocessor verification problems that arise from the Aspect-Oriented Design (AOD) methodology used at IBM. The verification problem checks sequential equivalence of an original design against a new version of the design with added aspects (e.g., clock-gating, logging, or debug interfaces). The complex verification challenge is broken into many sub-tasks using a combination of the usual sequential equivalence checking (SEC) approaches, including k -induction, speculative reduction, and localization [8]–[11]. Verification sub-tasks that are not solved by these techniques are then checked using Interpolation-based Model Checking (IMC) or IC3. Traditional IC3 scales very poorly for these verification problems. On the other hand, IMC works rather well but is not stable – small changes in the

design negatively impact verification times. The proposed IC3 algorithm with internal signals significantly outperforms both IMC and traditional IC3.

The proprietary nature of IBM AOD verification problems prohibits detailed public disclosure. Nevertheless, we apply the IBM AOD sequential equivalence checking flow on two selected benchmarks from the Hardware Model Checking Competition (HWMCC) to validate equivalence between the original design and its retimed [12] versions. Each such equivalence-check generates hundreds of verification problems of which some are solved by k -induction, but a significant number remain unsolved. We note that IC3 with internal signals is more effective than traditional IC3 in solving the remaining equivalences for both SEC problems. We also apply our algorithm on a small set of publicly available SEC benchmarks [13] from academia, and note that our proposed algorithm is able to solve a higher number of equivalences compared to traditional IC3. This suggests that using internal nets in IC3 is especially effective for difficult sequential equivalence checking problems.

To further validate the efficacy of IC3 with internal signals, we apply the proposed algorithm to a variety of single-property benchmarks from HWMCC. However, the technique does not show a significant improvement unlike our experience with IBM AOD and other benchmarks. There are a few HWMCC benchmarks that are solved significantly faster and some that are uniquely solved by our algorithm, but overall, traditional IC3 is superior. Interestingly, the number of designs where the new technique succeeds increases in the latest competition editions that are based on word-level designs. This points to a deficiency of any benchmark set – the distribution of problems in the set does not necessarily correspond to their distribution in practice. Techniques that perform well on only a few benchmarks in the set, might actually be very effective in some practical application!

The rest of the paper is organized as follows. Section II provides the necessary background. Section III describes motivating examples to highlight the core deficiency of IC3 addressed by our approach. Section IV describes the IC3 algorithm with internal signals, while Section V reports on our experimental evaluation. Section VI discusses related and future work, and Section VII concludes.

II. BACKGROUND

A. Safety Verification Problem

We represent a finite state transition system S as a tuple $\langle i, x, \text{Init}(x), \text{Tr}(i, x, x') \rangle$, which consists of primary inputs i , state variables x , predicate $\text{Init}(x)$ defining the initial states, and predicate $\text{Tr}(i, x, x')$ defining the transition relation. Next-state variables are denoted as x' . We assume that Tr is represented as a *netlist*, that is, a directed acyclic graph with nodes corresponding to logic gates. Given the values of x and i , the values of x' may thus be uniquely computed by (constant) propagation – i.e., using Boolean or three-valued simulation. We say that a *net* is either an input, a state variable or a logic gate. We refer to state variables and their negations

as *latches*, and to internal logic gates and their negations as *innards*. We say that an innard is *input-free* if it does not have any inputs in its combinational cone-of-influence.

A *clause* is a disjunction of literals, where each literal is either a net or its negation. We say that a clause is *over latches* to emphasize all the literals in the clause are latches. A Boolean formula in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. A *cube* is a conjunction of literals. A Boolean formula in *Disjunctive Normal Form (DNF)* is a disjunction of cubes. It is often convenient to treat a clause or a cube as a set of literals, a CNF as a set of clauses, and DNF as a set of cubes. For example, given a CNF formula F , a clause c and a literal ℓ , we write $\ell \in c$ to mean that ℓ occurs in c , and $c \in F$ to mean that c occurs in F .

A *trace* is a sequence of Boolean valuations to the nets, starting with an initial state satisfying Init and with successive time-step valuations consistent with Tr . *Reachable states*, denoted by Reach , are states that can be reached on a trace. Let $\text{Bad}(x)$ be a predicate defining *bad* (or *unsafe*) states. The *safety verification problem* consists of checking whether $\text{Reach} \Rightarrow \neg \text{Bad}$, that is either finding a trace that leads to a state in Bad or showing that such a trace does not exist.

B. Traditional IC3

We give a very brief and high-level description of IC3, concentrating on the components that are relevant for this work. This description includes the classical IC3 algorithm [1], [2], and some of its variants such as [6]. In what follows, we refer to all these algorithms simply as IC3.

IC3 proves safety by finding a formula $\text{In}(x)$, called a *safe inductive invariant*, that satisfies the following conditions:

$$\text{Init}(x) \Rightarrow \text{In}(x) \quad (1)$$

$$(\text{In}(x) \wedge i \cdot \text{Tr}(i, x, x')) \Rightarrow \text{In}(x') \quad (2)$$

$$\text{In}(x) \Rightarrow \neg \text{Bad}(x) \quad (3)$$

The computed formula $\text{In}(x)$ is in CNF over latches. Internally, IC3 maintains sets of clauses F_0, F_1, \dots called an *inductive trace*. Each F_k in a trace is called a *frame*, each clause $c \in F_k$ is called a *lemma*, and the index of a frame is called a *level*. We assume that F_0 is initialized to Init and that $\text{Init} \Rightarrow \neg \text{Bad}$. IC3 maintains the following invariant:

$$F_0 = \text{Init} \quad F_{k+1} \subseteq F_k \quad F_k \wedge \text{Tr} \Rightarrow F'_{k+1}$$

Note that the inductive trace maintained by IC3 is syntactically monotone, and each F_{k+1} is inductive relative to F_k . Let $\text{Reach}_{\leq k}$ denote the set of states reachable from Init in k steps or less. It holds that $\text{Reach}_{\leq k} \Rightarrow F_k$, i.e., F_k is an over-approximation of states reachable in k steps or less.

Additionally, IC3 maintains a queue of *proof obligations* (or *CTI's*) of the form $\langle m, k \rangle$ where m is a cube over latches and $k > 0$ is a *level*. At each point of the execution, it considers a proof obligation $\langle m, k \rangle$, and makes an *initial* query $\text{SAT}?(\text{Init} \wedge \neg m)$ that checks whether a state in m is an initial state, and a *predecessor* query $\text{SAT}?(\neg m \wedge F_{k-1} \wedge \text{Tr} \wedge m')$ that checks whether a state in m can be reached from a

state in F_{k-1} . If both results are unsatisfiable, IC3 can add the lemma $\neg m$ to all F_j , for $j \leq k$, refining the inductive trace. However, for performance it is crucial to *inductively generalize* $\neg m$ first, finding a lemma $\varphi \subseteq \neg m$, that also satisfies $Init \Rightarrow \varphi$ and $\varphi \wedge F_{k-1} \wedge Tr \Rightarrow \varphi'$ (some IC3-variants such as *Quip* also keep an under-approximation of *Reach* and modify *Init* to include this under-approximation). The inductive generalization is typically done by removing literals from $\neg m$ while the two conditions remain satisfied. We refer the reader to [3] for more details.

IC3 periodically *pushes* all lemmas, by checking if a lemma $\varphi \in F_k \setminus F_{k+1}$ can be added to F_{k+1} as well. If at any point, $F_k = F_{k+1}$ and $F_k \Rightarrow \neg Bad$, then we can take $Inv = F_k$ as the safe inductive invariant.

III. MOTIVATING EXAMPLES

In this section, we motivate our work with several examples. Each is a series of problems such that inductive invariants in CNF over latches grow exponentially, while the corresponding inductive invariants over latches and innards grow linearly. The examples are sketched briefly here, we provide full details with AIGER and source files in the companion repository.¹ Note that the examples are distilled to their essence. For some, the property itself is inductive. Thus, traditional IC3 that learns invariants over latches *and* the property is able to solve them. However, the illustrated problems remain when the examples are parts of a larger design, and the property is more complex and is no longer inductive on its own.

Example 1 (Parity) Let x_1, \dots, x_n be the latches. The set of reachable states is characterized by $\{x_1, \dots, x_n \mid x_1 \oplus \dots \oplus x_n = 1\}$. The set of bad states is characterized by $\{x_1, \dots, x_n \mid x_1 \oplus \dots \oplus x_n = 0\}$. Note that the only safe inductive invariant over latches has 2^{n-1} clauses representing $x_1 \oplus \dots \oplus x_n = 1$ in CNF. Yet, there is a safe inductive invariant consisting of a single lemma, $(z = 1)$, for the innard $z = x_1 \oplus \dots \oplus x_n$. \square

Example 2 (from [14]) Consider two counters that count modulo- 2^n , whose state bits are $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{n-1})$, respectively. Let i be an input. When $i = 0$ both counters keep their values; when $i = 1$ both counters increment their values by one modulo 2^n . Suppose that the initial state is $\{s \neq t\}$, and the bad state is $\{s = t\}$. The work [14] argues that any safe inductive invariant for the usual IC3 must contain at least 2^n lemmas. Furthermore, there is a much smaller safe inductive invariant for the *Reverse IC3* that consists of $2n$ lemmas required to represent $s = t$ in CNF. With innards, there is an inductive invariant consisting of a single lemma, $(z = 1)$, for the innard $z = (s \neq t)$. \square

Example 3 (SEC) This example illustrates a sequential equivalence checking problem between an original and a retimed [12] design. Let the “original part” of the design consist of latches x_1, \dots, x_n and inputs i_1, \dots, i_n , such that $init(x_k) = 0$ and $next(x_k) = i_k$ for $k = 1, \dots, n$, and a net

$z = x_1 \oplus \dots \oplus x_n$. Let the “retimed part” of the design consist of a net $u = i_1 \oplus \dots \oplus i_n$ and a latch v with $init(v) = 0$ and $next(v) = u$. Let the bad state be $\{z \neq v\}$. The only safe inductive invariant is $v \leftrightarrow (x_1 \oplus \dots \oplus x_n)$, that consists of 2^n lemmas in CNF. With innards, an alternative invariant requires only two lemmas: $v \rightarrow z$ and $z \rightarrow v$. \square

Example 4 This example is motivated by the benchmark *rast-pl6* from HWMCC’20. The design contains latches x_1, \dots, x_n and y_1, \dots, y_n , and innards $z_1 = x_1 \wedge y_1, \dots, z_n = x_n \wedge y_n$. Assume that the lemma $C = (z_1 \vee \dots \vee z_n)$ over innards is inductive. Representing C in CNF over latches requires 2^n lemmas. For example, for $n = 3$, the lemma $(z_1 \vee z_2 \vee z_3)$ is equivalent to 8 lemmas $(x_1 \vee x_2 \vee x_3)$, $(x_1 \vee x_2 \vee y_3)$, $(x_1 \vee y_2 \vee x_3)$, $(x_1 \vee y_2 \vee y_3)$, $(y_1 \vee x_2 \vee x_3)$, $(y_1 \vee x_2 \vee y_3)$, $(y_1 \vee y_2 \vee x_3)$, $(y_1 \vee y_2 \vee y_3)$. \square

IV. FINDING LEMMAS OVER INNARDS

In this section, we provide an overview of our approach (Sec. IV-A), followed by an algorithm for extending IC3 lemmas with innards (Sec. IV-B), and finally an algorithm for inductive generalization in the presence of innards (Sec. IV-C).

A. The overall approach

Traditional IC3 learns lemmas by inductively generalizing negations of blocked proof obligations. Both proof obligations and lemmas are over latches. These lemmas are then added to IC3’s inductive trace and used in future predecessor and convergence checks. In our approach, proof obligations are also over latches (exactly the same as in traditional IC3), however, we extend learning lemmas over both latches and innards. Our results apply to arbitrary innards, but for simplicity of presentation in the rest of the paper, we restrict to input-free innards, calling them simply innards. Note that unlike [7], our restriction is for presentation only. Throughout the section, we use the following running example.

Example 5 Let w, x, y, z be latches and i be an input. Let

$$\begin{aligned} Init &\triangleq w \wedge x \wedge y \wedge z \\ Tr &\triangleq (w' = \neg w) \wedge (x' = w) \wedge (y' = w) \wedge \\ &\quad (g = x \wedge y) \wedge (h = g \wedge i) \wedge (z' = h) \end{aligned}$$

This design has two gates: $g = x \wedge y$ and $h = g \wedge i$, where g is input-free and h depends on the input i . Hence, the set of (input-free) innards is $\{g\}$. \square

We extend IC3 to reason about innards in the initial state and the next state. To this end, let Tr_{inn} be the part of the transition relation that defines innards, and let $\widehat{Init} \triangleq Init \wedge Tr_{inn}$ and $\widehat{Tr} \triangleq Tr \wedge Tr_{inn}'$. In Example 5,

$$\begin{aligned} Tr_{inn} &= (g = x \wedge y) \quad \widehat{Init} = Init \wedge (g = x \wedge y) \\ \widehat{Tr} &= Tr \wedge (g' = x' \wedge y') \end{aligned}$$

where g' is a copy of g in “the next state”. The following definition extends relative induction [1] to lemmas over latches and innards.

¹<https://github.com/agurfinkel/innard-benchmarks>.

<p>Input: Frame k, Lemma C over latches, s.t. C is inductive relative to F_k</p> <p>Output: Lemma C_2 over latches and innards, s.t. C_2 is inductive relative to F_k</p> <ol style="list-style-type: none"> 1 $C_1 \leftarrow \text{ExtendLemma}(C)$ 2 $C_2 \leftarrow \text{InductivelyGeneralize}(k, C_1)$ 3 return C_2
--

Fig. 1. Procedure LearnAdditionalLemma.

Definition 1 A lemma C over latches and innards is inductive relative to a set of lemmas G iff (i) $\widehat{\text{Init}} \Rightarrow C$, and (ii) $G \wedge \widehat{\text{Tr}} \wedge C \Rightarrow C'$.

Def. 1 generalizes the original definition: if a lemma C over latches is relatively inductive in the original sense of [1], then C is also relatively inductive by Def. 1. In what follows, by *relatively inductive*, we always mean Def. 1. Continuing our running example, let $C = (w \vee x)$ (note that C is over latches), and $C_1 = (w \vee x \vee g)$ (note that C_1 is over latches and innards). Then, both C and C_1 are inductive relative to $G = \top$. Note that $\widehat{\text{Init}} \Rightarrow C$, $\top \wedge \widehat{\text{Tr}} \wedge C \Rightarrow C'$, $\widehat{\text{Init}} \Rightarrow C_1$, $\top \wedge \widehat{\text{Tr}} \wedge C_1 \Rightarrow C'_1$ hold.

The following lemma shows that using relatively inductive (in the sense of Def. 1) lemmas in IC3 is sound.

Lemma 1 (Soundness) For any lemma C over latches and innards, if $\widehat{\text{Init}} \Rightarrow C$ and $F_k \wedge \widehat{\text{Tr}} \wedge C \Rightarrow C'$ hold, then C includes $R_{\leq k+1}$ (all the states reachable in up to $k+1$ steps from $\widehat{\text{Init}}$). In particular, C can be added to IC3's inductive trace up to the frame $k+1$.

Our approach of learning lemmas over innards is a form of inductive generalization. Each time that IC3 blocks a proof obligation and learns a (relatively inductive) lemma over latches, we generalize it into an (additional) lemma over latches and innards. The overall algorithm LearnAdditionalLemma is shown in Fig. 1. We give a high-level overview of LearnAdditionalLemma, while the details of key functions are described in later sections. The approach consists of two steps:

Step 1: The procedure ExtendLemma extends lemma C (over latches) to a lemma $C_1 = C \vee C_0$ (over latches and innards) such that $\text{Tr}_{\text{inn}} \Rightarrow (C \Leftrightarrow C_1)$, i.e. C and C_1 are equivalent modulo Tr_{inn} . The details are in section IV-B. For instance, in our example lemmas $C = (w \vee x)$ and $C_1 = (w \vee x \vee g)$ are equivalent, given that $g = x \wedge y$. Indeed, modulo Tr_{inn} : $(w \vee x \vee g) \equiv (w \vee x \vee (x \wedge y)) \equiv (w \vee x)$. It also follows (see Lemma 1) that C_1 remains relatively inductive.

Step 2: The procedure InductivelyGeneralize inductively generalizes C_1 by removing literals, while prioritizing removal of latches (the original literals of C), and more generally trying to leave only the “interesting” innards. The details are in section IV-C. In our example, lemma $C_1 = (w \vee x \vee g)$ can be generalized to $C_2 = (w \vee g)$.

By construction, it follows that C_2 remains inductive relative to F_k . Moreover, as $\text{Tr}_{\text{inn}} \Rightarrow (C \Leftrightarrow C_1)$, and $C_2 \Rightarrow C_1$, then C_2 is potentially stronger than the original lemma C (but the converse might not hold). In our example, $C_2 = (w \vee g)$ is equivalent to $(w \vee (x \wedge y)) = (w \vee x) \wedge (w \vee y)$, i.e. the lemma C_2 over latches and innards represents two different lemmas over latches only. It is also interesting to note that while the original lemma C was over latches $\{w, x\}$, the “additional” lemma $(w \vee y)$ is over a different set of latches $\{w, y\}$.

Whenever ExtendLemma does not add any innards to C , the procedure LearnAdditionalLemma stops immediately, without calling InductivelyGeneralize. However, note that even when ExtendLemma adds new literals, it is possible that InductivelyGeneralize removes them, resulting in the original lemma C ! When LearnAdditionalLemma returns a lemma C_2 that is different from C , C_2 is also added to IC3's inductive trace (up to frame F_{k+1}), and hence is also used in future predecessor and pushing queries.

B. Extending lemmas with innards

The procedure ExtendLemma receives a lemma C over latches as input and returns a lemma C_1 over latches and innards as output. It iteratively finds innards z such that $\text{Tr}_{\text{inn}} \Rightarrow (z \Rightarrow C)$ and replaces C with $C \vee z$. It works as follows: instead of searching for an innard z that implies C , it searches for all innards $\neg z$ that are implied by $\neg C$ and take their negations. Specifically, given a lemma $C = (c_1 \vee \dots \vee c_m)$, we set each $c_i \in C$ to 0 and find which innards are implied by constant propagation in the Tr_{inn} part of the netlist. The algorithm for constant propagation in a netlist is standard and is not presented here.

Going back to our running example, given a lemma $C = (w \vee x)$, we are looking for innards implied by the partial assignment $(w = 0) \wedge (x = 0)$. Since $g = x \wedge y$, by propagation we obtain that $g = 0$. Thus, modulo Tr_{inn} , $g \Rightarrow C$, and hence C is equivalent to $(C \vee g) = (w \vee x \vee g)$. Note that by not considering input-free innards only (recall, we consider only input-free innards for simplicity of presentation), then, by propagation, we would also obtain that $h = (g \wedge i) = 0$. This would allow us to extend C to $(C \vee g \vee h) = (w \vee x \vee g \vee h)$. The following lemma follows by construction.

Lemma 2 Given lemma C over latches, the procedure ExtendLemma returns a lemma C_1 over latches and innards such that $\text{Tr}_{\text{inn}} \Rightarrow (C_1 \Leftrightarrow C)$.

Corollary 1 Let C and C_1 be lemmas over latches and innards respectively, such that (i) C is inductive relative to some G , and (ii) $\text{Tr}_{\text{inn}} \Rightarrow (C_1 \Leftrightarrow C)$. Then, C_1 is also inductive relative to G .

We remark that extending lemmas with literals that imply it is closely related to *asymmetric literal addition* [15] in SAT. We also remark that the condition that the original lemma C is over latches is not essential, and ExtendLemma can be used to extend lemmas that already have innards in them. This may be potentially useful for additional IC3 extensions.

Input: Frame k , lemma C over latches and innards, s.t. C is inductive relative to F_k

Output: (Inductively generalized) lemma $C_2 \subseteq C$ over latches and innards, s.t. C_2 is inductive relative to F_k

```

1  $C \leftarrow \text{SortLemma}(C)$  //  $C = \{c_1, \dots, c_n\}$ 
2 for  $i = 1, \dots, n$  do
3   if  $c_i$  has already been removed from  $C$  then
4     // do nothing
5   else if  $\text{Tr}_{inn} \Rightarrow ((C \setminus c_i) \Leftrightarrow C)$  then
6      $C \leftarrow C \setminus c_i$ 
7   else if  $\widehat{\text{Init}} \Rightarrow C \setminus c_i$  and
8      $F_k \wedge \widehat{\text{Tr}} \wedge (C \setminus c_i) \Rightarrow (C \setminus c_i)'$  then
9      $C \leftarrow C \setminus c_i$ 
10    for  $j = i + 1, \dots, n$  do
11      if  $c_j$  not used in the above proofs then
12         $C \leftarrow C \setminus c_j$ 
13    break
14 return  $C$ 

```

Fig. 2. Procedure InductivelyGeneralize: inductively generalizes lemmas over latches and innards.

C. Inductively generalizing lemmas with innards

Inductive generalization in traditional IC3 starts with a relatively inductive lemma C over latches (satisfying the conditions $\text{Init} \Rightarrow C$ and $F_k \wedge \text{Tr} \wedge C \Rightarrow C'$ with respect to a given frame F_k), and attempts to remove literals from C as long as C remains relatively inductive. The same procedure can be immediately applied to a lemma over latches and innards, once $\widehat{\text{Init}}$ and $\widehat{\text{Tr}}$ are used instead of Init and Tr , respectively. However, we found that a naive application of inductive generalization gives poor results. In most cases, it simply removes the innards that were previously added by `ExtendLemma`, and therefore, ends up with the original lemma over latches. Moreover, regular inductive generalization does not exploit possible dependencies between innards.

Fig. 2 shows a variant of inductive generalization that is better suited for generalizing lemmas over innards. The first step (line 1), consists of sorting the nets in the lemma, from the nets that we want to remove most to the nets that we want to remove least. In particular, we want to prioritize removal of latches, so as to obtain a different lemma that we started with. In our current implementation, we sort the nets by their *logic level*, so that latches have the lowest level and deeper nets in general have higher level. This way deeper nets are considered “more interesting” and the algorithm attempts to remove shallower nets first. Other heuristics can be considered as well, e.g., sorting the nets by the *size of the supporting logic*, or even dynamic heuristics that measure the *activity* of a net in previously generalized lemmas.

The main loop (lines 3–12) corresponds to inductive generalization in regular IC3: essentially, we remove literals of C one by one, as long as C remains relatively inductive. We

provide a detailed description of one iteration of the loop. Suppose that c_i is the literal under consideration.

1) Note that *multiple* literals can be removed from C in a single iteration of the loop (this optimization is also present in regular IC3 inductive generalization), so at the start of the iteration (line 3), we check if c_i has already been removed. If so, nothing needs to be done.

2) Lines 4–5 correspond to a special optimization that exploits dependencies between innards: in some cases, we can detect that c_i can be removed without requiring a SAT query. For instance, c_i can be removed when one of the following conditions holds:

- (i) $c_i = a \wedge b$, with $a \in C$,
- (ii) $c_i = a \vee b$, with $a, b \in C$, or
- (iii) there is an innard $d \in C$ with $d = c_i \vee b$.

For example, suppose that $C = (a \vee c \vee d)$ and $\{d = (b \vee c)\} \in \text{Tr}_{inn}$. Then, modulo Tr_{inn} , $C \Leftrightarrow (C \setminus c)$, i.e. $(a \vee c \vee d)$ can be replaced by $(a \vee d)$. This closely corresponds to *hidden literal elimination* technique in SAT [16], and can be viewed as the inverse of the argument used in `ExtendLemma`.

3) Line 6 checks whether c_i can be removed using two SAT-queries. One query checks the validity of $\widehat{\text{Init}} \Rightarrow (C \setminus c_i)$, by checking whether $\widehat{\text{Init}} \wedge \neg(C \setminus c_i)$ is unsatisfiable. The other query checks the validity of $F_k \wedge \widehat{\text{Tr}} \wedge (C \setminus c_i) \Rightarrow (C \setminus c_i)'$ by checking whether $F_k \wedge \widehat{\text{Tr}} \wedge (C \setminus c_i) \wedge \neg(C \setminus c_i)'$ is unsatisfiable. If both of these queries are unsatisfiable, c_i can be removed.

4) IC3 has the following standard optimization based on considering which of the literals of $(C \setminus c_i)$ were potentially required for unsatisfiability: if $c_j \in C$ was not required for either checks, then c_j can be removed. This is typically implemented by passing the literals of $\neg(C \setminus c_i)$ via SAT *assumptions* and analyzing the set of *conflicting assumptions*; a mechanism supported by most modern SAT-solvers, following MINISAT [17]. However, simply removing all non-required literals regardless of their order in C is more likely to remove the “more interesting” literals that we want to keep. So, our variant of this optimization (lines 8–12) only removes non-required literals with respect to the order. As an example, suppose that $C = (c_1 \vee c_2 \vee c_3 \vee c_4 \vee c_5 \vee c)$ (in this order), and that only the literals c_4 and c were potentially required for unsatisfiability queries involving $C \setminus c_1$. In addition to removing c_1 , we also remove c_2 and c_3 , but not c_5 , and at the end of the iteration of the loop, $C = (c_4 \vee c_5 \vee c)$. Intuitively, this works better because leaving c_5 in the lemma increases the chances to remove c_5 and to leave c (and not vice versa) on the following iterations of the loop. Lastly, in most cases an assumption-based SAT-solver applies assumptions in the order as they are given, hence, the assumptions appearing earlier are more likely to remain (while later assumptions are more likely to be removed). Therefore, when performing the SAT queries, we *reverse* the order of assumption literals, for instance when checking whether c_1 can be removed from $C = (c_1 \vee c_2 \vee c_3 \vee c_4 \vee c_5 \vee c)$, the assumptions are ordered from c to c_2 (and not from c_2 to c).

Note that during the regular inductive generalization (i.e.,

when computing the original lemma over latches) it is beneficial to make multiple passes over the main loop (lines 3–12). However, when generalizing lemmas over innards, performing multiple passes has not proven to be useful, so we only perform a single pass.

Lemma 3 *Given a lemma C_1 over latches and innards, the `InductivelyGeneralize` procedure returns a lemma C_2 that is relatively inductive with respect to F_k .*

Going back to our running example, suppose that $C_1 = (w \vee x \vee g)$ is inductive relative to $F_k = \top$. The procedure `SortLemma` is not likely to change the order of nets, as the latches already appear first. On the first iteration of the main loop, we attempt to remove w , but this fails as the SAT query $\top \wedge \widehat{Tr} \wedge (x \vee g) \wedge \neg x' \wedge \neg g'$ is satisfiable. On the second iteration, we attempt to remove x , and succeed, reducing C_1 to $(w \vee g)$. Finally, we attempt to remove g , which again fails. The final lemma returned by the algorithm is $C_2 = (w \vee g)$.

V. EXPERIMENTS

In this section, we present our experimental results. The techniques described in this paper are implemented in the IBM formal verification tool *Rulebase: SixthSense Edition* [18]. In what follows, we denote by IC3 the default variant of IC3 used by the tool (see [6]), and by IC3-INN the variant with the additional learning of lemmas over innards. For these experiments, we restrict to input-free innards. Table I summarizes the experiments. The table contains the benchmark set (explained in detail later), the number of instances in this set, time-limit per instance, and the data on performance of IC3 and IC3-INN. All the instances either are or expected to be unsatisfiable. For both IC3 and IC3-INN, we list the number of solved instances, and in parentheses – the number of uniquely solved instances (that is, not solved by the other configuration), and the cumulative runtime in seconds. Next, we describe each benchmark set in detail.

A. IBM-AOD-SEC

This set of benchmarks comes from checking sequential equivalence between two designs in the Aspect Oriented Design flow at IBM. This SEC problem is very challenging, and is traditionally solved as described in [8], [9], using speculative reduction to reduce the problem into multiple simpler (but still hard) sub-problems. These are then solved using a dedicated engine configuration consisting of combinational rewriting, k -induction, localization, and, eventually, a proof-based technique like IC3. Historically, Interpolation (IMC) was used for the final step. Generally IMC works well, but unfortunately, it’s not stable – small changes in the design or in the solving configuration significantly affect verification times. While trying to find an alternative configuration, it was discovered that IC3 performs very poorly, while IC3-INN significantly outperforms all other approaches.

In total, there are 3605 sub-problems. Each sub-problem contains 1–45 properties, 11–165 state elements, 126–2290 inputs, and 754–15924 gates. The (input-free) innards on

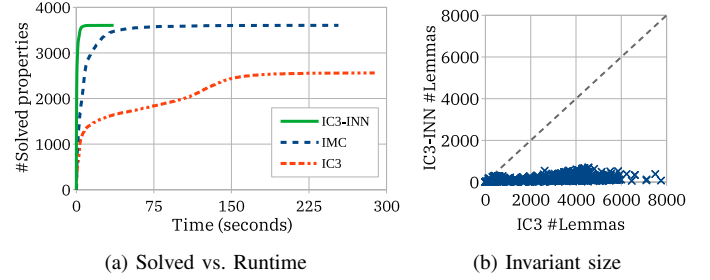


Fig. 3. Performance of IC3 and IC3-INN on AOD SEC benchmarks.

average constitute 3% of the gates. For this experiment, we run both IC3 and IC3-INN with a time-limit of 300 seconds per problem. Referring to Table I, regular IC3 performs very poorly: it can solve only 2562 of the sub-problems and times out in the 1043 remaining cases. On the other hand, IC3-INN performs extremely well: it can solve all of the problems, with the maximum run-time being only 36 seconds. Interestingly, IMC performs much better than IC3 on this set of problems and is also able to solve all problems (albeit about 13 times slower than IC3-INN). See the cactus plot in Fig. 3a for the detailed comparison between IC3, IC3-INN, and IMC.

A further comparison consists of comparing the number of lemmas in the safe inductive invariants discovered by IC3 and IC3-INN respectively. The scatter plot Fig. 3b shows this data for the 2562 instances solved by both configurations. We can see that IC3-INN discovers invariants that are significantly more compact, with the inductive invariants discovered by IC3-INN being on average 12× smaller than the invariants discovered by IC3. This partially explains the success of IC3-INN compared to IC3 on this set of benchmarks.

We also give data on the effectiveness of `LearnAdditionalLemma`, averaged across all 3605 test-cases. On average, the original lemma C (over latches) has 7 latches; `ExtendLemma` adds 10 innards; `InductivelyGeneralize` shrinks the lemma to 2 latches and 1 innards. The average logic level of innards is 7. Thus, `LearnAdditionalLemma` is able to produce significantly shorter lemmas using deep innards in the design.

Unfortunately, this benchmark set is proprietary and cannot be publicly released at this time.

B. 6s119-SEC, 6s22-SEC

Inspired by the success of IC3-INN on internal IBM benchmarks, we tried to manually create similar test-cases starting from publicly available benchmarks. Specifically, we have taken several HWMCC designs, and created problems to check sequential equivalence between the original design and the retimed design [12]. We have further applied the SEC flow described above, consisting of breaking the main problem into multiple sub-problems using speculative reduction. It turns out that creating interesting benchmark sets in this way is non-trivial: in many cases the speculatively reduced problems turn out to be very easy, in many other cases some of these speculatively reduced problems turn out to be satisfiable (in

TABLE I
SUMMARY OF EXPERIMENTAL RESULTS

benchmarks	#instances	time-limit per instance	IC3 solved (unique)	IC3 time	IC3-INN solved (unique)	IC3-INN time
IBM-AOD-SEC	3 605	300	2 562 (0)	424 885	3 605 (1 043)	2 465
6s119-SEC	364	600	364 (0)	2 906	364 (0)	1 207
6s22-SEC	310	600	262 (22)	32 701	278 (38)	24 774
AES-SEC	16	3 600	13 (0)	11 186	15 (2)	5 601
HWMCC11	278	3 600	277 (6)	40 186	272 (1)	55 557
HWMCC17	76	3 600	76 (0)	7 963	76 (0)	11 221
HWMCC20	192	3 600	190 (5)	35 907	187 (2)	41 448

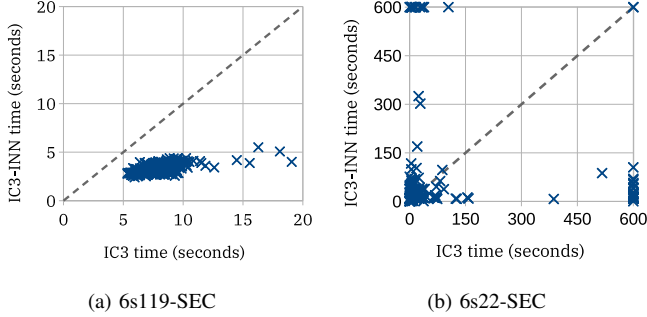


Fig. 4. Runtime of IC3 and IC3-INN on 6s119-SEC and 6s22-SEC.

the real SEC flow this would trigger refinement and another speculative reduction). Nevertheless, we have created two benchmark sets 6s22-SEC and 6s119-SEC, available at <https://github.com/agurfinkel/innard-benchmarks>. The set 6s119-SEC consists of 364 rather easy problems, so that both IC3 and IC3-INN can solve all of them within 600 seconds, with IC3-INN being about $2.4\times$ faster. The set 6s22-SEC consists of 310 problems, out of which IC3 can solve 262 problems and IC3-INN can solve 278 within 600 seconds. Please refer to Table I. Again, IC3-INN performs better than IC3, and is on average $1.3\times$ faster. A more precise comparison is given in scatter plots in Fig. 4. A detailed comparison against IMC is not included as on both sets of problems IMC performs significantly worse than either IC3 or IC3-INN (for instance, within 600 seconds IMC cannot solve 64 out of 364 problems even for the easy set 6s119-SEC).

C. Other SEC benchmarks; AES-SEC

As far as we know, there are no publicly available large SEC benchmark sets. HWMCC competitions do include several SEC benchmarks. However, in general we do not know which benchmarks come from SEC or what kind of application they represent. We believe it would be valuable to have a dedicated repository for SEC benchmarks.

The AES-SEC benchmark set was used in [13]. We have obtained this set from the authors of [13] in BTOR format, and translated it to AIGER. The AIGER benchmarks are available at <https://github.com/agurfinkel/innard-benchmarks>. In total, there are 16 problems, 12 of which turn out to be very easy for both IC3 and IC3-INN. Out of the 4 remaining

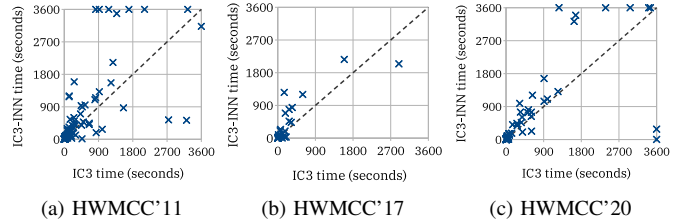


Fig. 5. Runtime of IC3 and IC3-INN on HWMCC benchmarks.

problems, IC3 can solve 1, and IC3-INN can solve 3. Please see Table I for details.

D. HWMCC benchmarks

We have run extensive experiments on the single-property benchmarks from HWMCC'11, HWMCC'17 and HWMCC'20 competitions (for the latter, we used the benchmarks in the AIGER format). In each case, we run simple combinational reductions prior to running IC3, and used the time-limit of 3 600 seconds. In Table I, we only report data for passing benchmarks that were solved either by IC3 or IC3-INN. In general, IC3-INN performs worse than IC3 both in terms of the number of properties solved and the total runtime. Detailed comparisons are presented as scatter plots in Fig. 5.

Table II presents data for 4 selected benchmarks. The benchmark *rast-p16* is very interesting: regular IC3 times out, yet IC3-INN solves the testcase in just 2 seconds. Furthermore, this benchmark was solved by relatively few tools in the HWMCC'20 competition. By closely examining the lemmas learned by IC3-INN exposed the pattern from Example 4 from Section III. In other words, IC3-INN learns lemmas over innards, each equivalent to a very large number of lemmas over latches. This potentially explains the success of IC3-INN in this case. Another noteworthy benchmark is *zipversa_composecrc_prf-p10*, which IC3-INN solves under 5 minutes, and which was solved only by one tool in the HWMCC'20 competition. The other two benchmarks exposed a certain inefficiency in our current implementation of IC3-INN. One can check that there are significantly more innards in the selected test-cases (and in HWMCC test-cases in general) as compared to IBM-AOD-SEC designs. The procedure `InductivelyGeneralize` starts taking a significant portion of the overall runtime, which negatively

TABLE II
SELECTED DESIGNS FROM HWMCC’20

Benchmark	#gates	#innards	IC3 time	IC3-INN time
rast-p16	3 019	332	timed-out	2
zipversa...prf-p10	1 688	694	timed-out	282
h_RCU	920	442	3 410	timed-out
dspfilters_fastfir...p45	21 301	5 289	2 381	timed-out

affects performance of IC3 when the lemmas over innards do not seem to help.

VI. RELATED AND FUTURE WORK

The technique presented in this paper can be viewed as an extension of regular IC3 that simply learns an additional lemma during inductive generalization. As such, it is reasonably easy to integrate it in an existing IC3 implementation. The main technical point being replacing *Init* by \widehat{Init} and *Tr* by \widehat{Tr} in IC3’s SAT queries. The key difference with other inductive generalization schemes (see for instance [3]) is that we are able to learn lemmas over both state variables and internal nets, which, in some cases, may exponentially reduce the size of the inductive invariant.

Backes and Riedel [7] also exploit internal nets in the design. However, the two approaches are very different: [7] uses input-free innards to generalize proof obligations (POBs), while we use arbitrary innards to generalize lemmas. Additionally, [7] uses only *input-free* innards (and, in fact, only the nets on the *boundary* between input-free and non input-free parts of the netlist), while we use all internal nets. Even more importantly, in our work the decision of which innards to include in the lemma was based on the ability to inductively generalize this lemma and not whether the innards are “boundary” or not. Above notwithstanding, it is interesting to combine the two approaches, i.e., to allow both proof-obligations and lemmas over internal nets. It is also interesting to more carefully integrate our approach with Quip [6]. Quip uses negations of lemmas as proof obligations, which would also introduce innards into POBs.

Another very interesting direction for further research is to extend the approach to learn lemmas over signals that are not present in the original netlist. Our framework allows such an extension: by including additional logic into the netlist (that is, creating additional innards), we would be able to learn lemmas over this new logic (even if this new logic is not in the cone-of-influence of the original problem!). This is closely related to implicit predicate abstraction of Tonetta et al. [19] that is used to lift propositional IC3 to SMT-based logics.

Finally, we believe that there is a lot of room to improve the current implementation. Currently, when there are many innards in the design, the procedure *InductivelyGeneralize* may require a large number of SAT queries, and, hence, may take a considerable portion of the overall runtime. Possibly, one can find better heuristics of which innards to consider (e.g., only to consider innards

with high logic level, or only to consider *higher-priority* innards), or find more efficient procedures to perform inductive generalization (e.g., instead of the top-down approach that removes literals one can consider a bottom-up approach that adds literals). In the worst-case, if learning additional lemmas takes a considerable amount of time, but does not seem useful, the technique can be simply turned off.

A further extension of our approach is to allow lemmas to be arbitrary formulas, not restricted to clauses in CNF. This is commonly done in SMT-based extensions of IC3 algorithms. For example, Sally [20] uses arbitrary SMT-formulas as lemmas, and Spacer [21] uses clauses over complex First Order signature. However, these techniques are difficult to port efficiently in the context of Hardware Model Checker since they rely on dynamic cnfization that is common in SMT-solvers but not in SAT-solvers.

VII. CONCLUSION

Currently, IC3 is unquestionably the most effective technique for formal symbolic model checking. It has received a lot of research attention, and has been extended in variety of ways including better inductive generalization, better lemma management, and search direction. However, one significant hidden limitation remains – IC3 is limited to learning inductive invariants in CNF over the latches (i.e., state variables) of the design. Therefore, IC3 cannot be effective for any design whose invariant has no concise CNF representation. No improvements in core IC3 parts can solve this problem.

In this paper, we propose to address this limitation by extending IC3 to learn lemmas not only over latches, but also over internal signals, that we call *innards*. We show learning lemmas over innards is a natural generalization of *inductive generalization*. Instead of simply dropping literals to strengthen the lemma, we propose to replace literals by internal signals that are forced by them. We also propose several improvements to a naive strategy that lead to significantly improved performance.

Our work is motivated by a specialized set of Sequential Equivalence Checking (SEC) benchmarks at IBM. These benchmarks have been traditionally difficult for IC3, but not for Interpolation (IMC). However, the performance of interpolation was not stable – being affected by small changes in the verification flow. Our new implementation excels on these benchmarks and leads to an order of magnitude improvement in performance.

Unfortunately, similar performance gains do not manifest on the publicly available HWMCC benchmarks that are the de-facto metric for academic model checking research. We believe this shows deficiency in the currently available benchmarks. Techniques that might be effective in industry might be missed by researchers since they do not perform well on these benchmarks. To remedy this, we identified some publicly available benchmarks, and created new benchmarks based on SEC flow, that illustrate the advantage of our technique. We hope this can stimulate further research and improvements to IC3.

In the current work, we assume that the design is fixed, and use internal signals that are already available. We think that this opens an interesting direction by allowing IC3 to change the design by synthesizing new innards that are useful for a current verification run. This brings IC3 and interpolation much closely together, and also paves way for bringing algorithms from hardware verification to software verification, and/or to word level.

ACKNOWLEDGMENTS


The authors would like to thank Jason Baumgartner, Robert Kanzelman, Raj Kumar Gajavelly, Ziv Nevo, Hongce Zhang, Sharad Malik, Alan Mishchenko, and Baruch Sterin. This work was supported, in part, by Individual Discovery Grant from the Natural Sciences and Engineering Research Council of Canada and IBM Faculty Fellowship.

REFERENCES

- [1] A. R. Bradley, “Sat-based model checking without unrolling,” in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87. [Online]. Available: https://doi.org/10.1007/978-3-642-18275-4_7
- [2] N. Eén, A. Mishchenko, and R. K. Brayton, “Efficient implementation of property directed reachability,” in *International Conference on Formal Methods in Computer-Aided Design, FMCAD ’11, Austin, TX, USA, October 30 - November 02, 2011*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 125–134. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2157675>
- [3] A. Griggio and M. Roveri, “Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking,” *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, no. 6, pp. 1026–1039, Jun 2016.
- [4] Y. Vizel, O. Grumberg, and S. Shoham, “Lazy abstraction and sat-based reachability in hardware model checking,” in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, G. Cabodi and S. Singh, Eds. IEEE, 2012, pp. 173–181. [Online]. Available: <http://ieeexplore.ieee.org/document/6462570/>
- [5] Z. Hassan, A. R. Bradley, and F. Somenzi, “Better generalization in IC3,” in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 157–164. [Online]. Available: <http://ieeexplore.ieee.org/document/6679405/>
- [6] A. Gurfinkel and A. Ivrii, “Pushing to the top,” in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, R. Kaivola and T. Wahl, Eds. IEEE, 2015, pp. 65–72.
- [7] J. D. Backes and M. D. Riedel, “Using cubes of non-state variables with property directed reachability,” in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, E. Macii, Ed. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 807–810. [Online]. Available: <https://doi.org/10.7873/DATE.2013.171>
- [8] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, “Scalable sequential equivalence checking across arbitrary design transformations,” in *24th International Conference on Computer Design (ICCD 2006), 1-4 October 2006, San Jose, CA, USA*. IEEE, 2006, pp. 259–266. [Online]. Available: <https://doi.org/10.1109/ICCD.2006.4380826>
- [9] H. Mony, J. Baumgartner, A. Mishchenko, and R. K. Brayton, “Speculative reduction-based scalable redundancy identification,” in *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, L. Benini, G. D. Micheli, B. M. Al-Hashimi, and W. Müller, Eds. IEEE, 2009, pp. 1674–1679. [Online]. Available: <https://doi.org/10.1109/DATE.2009.5090932>
- [10] R. Brayton, N. Een, and A. Mishchenko, “Using speculation for sequential equivalence checking,” in *21st International Workshop on Logic & Synthesis, IWLS 2012, 2012*.
- [11] R. Dureja, J. Baumgartner, R. Kanzelman, M. Williams, and K. Y. Rozier, “Accelerating Parallel Verification via Complementary Property Partitioning and Strategy Exploration,” in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. Haifa, Israel: IEEE/ACM, Sep. 2020.
- [12] A. Kuehlmann and J. Baumgartner, “Transformation-based verification using generalized retiming,” in *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001. Proceedings*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Springer, 2001, pp. 104–117. [Online]. Available: https://doi.org/10.1007/3-540-44585-4_10
- [13] H. Zhang, W. Yang, G. Fedyukovich, A. Gupta, and S. Malik, “Synthesizing environment invariants for modular hardware verification,” in *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020. Proceedings*, ser. Lecture Notes in Computer Science, D. Beyer and D. Zufferey, Eds., vol. 11990. Springer, 2020, pp. 202–225. [Online]. Available: https://doi.org/10.1007/978-3-030-39322-9_10
- [14] T. Seufert and C. Scholl, “Sequential verification using reverse PDR,” in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2017, Bremen, Germany, February 8-9, 2017*, D. Große and R. Drechsler, Eds. Shaker Verlag, 2017, pp. 79–90.
- [15] 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
- [16] 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
- [17] 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
- [18] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, “Scalable automated verification via expert-system guided transformations,” in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings*, ser. Lecture Notes in Computer Science, A. J. Hu and A. K. Martin, Eds., vol. 3312. Springer, 2004, pp. 159–173. [Online]. Available: https://doi.org/10.1007/978-3-540-30494-4_12
- [19] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “IC3 modulo theories via implicit predicate abstraction,” in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 46–61. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_4
- [20] D. Jovanovic and B. Dutertre, “Property-directed k -induction,” in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 85–92.
- [21] A. Komuravelli, A. Gurfinkel, and S. Chaki, “SMT-Based Model Checking for Recursive Programs,” 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*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 17–34.


Single Clause Assumption without Activation Literals to Speed-up IC3

Nils Froleys

nils.froleys@jku.at 

Johannes Kepler University, Linz, Austria

Armin Biere

biere@cs.uni-freiburg.de 

Albert-Ludwigs-University, Freiburg, Germany

Abstract—We extend the well-established assumption-based interface of incremental SAT solvers to clauses, allowing the addition of a temporary clause that has the same lifespan as literal assumptions. Our approach is efficient and easy to implement in modern CDCL-based solvers. Compared to previous approaches, it does not come with any memory overhead and does not slow down the solver due to disabled activation literals, thus eliminating the need for algorithms like IC3 to restart the SAT solver. All clauses learned under literal and clause assumptions are safe to keep and not implicitly invalidated for containing an activation literal. These changes increase the quality of learned clauses, resulting in better generalization for IC3. We implement the extension in the SAT solver CaDiCaL and evaluate it with the IC3 implementation in the model checker ABC. Our experiments on the benchmarks from a recent hardware model checking competition show a speedup for the average SAT call and a reduction in number of calls per verification instance, resulting in a substantial improvement in model checking time.

INTRODUCTION

Modern SAT solving is based on Conflict-Driven Clause Learning (CDCL) [1]. Many applications require solving a sequence of related SAT problems incrementally [2], [3], making use of inprocessing techniques [4], [5], [6] that make modern SAT solvers so efficient. Among those applications is the symbolic model checking algorithm IC3. In contrast to other incremental SAT-based techniques, such as bounded model checking (BMC) [7], [8] and k-induction [9], [10], IC3 does not rely on unrolling the transition function. As a result the SAT queries that IC3 poses are significantly smaller and faster to solve. However, the number of queries that IC3 makes over the course of one model checking procedure is significantly higher. We illustrate the kind of queries that IC3 makes in the following example.

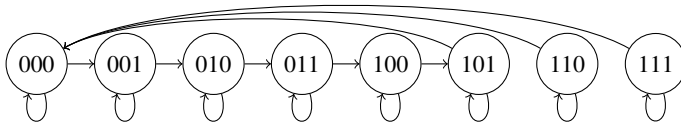


Fig. 1. Transition system

Consider the transition system of a three-bit ($b_2b_1b_0$) counter, encoding integers up to seven, in Fig. 1. Non-deterministically, the counter is incremented, remains unchanged or is reset to zero after reaching five. Suppose we want to ensure that starting at state zero, all states with

values greater than five are unreachable. A typical query asks “is state six reachable from any other state?”, expressed as $SAT?[T \wedge (\neg b_2 \vee \neg b_1 \vee b_0) \wedge b'_2 \wedge b'_1 \wedge \neg b'_0]$, where T encodes the transition system for one step from $b_2b_1b_0$ to $b'_2b'_1b'_0$. It is unsatisfiable, telling us that state six is in fact unreachable. We can try to generalize this result to a set of states by considering a *cube* – an assignment to a subset of variables. The query $SAT?[T \wedge (\neg b_2 \vee b_0) \wedge b'_1 \wedge \neg b'_0]$ is satisfiable because state two can be reached from state one and $SAT?[T \wedge (\neg b_2 \vee b_0) \wedge b'_2 \wedge \neg b'_0]$ is satisfiable due to the transition from state three to state four. However, the query $SAT?[T \wedge (\neg b_2 \vee \neg b_1) \wedge b'_2 \wedge b'_1]$ is unsatisfiable, allowing us to conclude that all states in the cube $b_2 \wedge b_1$ are not reachable from outside the cube. We can use that insight to strengthen T by adding $\neg b'_2 \vee \neg b'_1$ to all future queries. This is in contrast to the clauses we previously added for only one query.

The popular assumption-based interface pioneered by MiniSat [2], [8] allows the user to specify a set of literals that are assumed to be true and picked by the solver as the first decisions. This allows us to add the assumption that a state is within a certain cube after the transition ($b'_2 \wedge b'_1$), however we still need to assume an additional clause encoding that the state is currently not within said cube ($\neg b_2 \vee \neg b_1$). The most common way to implement clause assumption, is to simulate the desired behavior using activation literals [8], [11]. Let C be a clause to add temporarily and a , the activation literal, a free variable, *i.e.*, it does not occur in the formula. By adding $C \vee a$ to the formula and assuming $\neg a$, we achieve the same as adding C to the formula. After a solution is found, the clause a is added, effectively removing C from the formula.

The problem with IC3 specifically, is the large number of queries made over the course of a single verification procedure. After a few hundred calls the activation literals clutter up the variable space and slow down the SAT solvers propagation. The common solution to this problem is to fully restart the SAT solver by replacing it with a fresh instance periodically, thus also deleting all learned clauses and heuristic scores. How to schedule these restarts in IC3 specifically, has been the topic of a full journal paper [12]. Using the technique presented in this paper, restarts are not necessary at all. Additionally learned clauses are safe to keep and will not contain an activation literal, which would make them useless for future calls.

Other approaches to clause assumption have been explored: The logic solver Satire [13] supports pseudo-Boolean and

other constraints. It records the dependencies of learned constraints explicitly, thus allowing the deletion of arbitrary clauses. In the SMT community, an interface based on pushing and popping on the assertion stack is prevalent [14]. Since constraints are removed in order, it is possible to mark a point in the data structures that maintain learned knowledge and remove everything past it, when a pop operation is executed. The first implementation of IC3 [15] used the SAT solver Zchaff [16]. It assigns an additional 32-bit integer to each clause. When learning a clause the bits of all dependencies are combined. The user can delete a group of clauses with a certain bit. This approach mostly simulates the use of activation literals and comes with a significant memory overhead.

This paper presents an extension of the prevalent assumption mechanism to additionally allow the assumption of a single clause, called *constraint* in the following. The extension can be implemented by a simple modification to the decision mechanism in a CDCL-based SAT solver. We implemented it in under 100 lines of code in the state-of-the-art SAT solver CaDiCaL. To evaluate our implementation we modify the IC3 engine in the model checker ABC to use CaDiCaL and clause assumption. As a first result, the changes simplify SAT solver usage and eliminate the need for restarts as well as some book-keeping for activation literals. An empirical evaluation on the 2019 hardware model checking competition [17] benchmark set shows that ABC spends less time outside of computing SAT queries, the number of queries per verification is reduced and the average SAT call is faster. Overall using clause assumptions yields a substantial speedup in verification time.

INCREMENTAL SAT AND IC3

An *incremental SAT* solver solves a series of related formulas efficiently. It communicates with an application integrating it through an *interface* such as IPASIR [11]. It is implemented by all solvers participating in the incremental library track of the SAT Competition since 2015. The popular solver MiniSat along with all of its incremental descendants implement something very similar. We describe the relevant subset:

- `add(lit)` Add a literal to the current clause or if it equals 0, add the clause to the formula.
- `assume(lit)` Assume the literal to be true for the next solving attempt.
- `solve()` Return SAT if an assignment exists satisfying the formula and all assumptions, otherwise UNSAT.
- `val(lit)` Valid in SAT-case. Return the truth value of a literal in the satisfying assignment.
- `failed(lit)` Valid in UNSAT-case. Return *true* if the literal was assumed and used to prove unsatisfiability.

A prominent applications of incremental SAT-solving is the symbolic model checking algorithm IC3 by Bradley [15]. Given a transition system and a property P , IC3 tries to prove that it is not possible to reach a state that violates the property. It maintains a sequence of *frames* F_0, F_1, \dots, F_k , each frame F_i is a formula encoding an overapproximation of the set of states reachable in at most i steps. The frames are refined by adding additional clauses until one of the frames contains all reachable

states and none violates the property or a counterexample is found. Each frame has its own SAT solver instance that is initialized with an encoding of the transition function and updated with the new frame clauses.

The solvers are used almost exclusively to answer queries for predecessors of the form $SAT?[T \wedge F_i \wedge \neg s \wedge s']$, where T is the transition function and s is a cube. To refine the frames, a state s in the last frame that violates the property is identified with the query $SAT?[F_k \wedge \neg P]$. If no such state exists, a new frame is appended, otherwise IC3 tries to prove that the state is not actually reachable. The frames are queried for predecessors until an initial state is reached, thus producing a counterexample, or one of the frames returns unsat. In the latter case `failed` can be used to generalize the unreachable state to a cube, the negation of which is added to the frame. IC3 is guaranteed to eventually terminate with two consecutive frames containing the same set of states.

ASSUMING CLAUSES

Our main contribution is an extension to incremental SAT solvers that allows the assumption of an additional clause, called *constraint*, which is only valid during the next satisfiability query. Two functions are added to the interface:

- `constrain(lit)` Adds a literal to constraint. If a finalized constraint exists, delete it. If the literal equals zero, finalizes the current constraint.
- `constraint_failed()` Valid in *UNSAT* case. Return whether constraint was used to prove unsatisfiability.

Our approach is similar to the idea of model elimination [18]. We modify the decision heuristic to restrict the search to assignments that satisfy the constraint. The modified decision procedure is outlined in Fig. 2. The function `decide` is called initially at decision level 0. Decisions assigned to the trail are propagated outside of the function to assign truth values. Whenever a conflict arises, the decision level decreases and the assignments are backtracked [1]. Every assumption has a fixed decision level. In the case where an assumption is already satisfied, a *pseudo* decision level is introduced. Otherwise if an assumed literal is assigned to false at this point, the assignment is the result of propagating other assumptions together with original or learned clauses. Therefore the formula is proven unsatisfiable under the current assumptions if line 4 is reached.

At the first decision level after all assumptions have been assigned, three cases need to be considered: if one of the literals in the constraint is already satisfied, the search is not restricted. Otherwise one of the literals is picked as a decision to satisfy the constraint. In line 13 a variable selection heuristic can be used to pick the most promising literals first, similarly to [19], [20]. In the case where all literals are assigned to false, they are implied by the assumptions, thus cannot be assigned differently. The formula is therefore declared unsatisfiable under the assumptions and the constraint. This might only happen after additional clauses have been learned.

This approach to handle assumptions was pioneered by MiniSat [2]. It has been improved upon by collectively propagating the assumptions, using trail saving between incremental

```

decide ()
1  if level < lassumptionsl
2    ℓ = assumptions[level]
3    if val(ℓ) = false
4      analyzeFinal()
5    else if val(ℓ) = true
6      level++ // pseudo decision level
7    else trail[level++] = ℓ
8  else if level = lassumptionsl
9    unassignedLit = 0
10   for ℓ in constraint
11     if val(ℓ) = true
12       level++ // pseudo decision level
13     else if val(ℓ) = unassigned
14       unassignedLit = ℓ
15   if unassignedLit = 0
16     analyzeFinalConstraint() // cannot be satisfied
17   else trail[level++] = unassignedLit
18 else
19   ℓ = literalSelectionHeuristic()
20   trail[level++] = ℓ

```

Fig. 2. Algorithm `decide` picks the next decision to propagate.

calls [21] or factoring out assumptions [22]. These techniques can be combined with the presented constraint mechanism.

Modern SAT solvers not only report unsatisfiability as a result, but also allow the user to query whether a particular assumption failed, *i.e.*, was used to prove unsatisfiability. This concept, introduced as `analyzeFinal` by MiniSat [23], is essential for the efficiency of many applications. If an original or learned clause is inconsistent with the assumptions, the last assumption picked as a decision is already assigned to false. Using a simple breadth-first search, the reasons for this assignment can be traced back through the implication graph [1]. The assumptions at the leaves of the search tree are marked as failed. In line 16, a similar search is initialized with the negation of every literal in the constraint. Thus, all assumptions necessary to prove unsatisfiability of the constraint in conjunction with the formula are marked as failed.

EXPERIMENTS

We implemented the constraint interface in CaDiCaL [24] version 1.3.1. To increase confidence in the correctness of the SAT solver and its new extension, we used the model-based tester [25] that is integrated with CaDiCaL. It generates random sequences of API calls including assumptions and constraints together with random configurations for the solver. The returned models and failed assumption sets are checked for correctness. We ran the tester on 8 cores for multiple days to validate 1.2 billion test runs.

To evaluate our approach, we integrated CaDiCaL into the bit-level model checker ABC¹ [26], replacing the integrated version of MiniSat [2]. There are two places where activation literals are used in ABC. The first is an alternative implementation of cube generalization, that is not used in the default configuration. In fact, it seems to not work correctly in the default version of ABC¹. The other usage of activation literals is in the function that implements the predecessor query $SAT?[T \wedge F_i \wedge \neg s \wedge s']$. The transition function T and the frame F_i will only be extended with additional clauses, the cube s however changes at each query. The next-step cube s' is in conjunction with the rest of the formula and therefore translates to a set of unit clauses that can be implemented with assumptions. To combat the slowdown due to unused activation literals cluttering up the variable space, ABC replaces the SAT solver with a new instance after adding 300 activation literals. Using the extended interface, the negated cube $\neg s$ can be added as a constraint, thus eliminating the restarts.

We tested five configurations: the original version of ABC (Og), disabled SAT solver restarts (Di), a version with CaDiCaL as backend using activation literals (Ca) and one also using CaDiCaL but the new constraint interface instead of activation literals (Co). As an additional result we present a slight modification to the last configuration that defers model reconstruction [6] in the SAT-case and failed literal collection in the UNSAT-case until a model or a failed literal is queried respectively (De). Using a heuristic to pick the literals from the constraint has not been successful. ABC uses a priority metric to order the literals of the cube s by default. Using this order for the constraint turned out to be superior to the heuristics available in CaDiCaL.

Our evaluation follows the principles laid out in SAT manifesto v1.0. [27]. The source code used for the evaluation and the generated log files are available on our website². The experiments are run in parallel on 32 nodes of our cluster. Each node has 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 4 instances of ABC to every node. The time limit is set to 1 hour of wall-clock time, memory is limited to 30GB per instance. The memory limit is the only aspect that differs from the setup used in the hardware model checking competition. However, the maximum memory consumption was observed to be below 1.5GB.

The evaluation is based on the benchmark set used in the 2019 model checking competition [17]. It contains 219 instances, 15 of which we removed because they were not solved by any tested configuration. We use PAR-2 scoring to compare the configurations. PAR-2 assigns the runtime in seconds or twice the time limit (7200) if an instance was not solved. The other columns list additional measurements for the two configurations using CaDiCaL, one with activation literals (Ca) and the other using constraints instead (Co). The number of restarts is zero if constraints are used and

¹commit f87c8b4

²<http://fmv.jku.at/assumingclauses>

TABLE I
EXPERIMENTAL RESULTS.

	PAR-2					Res.	Calls		TpC	
	Di	Og	Ca	Co	De	Ca	Ca	Co	Ca	Co
Mean	80	46	16	8.93	8.21	61	19	15	0.61	0.51
beemTele6Int	136	7200	53	181	101	520	157	574	0.24	0.27
toyLock4	7200	483	1731	357	359	7459	2251	1098	0.42	0.25
visArraysField5	7200	1.6	0.58	51	34	1	1	113	0.53	0.41
nan	208	421	163	158	140	1381	420	423	0.29	0.32
beemColl6Int	241	258	322	133	108	398	123	91	2.31	1.24
cal110	213	168	130	110	122	191	59	42	1.96	2.39
cal109	179	197	102	117	86	110	34	44	2.71	2.44
cal93	186	136	121	118	140	206	63	58	1.69	1.8
cal94	127	160	115	95	131	171	52	41	1.94	2.1
cal100	112	42	67	67	54	148	45	44	1.23	1.29
cal131	46	44	77	58	60	136	42	35	1.58	1.41
cal146	47	39	71	42	38	131	41	23	1.51	1.55
cal136	34	46	59	43	35	100	31	23	1.62	1.59
cal128	52	38	46	37	40	99	31	25	1.29	1.27
beemExit5Int	51	17	26	16	15	357	110	86	0.18	0.15
cal134	38	47	50	48	36	79	25	26	1.72	1.57
cal132	39	36	48	42	32	83	26	24	1.57	1.54
cal144	30	34	41	33	42	64	20	17	1.7	1.64
beemLampNat5Int	26	23	23	35	31	193	61	102	0.28	0.3
cal89	16	14	32	33	25	68	22	18	1.23	1.6
beemRether4Bstep	13	4.29	16	7.16	6.99	91	29	13	0.42	0.49
beemBrp2Int	16	5.1	3.6	0.76	0.74	86	29	7	0.08	0.07
beemFrogs2Bstep	2.47	2.53	12	5.59	4.74	31	10	4	1.12	1.27
beemAdding5Int	1.78	3.9	2.07	1.12	1.09	53	17	11	0.08	0.07
visArraysTwo	1.35	2.89	3.89	0.57	0.55	99	30	5	0.09	0.07
Heap	2.02	1.9	3.38	1.68	1.63	57	22	13	0.11	0.09

Disable restarts, Original version of ABC, CaDiCaL backend, Constraint interface used, Defer model reconstruction

therefore not shown. Besides that, we list the number of SAT calls (in thousands), along with the average time per call in milliseconds. Table I presents the measured data for instances, where at least one configuration took more than two seconds, along with an average over all 204 instances.

Comparing the first two columns, it is evident that if activation literals are used, solver restarts are necessary. It has been suggested [12] that because the queries posed by IC3 are small but numerous, IC3 implementations should prefer faster SAT solvers to more powerful ones. Comparing the original with the CaDiCaL version shows that while using MiniSat is faster on a number of instances, using CaDiCaL seems to be an advantage on the harder instances. In fact, using the newer SAT solver, one additional instance can be verified. Over all instances a speedup of 2.82 is observed.

With the version using CaDiCaL and activation literals as a baseline, we observe a speedup of 1.84 when switching to constraints. The time spend outside the SAT solver is reduced to below 20%, by eliminating the actual SAT solver restarts and the repeated loading of the transition relation [28]. Beyond that, the average SAT call is 16% faster. This can partially be explained by the solver not being slowed down by activation literals. We conjecture that, more importantly, the “quality” of the learned clauses in the solvers database is higher. Since clauses are not deleted by restarts and none of the learned clauses are implicitly disabled for containing an activation literal, the solver can profit from shorter and more useful

clauses. Measuring this quality however, is outside the scope of this paper. An additional effect is that these clauses allow conflicts earlier in the search tree, resulting in fewer failed literals and thus allows for better generalization in IC3. This can explain why 21% fewer calls are made.

The last two columns listing PAR-2 scores reflect small changes in the solver. Deferring the model reconstruction results in an additional speedup of 9%, increasing the total speedup compared to the original version to 5.64.

CONCLUSION

We present a simple extension to the commonly used incremental SAT solver interface IPASIR that simplifies solver usage and is easy to implement by modern SAT solvers. The extension gives an alternative to the techniques described in the journal paper [12] and partially implemented in ABC. Our experiments using the new technique with ABC show a substantial improvement in model checking time. Compared to the original IC3 engine, our final implementation is more than five times faster.


Handling more than one constraint can be achieved by using a complete model elimination search over the constraints. This would however increase the implementation effort. Additionally, inprocessing techniques cannot be applied, therefore model elimination might be less effective than using activation literals, if the number of temporary clauses is high. We leave this investigation to future work.


Acknowledgements: This work is supported by the Austrian Science Fund (FWF); projects W1255-N23 / S11408-N23 as well as the LIT AI Lab funded by the State of Upper Austria.


REFERENCES

- [1] Marques-Silva, Joao and Lynce, Ines and Malik, Sharad, “Chapter 4. Conflict-Driven Clause Learning SAT Solvers,” in *Handbook of Satisfiability: Second Edition*, Biere, Armin and Heule, Marijn and van Maaren, Hans and Walsh, Toby, Ed. IOS Press, feb 2021.
- [2] Eén, Niklas and Sörensson, Niklas, “An Extensible SAT-Solver,” in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, Giunchiglia, Enrico and Tacchella, Armando, Ed. Berlin, Heidelberg: Springer, 2004, pp. 502–518.
- [3] Audemard, Gilles and Lagniez, Jean-Marie and Simon, Laurent, “Improving Glucose for Incremental SAT Solving with Assumptions,” in *Theory and Applications of Satisfiability Testing – SAT 2013*, ser. Lecture Notes in Computer Science, Järvisalo, Matti and Van Gelder, Allen, Ed. Berlin, Heidelberg: Springer, 2013, pp. 309–317.
- [4] Eén, Niklas and Biere, Armin, “Effective Preprocessing in SAT Through Variable and Clause Elimination,” in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, Bacchus, Fahiem and Walsh, Toby, Ed. Berlin, Heidelberg: Springer, 2005, pp. 61–75.
- [5] Järvisalo, Matti and Heule, Marijn J. H. and Biere, Armin, “Inprocessing Rules,” in *Automated Reasoning*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 355–370.
- [6] Fazekas, Katalin and Biere, Armin and Scholl, Christoph, “Incremental Inprocessing in SAT Solving,” in *Theory and Applications of Satisfiability Testing – SAT 2019*, ser. Lecture Notes in Computer Science, Janota, Mikoláš and Lynce, Inês, Ed. Cham: Springer International Publishing, 2019, pp. 136–154.
- [7] Biere, Armin and Cimatti, Alessandro and Clarke, Edmund and Zhu, Yunshan, “Symbolic Model Checking without BDDs,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, Cleaveland, W. Rance, Ed. Berlin, Heidelberg: Springer, 1999, pp. 193–207.
- [8] Eén, Niklas and Sörensson, Niklas, “Temporal Induction by Incremental SAT Solving,” *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 4, pp. 543–560, jan 2003.
- [9] Bjesse, Per and Claessen, Koen, “SAT-Based Verification without State Space Traversal,” in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, Hunt, Warren A. and Johnson, Steven D., Ed. Berlin, Heidelberg: Springer, 2000, pp. 409–426.
- [10] Sheeran, Mary and Singh, Satnam and Stålmarck, Gunnar, “Checking Safety Properties Using Induction and a SAT-Solver,” in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, Hunt, Warren A. and Johnson, Steven D., Ed. Berlin, Heidelberg: Springer, 2000, pp. 127–144.
- [11] Balyo, Tomáš and Biere, Armin and Iser, Markus and Sinz, Carsten, “SAT Race 2015,” *Artificial Intelligence*, vol. 241, pp. 45–65, dec 2016.
- [12] Cabodi, G. and Camurati, P. E. and Mishchenko, A. and Palena, M. and Pasini, P., “SAT Solver Management Strategies in IC3: An Experimental Approach,” *Formal Methods in System Design*, vol. 50, pp. 39–74, mar 2017.
- [13] Whitemore, J. and Kim, J. and Sakallah, K., “SATIRE: A New Incremental Satisfiability Engine,” in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, jun 2001, pp. 542–545.
- [14] Barrett, Clark and Stump, Aaron and Tinelli, Cesare and others, “The Smt-Lib Standard: Version 2.0,” in *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, England)*, vol. 13, 2010, p. 14.
- [15] Bradley, Aaron R., “SAT-Based Model Checking without Unrolling,” in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, Jhala, Ranjit and Schmidt, David, Ed. Berlin, Heidelberg: Springer, 2011, pp. 70–87.
- [16] Fu, Zhaohui and Marhajan, Yogesh and Malik, Sharad, “Zchaff Sat Solver,” 2004.
- [17] Preiner, Mathias and Biere, Armin, “Hardware Model Checking Competition 2019,” <http://fmv.jku.at/hwmc19/>, 2019.
- [18] Van Gelder, Allen, “Autarky Pruning in Propositional Model Elimination Reduces Failure Redundancy,” *Journal of Automated Reasoning*, vol. 23, no. 2, pp. 137–193, aug 1999.
- [19] Goldberg, Evgenii I. and Novikov, Yakov, “BerkMin: A Fast and Robust Sat-Solver,” in *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002)*, 4-8 March 2002, Paris, France. IEEE Computer Society, 2002, pp. 142–149.
- [20] Gershman, Roman and Strichman, Ofer, “HaifaSat: A New Robust SAT Solver,” in *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, ser. Lecture Notes in Computer Science, Ur, Shmuel and Bin, Eyal and Wolfsthal, Yaron, Ed., vol. 3875. Springer, 2005, pp. 76–89.
- [21] Hickey, Randy and Bacchus, Fahiem, “Speeding Up Assumption-Based SAT,” in *Theory and Applications of Satisfiability Testing – SAT 2019*, ser. Lecture Notes in Computer Science, Janota, Mikoláš and Lynce, Inês, Ed. Cham: Springer International Publishing, 2019, pp. 164–182.
- [22] Lagniez, Jean-Marie and Biere, Armin, “Factoring Out Assumptions to Speed Up MUS Extraction,” in *Theory and Applications of Satisfiability Testing – SAT 2013*, ser. Lecture Notes in Computer Science, Järvisalo, Matti and Van Gelder, Allen, Ed. Berlin, Heidelberg: Springer, 2013, pp. 276–292.
- [23] Eén, Niklas and Sörensson, Niklas, “MiniSat Page,” <http://minisat.se/>.
- [24] Biere, Armin, “Cadical, Lingeling, Plingeling, Treengeling and Yalsat Entering the Sat Competition 2018,” *Proceedings of SAT Competition*, pp. 14–15, 2017.
- [25] Artho, Cyrille and Biere, Armin and Seidl, Martina, “Model-Based Testing for Verification Back-Ends,” in *Tests and Proofs*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 39–55.
- [26] Brayton, Robert and Mishchenko, Alan, “ABC: An Academic Industrial-Strength Verification Tool,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, Touili, Tayssir and Cook, Byron and Jackson, Paul, Ed. Berlin, Heidelberg: Springer, 2010, pp. 24–40.
- [27] Biere, Armin and Järvisalo, Matti and Le Berre, Daniel and Meel, Kuldeep S. and Mengel, Stefan, “The SAT Practitioner’s Manifesto,” sep 2020.
- [28] Vizel, Y. and Grumberg, O. and Shoham, S., “Lazy Abstraction and SAT-Based Reachability in Hardware Model Checking,” in *2012 Formal Methods in Computer-Aided Design (FMCAD)*, oct 2012, pp. 173–181.

Logical Characterization of Coherent Uninterpreted Programs

Hari Govind V K 
University of Waterloo

Sharon Shoham 
Tel-Aviv University

Arie Gurfinkel 
University of Waterloo

Abstract—An uninterpreted program (UP) is a program whose semantics is defined over the theory of uninterpreted functions. This is a common abstraction used in equivalence checking, compiler optimization, and program verification. While simple, the model is sufficiently powerful to encode counter automata, and, hence, undecidable. Recently, a class of UP programs, called coherent, has been proposed and shown to be decidable. We provide an alternative, logical characterization, of this result. Specifically, we show that every coherent program is bisimilar to a finite state system. Moreover, an inductive invariant of a coherent program is representable by a formula whose terms are of depth at most 1. We also show that the original proof, via automata, only applies to programs over unary uninterpreted functions. While this work is purely theoretical, it suggests a novel abstraction that is complete for coherent programs but can be soundly used on *arbitrary* uninterpreted (and partially interpreted) programs.

I. INTRODUCTION

The theory of Equality with Uninterpreted Functions (EUF) is an important fragment of First Order Logic, defined by a set of functions, equality axioms, and congruence axioms. Its satisfiability problem is decidable. It is a core theory of most SMT solvers, used as a glue (or abstraction) for more complex theories. A closely related notion is that of Uninterpreted Programs (UP), where all basic operations are defined by uninterpreted functions. Feasibility of a UP computation is characterized by satisfiability of its path condition in EUF. UPs provide a natural abstraction layer for reasoning about software. They have been used (sometimes without explicitly being named), in equivalence checking of pipelined microprocessors [1], and equivalence checking of C programs [17]. They also provide the foundations of Global Value Numbering (GVN) optimization in many modern compilers [6], [8], [12].

Unlike EUF, reachability in UP is undecidable. That is, in the *lingua franca* of SMT, the satisfiability of Constrained Horn Clauses over EUF is undecidable. Recently, Mathur et al. [9], have proposed a variant of UPs, called *coherent uninterpreted program* (CUPs). The precise definition of coherence is rather technical (see Def. 3), but intuitively the program is restricted from depending on arbitrarily deep terms. The key result of [9] is to show that both reachability of CUPs and deciding whether an UP is coherent are decidable. This makes CUP an interesting infinite state abstraction with a *decidable* reachability problem.

Unfortunately, as shown by our counterexample in Fig. 4 (and described in Sec. VI), the key construction in [9] is incorrect. More precisely, the proofs of [9] hold only of

CUPs restricted to unary functions. In this paper, we address this bug. We provide an alternative (in our view simpler) proof of decidability and extend the results from reachability to arbitrary model checking. The case of non-unary CUPs is much more complex than unary. This is not surprising, since similar complications arise in related results on Uniform Interpolation [4] and Cover [5] for EUF.

Our key result is a logical characterization of CUP. We show that the set of reachable states (i.e., the strongest inductive invariant) of a CUP is definable by an EUF formula, over program variables, with terms of depth at most 1. That is, the most complex term that can appear in the invariant is of the form $v \approx f(\vec{w})$, where v and \vec{w} are program variables, and f a function.

This characterization has several important consequences since the number of such bounded depth formulas is finite. Decidability of reachability, for example, follows trivially by enumerating all possible candidate inductive invariants. More importantly from a practical perspective, it leads to an efficient analysis of *arbitrary* UPs. Take a UP P , and check whether it has a safe inductive invariant of bounded terms. Since the number of terms is finite, this can be done by implicit predicate abstraction [3]. If no invariant is found, and the counterexample is not feasible, then P is not a CUP. At this point, the process either terminates, or another verification round is done with predicates over deeper terms. Crucially, this does not require knowing whether P is a CUP apriori – a problem that itself is shown in [9] to be at least PSPACE.

We extend the results further and show that CUPs are bisimilar to a finite state system, showing, in particular, that arbitrary model checking for CUP (not just reachability) is decidable.

Our proofs are structured around a series of abstractions, illustrated in a commuting diagram in Fig. 1. Our key abstraction is the base abstraction α_b . It forgets terms deeper than depth 1, while maintaining all their consequences (by using additional fresh variables). We show that α_b is sound and complete (i.e., preserves all properties) for CUPs (while, sound, but not complete for UP). It is combined with a cover abstraction α_C , that we borrow from [5]. The cover abstraction ensures that reachable states are always expressible over program variables. It serves the purpose of existential quantifier elimination, that is not available for EUF. Finally, a renaming abstraction α_r is a technical tool to bound the occurrences of constants in abstract reachable states.

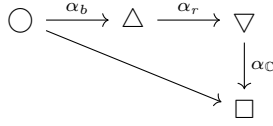


Fig. 1: Sequence of abstractions used in our proofs.

The rest of the paper is structured as follows. We review the necessary background on EUF in Sec. II. We introduce our formalization of UPs and CUPs in Sec. III. Sec. IV presents bisimulation inducing abstractions for UP. Sec. V presents our base abstraction and shows that it induces a bisimulation for CUPs. Sec. VI develops logical characterization for CUPs, presents our decidability results, and shows that a finite state abstraction of CUPs is computable. We conclude the paper in Sec. VII with summary of results and a discussion of open challenges and future work.

II. BACKGROUND

We assume that the reader is familiar with the basics of First Order Logic (FOL), and the theory of Equality and Uninterpreted Functions (EUF). We use $\Sigma = (\mathcal{C}, \mathcal{F}, \{\approx, \not\approx\})$ to denote a FOL signature with constants \mathcal{C} , functions \mathcal{F} , and predicates $\{\approx, \not\approx\}$, representing equality and disequality, respectively. A term is a constant or (well-formed) application of a function to terms. A literal is either $x \approx y$ or $x \not\approx y$, where x and y are terms. A formula is a Boolean combination of literals. We assume that all formulas are quantifier free unless stated otherwise. We further assume that all formulas are in Negation Normal Form (NNF), so negation is defined as a shorthand: $\neg(x \approx y) \triangleq x \not\approx y$, and $\neg(x \not\approx y) \triangleq x \approx y$. Throughout the paper, we use \bowtie to indicate a predicate in $\{\approx, \not\approx\}$. For example, $\{x \bowtie y\}$ means $\{x \approx y, x \not\approx y\}$. We write \perp for false, and \top for true. We do not differentiate between sets of literals Γ and their conjunction $(\bigwedge \Gamma)$. We write $\text{depth}(t)$ for the maximal depth of function applications in a term t . We write $\mathcal{T}(\varphi)$, $\mathcal{C}(\varphi)$, and $\mathcal{F}(\varphi)$ for the set of all terms, constants, and functions, in φ , respectively, where φ is either a formula or a collection of formulas. Finally, we write $t[x]$ to mean that the term t contains x as a subterm.

For a formula φ , we write $\Gamma \models \varphi$ if Γ entails φ , that is every model of Γ is also a model of φ . For any literal ℓ , we write $\Gamma \vdash \ell$, pronounced ℓ is *derived* from Γ , if ℓ is derivable from Γ by the usual EUF proof system \mathcal{P}_{EUF} .¹ By refutational completeness of \mathcal{P}_{EUF} , Γ is unsatisfiable iff $\Gamma \vdash \perp$.

Given two EUF formulas φ_1 and φ_2 and a set of constants $V \subseteq \mathcal{C}$, we say that the formulas are V -equivalent, denoted $\varphi_1 \equiv_V \varphi_2$, if, for all quantifier free EUF formulas ψ such that $\mathcal{C}(\psi) \subseteq V$, $(\varphi_1 \wedge \psi) \models \perp$ if and only if $(\varphi_2 \wedge \psi) \models \perp$.

Example 1 Let $\varphi_1 = \{x_1 \approx f(a_0, x_0), y_1 \approx f(b_0, y_0), x_0 \approx y_0\}$, $\varphi_2 = \{x_1 \approx f(a_0, w), y_1 \approx f(b_0, w)\}$, $\varphi_3 = \{x_1 \approx f(a_0, x_0), y_1 \approx f(b_0, y_0)\}$, and $V = \{x_1, y_1, a_0, b_0\}$. Then, $\varphi_1 \equiv_V \varphi_2$ but $\varphi_1 \not\equiv_V \varphi_3$. \square

¹Presented in our companion technical report [7].

$$\begin{aligned}
 \langle \text{stmt} \rangle &::= \mathbf{skip} \mid \langle \text{var} \rangle := \langle \text{var} \rangle \mid \langle \text{var} \rangle := f(\langle \vec{\text{var}} \rangle) \mid \\
 &\quad \mathbf{assume}(\langle \text{cond} \rangle) \mid \langle \text{stmt} \rangle ; \langle \text{stmt} \rangle \mid \\
 &\quad \mathbf{if}(\langle \text{cond} \rangle) \mathbf{then} \langle \text{stmt} \rangle \mathbf{else} \langle \text{stmt} \rangle \mid \\
 &\quad \mathbf{while}(\langle \text{cond} \rangle) \langle \text{stmt} \rangle \\
 \langle \text{cond} \rangle &::= \langle \text{var} \rangle = \langle \text{var} \rangle \mid \langle \text{var} \rangle \neq \langle \text{var} \rangle \\
 \langle \text{var} \rangle &::= x \mid y \mid \dots
 \end{aligned}$$

Fig. 2: Syntax of the programming language UPL.

While EUF does not admit quantifier elimination, it does admit elimination of constants while preserving quantifier free consequences. Formally, a *cover* [2], [4], [5] of an EUF formula φ w.r.t. a set of constants V is an EUF formula ψ such that $\mathcal{C}(\psi) \subseteq \mathcal{C}(\varphi) \setminus V$ and $\varphi \equiv_{\mathcal{C}(\varphi) \setminus V} \psi$. By [5], such ψ exists and is unique up to equivalence; we denote it by $\mathbb{C}V \cdot \varphi$.

III. UNINTERPRETED PROGRAMS

An *uninterpreted program* (UP) is a program in the *uninterpreted programming language* (UPL). The *syntax* of UPL is shown in Fig. 2. Let V denote a fixed set of program variables. We use lower case letters in a special font: x, y , etc. to denote individual variables in V . We write \vec{y} for a list of program variables. Function symbols are taken from a fixed set \mathcal{F} . As in [9], w.l.o.g., UPL does not allow for Boolean combination of conditionals and relational symbols.

The small step symbolic operational semantics of UPL is defined with respect to a FOL signature $\Sigma = (\mathcal{C}, \mathcal{F}, \{\approx, \not\approx\})$ by the rules shown in Fig. 3. A program *configuration* is a triple $\langle s, q, pc \rangle$, where s , called a *statement*, is a UP being executed, $q : V \rightarrow \mathcal{C}$ is a *state* mapping program variables to constants in \mathcal{C} , and pc , called the *path condition*, is a EUF formula over Σ . We use $\mathcal{C}(q) \triangleq \{c \mid \exists v \cdot q(v) = c\}$ to denote the set of all constants that represent current variable assignments in q . With abuse of notation, we use $\mathcal{C}(q)$ and q interchangeably. We write \equiv_q to mean $\equiv_{\mathcal{C}(q)}$.

For a state q , we write $q[x \mapsto x']$ for a state q' that is identical to q , except that it maps x to x' . We write $\langle e, q \rangle \Downarrow v$ to denote that v is the value of the expression e in state q , i.e., the result of substituting each program variable x in e with $q(x)$, and replacing functions and predicates with their FOL counterparts. The value of e is an FOL term or an FOL formula over Σ . For example, $\langle x = y, [x \mapsto x, y \mapsto y] \rangle \Downarrow x \approx y$.

Given two configurations c and c' , we write $c \rightarrow c'$ if c reduces to c' using one of the rules in Fig. 3. Note that there is no rule for **skip** – the program terminates once it gets into a configuration $\langle \mathbf{skip}, q, pc \rangle$.

Let $\mathcal{C}_0 = \{v_0 \mid v \in V\} \subseteq \mathcal{C}$ be a set of initial constants. In the initial state q_0 of a program, every variable is mapped to the corresponding initial constant, i.e., $q_0(v) = v_0$.

The operational semantics induces, for an UP P , a transition system $\mathcal{S}_P = \langle \mathcal{C}, c_0, \mathcal{R} \rangle$, where \mathcal{C} is the set of configurations, $c_0 \triangleq \langle P, q_0, \top \rangle$ is the initial configuration, and $\mathcal{R} \triangleq \{(c, c') \mid c \rightarrow c'\}$. A configuration c of P is *reachable*

$$\begin{array}{c}
\langle \text{skip} ; s, q, pc \rangle \rightarrow \langle s, q, pc \rangle \\
\hline
\langle s_1, q, pc \rangle \rightarrow \langle s'_1, q', pc' \rangle \\
\hline
\langle s_1 ; s_2, q, pc \rangle \rightarrow \langle s'_1 ; s_2, q', pc' \rangle \\
\hline
\langle c, q \rangle \Downarrow v \quad (pc \wedge v) \not\models \perp \\
\hline
\langle \text{assume}(c), q, pc \rangle \rightarrow \langle \text{skip}, q, pc \wedge v \rangle \\
\hline
\langle e, q \rangle \Downarrow v \quad x' \in \mathcal{C}(\Sigma) \text{ is fresh in } pc \\
\hline
\langle x := e, q, pc \rangle \rightarrow \langle \text{skip}, q[x \mapsto x'], pc \wedge x' = v \rangle \\
\hline
\langle \text{if } (c) \text{ then } s_1 \text{ else } s_2, q, pc \rangle \rightarrow \langle \text{assume}(c) ; s_1, q, pc \rangle \\
\hline
\langle \text{if } (c) \text{ then } s_1 \text{ else } s_2, q, pc \rangle \rightarrow \langle \text{assume}(\neg c) ; s_2, q, pc \rangle \\
\hline
\langle \text{while } (c) \text{ } s, q, pc \rangle \rightarrow \\
\langle \text{if } (c) \text{ then } (s ; \text{while } (c) \text{ } s) \text{ else skip}, q, pc \rangle
\end{array}$$

Fig. 3: Small step symbolic operational semantics of UPL, where $\neg c$ denotes $x \neq y$ when c is $x = y$, and $x = y$ when c is $x \neq y$.

if c is reachable from c_0 in S_P . We denote the set of all reachable configurations in S_P using $\text{Reach}(S_P)$. The set of all statements in the semantics of P , including the intermediate statements, are called *locations* of P , and are denoted by $\mathcal{L}(P)$. We often use P and S_P interchangeably.

Our semantics of UPL differs in some respects from the one in [9]. First, we follow a more traditional small-step operational semantics presentation, by providing semantics rules and the corresponding transition system. However, this does not change the semantics conceptually. More importantly, we ensure that the path condition remains satisfiable in all reachable configurations (by only allowing an assume statement to execute when it results in a satisfiable path condition). We believe this is a more natural choice that is also consistent with what is typically used in other symbolic semantics. UP reachability under our semantics coincides with the definition of [9].

Definition 1 (UP Reachability) Given an UP P , determine whether there exists a state q and a path condition pc s.t., the configuration $\langle \text{skip}, q, pc \rangle$ is reachable in P . \square

A certificate for unreachability of location s , is an inductive assertion map η (or an inductive invariant) s.t. $\eta(s) = \perp$.

Definition 2 (Inductive Assertion Map) Let $\Sigma_0 \triangleq (\mathcal{C}_0, \mathcal{F}, \{\approx, \not\approx\})$, be restriction of Σ to \mathcal{C}_0 . An *inductive assertion map* of an UP P , is a map $\eta : \mathcal{L}(P) \rightarrow \text{EUF}(\Sigma_0)$ s.t. (a) $\eta(P) = \top$, and (b) if $\langle s, q_0, \eta(s) \rangle \rightarrow \langle s', q', pc' \rangle$, then $pc' \models (\eta(s')[v_0 \mapsto q'(v) \mid v \in \mathcal{V}])$. \square

In [9], a special sub-class of UPs has been introduced with a decidable reachability problem.

Definition 3 (Coherent Uninterpreted Program [9]) An UP P is *coherent* (CUP) if all of the reachable configurations

1	$x := t;$	$x_0 \approx t_0$
2	$y := t;$	$x_0 \approx t_0 \wedge y_0 \approx t_0$
3	while ($c \neq d$) {	$x_0 \approx y_0$
4	$x := n(x);$	$x_0 \approx n(y_0) \wedge c_0 \not\approx d_0$
5	$y := n(y);$	$x_0 \approx y_0 \wedge c_0 \not\approx d_0$
6	$c := n(c);$	$x_0 \approx y_0$
7	}	
8	$x := f(a, x);$	$x_0 \approx f(a_0, y_0) \wedge c_0 \approx d_0$
9	$y := f(b, y);$	$(a_0 \approx b_0 \Rightarrow x_0 \approx y_0) \wedge c_0 \approx d_0$
10	assume ($a = b$);	$a_0 \approx b_0 \wedge x_0 \approx y_0 \wedge c_0 \approx d_0$
11	assume ($x \neq y$);	\perp

Fig. 4: An example CUP program and its inductive assertions.

of P satisfy the following two properties:

Memoizing for any configuration $\langle x := f(\vec{y}), q, pc \rangle$, if there is a term $t \in \mathcal{T}(pc)$ s.t. $pc \models t \approx f(q(\vec{y}))$, then there is $v \in \mathcal{V}$ s.t. $pc \models q(v) \approx t$.

Early assume for any configuration

$\langle \text{assume}(x = y), q, pc \rangle$, if there is a term $t \in \mathcal{T}(pc)$ s.t. $pc \models t \approx s$ where s is a superterm of either $q(x)$ or $q(y)$, then, there is $v \in \mathcal{V}$ s.t. $pc \models q(v) \approx t$. \square

Intuitively, memoization ensures that if a term is recomputed, then it is already stored in a program variable; early assumes ensures that whenever an equality between variables is assumed, any of their superterms that was ever computed is still stored in a program variable. Note that unlike the original definition of CUP in [9], we do not require the notion of an *execution*. The path condition accumulates the history of the execution in a configuration, which is sufficient.

Example 2 An example of a CUP is shown in Fig. 4. Some reachable states in the first iteration of the loop are shown below, where line numbers are used as locations, and pc_i stands for the path condition at line i :

$$\begin{aligned}
&\langle 2, q_0[x \mapsto x_1, y \mapsto y_1], x_1 \approx t_0 \wedge y_1 \approx t_0 \rangle \\
&\langle 6, q_0[x \mapsto x_2, y \mapsto y_2, c \mapsto c_1], pc_2 \wedge \\
&\quad c_0 \not\approx d_0 \wedge x_2 \approx n(x_1) \wedge y_2 \approx n(y_1) \wedge c_1 \approx n(c_0) \rangle \\
&\langle 9, q_0[x \mapsto x_3, y \mapsto y_3, c \mapsto c_1], pc_6 \wedge \\
&\quad c_1 \approx d_0 \wedge x_3 \approx f(a_0, x_2) \wedge y_3 \approx f(b_0, y_2) \rangle
\end{aligned}$$

The program is coherent because (a) no term is recomputed; (b) for the assume at line 10, the only superterms of a_0 and b_0 are $f(a_0, x_n)$ and $f(b_0, y_n)$, and they are stored in x and y , respectively; and (c) for the assume ($c_n = d_0$) introduced by the exit condition of the while loop, no superterms of c_n , d_0 are ever computed. The program does not reduce to **skip** (i.e., it does not reach a final configuration). Its inductive assertion map is shown in Fig. 4 (right). \square

Note that UP are closely related, but are not equivalent, to the Herbrand programs of [12]. While Herbrand programs use the syntax of UPL, they are interpreted over a fixed universe of Herbrand terms. In particular, in Herbrand programs $f(x) \approx g(x)$ is always false (since $f(x)$ and $g(x)$ have different top-level functions), while in UP, it is satisfiable.

IV. ABSTRACTION AND BISIMULATION FOR UP

In this section, we review abstractions for transition systems. We then define two abstraction for UP: cover and renaming, and show that they induce bisimulation. That is, for UP, these abstractions preserve all properties. Finally, we show a simple logical characterization result for UP to set the stage for our main results in the following sections.

Definition 4 Given a transition system $\mathcal{S} = (C, c_0, \mathcal{R})$ and a (possibly partial) abstraction function $\sharp : C \rightarrow C$, the induced *abstract transition system* is $\sharp(\mathcal{S}) = (C, c_0^\sharp, \mathcal{R}^\sharp)$, where

$$c_0^\sharp \triangleq \sharp(c_0) \\ \mathcal{R}^\sharp \triangleq \{(c_\sharp, c'_\sharp) \mid \exists c, c'. c \rightarrow c' \wedge c_\sharp = \sharp(c) \wedge c'_\sharp = \sharp(c')\}$$

We write $c \rightarrow^\sharp c'$ when $(c, c') \in \mathcal{R}^\sharp$. Note that \sharp must be defined for c_0 . \square

Throughout the paper, we construct several abstract transition systems. All transition systems considered are *attentive*. Intuitively, this means that their transitions do not distinguish between configurations that have q -equivalent path conditions. We say that two configurations $c_1 = \langle s, q, pc_1 \rangle$ and $c_2 = \langle s, q, pc_2 \rangle$ are equivalent, denoted $c_1 \equiv c_2$ if $pc_1 \equiv_q pc_2$.

Definition 5 (Attentive TS) A transition system $\mathcal{S} = (C, c_0, \mathcal{R})$ is *attentive* if for any two configurations $c_1, c_2 \in C$ s.t. $c_1 \equiv c_2$, if there exists $c'_1 \in C$ s.t. $(c_1, c'_1) \in \mathcal{R}$, then there exists $c'_2 \in C$, s.t. $(c_2, c'_2) \in \mathcal{R}$ and $c'_1 \equiv c'_2$ and vice versa. \square

Weak, respectively strong, preservation of properties between the abstract and the concrete transition systems are ensured by the notions of *simulation*, respectively *bisimulation*.

Definition 6 ([11]) Let $\mathcal{S} = (C, c_0, \mathcal{R})$ and $\sharp(\mathcal{S}) = (C, c_0^\sharp, \mathcal{R}^\sharp)$ be transition systems. A relation $\rho \subseteq C \times C$ is a *simulation* from \mathcal{S} to $\sharp(\mathcal{S})$, if for every $(c, c_\sharp) \in \rho$:

- if $c \rightarrow c'$ then there exists c'_\sharp such that $c_\sharp \rightarrow^\sharp c'_\sharp$ and $(c', c'_\sharp) \in \rho$.

$\rho \subseteq C \times C$ is a *bisimulation* from \mathcal{S} to $\sharp(\mathcal{S})$ if ρ is a simulation from \mathcal{S} to $\sharp(\mathcal{S})$ and $\rho^{-1} \triangleq \{(c_\sharp, c) \mid (c, c_\sharp) \in \rho\}$ is a simulation from $\sharp(\mathcal{S})$ to \mathcal{S} . We say that $\sharp(\mathcal{S})$ *simulates*, respectively *is bisimilar to*, \mathcal{S} if there exists a simulation, respectively, a bisimulation, ρ from \mathcal{S} to $\sharp(\mathcal{S})$ such that $(c_0, c_0^\sharp) \in \rho$. \square

We say that a bisimulation $\rho \subseteq C \times C$ is *finite* if its range, $\{\rho(c) \mid c \in C\}$, is finite. A finite bisimulation relates a (possibly infinite) transition system with a finite one.

Next, we define two abstractions for UP programs and show that they result in bisimilar abstract transition systems. The first abstraction eliminates all constants that are not assigned to program variables from the path condition, using the cover operation. The second abstraction renames the constants assigned to program variables back to the initial constants \mathcal{C}_0 . Both abstractions together ensure that all reachable configurations in the abstract transition system are defined over Σ_0 (i.e., the only constants that appear in states, as well as in path conditions, are from \mathcal{C}_0). There may still be infinitely many such

configurations since the depth of terms may be unbounded. We show that whenever the obtained abstract transition system has finitely many reachable configurations, the concrete one has an inductive assertion map that characterizes the set of reachable configurations.

Definition 7 (Cover abstraction) The cover abstraction function $\alpha_{\mathbb{C}} : C \rightarrow C$ is defined by

$$\alpha_{\mathbb{C}}(\langle s, q, pc \rangle) \triangleq \langle s, q, \mathbb{C}(\mathcal{C} \setminus \mathcal{C}(q)) \cdot pc \rangle \quad \square$$

Since $pc \equiv_q \mathbb{C}(\mathcal{C} \setminus \mathcal{C}(q)) \cdot pc$, the cover abstraction also results in a bisimilar abstract transition system.

Theorem 1 For any attentive transition system $\mathcal{S} = (C, c_0, \mathcal{R})$, the relation $\rho = \{(c, \alpha_{\mathbb{C}}(c)) \mid c \in \text{Reach}(\mathcal{S})\}$ is a bisimulation from \mathcal{S} to $\alpha_{\mathbb{C}}(\mathcal{S})$. \square

To introduce the renaming abstraction, we need some notation. Given a quantifier free formula φ , constants $a, b \in \mathcal{C}(\varphi)$ such that $a \neq b$, let $\varphi[a \mapsto b]$ denote $\varphi[b \mapsto x][a \mapsto b]$, where x is a constant not in $\mathcal{C}(\varphi)$. For example, if $\varphi = (a \approx c \wedge b \approx d)$, $\varphi[a \mapsto b] = (b \approx c \wedge x \approx d)$.

Given a path condition pc and a state q , let $r_0(pc, q)$ denote the formula obtained by renaming all constants in $\mathcal{C}(q)$ using their initial values. $r_0(pc, q) = pc[q(v) \mapsto v_0]$ for all $v \in V$ such that $q(v) \neq v_0$.

Definition 8 (Renaming abstraction) The renaming abstraction function $\alpha_r : C \rightarrow C$ is defined by

$$\alpha_r(\langle s, q, pc \rangle) \triangleq \langle s, q_0, r_0(pc, q) \rangle \quad \square$$

Theorem 2 For any attentive transition system $\mathcal{S} = (C, c_0, \mathcal{R})$, the relation $\rho = \{(c, \alpha_r(c)) \mid c \in \text{Reach}(\mathcal{S})\}$ is a bisimulation from \mathcal{S} to $\alpha_r(\mathcal{S})$. \square

Finally, we denote by $\alpha_{\mathbb{C}, r}$ the composition of the renaming and cover abstractions: $\alpha_{\mathbb{C}, r} \triangleq \alpha_{\mathbb{C}} \circ \alpha_r$ (i.e., $\alpha_{\mathbb{C}, r}(c) = \alpha_r(\alpha_{\mathbb{C}}(c))$). Since the composition of bisimulation relations is also a bisimulation, $\alpha_{\mathbb{C}, r}(\mathcal{S})$ is bisimilar to \mathcal{S} .

Theorem 3 (Logical Characterization of UP) If $\alpha_{\mathbb{C}, r}$ induces a finite bisimulation on an UP P , then, there exists an inductive assertion map η for P that characterizes the reachable configurations of P . \square

PROOF Define $\eta(s) \triangleq \bigvee \{pc \mid \langle s, q, pc \rangle \in \text{Reach}(\alpha_{\mathbb{C}, r}(P))\}$. Then, $\eta(s)$ is such an inductive assertion map. \blacksquare

Intuitively, Thm. 3 says that inductive invariant of UP, whenever it exists, can be described using EUF formulas over program variables. That is, any extra variables that are added to the path condition during program execution can be abstracted away (specifically, using the cover abstraction). There are, of course, infinitely many such invariants since the depth of terms is not bounded (only constants occurring in them). In the sequel, we systematically construct a similar result for CUP.

V. BISMULATION OF CUP

The first step in extending Thm. 3 to CUP is to design an abstraction function that bounds the depth of terms that appear in any reachable (abstract) state. It is easy to design such a function while maintaining soundness – simply forget literals that have terms that are too deep. However, we want to maintain precision as well. That is, we want the abstract transition system to be bisimilar to the concrete one. Just like cover abstraction, the base abstraction function also eliminates all constants that are not assigned to program variables. Unlike cover abstraction, the base abstraction does not maintain $\mathcal{C}(q)$ -equivalence of the path conditions, but, rather, forgets most literals that cannot be expressed over program variables.

In this section, we focus on the definition of the base abstraction and prove that it induces bisimulation for CUP. This result is used in Sec. VI, to logically characterize CUPs.

Intuitively, the base abstraction “truncates” the congruence graph induced by a path condition in nodes that have no representative in the set of constants assigned to the program variables (V in the following definition), and assigns to the truncated nodes fresh constants (from W in the following definition).

Congruence closure procedures for EUF use a *congruence graph* to concisely represent the deductive closure of a set of EUF literals [15], [16]. Here, we use a logical characterization of a congruence graph, called a *V-basis*. Let Γ be a set of EUF literals. A triple $\langle W, \beta, \delta \rangle$ is a V -basis of Γ relative to a set of constants V , written $\langle W, \beta, \delta \rangle \in \text{base}(\Gamma, V)$, iff (a) W is a set of fresh constants not in $\mathcal{C}(\Gamma)$, and β and δ are conjunctions of EUF literals; (b) $(\exists W \cdot \beta \wedge \delta) \equiv \Gamma$; (c) $\beta \triangleq \beta_{\approx} \cup \beta_{\not\approx} \cup \beta_{\mathcal{F}}$ and $\delta \triangleq \delta_{\approx} \cup \delta_{\not\approx} \cup \delta_{\mathcal{F}}$, where

$$\begin{aligned} \beta_{\approx} &\subseteq \{u \approx v \mid u, v \in V\} & \beta_{\not\approx} &\subseteq \{u \not\approx v \mid u, v \in V\} \\ \beta_{\mathcal{F}} &\subseteq \{v \approx f(\vec{w}) \mid v \in V, \vec{w} \subseteq V \cup W, \vec{w} \cap V \neq \emptyset\} \\ \delta_{\approx} &\subseteq \{w \approx u \mid w \in V \cup W, u \notin V \cup W\} \\ \delta_{\not\approx} &\subseteq \{u \not\approx w \mid u \in W, w \in W \cup V\} \\ \delta_{\mathcal{F}} &\subseteq \{v \approx f(\vec{w}) \mid v, \vec{w} \subseteq V \cup W, v \in V \Rightarrow \vec{w} \subseteq W\} \end{aligned}$$

(d) $\beta \wedge \delta \not\vdash v \approx w$ for any $v \in V, w \in W$; and (e) $\beta \wedge \delta \not\vdash w_1 \approx w_2$ for any $w_1, w_2 \in W$ s.t. $w_1 \neq w_2$.

Note that we represent both equalities and disequalities in the V -basis as common in implementations (but not in the theoretical presentations) of the congruence closure algorithm. Intuitively, V are constants in $\mathcal{C}(\Gamma)$ that represent equivalence classes in Γ , and W are constants added to represent equivalence classes that do not have a representative in V . A V -basis, of any satisfiable set Γ , is unique up to renaming of constants in W and ordering of equalities between constants in V .

Example 3 Let $\Gamma = \{x \approx f(a, v_1), y \approx f(b, v_2), v_1 \approx v_2\}$ and $V = \{a, b, x, y\}$. A V -basis of Γ is $\langle W, \beta, \delta \rangle$, where $W = \{w\}$, $\beta = \{x \approx f(a, w), y \approx f(b, w)\}$, $\delta = \{w \approx v_1, w \approx v_2\}$. Renaming w to w' is a different V -basis: $\langle W', \beta', \delta' \rangle \in \text{base}(\Gamma, V)$ where $W' = \{w'\}$, $\beta' = \beta[w \mapsto w']$ and $\delta' = \delta[w \mapsto w']$.

As another example, consider $\Gamma = \{x \approx f(a, p), x \approx f(a, n(p)), y = f(b, p), y = f(c, n(p))\}$ and $V = \{x, y, a, b, c\}$. A V -basis of Γ is $\langle W, \beta, \delta \rangle$, where $W = \{w_0, w_1\}$, $\delta_2 = \{w_0 \approx p, w_1 \approx n(w_0)\}$, and

$$\beta_2 = \left\{ \begin{array}{ll} x \approx f(a, w_0) & x \approx f(a, w_1) \\ y \approx f(b, w_0) & y \approx f(c, w_1) \end{array} \right\} \quad \square$$

While a basis maintains all consequences of Γ (since $(\exists W \cdot \beta \wedge \delta) \equiv \Gamma$), the V -base abstraction of Γ , defined next, is weaker. It preserves consequences of β only:

Definition 9 (V-base abstraction) The V -base abstraction α_V for a set of constants V , is a function between sets of literals s.t. for any sets of literals Γ and Γ' :

- (1) $\alpha_V(\Gamma) \triangleq \beta$, where $\langle W, \beta, \delta \rangle \in \text{base}(\Gamma, V)$,
- (2) if there exists a β s.t. $\langle W_1, \beta, \delta_1 \rangle \in \text{base}(\Gamma, V)$ and $\langle W_2, \beta, \delta_2 \rangle \in \text{base}(\Gamma', V)$, then $\alpha_V(\Gamma) = \alpha_V(\Gamma')$. \square

The second requirement of Def. 9 ensures that two formulas that have the same V -consequences, have the same V -abstraction. For example, for a set of constants $V = \{u, v\}$, the formulas $\varphi_1 = \{v \approx f(u, x)\}$ and $\varphi_2 = \{v \approx f(u, y)\}$, have the same V -base abstraction: $v \approx f(u, w)$. Note that at this point, we only require that α_V is well defined (for example, it does not have to be computable.)

We now extend V -base abstraction to program configuration, calling it simply *base abstraction*, since the set of preserved constants is determined by the configuration:

Definition 10 (Base abstraction) The base abstraction $\alpha_b : C \rightarrow C$ is defined for configurations $\langle s, q, pc \rangle \in C$, where pc is a *conjunction* of literals: $\alpha_b(\langle s, q, pc \rangle) \triangleq \langle s, q, \alpha_{\mathcal{C}(q)}(pc) \rangle$. \square

Namely, the base abstraction $\alpha_{\mathcal{C}(q)}$ applied to the path condition is determined by the state q in the configuration. We often write $\alpha_q(\varphi)$ as a shorthand for $\alpha_{\mathcal{C}(q)}(\varphi)$.

We are now in position to state the main result of this section. Given a CUP P , the abstract transition system $\alpha_b(\mathcal{S}_P) = (C, c_0^{\alpha_b}, \mathcal{R}^{\alpha_b})$ is bisimilar to the concrete transition system $\mathcal{S}_P = (C, c_0, \mathcal{R})$. Note that at this point, we do not claim that $\alpha_b(\mathcal{S}_P)$ is finite, or that it is computable. We focus only on the fact that the literals that are forgotten by the base abstraction do not matter for any future transitions. The key technical step is summarized in the following theorem:

Theorem 4 Let $\langle s, q, pc \rangle$ be a reachable configuration of a CUP P . Then,

- (1) $\langle s, q, pc \rangle \rightarrow \langle s', q', pc \wedge pc' \rangle$ iff $\langle s, q, \alpha_q(pc) \rangle \rightarrow \langle s', q', \alpha_q(pc) \wedge pc' \rangle$, and
- (2) $\alpha_{q'}(pc \wedge pc') = \alpha_{q'}(\alpha_q(pc) \wedge pc')$. \square

The proof of Thm. 4 is not complicated, but it is tedious and technical. It depends on many basic properties of EUF. We summarize the key results that we require in the following lemmas. The proofs of the lemmas are provided in our companion technical report [7].

We begin by defining a *purifier* – a set of constants sufficient to represent a set of EUF literals with terms of depth one.

Definition 11 (Purifier) We say that a set of constants V is a *purifier* of a constant a in a set of literals Γ , if $a \in V$ and for every term $t \in \mathcal{T}(\Gamma)$ s.t. $\Gamma \vdash t \approx s[a]$, $\exists v \in V$ s.t. $\Gamma \vdash v \approx t$.

For example, if $\Gamma = \{c \approx f(a), d \approx f(b), d \not\approx e\}$. Then, $V = \{a, b, c\}$ is a purifier for a , but not a purifier for b , even though $b \in V$.

In all the following lemmas, Γ , φ_1 , φ_2 are sets of literals; V a set constants; $a, b \in \mathcal{C}(\Gamma)$; $u, v, x, y \in V$; V is a purifier for $\{x, y\}$ in Γ , φ_1 , and in φ_2 ; $\beta = \alpha_V(\Gamma)$; and $\alpha_V(\varphi_1) = \alpha_V(\varphi_2)$.

Lemma 1 says that anything newly derivable from Γ and a new equality $a \approx b$ is derivable using superterms of a and b :

Lemma 1 *Let t_1 and t_2 be two terms in $\mathcal{T}(\Sigma)$ s.t. $\Gamma \not\vdash (t_1 \approx t_2)$. Then, $(\Gamma \wedge a \approx b) \vdash (t_1 \approx t_2)$, for some constants a and b in $\mathcal{C}(\Gamma)$, iff there are two superterms, $s_1[a]$ and $s_2[b]$, of a and b , respectively, s.t. (i) $\Gamma \vdash (t_1 \approx s_1[a])$, (ii) $\Gamma \vdash (t_2 \approx s_2[b])$, and (iii) $(\Gamma \wedge a \approx b) \vdash (s_1[a] \approx s_2[b])$.*

Lemma 2 and Lemma 3 say that all consequences of Γ that are relevant to V are present in $\beta = \alpha_V(\Gamma)$ as well.

Lemma 2 $(\Gamma \wedge x \approx y \vdash u \approx v) \iff (\beta \wedge x \approx y \vdash u \approx v)$.

Lemma 3 $(\Gamma \wedge x \approx y \vdash u \not\approx v) \iff (\beta \wedge x \approx y \vdash u \not\approx v)$.

Lemma 4 says that $\beta = \alpha_V(\Gamma)$ can be described using terms of depth one using constants in V .

Lemma 4 V is a purifier for $x \in V$ in β .

Lemma 5 says that α_V is idempotent.

Lemma 5 $\alpha_V(\Gamma) = \alpha_V(\alpha_V(\Gamma))$.

Lemma 6 and Lemma 7 say that α_V preserves addition of new literals and dropping of constants.

Lemma 6 $\alpha_V(\varphi_1 \wedge x \approx y) = \alpha_V(\varphi_2 \wedge x \approx y)$.

Lemma 7 If $U \subseteq V$, then

$$(\alpha_V(\varphi_1) = \alpha_V(\varphi_2)) \Rightarrow (\alpha_U(\varphi_1) = \alpha_U(\varphi_2))$$

Lemma 8 extends the preservation results to disequalities. V is a set of constants, $x, y \in V$. V is not required to be a purifier (as it was in the previous lemmas).

Lemma 8 $\alpha_V(\varphi_1 \wedge x \not\approx y) = \alpha_V(\varphi_2 \wedge x \not\approx y)$.

Lemma 9 extends the preservation results for equalities involving a fresh constant x' s.t. $x' \notin \mathcal{C}(\varphi_1) \cup \mathcal{C}(\varphi_2)$. $\vec{y} \subseteq V$, $V' = V \cup \{x'\}$, and $f(\vec{y})$ be a term s.t. there does not exists a term $t \in \mathcal{T}(\varphi_1) \cup \mathcal{T}(\varphi_2)$ s.t. $\varphi_1 \vdash t \approx f(\vec{y})$ or $\varphi_2 \vdash t \approx f(\vec{y})$.

Lemma 9

$$\alpha_{V'}(\varphi_1 \wedge x' \approx y) = \alpha_{V'}(\varphi_2 \wedge x' \approx y) \quad (1)$$

$$\alpha_{V'}(\varphi_1 \wedge x' \approx f(\vec{y})) = \alpha_{V'}(\varphi_2 \wedge x' \approx f(\vec{y})) \quad (2)$$

We are now ready to present the proof of Thm. 4:

PROOF (THEOREM 4) In the proof, we use $x = q(x)$, and $y = q(y)$. For part (1), we only show the proof for $s = \mathbf{assume}(x \bowtie y)$ since the other cases are trivial.

The only-if direction follows since $\alpha_q(pc)$ is weaker than pc . For the if direction, $pc \not\vdash \perp$ since it is part of a reachable configuration. Then, there are two cases:

- case $s = \mathbf{assume}(x = y)$. Assume $(pc \wedge x \approx y) \models \perp$. Then, $(pc \wedge x \approx y) \vdash t_1 \approx t_2$ and $pc \vdash t_1 \not\approx t_2$ for

some $t_1, t_2 \in \mathcal{T}(pc)$. By Lemma 1, in any new equality $(t_1 \approx t_2)$ that is implied by $pc \wedge (x \approx y)$ (but not by pc), t_1 and t_2 are equivalent (in pc) to superterms of x or y . By the early assume property of CUP, $\mathcal{C}(q)$ purifies $\{x, y\}$ in pc . Therefore, every superterm of x or y is equivalent (in pc) to some constant in $\mathcal{C}(q)$. Thus, $(pc \wedge x \approx y) \vdash u \approx v$ and $(pc \wedge x \approx y) \vdash u \not\approx v$ for some $u, v \in \mathcal{C}(q)$. By Lemma 2, $(\alpha_q(pc) \wedge x \approx y) \vdash u \approx v$. By Lemma 3, $(\alpha_q(pc) \wedge x \approx y) \vdash u \not\approx v$. Thus, $(\alpha_q(pc) \wedge x \approx y) \models \perp$.

- case $s = \mathbf{assume}(x \neq y)$. $(pc \wedge x \not\approx y) \models \perp$ if and only if $pc \vdash x \approx y$. Since $x, y \in \mathcal{C}(q)$, $\alpha_q(pc) \vdash x \approx y$. ■

For part (2), we only show the cases for assume and assignment statements, the other cases are trivial.

- case $s = \mathbf{assume}(x = y)$. Since $q' = q$, we need to show that $\alpha_q(pc \wedge x \approx y) = \alpha_q(\alpha_q(pc) \wedge x \approx y)$. From the early assumes property, $\mathcal{C}(q)$ purifies $\{x, y\}$ in pc . By Lemma 4, $\mathcal{C}(q)$ purifies $\{x, y\}$ in $\alpha_q(pc)$ as well. By Lemma 5, $\alpha_q(pc) = \alpha_q(\alpha_q(pc))$. By Lemma 6, $\alpha_q(pc \wedge x \approx y) = \alpha_q(\alpha_q(pc) \wedge x \approx y)$.
- case $s = \mathbf{assume}(x \neq y)$. Since $q' = q$, we need to show that $\alpha_q(pc \wedge x \not\approx y) = \alpha_q(\alpha_q(pc) \wedge x \not\approx y)$. By Lemma 5, $\alpha_q(pc) = \alpha_q(\alpha_q(pc))$. By Lemma 8, $\alpha_q(pc \wedge x \not\approx y) = \alpha_q(\alpha_q(pc) \wedge x \not\approx y)$.
- case $s = x := y$. W.l.o.g., assume $q' = q[x \mapsto x']$, for some constant $x' \notin \mathcal{C}(pc)$. By Lemma 5, $\alpha_q(pc) = \alpha_q(\alpha_q(pc))$. By Lemma 9 (case 1), $\alpha_{\mathcal{C}(q) \cup \{x'\}}(pc \wedge x' \approx y) = \alpha_{\mathcal{C}(q) \cup \{x'\}}(\alpha_q(pc) \wedge x' \approx y)$. By Lemma 7, $\alpha_{q'}(pc \wedge x' \approx y) = \alpha_{q'}(\alpha_q(pc) \wedge x' \approx y)$, since $\mathcal{C}(q') \subseteq (\mathcal{C}(q) \cup \{x'\})$.
- case $s = x := f(\vec{y})$. W.l.o.g., $q' = q[x \mapsto x']$ for some constant $x' \notin \mathcal{C}(pc)$. There are two cases: (a) there is a term $t \in \mathcal{T}(pc)$ s.t. $pc \vdash t \approx f(\vec{y})$, (b) there is no such term t .

(a) By the memoizing property of CUP, there is a program variable z s.t. $q(z) = z$ and $pc \vdash z \approx f(\vec{y})$. Therefore, by definition of α_q , $\alpha_q(pc) \vdash z \approx f(\vec{y})$. The rest of the proof is identical to the case of $s = x := z$.

(b) Since there is no term $t \in \mathcal{T}(pc)$ s.t. $pc \vdash t \approx f(\vec{y})$, there is also no such term in $\mathcal{T}(\alpha_q(pc))$ as well. By Lemma 5, $\alpha_q(pc) = \alpha_q(\alpha_q(pc))$. By Lemma 9 (case 2), $\alpha_{\mathcal{C}(q) \cup \{x'\}}(pc \wedge x \approx f(\vec{y})) = \alpha_{\mathcal{C}(q) \cup \{x'\}}(\alpha_q(pc) \wedge x \approx f(\vec{y}))$. By Lemma 7, $\alpha_{q'}(pc \wedge x \approx f(\vec{y})) = \alpha_{q'}(\alpha_q(pc) \wedge x \approx f(\vec{y}))$ since $\mathcal{C}(q') \subseteq (\mathcal{C}(q) \cup \{x'\})$. ■

Corollary 1 For a CUP P , the relation $\rho \triangleq \{(c, \alpha_b(c)) \mid c \in \text{Reach}(S_P)\}$ is a bisimulation from S_P to $\alpha_b(S_P)$. □

Note that for an arbitrary UP, α_b induces a simulation (since α_b only weakens path conditions).

By construction, for any configuration in an abstract system constructed using α_b , the path condition will be at most depth-1. In Sec. VI, we use this property to build a logical characterization of CUP and show that reachability of CUP programs is decidable.

VI. LOGICAL CHARACTERIZATION OF CUP

In this section, we show that for any CUP program P , all reachable configurations of P can be characterized using formulas in EUF, whose size is bounded by the number of program variables in P .

Theorem 5 (Logical Characterization of CUP) *For any CUP P , there exists an inductive assertion map η , ranging over EUF formulas of depth at most 1, that characterizes the reachable configurations of P .* \square

The first step in the proof is to compose the renaming abstraction (Def. 8) with the base abstraction (Def. 10). We denote the composition with $\alpha_{b,r}$, i.e., $\alpha_{b,r} \triangleq \alpha_b \circ \alpha_r$. Cor. 1 and Thm. 2 ensures that $\alpha_{b,r}$ is sound and complete for CUP. We split the rest of the proof into two cases: CUPs restricted to unary functions, called 1-CUP, followed by arbitrary CUPs.

PROOF (THM. 5, 1-CUP) Let Σ^1 be a signature containing function symbols of arity atmost 1, $\Sigma^1 \triangleq (\mathcal{C}, \mathcal{F}^1, \{\approx, \not\approx\})$. Let Γ be a set of literals in Σ^1 and V be a set of constants. By the definition of V -base abstraction (Def. 9), $\alpha_V(\Gamma) = \beta_{\approx} \wedge \beta_{\not\approx} \wedge \beta_{\mathcal{F}}$. β_{\approx} and $\beta_{\not\approx}$ are over constants in V . $\beta_{\mathcal{F}}$ contains two types of literals: $\beta_{\mathcal{F}_V}$ and $\beta_{\mathcal{F}_W}$. $\beta_{\mathcal{F}_V}$ are 1 depth literals over constants in V . $\beta_{\mathcal{F}_W}$ are literals of the form $v \approx f(\vec{w})$ where $v \in V$ and \vec{w} is a list of constants, at least one of which is in V : $\vec{w} \cap V \neq \emptyset$ and $\vec{w} \not\subseteq V$. Since Γ can only have unary functions, $\beta_{\mathcal{F}_W} = \emptyset$. Therefore, all literals in $\alpha_V(\Gamma)$ are of depth at most 1 and only contain constants from V . Hence, there are only finitely many configurations in $\alpha_{b,r}(\mathcal{S}_P)$. Therefore,

$$\eta(s) \triangleq \bigvee \{pc \mid \langle s, q_0, pc \rangle \in \text{Reach}(\alpha_{b,r}(\mathcal{S}_P))\}$$

is an inductive assertion map, ranging over formulas for depth at most 1, that characterizes the reachable configurations of P . Moreover, the size of each disjunct in $\eta(s)$ is polynomial in the number of program variables and functions in P . \blacksquare

An interesting consequence of the above proof is that, for 1-CUPs, α_b is efficiently computable (since, $\beta_{\mathcal{F}_W} = \emptyset$). Thus, the transition system $\alpha_{b,r}(\mathcal{S}_P)$ is finite, and can be constructed on-the-fly. Hence, reachability of 1-CUP is in PSPACE.

PROOF (THM. 5, GENERAL CASE) In general, CUP programs can contain unary and non-unary functions. Therefore, the V -base abstraction (Def. 9) may introduce fresh constants. We use the cover abstraction (Def. 7) to eliminate these fresh constants. By Thm. 1, $\alpha_{\mathcal{C}}(\alpha_{b,r}(\mathcal{S}_P))$ is bisimilar to $\alpha_{b,r}(\mathcal{S}_P)$. Notice that all the fresh constants introduced by the V -base abstraction are arguments to function applications. Therefore, all consequences of eliminating the fresh constants are Horn clauses of the form $\bigwedge_i (x_i \approx y_i) \Rightarrow x \approx y$, where $x_i, y_i, x, y \in \mathcal{C}_0$. Since V -basis is of depth at most 1, cover of the V -basis is also of depth at most 1. Since there are only finitely many formulas of depth at most 1 over \mathcal{C}_0 , $\alpha_{\mathcal{C}}(\alpha_{b,r}(\mathcal{S}_P))$ has only finitely many configurations. Hence,

$$\eta(s) \triangleq \bigvee \{pc \mid \langle s, q_0, pc \rangle \in \text{Reach}(\alpha_{\mathcal{C}}(\alpha_{b,r}(\mathcal{S}_P)))\}$$

is an inductive assertion map that characterizes the reachable configurations of P and ranges over depth-1 formulas. \blacksquare

Consider the CUP shown in Fig. 4. At line 9, the $\alpha_{b,r}$ abstraction produces the following abstract pc : $x_0 \approx f(a_0, w) \wedge y_0 \approx f(b_0, w) \wedge c_0 \approx d_0$. Using cover to eliminate the constant w gives us $\mathcal{C}w \cdot pc = (a_0 \approx b_0 \Rightarrow x_0 \approx y_0) \wedge c_0 \approx d_0$, which is exactly the invariant assertion mapping $\eta(9)$ at line 9.

We have seen that all CUP programs have an inductive assertion map that characterizes their reachable configurations and ranges over a finite set of formulas. Therefore,

Corollary 2 *CUP reachability is decidable.* \square

A. Relationship to [9]

In [9], Cor. 2 is proven by constructing a deterministic finite automaton that accepts all *feasible* coherent executions.² However, the construction fails for the executions of the CUP in Fig. 4: the execution that reaches a terminal configuration is infeasible, but it is (wrongfully) accepted by the automaton. Intuitively, the reason is that the automaton is deterministic and its states are not sufficiently expressive. The states of the automaton keep track of equalities between program variables (which correspond to β_{\approx} in our abstraction), disequalities between them ($\beta_{\not\approx}$ in our case), and partial function interpretations ($\beta_{\mathcal{F}}$). However, the partial function interpretations are restricted to $\beta_{\mathcal{F}_V}$, i.e., do not allow auxiliary constants that are not assigned to program variables. Thus, they are unable to keep track of $x_0 \approx f(a_0, w) \wedge y_0 \approx f(b_0, w) \wedge c_0 \approx d_0$ in line 9, which is essential for showing infeasibility of the execution. Eliminating the auxiliary constants, as we do in the cover abstraction, does not remedy the situation since it introduces a disjunction $(a_0 \not\approx b_0 \wedge c_0 \approx d_0) \vee (x_0 \approx y_0 \wedge c_0 \approx d_0)$, which the deterministic automaton does not capture.

B. Computing a Finite Abstraction

We have shown that CUP programs are bisimilar to finite state systems. However, all our proofs depend on α_b , which was not assumed to be computable. In this section, we show how to implement α_b , and, thereby, show how to compute a finite state system that is bisimilar to a CUP program. Note that our prior results are independent of this section.

The main difficulty is in naming the fresh constants, which we always refer to as W , that are introduced by the base abstraction. Since we require that base abstraction is canonical, the naming has to be unique. Furthermore, we have to show that the number of such W constants is bounded. We solve both of these problems by proposing a deterministic naming scheme. The scheme is determined by a normalization function n_V that replaces all the fresh constants in a V -basis with canonical constants.

Let β be a V -basis. We denote the auxiliary constants in β ($\mathcal{C}(\beta) \setminus V$) by $W = \{w_0, w_1, \dots\}$, and by ‘?’ some unused constant that we call a *hole*. Recall that constants from W may only appear in literals of the form $v \approx f(\vec{w})$. We define

²In our setting, feasible coherent executions correspond to paths in the transition system of any CUP.

the set of W -templates as the set of all terms $f(\vec{a})$, where each element in \vec{a} is either a hole or a constant in W . A term t matches a template $f(\vec{a})$ if $t = f(\vec{b})$, and \vec{a} and \vec{b} agree on all constants in W . For example, let ξ be the template $f(?, w_1, ?, w_2)$. The term $f(a, w_1, b, w_2)$ matches ξ , but $f(w_0, w_1, b, w_2)$ does not, because one of the holes is filled with $w_0 \in W$. We say that a literal $v \approx f(\vec{b})$ matches a template ξ if $f(\vec{b})$ matches ξ . The W -context of a W -template ξ in a set of literals L , denoted $Z_L(\xi)$, is the set $Z_L(\xi) \triangleq \{\ell[W \mapsto ?] \mid \ell \in L \wedge \ell \text{ matches } \xi\}$, where $\ell[W \mapsto ?]$ means that all occurrences of constants in W are replaced with a hole. For example, let $\xi = f(?, w_1, w_2, ?)$ and $L = \{v \approx f(a, w_1, w_2, b), u \approx f(c, w_1, w_2, a), w \approx f(x, w_1, w_2, b), x \approx g(x, w_1, w_2, b)\}$ then $Z_L(\xi) = \{v \approx f(a, ?, ?, b), u \approx f(c, ?, ?, a), w \approx f(x, ?, ?, b)\}$.

Since V and \mathcal{F} are finite, the number of W -contexts is finite, independent of W . Let w_Z be a fresh constant for context Z .

Definition 12 (Normalization Function) The normalization function $n_V(\beta)$ is defined as follows:

- (1) for each $t \in \mathcal{T}(\Gamma)$ s.t. $\mathcal{C}(t) \cap W \neq \emptyset$, create a template ξ by dropping all constants not in W . Let Ξ denote the set of templates so obtained.
- (2) Let $Ctx \triangleq \{Z_\Gamma(\xi) \mid \xi \in \Xi\}$.
- (3) For each $\ell \in \Gamma$, if $\ell[W \mapsto ?] \in Z$ for some $Z \in Ctx$, then replace all occurrences of W in ℓ with w_Z . \square

The normalization preserves V -equivalence of β because it renames local constants, while maintaining all consequences that are derivable through them. That is, $n_V(\beta) \equiv_V \beta$. Furthermore, $n_V(\beta)$ is canonical.

Therefore, given a set of literals Γ , we use $n_V(\beta)$ as a computable implementation of the V -base abstraction, α_V (Def. 9). That is, $\alpha_V(\Gamma) \triangleq n_V(\beta)$ where $\langle W, \beta, \delta \rangle \in \text{base}(\Gamma, V)$. Even though $n_V(\beta)$ may not be a part of a V -basis for Γ , it satisfies all the properties used in the proof of Thm. 4.

We define the normalizing abstraction in the usual way:

Definition 13 (Normalizing abstraction) The normalizing abstraction function $\alpha_n : C \rightarrow C$ is defined by

$$\alpha_n(\langle s, q_0, pc \rangle) \triangleq \langle s, q_0, n(pc) \rangle \quad \square$$

Let $\alpha_{b,r,n} \triangleq \alpha_b \circ \alpha_r \circ \alpha_n$ be the composition of normalization abstraction with renaming and base abstraction where α_b is implemented using normalization. Notice that, for any state $c = \langle s, q, pc \rangle$, $\alpha_{b,r,n}(c)$ is computed by first computing any V -basis of pc , applying n_q , renaming all $\mathcal{C}(q)$ constants to q_0 , and applying n_{q_0} . The second normalization is required to ensure that the fresh constants are canonical with respect to q_0 . By definition $\alpha_{b,r,n}$ is computable. Hence, it can be used to compute the finite abstraction of any CUP.

Theorem 6 For a CUP P , the finite abstract transition system $\alpha_{b',r,n}(S_P)$ is bisimilar to P and is computable. \square

Thm. 6 implies that any property that is decidable over a finite transition system is also decidable over CUPs. In particular, temporal logic model checking is decidable.

VII. CONCLUSION

In this paper, we study theoretical properties of Coherent Uninterpreted Programs (CUPs) that have been recently proposed by Mathur et al. [9]. We identify a bug in the original paper, and provide an alternative proof of decidability of the reachability problem for CUP. More significantly, we provide a logical characterization of CUP. First, we show that inductive invariant of CUP is describable by shallow formulas. Hence, the set of all candidate invariants can be effectively enumerated. Second, we show that CUPs are bisimilar to finite transition systems. Thus, while they are formally infinite state, they are not any more expressive than a finite state system. Third, we propose an algorithm to compute a finite transition system of a CUP. This lifts all existing results on finite state model checking to CUPs.

In the paper, we have focused on the core result of Mathur et al, and have left out several interesting extensions. In [9], the notion of CUP is extended with k -coherence – a UP P is k -coherent if it is possible to transform P into a CUP \hat{P} by adding k ghost variables to P . This is an interesting extension since it makes potentially many more programs amenable to decidable verification. We observe that addition of ghost variables is a form of abstraction. Thus, invariants of \hat{P} can be translated to invariants of P using techniques of Namjoshi et al. [13], [14]. This essentially amounts to existentially eliminating ghost variables from the invariant of \hat{P} . Such elimination increases the depth of terms in the invariant at most by one for each variable eliminated. Thus, we conjecture that k -coherent programs are characterized by invariants with terms of depth at most k .

Mathur et al. [9] extend their results to recursive UP programs (i.e., UP programs with recursive procedures). We believe our logical characterization results extend to this setting as well. In this case, both the invariants and procedure summaries (i.e., procedure pre- and post-conditions) are described using terms of depth at most 1.

Our results also hold when CUPs are extended with simple axiom schemes, as in [10], while for most non-trivial axiom schemes CUPs become undecidable.


Perhaps most interestingly, our results suggest efficient verification algorithms for CUPs and interesting abstraction for UPs. Since the space of invariant candidates is finite, it can be enumerated, for example, using implicit predicate abstraction. For CUPs, this is a complete verification method. For UPs it is an abstraction. Most importantly, it does not require prior knowledge to whether an UP is a CUP!

Acknowledgment: The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the United States-Israel Binational Science Foundation (BSF) grant No. 2016260, and the Israeli Science Foundation (ISF) grant No. 1810/18. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).


REFERENCES

- [1] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessor control,” in *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Springer, 1994, pp. 68–80. [Online]. Available: https://doi.org/10.1007/3-540-58179-0_44
- [2] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, and A. Rivkin, “Model completeness, covers and superposition,” in *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, ser. Lecture Notes in Computer Science, P. Fontaine, Ed., vol. 11716. Springer, 2019, pp. 142–160. [Online]. Available: https://doi.org/10.1007/978-3-030-29436-6_9
- [3] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, “IC3 modulo theories via implicit predicate abstraction,” in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 46–61. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_4
- [4] S. Ghilardi, A. Gianola, and D. Kapur, “Computing uniform interpolants for EUF via (conditional) dag-based compact representations,” in *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020*, ser. CEUR Workshop Proceedings, F. Calimeri, S. Perri, and E. Zuppano, Eds., vol. 2710. CEUR-WS.org, 2020, pp. 67–81. [Online]. Available: <http://ceur-ws.org/Vol-2710/paper5.pdf>
- [5] S. Gulwani and M. Musuvathi, “Cover algorithms and their combination,” in *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 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*, ser. Lecture Notes in Computer Science, S. Drossopoulou, Ed., vol. 4960. Springer, 2008, pp. 193–207. [Online]. Available: https://doi.org/10.1007/978-3-540-78739-6_16
- [6] S. Gulwani and G. C. Necula, “A polynomial-time algorithm for global value numbering,” in *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, ser. Lecture Notes in Computer Science, R. Giacobazzi, Ed., vol. 3148. Springer, 2004, pp. 212–227. [Online]. Available: https://doi.org/10.1007/978-3-540-27864-1_17
- [7] H. G. V. K., S. Shoham, and A. Gurfinkel, “Logical characterization of coherent uninterpreted programs,” *CoRR*, vol. abs/2107.12902, 2021. [Online]. Available: <https://arxiv.org/abs/2107.12902>
- [8] G. A. Kildall, “A unified approach to global program optimization,” in *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, P. C. Fischer and J. D. Ullman, Eds. ACM Press, 1973, pp. 194–206. [Online]. Available: <https://doi.org/10.1145/512927.512945>
- [9] U. Mathur, P. Madhusudan, and M. Viswanathan, “Decidable verification of uninterpreted programs,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 46:1–46:29, 2019. [Online]. Available: <https://doi.org/10.1145/3290359>
- [10] —, “What’s decidable about program verification modulo axioms?” 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 II*, ser. Lecture Notes in Computer Science, A. Biere and D. Parker, Eds., vol. 12079. Springer, 2020, pp. 158–177. [Online]. Available: https://doi.org/10.1007/978-3-030-45237-7_10
- [11] R. Milner, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989.
- [12] M. Müller-Olm, O. Rüthing, and H. Seidl, “Checking herbrand equalities and beyond,” in *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, ser. Lecture Notes in Computer Science, R. Cousot, Ed., vol. 3385. Springer, 2005, pp. 79–96. [Online]. Available: https://doi.org/10.1007/978-3-540-30579-8_6
- [13] K. S. Namjoshi, “Lifting temporal proofs through abstractions,” in *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*, ser. Lecture Notes in Computer Science, L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, Eds., vol. 2575. Springer, 2003, pp. 174–188. [Online]. Available: https://doi.org/10.1007/3-540-36384-X_16
- [14] K. S. Namjoshi and L. D. Zuck, “Witnessing program transformations,” in *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013, Proceedings*, ser. Lecture Notes in Computer Science, F. Logozzo and M. Fähndrich, Eds., vol. 7935. Springer, 2013, pp. 304–323. [Online]. Available: https://doi.org/10.1007/978-3-642-38856-9_17
- [15] G. Nelson and D. C. Oppen, “Fast decision procedures based on congruence closure,” *J. ACM*, vol. 27, no. 2, pp. 356–364, 1980. [Online]. Available: <https://doi.org/10.1145/322186.322198>
- [16] R. Nieuwenhuis and A. Oliveras, “Proof-producing congruence closure,” in *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, ser. Lecture Notes in Computer Science, J. Giesl, Ed., vol. 3467. Springer, 2005, pp. 453–468. [Online]. Available: https://doi.org/10.1007/978-3-540-32033-3_33
- [17] O. Strichman and B. Godlin, “Regression verification - A practical way to verify programs,” in *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, ser. Lecture Notes in Computer Science, B. Meyer and J. Woodcock, Eds., vol. 4171. Springer, 2005, pp. 496–501. [Online]. Available: https://doi.org/10.1007/978-3-540-69149-5_54

Data-driven Optimization of Inductive Generalization

Nham Le 
University of Waterloo
nv3le@uwaterloo.ca

Xujie Si 
McGill University
CIFAR AI Chair, Mila
xsi@cs.mcgill.ca

Arie Gurfinkel 
University of Waterloo
arie.gurfinkel@uwaterloo.ca

Abstract—Inductive generalization (IG) is the key to the efficiency of modern Symbolic Model Checkers (SMCs). In this paper, we introduce a *data-driven* method for inductive generalization, whose performance can be automatically improved through historical runs over similar instances. Our method is inspired by recent advances for the part-of-speech (PoS) tagging problem in natural language processing (NLP). Specifically, we use a hierarchical recurrent neural network augmented with syntactic and semantic information to predict essential parts of a proof obligation that could be generalized, instead of checking each part one by one. We develop a prototype called ROPEY by incorporating our method into SPACER – a state-of-the-art SMC, and perform evaluations on the KIND2’s simulation benchmarks. ROPEY is evaluated in two settings: *online learning* – for a given instance, we run SPACER for a number of iterations and collect its trace on which ROPEY is trained, and then use ROPEY to guide SPACER to finish the remaining solving process; and *transfer learning* – ROPEY is trained over historical runs of SPACER in advance, and for future instances, ROPEY is used directly to guide SPACER from the very beginning. For non-trivial benchmarks, ROPEY perfectly answers 72% and 77% of the queries in the online and transfer learning settings, respectively. While the speed improvement is not the focus of the paper, our preliminary results are promising: for non-trivial instances, ROPEY’s end-to-end running time is 25% faster.

I. INTRODUCTION

Model checking has been widely used in various important areas such as robustness analysis of deep neural networks [27], verification of hardware designs [16], software verification [3], analysis [20] and testing [41], parameter synthesis in biology [5], and many others. The central challenge of model checking is to find a concise and sound approximation of all possible states a given system may reach, which does not cover any undesired states (i.e. violating given specifications). Tremendous progress has been made by innovations in efficient data representations [10], scalable SAT solvers [43], [35], [18], and effective heuristics [14], [13], [32]. Modern model checkers share a common basis, namely, IC3 [7], of which the key insight is *inductive generalization (IG)*. This idea has been generalized to support rich theories [26] that are crucial for many verification tasks [30], [22] beyond hardware verification. The generalized IC3 with rich theories, also known as satisfiability checking for Constrained Horn

Clauses modulo Theory (CHC) [6], becomes the core part of a broad range of verification tasks.

Existing IG techniques follow either an enumerative search process [7], [8] or ad-hoc heuristics [21], [31]. These heuristics are effective but demand non-trivial domain-specific (or even problem-specific) expertise. In this work, we aim to learn such heuristics automatically from the past successful IGs. We observe that verification problems as well as associated IGs are not isolated from each other. Taking software verification as an example, verifying different properties of the same program involves similar or same IGs; different versions of programs have a similar code base; and different software may use the same conventions, idioms, libraries and frameworks, resulting in similar structures.

Our approach is inspired by recent advances in deep learning, especially in NLP where non-trivial semantic correlations between words are learned automatically using Neural Networks (NNs) [33]. However, IG raises many new challenges for deep learning. First, the input and the output of IG are symbolic expressions, which are *highly structured* with *rich semantics*. Slight syntactic variations can lead to dramatic changes in semantics. Second, more importantly, given that neural networks hardly provide any reliable guarantees, how to design a data-driven system based on deep neural networks, which exhibits *learnability* from past experiences but still preserves *soundness*? All these challenges have to be properly addressed in building a data-driven reasoning framework. In this work, we share our design choices and empirical findings in building a data-driven inductive generalization engine ROPEY, which introduces a neural component into SMC. Specifically, we make the following contributions:

- we adapt standard deep learning models to effectively represent symbolic expressions by incorporating both syntactic and semantic information;
- we design a simple but effective learning objective so that training data can be collected with nearly no changes of existing model checkers;
- our integration algorithm achieves soundness by design, and in the worst case, the learning component may only hurt the running time performance;
- we implement ROPEY on top of SPACER, a state-of-the-art CHC-solver. Our empirical evaluations indicate that ROPEY can effectively predict perfect answers for IG

This work was supported, in part, by an Individual Discovery Grant from the Natural Sciences and Engineering Research Council of Canada, and the Canada CIFAR AI Chair Program.

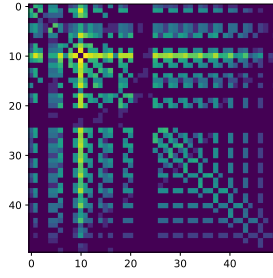


Fig. 1: Literal co-occurrences in solving PRODUCER_CONSUMER_luke_2_e7_1068_e8_1019.

queries, and this predictive power directly translates to an improvement in end-to-end running time.

The utility of our current solution is modest since its applications are restricted to two use-cases: verification of *multiple* properties of a *single* system (transfer learning), and guiding verification of a hard property using its partial run (online learning). This, however, is already useful in the context of multi-property verification that is common both in hardware and software verification domain [12]. More importantly, we demonstrate that NN-based heuristics can be effective in IC3-style algorithms. We believe this will lead to many further improvements, including heuristics that will eventually transfer between systems.

The rest of the paper is structured as follows. Sec. II shows a motivating example. Sec. III gives an overview of our approach. Sec. IV describes two novel embedding methods for converting symbolic expressions into numerical vectors. Sec. V formalizes the learning problem and describes our neural network architecture. Sec. VI presents our empirical evaluation and ablation study. Finally, Sec. VII discusses closely related work, and Sec. VIII concludes the paper.

II. A MOTIVATING EXAMPLE

In this section, we motivate our approach by illustrating the solving process of a particular CHC problem – the variant e7_1068_e8_1019 of the problem PRODUCER_CONSUMER_luke_2 from KIND2 [11] benchmarks. We identify a bottle neck in IG, observe a pattern in the solving process, and explain how this leads to our intuition. While we use a specific instance for illustration, the results generalize to others in our benchmarks. We assume familiarity with SMC [15] and inductive generalization of IC3 [7]. These are also summarized in Sec. III.

SPACER cannot solve this variant in less than 930s. SPACER proves that the instance is safe up to depth 29 in 883s, in which 545s (61%) is spent on IG – so this is the bottleneck.

During inductive generalization process, SPACER takes a candidate lemma L , and uses an SMT solver to check whether each literal of L can be dropped. Each call to the SMT solver is potentially very costly. Thus, it is desirable to drop or skip multiple literals together.

We conjecture that there is a pattern between literals: some groups of literals may always be dropped or kept together. If this correlation is known, it can be used to speed up IG.

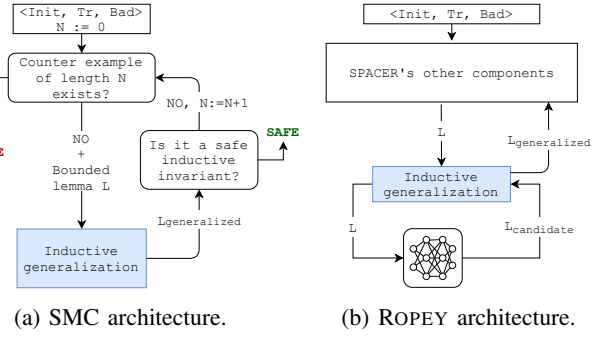


Fig. 2: Overview of Symbolic Model Checking and ROPEY.

To verify our hypothesis, in Fig. 1 we visualize the co-occurrences of kept literals in the instance. Literals are ordered by the time they are learned. Each cell X_{ij} in the grid is the number of times the literals ℓ_i and ℓ_j appear together in some generalized lemma (normalized by the largest value). In the figure, brighter cells indicate larger values.

The figure shows a strong geometric pattern, with literals clustered into unusual groups. However, we are not able to tell the exact heuristics describing those patterns. In this paper, we turn this observation into a practical inductive generalization method with the help of data-driven approach.

III. OVERVIEW

In this section, we give an overview of our technique, outline the challenges involved, and our key insights to address them. The context is symbolic SMT-based Model Checking (SMC) [7], [26], [29], also known as satisfiability checking for Constrained Horn Clauses modulo Theory (CHC) [6]. In Model Checking, the high-level goal is to show that an infinite state transition system (Tr) does not have an execution/path that reaches a set of bad states (Bad) by finding a formula Inv that is an inductive invariant of Tr and does not intersect with Bad . The goal of CHC solving is to show that a set of First Order Logic formulas Φ that satisfy the Horn restriction [6] is satisfiable by exhibiting a symbolic formula $Model$ that defines an FOL model that satisfies Φ . The two problems are closely related. Model Checking is often reduced to CHC solving. Both problems are in general undecidable.

Fig. 2a shows the basic structure of an SMC algorithm based on IC3 architecture. In the paper, we use SMC SPACER [29], but the architecture is common to many engines. SMC iteratively unrolls Tr , uses an SMT solver to find a bounded counterexample (which is usually decidable), and, if no counterexample is found, attempts to create an inductive invariant. The invariant is constructed as a set of so called *lemmas*, where each lemma blocks a predecessor of Bad (a *proof obligation*), and is a disjunction of atomic formulas. An example lemma is $x \leq 0 \vee y$, which is often written as a set for convenience, i.e. $\{x \leq 0, y\}$. Many of the details of the algorithm are not important, and we omit them here. The step we focus on in this paper is *inductive generalization* (IG) (highlighted in blue in Fig. 2a), that is responsible for generalizing learned lemmas. In practice, IG is crucial for the performance of SMC.

Input: the original F-inductive lemma $L = \{\ell_1, \ell_2, \dots, \ell_n\}$
Output: a generalized F-inductive lemma $K \subseteq L$

```

1  $K \leftarrow \emptyset$  // kept literals
2  $C \leftarrow L$  // literals to check
3 while  $C \neq \emptyset$  do
4    $K, C \leftarrow \text{dropOne}(K, C)$ 
5 return  $K$ 

6 function  $\text{dropOne}(K, C)$ 
7    $\text{lit} \leftarrow \text{pick}(C)$ 
8   if  $\text{isInductive}(K \cup C \setminus \{\text{lit}\})$  then
9      $C \leftarrow C \setminus \{\text{lit}\}$ 
10  else
11     $K \leftarrow K \cup \{\text{lit}\}$ 
12     $C \leftarrow C \setminus \{\text{lit}\}$ 
13  return  $K, C$ 

```

Fig. 3: ITERDROP algorithm.

Conceptually, inductive generalization is a simple process, usually done with an algorithm similar to the one we call ITERDROP¹, shown in Fig. 3. ITERDROP starts with a valid lemma $L = \{\ell_1, \dots, \ell_n\}$, and proceeds to generalize L by removing an arbitrary chosen literal from L , and using an SMT solver to check whether the lemma is still valid (by calling isInductive). The details of isInductive are not important – but it can be quite expensive. If the call succeeds, the literal is removed, otherwise it is kept. The goal is to generalize to a valid lemma with a minimal number of literals. From now on, when the context is clear, we use *generalization* instead of inductive generalization.

We illustrate ITERDROP with a sample run, shown in Fig. 4a. Start from the given lemma $L = \{x_3, x_1, x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$, ITERDROP proceeds as follows:

- 1) it tries to drop the first literal, x_3 , by checking whether $L'_1 = \{x_1, x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$ is valid;
- 2) assume that L'_1 is valid, then $L \leftarrow L'_1$, x_1 is chosen next;
- 3) now, assume that $L'_2 = \{x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$ is not valid. L remains as is and $x_6 = 1$ is chosen next;
- 4) assume that $L'_3 = \{x_1, x_9 - x_{10} \geq 41, x_5 = 1\}$ is valid, then $L \leftarrow L'_3$, and $x_9 - x_{10} \geq 41$ is chosen next;
- 5) assume that $L'_4 = \{x_1, x_5 = 1\}$ is not valid, then L is unchanged, and $x_5 = 1$ is chosen next;
- 6) assume that $L'_5 = \{x_1, x_9 - x_{10} \geq 41\}$ is valid, then L'_5 is the final generalized lemma.

The example highlights the difficulty of inductive generalization. First, each call to isInductive is potentially very expensive. Thus, reducing the number of the calls is highly desirable. Second, many of the calls, like steps 3 and 5 are “useless” – no new lemma is learned from them. Thus, reducing such “useless” calls is also highly desirable. Finally, a solver makes many (up to thousands) such inductive generalization calls per run.

Our *key insight* is that since generalization happens frequently, and, while the lemmas are different, the literals are similar, *it is possible to learn the co-occurrence between*

literals that do and do not occur in the same lemma together. This co-occurrence, if learned, could then be used to improve inductive generalization!

Crucially, SPACER learns new literals all the time, and literals between different instances of the same problem are often similar, for instance, $x_1 - 2x_3 \geq 20$ and $x_1 - 2x_3 \geq 25$. Thus, an ML-based solution is useful to transfer knowledge between different sets of literals. Our method is inspired by the PoS-tagging problem in NLP, in which NNs automatically learn co-occurrence patterns between words and their tags. We elaborate more on this inspiration in Sec. V. We have also tried creating our own hand-crafted heuristics for directly calculating co-occurrence (for example, by using Boolean abstraction of literals), but none worked well in practice.

Concretely, we propose a novel neural network architecture, denoted by \mathcal{M} , that learns from past IG queries, and is then used to predict answers for new IG queries. As shown in Fig. 4c, \mathcal{M} outputs a binary mask (a list of zeros and ones) corresponding to literals that should be dropped or kept in the lemma. To evaluate \mathcal{M} in the context of an SMC, we devise a new neural-based IG algorithm called XDROP, that has at its core (Fig. 6). We have developed ROPEY, a prototype SMC that uses XDROP to guide SPACER. (Fig. 2b).

In Fig. 4b, we illustrate a run of XDROP on our example: (1) it runs \mathcal{M} on the input L ; (2) it creates a mask $\{0, 1, 0, 1, 0\}$, corresponding to a candidate $L_{\text{cand}} = \{x_1, x_9 - x_{10} \geq 41\}$; (3) it checks the inductiveness of L_{cand} ; (4) it accepts L_{cand} , and runs ITERDROP starting from L_{cand} . Note that XDROP runs only 3 inductiveness checks, compared to 5 used by ITERDROP.

Challenges. To make ROPEY a practical verification engine, we have to address challenges in both the machine learning and the logical soundness aspect. For machine learning, the challenge is in representing symbolic expressions as vectors, while still maintaining their rich semantic structure. For logical soundness, the challenge is in setting up the learning objective and using the neural net in a way that guarantees the soundness of a verification engine.

Representation learning of symbolic formulas. Literals are symbolic formulas, which are structured and have meaning sensitive to small changes. Simply viewing a literal as a sequence of tokens fails to capture the subtle semantic differences between structurally similar formulas.

We incorporate both syntactic and semantic information of a literal into its representation. Our approach views a literal as a directed acyclic graph (DAG), which is post-processed from its abstract syntax tree (AST), and then adapts TREELSTM [44] to embed such a DAG structure. Our approach also takes semantic information into consideration so that specific properties of values are respected: embedding of numbers and variables should preserve their relative order and equality.

Learning for inductive generalization. Directly using ML to address the generalization problem is a non-trivial structure prediction problem. It takes in a set of symbolic formulas and outputs another set of symbolic formulas that are more general and more concise. Rather than having an

¹While there are more advanced IG techniques, such as [23], we choose ITERDROP since it is used in SPACER – a state-of-the-art CHC solver.

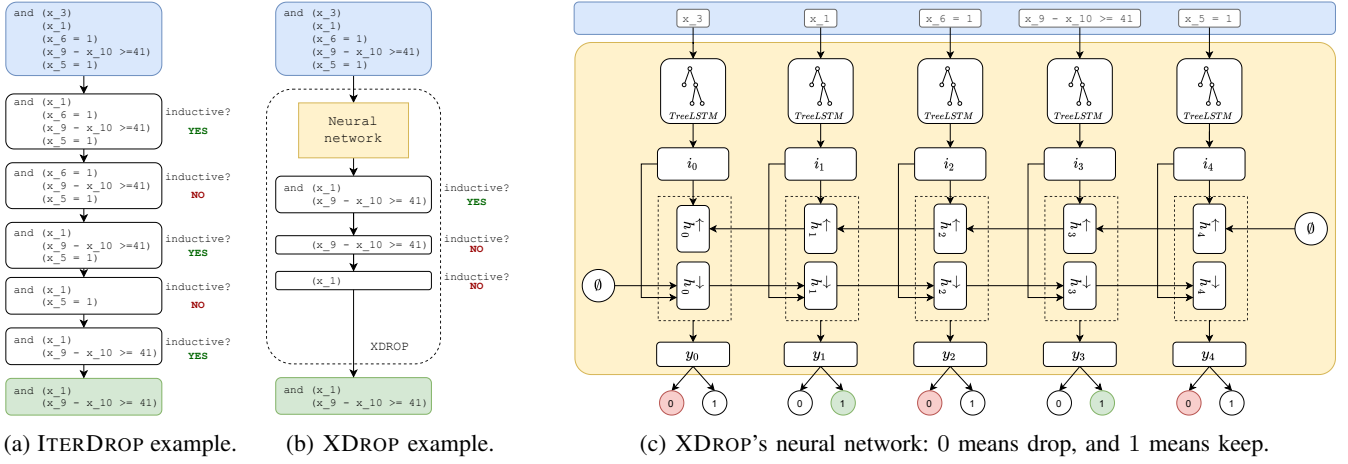


Fig. 4: Examples of how ITERDROP and XDROP do inductive generalization on the same query.

end-to-end ML solution, we embed a learning component in a classic symbolic approach of generalization. Specifically, the learning component captures the co-occurrence between literals appearing in past runs and predicts the likelihood of keeping or dropping a literal in the current run. Furthermore, uncertainties introduced by the learning component have to be carefully controlled, which otherwise could lead to unsound conclusion. ROPEY is designed to make sound progress no matter what predictions the learning component provides. Bad predictions may be harmful to the performance, but not to soundness!

IV. REPRESENTATION LEARNING

Machine learning frameworks [36] and algorithms [44], [38] operate over fixed-length numerical vectors. One challenge for applying machine learning for IG is converting discrete structures with rich semantic meanings into such numerical representations. In this section, we describe how we embed the basic unit of our inputs – symbolic formulas – into fixed-length vectors, while still maintaining their syntactic and semantic meaning to a certain extent.

A. Representing and normalizing symbolic formulas

Abstract Syntax Trees (ASTs) are natural representations of formulas that are traditionally used in parsing and compilers. They preserve the key structure of the formula, while hiding (or abstracting) unnecessary details such as white space, commas and parentheses. Alternative representations such as sequences of tokens abstract too much of the structure of the formula, while highlighting unnecessary differences. Thus, we represent logical formulas using their ASTs: operators label nodes of the tree, operands are children, constants (boolean and numeric) and variables are leaves. An example of an AST is shown in Fig. 5b.

Ideally, we would like to represent semantically equivalent formulas with the same AST. However, this is not guaranteed if one naively parses a formula into an AST. For example, $x + 0 > y$ and $x > y$ are semantically equivalent, yet differ in the concrete syntax, and have different ASTs. To address this, we rewrite each formula in a “normal” form by simplifying as

well as ordering commutative operators. Specifically, we use a simplification engine of Z3 [17]. Our normalizer cannot handle sophisticated semantic equivalences, such as normalizing $2/7 \cdot x_9 - 4/7 \cdot x_{10} \geq 6$ into $1/7 \cdot x_9 - 2/7 \cdot x_{10} \geq 3$. Improving the normalization process to handle such cases would be an interesting future work.

Note that semantically equivalent rewriting and normalization make our representations of symbolic formulas essentially *directed acyclic graphs (DAGs) modulo semantic equivalence*, because semantically equivalent subtrees share the exact same embedding. Indeed, representations of symbolic formulas in our implementation are DAGs, although they are viewed as if they were trees by the embedding model. Without further notice, when we refer to a node in a tree, we actually mean its corresponding node in the DAG.

We use TREE LSTM [44] to embed a symbolic formula, or more concretely its AST representation, into a fixed-length vector. TREE LSTM is essentially a recursive process, where the embedding of a (sub-)tree is an aggregation of the embedding of the root node and embeddings of its subtrees. The basic requirement of using TREE LSTM is to have an embedding for each node. In the rest of this section, we describe the features used to embed each AST node into a fixed-length vector.

B. Embedding features of an AST node

A common technique to map a node N to a vector is to first map the infinite (or simply large) set Σ of all possible nodes into a finite set T of tokens (a.k.a. *encoding*), and then *embed* each token into a vector using an embedding matrix of size $|T| \times d_{\text{emb}}$.

a) *Encoding*: Under the standard encoding scheme, many nodes have to be mapped into the same token. For example, in NLP, all out-of-vocabulary words are mapped into a token $\langle \text{UNK} \rangle$. Similarly, variable names, and numerical constants over an expression can be mapped into two tokens: $\langle \text{VAR} \rangle$ and $\langle \text{NUM} \rangle$, respectively.

Unfortunately, this encoding scheme is inadequate in our setting. We believe that both the variable names and the values

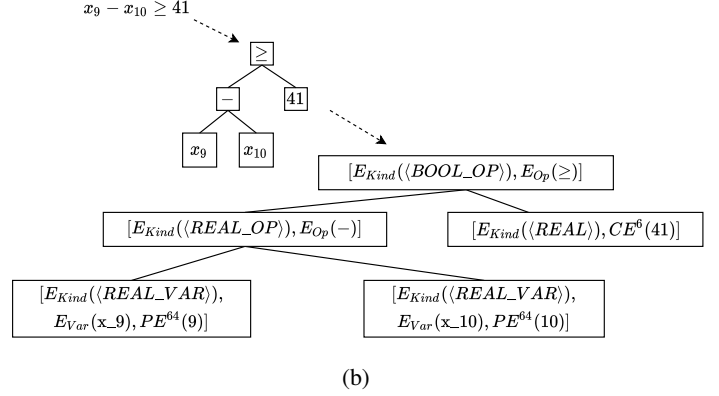
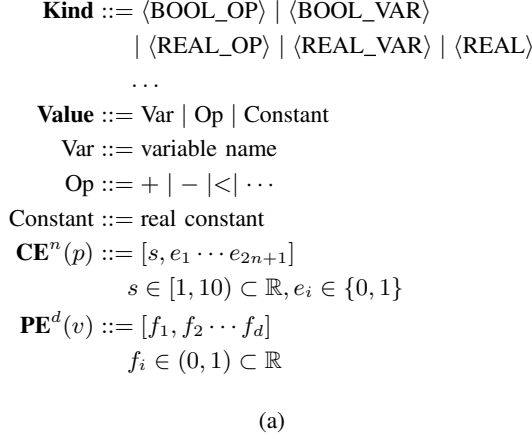


Fig. 5: (a) The grammar for AST node features, and (b) an example AST and its semantic features.

of the numeric constants are highly relevant for successful generalizations! For example, consider two pairs of formulas:

$$\begin{aligned}
 x_1 - 2x_3 + 7x_5 &\geq 10 & x_1 - 2x_3 + 7x_5 &\geq 14 & (1) \\
 x_1 - 2x_3 + 7x_5 &\geq 10 & x_1 + x_3 - x_5 &\geq 0 & (2)
 \end{aligned}$$

Pair (1) represents two parallel hyperplanes, with the first subsuming the second. Pair (2) represents two intersecting hyperplanes and cannot be simplified any further. The difference between the two pairs disappears when all numeric constants are mapped to a small finite set of tokens. Yet, this difference is crucial for successful learning in our context!

Instead of abstracting variables (or constants) into a single token, we propose a finer granularity abstraction as follows. Each node is abstracted as a pair of **Kind**, **Value**, whose grammar is shown in Fig. 5a. **Kind** captures the type (or sort) of the expression of an AST node. The encoding is one of the pre-defined symbols, such as $\langle \text{BOOL_OP} \rangle$ for a Boolean operator, etc. **Value** captures the content of an AST node. It could be a *Variable Name*, an *Operator*, or a *Constant*. Operators are encoded as their string representation. Constants are encoded as their string representations. Variable Names are encoded using the form x_i , where x is some fixed string, and i a numeric id of the variable.

Next, we describe how we embed the pair **Kind**, **Value** into a fixed-length vector.

b) Embedding: **Kind** is embedded into a fixed-length vector of length d_{Kind} using a standard embedding matrix [34] E_{Kind} of the size $|\text{Kind}| \times d_{\text{Kind}}$. **Value** could be embedded in the same manner. However, given **Value** is quite diverse, we propose different embedding methods for different kinds of values. When **Value** is an Op, we introduce the second embedding matrix E_{Op} of the size $|\text{Op}| \times d_{\text{Op}}$.

When **Value** is a Variable Name, we combine two embedding methods. The first method, which we call *Naive Embedding*, is the same as above, in which we use another embedding matrix E_{Var} of the size $|\text{Var}| \times d_{\text{Var}}$. The second method, which we call *Positional Embedding*, based on the method introduced in [46]. It embeds the id t of the normalized variable name x_t as follows: The embedding of the position

t is a vector $\text{PE}^d(t)$ of length d . The value for the i^{th} entry in the vector $\text{PE}^d(t)$ is defined as follows:

$$\text{PE}^d(t)_i = \begin{cases} \sin(\omega_k \cdot t) & \text{if } i = 2k \\ \cos(\omega_k \cdot t) & \text{if } i = 2k + 1 \end{cases}$$

where $\omega_k = 10000^{-2k/d}$. This embedding satisfies many nice properties: each position is mapped to a unique value, all entries in the vector are between 0 and 1 (which makes learning easier), and, lastly, for every fixed offset k , there exists a transformation matrix $T \in \mathbb{R}^{d \times d}$ s.t. $T \cdot \text{PE}^d(t)_i = \text{PE}^d(t+k)_i$ holds for any position t and index i [46]. This last property allows the model to learn relative positions easily. In practice, we combine the two methods by concatenating their vectors.

When **Value** is a Constant, we want to embed it in a way that allows the network to quickly extract magnitudes of constants along with their values. We propose the following *Constant Embedding* method: Given a numerical value p , its embedding is a vector $\text{CE}^n(p)$ of length $2(n+1)$. To embed it, we first write p in its scientific notation: $p = s \times 10^e$. The entries in $\text{CE}^n(p)$ are then calculated as follows:

$$\begin{aligned}
 \text{CE}^n(p)_1 &= s \\
 \text{CE}^n(p)_{i \neq 1} &= \begin{cases} 1 & \text{if } i = 2 + n + e \\ 0 & \text{if } i \neq 2 + n + e \end{cases}
 \end{aligned}$$

Simply put, we embed the significant s as the first entry in the vector, and the rest is the one-hot encoding of e in the range $[-n, n]$. For example, with $n = 2$, $p = 42 = 4.2 \times 10^1$, its embedding is $\text{CE}^2(42) = [4.2 \ 0 \ 0 \ 1 \ 0]$. Similarly, $\text{CE}^3(0.42) = [4.2 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0]$.

The final feature vector for a node is then the concatenation of the embedding of **Kind** and **Value**. In our experiments, we set $d_{\text{Kind}} = d_{\text{Op}} = d_{\text{Var}} = d = 64$, and $n = 6$. We conclude this section with an example. Fig. 5b shows an AST for $x_9 - x_{10} \geq 41$ and its transformation into a tree of feature vectors, with $n = 6$ and $d = 64$.

V. LEARNING TO GENERALIZE

In this section, we elaborate on our insight first mentioned in Sec. III, then we describe the details of our model.

Word	Tag	Literal	Tag
Travelers	noun	x	drop
love	verb	x_1	keep
to	preposition	$x_6 = 1$	drop
park	verb	$x_9 - x_{10} \geq 41$	keep
here	adverb	$x_5 = 1$	drop

TABLE I: Two examples for PoS-tagging (left) and IG (right).

A. Lemma Labeling Problem

In Natural Language Processing, part-of-speech tagging (PoS-tagging) is the process of labeling each word in a text (corpus) a particular part of speech, based on *its definition and its context*. Table I (left) shows an example of tagging a sentence. To correctly tag each word, a tagger needs to know that “park” in this context is a verb, not a noun. State-of-the-art PoS-tagger tackles this problem purely from the probabilistic view [45]: in the dataset, how many times “park” is tagged as a NOUN, how many times “park” is tagged as a VERB given that the following word is tagged as an ADVERB, etc.

Our insight is that the inductive generalization could be viewed as a special case of PoS-tagging in which there are only two tags: drop and keep. Table I (right) shows one such example. We also view the problem in the same probabilistic way: in the dataset, how many times x_3 is kept, how many times x_3 is dropped given that x_1 is kept, etc. It is reasonable to expect there are shared patterns between different properties of the same system, or between different points in time of the same solving process. However, it is not expected that the learned pattern is transferable between different systems (x_3 in one system is completely different from x_3 in the others, just like “park” in English and Korean are completely different).

Formally, we define our problem as an instance of the *sequence labeling problems*:

Problem 1 (Lemma labeling problem) \mathcal{L} is the set of all possible literals. Given a list of literals L of length n and a vector \mathcal{M} of zeros and ones, $|\mathcal{M}| = n$, train a tagger $: \mathcal{L}^n \mapsto \{0, 1\}^n$ s.t. $(L) \approx \mathcal{M}$.

Note that in the problem definition we keep the lemma as a list instead of a set of literals. This means that given a different ordering from the same set of literals, we might end up with a different result. However, this is also the behavior of SPACER, because SPACER maintains the lemma as a list of literals, and *pick*(C) in Fig. 3 simply returns the first element in C .

B. Model

To handle inputs of different lengths, we use two variants of the Long Short-Term Memory (LSTM) [25] network. At the high level, the information (hidden state) at each timestep t in a vanilla LSTM is $\vec{h}_t = LST(\vec{i}_t, \vec{h}_{t-1})$, where \vec{i}_t is the input at timestep t , and a vector of zeros is used for the initial \vec{h}_0 . Intuitively, the formula says that the hidden state at timestep t captures information from every *prior* timestep.

The first variant, Bidirectional-LSTM [38], has been shown to improve the labeling performance in NLP tasks [47]. It extends LSTM by including information from *later* timesteps as

Input: the original F-inductive lemma $L = \{\ell_1, \ell_2, \dots, \ell_n\}$

Output: a generalized F-inductive lemma

```

1  $L_{Cand} \leftarrow \{\ell_i \mid \ell \in L, (L)[i] = 1\}$ 
2 if isInductive( $L_{Cand}$ ) then
3   | return iterDrop( $L_{Cand}$ )
4 else
5   | return iterDrop( $L$ )

```

Fig. 6: XDROP algorithm.

well, thus, allowing the network to use better context information. Concretely, it adds the backward $\overleftarrow{h}_t = LST(\overleftarrow{i}_t, \overleftarrow{h}_{t+1})$. Then, the hidden state h_t is the concatenation $[\overleftarrow{h}_t, \overrightarrow{h}_t]$.

The second variant, TREELSTM [44], has been shown to be suitable for tree-like inputs, such as ASTs. It extends LSTM by considering the linear chain of timesteps as a special case of a tree, in which each node has exactly one child. Given a node i_j in a tree, with $H(i_j)$ is the set of hidden states corresponding to each child node of i_j , TREELSTM extends the equations with $h_j = TreeLST(i_j, H(i_j))$. Intuitively, TREELSTM passes information from all children to their parent, allowing better topology information to be learned. In this work, we use the information at the root node as the summary of the whole tree.²

Fig. 4c shows our full model with a Bidirectional LSTM layer on top of a TREELSTM layer in a hierarchical manner. From top to bottom in Fig. 4c, at a literal ℓ_t corresponding to an AST with root $Root_t$, we calculate the following:

$$\begin{aligned}
i_t &= TreeLSTM(Root_t, H(Root_t)) \\
\overleftarrow{h}_t &= LSTM(i_t, \overleftarrow{h}_{t+1}) \quad \overrightarrow{h}_t = LSTM(i_t, \overrightarrow{h}_{t-1}) \\
h_t &= [\overleftarrow{h}_t, \overrightarrow{h}_t] \quad y_t = W \cdot h_t + b
\end{aligned}$$

where $W \in \mathbb{R}^{|h_t| \times 2}$ and $b \in \mathbb{R}^2$ are the weight matrix and bias that transforms h_t to a vector of size 2. Each equation above corresponds to a layer in Fig. 4c. Finally, the predicted label for ℓ_t is the index of the max value of y_t .

Fig. 6 describes how we use the learned model in our neural-based IG algorithm XDROP. Given that deep learning models could make arbitrary predictions, special care need to be taken in order to preserve soundness. In the worst case, XDROP should be effectively the same as ITERDROP. More formally, we have the following important yet straightforward theorem.

Theorem 1 XDROP is sound and terminating.

XDROP is implemented in Python using PyTorch [36], while SPACER is implemented in C++. We implement a client-server architecture in which XDROP is wrapped in a gRPC server which connects to a gRPC client inside SPACER.

C. Discussion

Using NNs to guide generalization might seem arbitrary at first. Perhaps a simpler heuristic based on counting frequency is sufficient. In fact, we have tried many different handcrafted heuristics first. However, two common problems arose: (a) the

²It is also possible to use the sum of every node in the tree as the summary, as mentioned in [44].

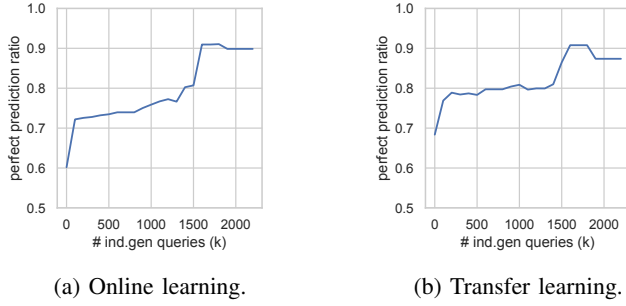


Fig. 7: ROPEY 's predictive power for benchmarks with at least k IG queries.

heuristics do not work consistently across different benchmarks; (b) even if a heuristic works, it does not transfer to different properties since different literals are learned for different properties and systems.

There are many alternative ways to guide generalization using a neural component than the one we chose. Perhaps most desirable is to have an end-to-end solution in which the neural component takes an original lemma as input and produces a generalized lemma as output. However, the symbolic reasoning required for this is so complex that we believe that such a solution is much harder to train and scale up. Another alternative is to learn an approximation of the inductive check, i.e., the function $\text{isInductive}(\text{Context}, L) \mapsto \{\text{true}, \text{false}\}$ that determines whether a candidate lemma L is inductive in the current context. We have tried such an approach, but could not make it effective. The difficulty is that the *Context* that is used by the inductive checker is a large symbolic formula. This makes training the network difficult. We suspect it is as hard as *learning a neural SMT-solver* [40], [39].

VI. EMPIRICAL EVALUATION

A. Benchmarks and environment setup

We evaluate *ROPEY* on a set of simulation benchmarks publicly available³ for the KIND2 model checker [11] (simply called KIND2 from now on). This benchmark suite corresponds to verification of systems that are known to be challenging for IG, for which *SPACER* behaves poorly. Furthermore, KIND2 benchmarks can be easily grouped into training set (i.e. a set of original benchmarks) and testing set (i.e. a set of corresponding variants). In total, KIND2 consists of 324 benchmarks.

We train *ROPEY*'s neural network using Adam optimizer [28] with dropout rate 0.5. We set the hidden size of TreeLSTM to be 64, and use embedding dimensions mentioned in Sec. IV.⁴ We stop training when either the performance has not been improved over the last 250 epochs or the number of epochs reaches a predefined threshold (i.e. 1500). Naive Embedding, Positional Embedding and Constant Embedding are always used. Ablation study for those embeddings is

³<https://github.com/kind2-mc/kind2-benchmarks>.

⁴These dimensions could be further fine-tuned, which we leave as interesting future work.

discussed in Sec. VI-E. All experiments are performed on a Linux desktop equipped with an Intel® Xeon E5-2680 v2, an NVIDIA 1080 Ti GPU, and 64GBs of memory. The artifacts including code and data are available on the project website at <https://nhamlv-55.github.io/Ropey>.

Given that evaluating benchmarks with a short running time (i.e. less than one second) is susceptible to noise, for all experiments we report both the numbers for all benchmarks and the numbers for non-trivial benchmarks. We define a non-trivial benchmark as the one that takes at least 5 seconds to solve, or has at least 100 IG queries (depending on whether we are measuring running time or predictive power, respectively).

B. Predictive power

We evaluate the model in two settings, namely, *online learning* and *transfer learning*. Given a lemma in the form of a list of literals, ROPEY predicts a likely inductively generalized lemma, which is a sub-list of the given lemma. We define a prediction returned by ROPEY as a *perfect prediction* iff given the same input, vanilla *SPACER* produces the same exact answer. Note that this is a conservative criterion because there might be multiple valid inductive generalizations.

Online learning In this setting, we collect 144 benchmarks from KIND2 that have at least 2 IG queries in their solving trace. For each of them, we use *SPACER* to solve it until completion or until a time limit of 930 seconds is reached. Each solving trace is then split in half, and ROPEY is trained on the first half to predict the answers to queries seen in the second half of the trace (tail queries). We measure how many percent of the tail queries are perfectly predicted by ROPEY . The average length of queries is 9.75 literals.

ROPEY achieves 60.19% perfect prediction ratio for all benchmarks and 72.18% for non-trivial benchmarks. The trend of perfect prediction ratio along with the corresponding number of queries are plotted in Fig. 7a, where Y-axis is the perfect prediction ratio and X-axis is benchmarks ordered according to their total number of IG queries. The plot shows that ROPEY generally works better for larger benchmarks. For instance, ROPEY returns perfect predictions for more than 90% of the queries in benchmarks with 1600 or more IG queries.

Transfer learning In this setting, we use 123 benchmarks (i.e., 30 seed benchmarks and 93 variant benchmarks) from KIND2 based on their naming convention. For example, `metros_2_e1_1116.smt2` is one variant of `metros_2.smt2`. Note that we have fewer benchmarks in this task since some seed benchmarks can be solved without any IG queries, while its variants cannot. Those seeds and variants are all excluded from the task. The average length of the queries for this task is 8.43 literals.

We train ROPEY on traces generated by solving the seed benchmarks to completion or until timeout. The models are then used to predict queries asked during the solving process of the corresponding variants.

ROPEY achieves 68.36% and 76.89% perfect prediction ratio for all benchmarks and non-trivial benchmarks, respectively. We also plot the trend of perfect prediction ratio in Fig. 7b.

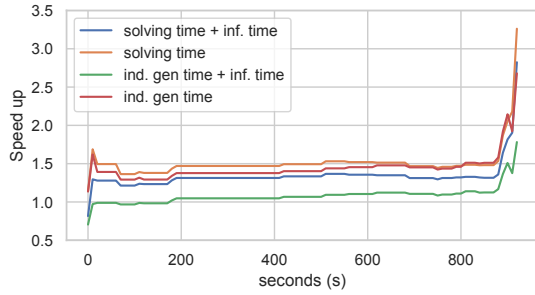


Fig. 8: ROPEY’s speedups for benchmarks taking more than s seconds to solve.

	All	Non-trivial
solving + inf. time	0.81560	1.25385
solving time	1.14085	1.69792
ind. gen time	1.13570	1.63041
ind. gen + inf. time	0.70519	0.91891

TABLE II: ROPEY’s speedups compared with SPACER.

Similar to the online learning setting, generally works better for larger benchmarks. It is a little surprising that the perfect prediction ratio of transfer learning setting is slightly better than the ratio of online learning. This might indicate that in our benchmarks, queries in the beginning and at the end of the same benchmark are more different than queries between seeds and variants. Quantifying this observation is an interesting direction for future work.

C. Running time

ROPEY’s running time can be broken down into few components: SPACER’s time (in which IG time is a subcomponent), communication time over gRPC, data parsing time, and model running time. We group the later three components into *inferencing time*. On average, inferencing takes 48.1% and 24% of the total running time for all and non-trivial benchmarks, respectively. For future work, we state that there are opportunities for engineering improvement to reduce the inferencing time.

We measure the speedup in IG time and SPACER’s solving time with and without the inferencing time. If ROPEY times out, we measure the running time that ROPEY needs to verify to the same depth as SPACER. The timeout is set to be 930 seconds, and in cases where ROPEY times out, we rerun it with the timeout set to 2790 seconds to allow it to verify to the same depth as SPACER. The results are in Table II. We also plot in Fig. 8 the speedups achieved at different running time threshold s , e.g for benchmarks that takes more than 50 seconds to solve, 100 seconds to solve, etc.

For unsolved benchmarks, notice the spikes at the tail of Fig. 8: ROPEY takes much less time to reach to the same depth as SPACER, up to $2.8\times$ faster (inferencing time included).

D. Training time

In this paper, we specifically consider realistic applications where training time is not a bottleneck – train once on one instance and apply to many similar instances (offline), or train during a very long run (days or weeks) and apply to the rest of

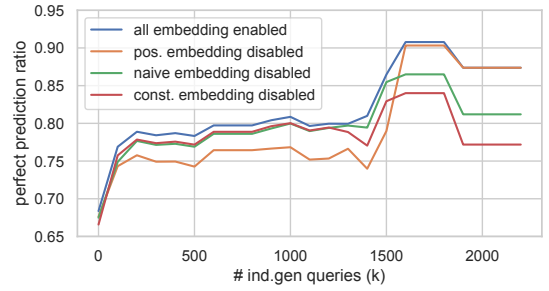


Fig. 9: Effects of using different embeddings for benchmarks with at least k IG queries.

the run (online). For that reason, we do not optimize training code, nor do we run training in an isolated environment where time measurements are meaningful. Nonetheless, we share some statistics of the training time – the minimum, median and maximum training time are 17, 1027 (17 minutes), and 165811 seconds (46 hours), respectively. More details are hosted on our project webpage https://nhamlv-55.github.io/ROPEY/training_time. Training any individual model (i.e., when GPU is used to train only a single model) is faster, but training all models sequentially is too slow. Since we do not consider training time itself to be of significant interest, we train as many models in parallel as possible.

E. Ablation study

Embedding variables and constants is crucial for our tasks. In this ablation study, we evaluate three embeddings we proposed in Sec. IV-B for handling variables and constants. Fig. 9 shows four plots of ROPEY with four different embedding configurations. ROPEY achieves the best performance when all embeddings are enabled. ROPEY’s performance drops dramatically when the positional embedding is disabled, indicating leveraging variable’s position information helps for capturing co-occurrence patterns. Disabling Naive Embedding or Constant Embedding does not affect the performance much for benchmarks with relatively small number (i.e. < 1000) of IG queries, however, the performance drops dramatically when the number of IG queries becomes large.

VII. RELATED WORK

There has been a number of work studying neural learning for symbolic reasoning. Some studied the capability of deep learning models on handling relatively simple symbolic reasoning tasks, such as symbolic expression equivalence [1] or logical entailment [19], which can be easily performed by a symbolic engine like SMT solver. [2] and [37] focus on learning embeddings of programs using paths over abstract syntax trees or control flows, and the learned embeddings are helpful for suggesting function or variable names. Our focus is on improving state-of-the-art symbolic engines on non-trivial symbolic reasoning tasks like symbolic model checking. The most relevant work is [4], which predicts a high-level strategy (or configuration) of an SMT solver based on *static* statistics of a verification instance. In contrast, our approach learns from

dynamic runs and provides guidance for decisions in a finer granularity. Two other related work are [24] and [42]. The former also uses deep learning to guide numerical analysis, where the soundness is not a concern as imperfect prediction results in less precise (but still acceptable) numerical approximations. Like our problem, the latter also faces the soundness issue and proposes an end-to-end reinforcement learning based approach, which however suffers from scalability issues.

VIII. CONCLUSION

In this paper, we explore how deep neural networks can be used in IC3. We chose inductive generalization because it is (a) a common bottleneck; and (b) seemed suitable to optimize with NNs. We view this as a first step in using data-driven NNs to guide IC3. Specifically, we propose a data-driven approach to improving inductive generalization, which effectively embeds symbolic formulas in fixed-length vectors and uses a hierarchical recurrent neural network to guide inductive generalization (i.e., predict which literals of a lemma should be kept or dropped). We build a prototype, ROPEY, and evaluate it on KIND2 benchmark suite. We observe promising predictive power of neural networks in inductive generalization and modest improvement in terms of absolute running time over the state-of-the-art SMC engine, SPACER, which boosts the solving time for non-trivial instances by 25%.

Our work shows that it is possible for NNs to learn complex symbolic patterns in IC3, and such learned patterns can be used to improve IC3. ROPEY's pure performance does not show a strong gain yet, but is still encouraging. We envision the performance gain would be much more significant by improving ROPEY with better engineering effort or leveraging advanced hardware acceleration for deep learning models in the future (like TPUs). Another orthogonal improvement is to explore more advanced transformer-based language models like GPT-3 [9] to further improve the prediction accuracy.

REFERENCES

- [1] M. Allamanis, P. Chanthirasegaran, P. Kohli, and C. A. Sutton, "Learning continuous semantic representations of symbolic expressions," in *ICML 2017*, vol. 70, 2017.
- [2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *POPL*, vol. 3, 2019.
- [3] T. Ball, "Secrets of software model checking," in *AGP 2002*, 2002.
- [4] M. Balunovic, P. Bielik, and M. T. Vechev, "Learning to Solve SMT Formulas," in *NeurIPS*, 2018.
- [5] J. Barnat, L. Brim, A. Krejci, A. Streck, D. Safranek, M. Vejnar, and T. Vejnostek, "On parameter synthesis by parallel model checking," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, no. 3, 2012.
- [6] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Gurevich 75*, 2015.
- [7] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011.
- [8] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," in *FMCAD*, 2011.
- [9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *NeurIPS*, vol. 33, 2020, pp. 1877–1901.
- [10] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, 1986.
- [11] A. Champion, A. Mebsout, C. Stickels, and C. Tinelli, "The Kind 2 Model Checker," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780. Springer, 2016, pp. 510–517. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_29
- [12] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, p. 135–143.
- [13] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods Syst. Des.*, vol. 19, no. 1, 2001.
- [14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000.
- [15] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*, 1st ed. Springer Publishing Company, Incorporated, 2018.
- [16] E. M. Clarke, K. L. McMillan, S. V. A. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *CAV*, 1996.
- [17] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008.
- [18] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003.
- [19] R. Evans, D. Saxton, D. Amos, P. Kohli, and E. Grefenstette, "Can neural networks understand logical entailment?" in *ICLR*, 2018.
- [20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *PLDI*, 2002.
- [21] A. Griggio and M. Roveri, "Comparing different variants of the ic3 algorithm for hardware model checking," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 6, 2016.
- [22] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *CAV*, 2015.
- [23] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 157–164.
- [24] J. He, G. Singh, M. Püschel, and M. T. Vechev, "Learning fast and precise numerical analysis," in *PLDI*, 2020.
- [25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997.
- [26] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, vol. 7317, 2012.
- [27] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett, "The marabou framework for verification and analysis of deep neural networks," in *CAV*, 2019.
- [28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.
- [29] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," in *CAV*, 2014.
- [30] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, "Automatic abstraction in smt-based unbounded software model checking," in *CAV*, 2013.
- [31] H. G. V. Krishnan, Y. Chen, S. Shoham, and A. Gurfinkel, "Global guidance for local generalization in model checking," in *CAV*, 2020.
- [32] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, 2006.
- [33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NeurIPS*, 2013.
- [34] —, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, 2013.
- [35] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC*, 2001.
- [36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *NeurIPS*, 2019.

- [37] V. K. S. R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant, “Ir2vec: A flow analysis based scalable infrastructure for program encodings,” *CoRR*, vol. abs/1909.06228, 2019.
- [38] M. Schuster and K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
- [39] D. Selsam and N. Bjørner, “Guiding High-Performance SAT Solvers with Unsat-Core Predictions,” in *SAT*, 2019.
- [40] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, “Learning a SAT Solver from Single-Bit Supervision,” in *ICLR*, 2019.
- [41] O. Sheyner, J. W. Haines, S. Jha, R. Lippmann, and J. M. Wing, “Automated generation and analysis of attack graphs,” in *SSP*, 2002.
- [42] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, “Learning loop invariants for program verification,” in *NeurIPS*, 2018.
- [43] J. P. M. Silva and K. A. Sakallah, “GRASP – a new search algorithm for satisfiability,” in *ICCAD*, 1996.
- [44] K. S. Tai, R. Socher, and C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” in *ACL*, 2015.
- [45] H. Tsai, J. Riesa, M. Johnson, N. Arivazhagan, X. Li, and A. Archer, “Small and practical BERT models for sequence labeling,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3632–3636. [Online]. Available: <https://www.aclweb.org/anthology/D19-1374>
- [46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.
- [47] X. Zhang and H. Wang, “A joint model of intent determination and slot filling for spoken language understanding,” in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI’16. AAAI Press, 2016, p. 2993–2999.

Model Checking AUTOSAR Components with CBMC

Timothee Durand*, Katalin Fazekas[†], Georg Weissenbacher[†] and Jakob Zwirchmayr*

*TTTech Auto AG, Vienna, Austria

[†]TU Wien, Vienna, Austria

Abstract—Automotive software needs to comply with stringent functional safety standards to reduce the risk of malfunction. In particular, the ISO 26262 standard highly recommends the use of formal verification for highly safety-critical software components. Automated formal verification techniques (such as Model Checking) enable the quick detection of intricate software bugs and can, to a limited extent, even guarantee their absence.

We report our efforts to deploy the openly available verification tool CBMC to verify AUTOSAR Software Components and Complex Device Drivers using Bounded Model Checking and k-induction combined with upfront static analysis.

I. INTRODUCTION

Modern cars now contain as many as 150 Electronic Control Units (ECUs) running software from different suppliers. AUTOSAR, an open and standardized software architecture for automotive applications, guarantees the interoperability of automotive software components. This platform provides a common development methodology based on a standardized exchange format for describing software components (ARXML), standardized communication interfaces and a Run-Time Environment (RTE), and a basic software (BSW) layer (see Fig. 1). The BSW comprises hardware-specific software modules (including Complex Device Drivers (CDDs)) that provide functions to the upper software layers. The RTE middleware provides interfaces and functions for inter- and intra-ECU communication between the application software components. Software Components (SWCs) in the application layer access the lower layers via the RTE, and can hence be readily deployed on different vehicle and platform variants.

The ISO 26262 [1] functional safety standard establishes safety requirements for automotive components (including software). The norm defines four Automotive Safety Integrity Levels (ASILs) ranging from A (low risk) to D (life-threatening hazards). ASIL-D requires the highest degree of rigor, including (semi-)formal verification in the development process. Consequently, formal methods are frequently applied in industrial dependable system design [2]. Moreover, ASIL-code needs to be reverified whenever the implementation is changed, re-generated, or re-configured.

In this context, automated static analysis techniques (such as abstract interpretation or software model checking [3], [4]) are particularly attractive, as they require comparatively little manual interaction and can detect intricate software bugs and, to a limited extent, even guarantee their absence.

We investigate the applicability of model checking to AUTOSAR code written in ANSI-C. While commercial tools for

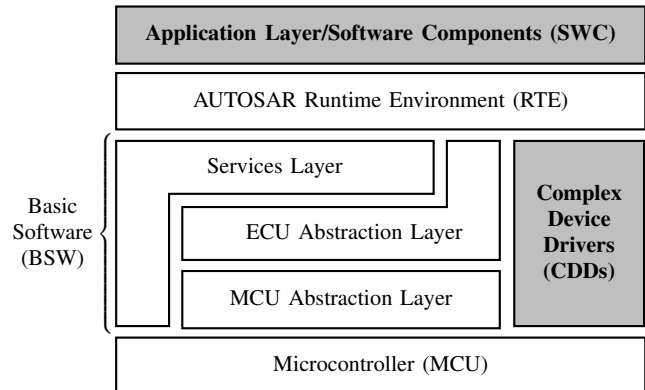


Fig. 1. AUTOSAR Architecture

static analysis of AUTOSAR code exist [5], we focus on the software model checking tool CBMC [6] because of the tool's availability, sustained development, and its permissive open source license. The latter allowed us to adapt CBMC to our work-flow and requirements: the specifics of AUTOSAR software and the ISO 26262 requirements (such as the ARXML description, the use of the RTE, and repeated verification runs) imposes the need for an automated tool chain.

Contributions. Our report (based on the master's thesis of the first author [7]) describes the following contributions:

- 1) To apply CBMC to AUTOSAR code, we generate a test harness and RTE-stubs from an ARXML description.
- 2) We deploy Bounded Model Checking (BMC) to detect bugs, *k*-Induction to prove their absence, and combine both techniques with an upfront static analysis to improve verification performance and results.
- 3) We present case studies for SWCs and CDDs and discuss the different challenges regarding their verification.
- 4) We report our learned lessons and the practicality of the approach and identify open challenges and future work.

II. METHODOLOGY

To verify our SWCs and CDDs (described in subsect. III-A), we need to (1) generate the verification environment and (2) instrument and augment the code with static analysis results.

A. The AUTOSAR Platform

AUTOSAR uses three abstraction levels to describe the SWCs of a system. The highest level—the Virtual Function

```

1 int main_k_base() {
2   SWC_Init();
3
4   for(i=0; i < K; i++) {
5     SWC_Step();
6     assert(P);
7   }
8 }
9
1 int main_k_step() {
2   SWC_Init();
3   mod_ndet_loop_variables();
4   for(i=0; i < K+1; i++) {
5     assume(P);
6     SWC_Step();
7   }
8   assert(P);
9 }

```

Fig. 2. Entry points for k -Induction experiments to prove property P

Bus (VFB)—describes types of SWCs and their connections to other SWCs (PortInterfaces and PortPrototypes), as well as the messages they exchange via their ports (DataTypes). At the middle level—the RTE—the execution behavior of SWCs, i.e., RunnableEntities and their trigger events, are defined. Finally, at the implementation level, these defined RunnableEntities are mapped to their implementations (given as source or object code).

System constraints and the system configuration are described in the ARXML format (see Fig. 3 for an example). In the given context, the SWC Description and the RTE Extract of the ECU Configuration are of relevance, since they describe the messages and data-types that SWCs can exchange.

B. Generating Verification Environment

The RunnableEntities of an SWC (defined in the corresponding ARXML model [8]) provide initialization and step functions, which are invoked periodically in an order we presume to be fixed (see also sect. V).

BMC focuses on checking the correctness of the program only up to a predetermined number of iterations of each loop, pruning all executions that require more. The entry point of our generated test harness for BMC is a function which, after initialization, calls the step functions of the RunnableEntities in an (unbounded) loop.

The test harness for k -Induction¹ has two entry points: one for the base case and another for the inductive step. Fig. 2 illustrates the principle of k -Induction: BMC is used to establish the base case by checking whether the assertion P holds for the first K loop iterations. Subsequently, we use BMC to check whether P holds after $K + 1$ steps under the assumption that it holds in the first K iterations starting from an *arbitrary* program state. If both the base case and induction step succeed, then P holds after any number of loop iterations.

SWCs exclusively interact with each other and with the BSW through the RTE (see Fig. 1), and RTE ports are their only external input [9]. We assume the correctness of the RTE implementation and replace it with an appropriate abstraction. This has two consequences: Firstly, it results in a smaller code base that is more tractable for verification tools. Secondly, as our RTE abstraction conservatively models the most general environment of the SWC, it takes arbitrary interactions with the environment (e.g., any communication via the RTE) into account. This modular approach guarantees that a change in

¹CBMC's built-in support for k -Induction did not cope with the nested loops in our SWCs, which is why we require a separate harness.

```

1 <IMPLEMENTATION-DATA-TYPE UUID="...">
2   <SHORT-NAME>Dt_Engine_RPM</SHORT-NAME>
3   ...
4   <COMPU-METHOD-REF DEST="COMPU-METHOD">
5     /DataTypes/CompuMethods/CM_Engine_RPM
6   </COMPU-METHOD-REF>
7   <IMPLEMENTATION-DATA-TYPE-REF DEST="...">
8     /AUTOSAR_Platform/ImplementationDataTypes/uint16
9   </IMPLEMENTATION-DATA-TYPE-REF>
10  ...
11 </IMPLEMENTATION-DATA-TYPE>
12 ...
13 <COMPU-METHOD UUID="...">
14   <SHORT-NAME>CM_Engine_RPM</SHORT-NAME>
15   ...
16   <COMPU-SCALE>
17     <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
18     <UPPER-LIMIT INTERVAL-TYPE="CLOSED">255
19   </UPPER-LIMIT>
20   <COMPU-RATIONAL-COEFFS>...</COMPU-RATIONAL-COEFFS>
21 </COMPU-SCALE>
22 ...
23 </COMPU-METHOD>

```

```

i void modif_nondet_Dt_Engine_RPM(Dt_Engine_RPM* tmp);
ii void modif_nondet_uint16(uint16* tmp);
iii Std_ReturnType get_nondet_Std_ReturnType();
iv Std_ReturnType
v   Rte_Read_Engine_RPM_stub(Dt_Engine_RPM* tmp);
vi
vii void modif_nondet_Dt_Engine_RPM(Dt_Engine_RPM* tmp) {
viii   modif_nondet_uint16(tmp);
ix     assume(0 <= *tmp && *tmp <= 255);
x }
xi
xii Std_ReturnType
xiii Rte_Read_Engine_RPM_stub(Dt_Engine_RPM* tmp) {
xiv   modif_nondet_Dt_Engine_RPM(tmp);
xv   return get_nondet_Std_ReturnType();
xvi }

```

Fig. 3. Parts of ARXML specification of data type Dt_Engine_RPM (above) and an example of using it in generated RTE function stubs (below)

the environment (e.g., the deployment of other components) does not invalidate prior verification results.

The ARXML specification [10] and the AUTOSAR meta model [8] describe the DataTypes of messages, allowing us to automatically generate an abstraction of the RTE communication functions. Fig. 3 depicts parts of a specification in the ARXML format that defines data types on different abstraction levels. Lines 7-9 state that Dt_Engine_RPM is implemented as uint16. Lines 4-6 refer to a CompuMethod element that specifies a range of valid values from 0 to 255 for the data type. These limits guarantee that the computation will result in a value representable by uint16. For a thorough definition of data types and their constraints see [8, Sect. 5].

In our RTE abstraction parameters and return values of RTE functions are first havoced and then constrained based on information provided in the ARXML specification. These constraints are automatically generated. We generate non-deterministic modifier and generator functions that are invoked in the generated RTE API stubs (see, e.g., function Rte_Read_Engine_RPM_stub in Fig. 3). Fig. 3 also illustrates how the data constraints defined by the XML in lines 17-18 translate into a C assumption (line viii) due to the type Dt_Engine_RPM.

C. Static Analysis and Instrumentation of Code

As a next step, the verification target SWC source code, its dependencies and the generated RTE stubs are built and linked into a single object with CBMC. Though our software project is complex and uses many architectural parameters, CBMC's `goto-cc` could seamlessly replace the compiler and linker in our build process. We note that, in accordance with the ISO 26262 standard, our code base is written in a well-specified and supported sub-set of the ANSI-C language.

Before starting the verification with CBMC, we perform an upfront static analysis of the code to support and complement the strengths of CBMC. To this end, we emit the complete target project into a single source file and run Frama-C [11] on the resulting code. While Frama-C provides a wide range of static analysis techniques, we only employed its Evolved Value Analysis (EVA [12]) plug-in, which is based on abstract interpretation techniques. We used its default parameters that do not rely on more advanced abstract domains. This analysis can infer relatively small value sets for the variables (including function pointers), which simplifies the task of CBMC, but also provides indispensable type constraints for constructing induction proofs in some of our k -Induction experiments. The results of the static analysis are automatically incorporated as assumptions constraining the values of global variables (which represent the entire state of the system) and as replacements of function pointers with explicit case statements.

Prior to instrumentation of the code with the constraints provided by Frama-C, we verify (in independent k -Induction runs) that the value sets provided by Frama-C are actually inductive invariants. To verify the results of the function pointer analysis, the bodies of functions that are unreachable according to Frama-C are replaced with failing assertions which are then checked using CBMC.

D. Implementation details

To automatically parse the ARXML specifications, RTE headers and to generate C stubs, we relied on several openly available Python modules (e.g. PyCParser [13], lxml [14], and cogu-autosar [15]). Some missing POSIX stubs were implemented manually, and we had to patch CBMC to emit proper C code for the SWCs in our experiments.

III. CASE STUDIES

A. Component Descriptions

We analyse four AUTOSAR SWCs of an automotive software platform that comprises of ECUs with multiple hosts. The platform provides services such as a common time-base for the hosts, global time-triggered scheduling, and time-triggered or time-sensitive communication between hosts. A custom RTE hides the fact that the underlying system is distributed and hosted on multiple SoCs/CPUs from the Application SWCs.

LifeCycle Service Server (LCS-S) component: This component is typically executed on the host with the highest ASIL and implements a state machine that determines the state (Init, Standby, Running, etc.) of each host. Running, for instance, indicates that the platform started up successfully

and all hosts are operating under supervision. State transitions are triggered by failing built-in self tests, or depend on the states of other services. The LCS-S sends requests to its clients to trigger transitions and ensures that all client hosts transition correctly and report the expected lifecycle states.

While the LCS-S communicates with other SWCs via the RTE, it is considered a CDD because it directly interacts with other health- and safety-related platform services implemented as CDDs. These interactions via non-standardized interfaces require a few LCS-specific extensions of the verification environment and hence knowledge about implementation details.

LifeCycle Service Client (LCS-C) component: implements the same state machine as the LCS-S and periodically checks whether state transitions are required or have been requested by the LCS-S. An example for a transition requested by the LCS-S and confirmed by the LCS-C is the power-off sequence, where clients might store data in non-volatile memory.

Vehicle Communication Service (ApCom) component: This Application SWC is typically either ASIL-B or D and receives messages from the CAN bus (via the corresponding service in the BSW) and transforms them into RTE data types. Thus, the developers need not be aware of the underlying CAN specifics.

As ApCom utilizes only RTE and BSW COM interfaces, it can be model checked with a generic abstraction of these interfaces. Since large parts of the configuration and the implementation are generated based on a mapping between the CAN and RTE messages, the repeated (automated) verification of this generated code is frequently necessary.

Middleware: This component is a CDD that communicates with other hosts through a Transport Layer (e.g. Ethernet or a time-sensitive version thereof), often relying on OS system calls. Since the exchanged messages contain RTE data, it requires non-standardized interaction with the RTE (such as access to its buffer management system), which complicates verification. While the implementation of the buffer management is static, generated or configurable parts of the code introduce the need for repeated analysis. Since it handles ASIL data, the Middleware may be classified up to ASIL-D.

Table I presents some code metrics for each SWC to illustrate their complexity. More details are available in [7, Section 5]. The components of the LifeCycle service are simpler than the other SWCs, with the LCS-S being the more complex one of both due to supervision and platform initialization tasks. The ApCom component relies heavily on calls-by-reference and function pointers, as evidenced by the amount of pointer arithmetic and dereference operations. Its buffer and data frame manipulation operations make the Middleware the most challenging component of our case study. The high complexity metrics for ApCom and Middleware also denote the presence of large chunks of generated code with repetitive structures within these components.

B. Checked program properties

Our goal is to automatically detect potential errors and vulnerabilities (expressed as assertions) in our code base. In addition to assertions added by developers, we check the

TABLE I
CODE METRICS OF TARGET SOFTWARE COMPONENTS

		LCS-C	LCS-S	ApCom	MW.
Operations	Pointer dereference	50	115	2222	2170
	Add. & Subst.	31	129	330	3662
	Mult. & Div.	36	76	898	471
	Bitwise operations	10	14	11	304
Control flow	If statements	119	243	1276	948
	Loops	4	17	77	76
	Function calls	129	309	1347	1328
	Function returns	66	136	365	329
Complexity	Lines of code	1469	4923	15973	16536
	Program locations	529	1182	5935	7061
	Global variables	34	94	427	584
	MacCabe Cycl. Compl.	187	410	1681	1895

TABLE II
RUNNING TIMES FOR STATIC ANALYSIS OF THE TARGET SWCs

SW Comp.	Frama-C EVA		Slicing	
	Mem. (MB)	Time (s)	LOC (before)	LOC (after)
LCS-C	1281.58	87.96	87340	1469
LCS-S	6564.27	474.04	216349	4923
ApCom	7635.43	596.77	216349	15973
Middleware	1628.26	360.34	106153	16536

properties automatically generated by CBMC (e.g. possible arithmetic overflows, safety of pointer dereferences; see [6]). To enable k -Induction, we instrumented our code base with the necessary assumptions and assertions similarly to Fig. 2. In the k -Induction experiments, we additionally checked constraints on permissible values of variables (e.g., to identify invalid states in the LifeCycle service). Note that defining these latter properties is a manual step that requires insights into the implementation details and the in-depth understanding of the application domain, while the other introduced assertions are automatically constructed.

C. Experiments and Results

For verification we used CBMC 5.23. All experiments were conducted on an Intel(R) Xeon(R) CPU E5345@2.33GHz equipped with 47.2 GB of memory, running Ubuntu 18.04.4. For each run, we set a memory limit of 40 GB and a CPU time limit of one hour, measured by the tool BenchExec [16].

1) *Static Analysis*: We introduced static analysis into our work-flow to address three challenges. First, to avoid spurious counter examples that were due to imprecise value analysis (see for example our k -Induction experiments later in this section). Second, in some of our benchmarks, due to the imprecise value analysis of the function pointers, cycles in the call graph led to non-termination of CBMC. Finally, the computed call graph allows us to identify and exclude code that is not part of the targeted code base, but is still included in the compilation process. The difference in size (lines of codes) before and after slicing unreachable functions in the input file is given Table II. Hence, in our experiments static analysis is an essential preprocessing step that provides valuable benefits.

To gain these benefits, however, an exhaustive static analysis of the code base for each SWC is necessary. Table II presents the running time and memory requirements of this step for each SWC. Note that this analysis includes a precise value analysis for every global variable and function pointer of the code base and removes the unreachable sections of the SWCs.

2) *Bounded Model Checking*: We considered 5 iterations of the loop calling the `RunnableEntities` of our SWCs (cf. subsect. II-B). As most loops in automotive real-time software are statically bounded, CBMC was able to automatically determine bounds for most other loops. In addition, CBMC can detect whether there exist executions that iterate the loop more often than pretermined by the given bound, which we used to identify loops that needed to be bounded manually (of which there were less than 10 overall).

Table III (left) summarizes our BMC results, providing for each SWC the number of checked assertions, memory usage, and run-time. Though no real bugs were found, our verification attempts revealed a modelling flaw in the ARXML specification of the ApCom SWC. In our first verification attempt, CBMC reported an arithmetic overflow in ApCom. Analyzing the report showed that the ARXML specification of the data type of one of the involved variables (whose value was provided by our ARXML-derived RTE abstraction) was too permissive. As the actual implementation of the RTE is more restrictive, this overflow cannot occur in practice.

We identified a similar problem with the ARXML-derived RTE model of the LCS-C component, which yielded a `Not Present` state that is unreachable in the actual implementation. This revealed a limitation of our modular verification approach, which lacks precise information about the states reachable in other (abstracted) components. As before, this bug cannot occur in the implementation.

The Middleware turned out to be too challenging to verify in our experiments. Attempts to simplify the program (by e.g. abstracting away the initialization of shared memory regions which introduced large arrays in the resulting formulas) led to numerous spurious error reports, rendering the approach impractical. Since CBMC did not support some necessary operations, our attempts to deploy a Satisfiability-Modulo-Theory (SMT) solver as back-end also failed.

3) *k -Induction*: The right part of Table III presents the results of our k -Induction experiments. The run-times are the sum and the memory requirements are the maximum of the two consecutive CBMC runs for the base case and induction step (see Fig. 2). In our experiments, we observed that a value of 1 is sufficient in all our (terminating) runs to prove the properties, which we attribute to the auxiliary constraints provided by the upfront static analysis. Hence, k -Induction uses fewer resources than BMC in our setting.

Moreover, the value constraints provided by Frama-C proved to be crucial. Our verification attempts without static analysis led to spurious reports of out-of-bound array accesses in the LCS-S component. This is owed to the fact that the initial states (of the state machine) in the induction step (Fig. 2) are arbitrary and hence potentially unreachable in

TABLE III
EXPERIMENTAL RESULTS OF BOUNDED MODEL CHECKING AND k -INDUCTION

SW Comp.	Bounded Model Checking				k -Induction			
	Assertions	Memory (MB)	Time (s)	Outcome	Assertions	Memory (MB)	Time (s)	Outcome
LCS-C	366	1766.5	102.64	Bounded-Success	370	711.6	44.65	Success
LCS-S	1806	2072.2	135.34	Bounded-Success	1824	1334.7	91.04	Success
ApCom	15562	3406.4	157.58	Bounded-Success	15597	3184.0	292.27	Success
Middleware	9680	14635.7	3600.00	Time out	9780	10043.1	3600.0	Time out

the actual implementation. The value set information provided by Frama-C constrains the initial states to reachable states and strengthens our induction hypothesis. Other components (LCS-C and ApCom) could be verified even without the use of Frama-C. As in our BMC experiments, our attempts to verify the Middleware timed out.

For a comparison of (an older version of) CBMC to alternative software model checking tools (such as CPAchecker [17] and Ultimate Automizer [18]) on the presented SWCs, see [7] (Section 6, pages 44-45).

IV. RELATED WORK

Ahmed and Safar [19] use the symbolic simulation tool KLEE [20] to automatically extract test cases from the C source code of an AUTOSAR BSW module. As testing of safety-critical applications must be requirements-based [1], generated test-cases need to be mapped to requirements. In their CBMC-based automated testing method for the avionic domain, Sun et al. [21] annotate the source code with low-level requirements (expressed as pre- and post-conditions) to establish such a mapping. Mittag [22] applies static analysis to AUTOSAR components, focusing on comparatively simple properties. Berger et al. [23] apply the CBMC-based verifier BTC [24] to check automotive code generated by Simulink, but do not address AUTOSAR. Fang et al. [25] use the SPIN model checker to verify a hand-crafted model of an AUTOSAR-based operating system. Westhofen [26] implements custom k -Induction on top of CBMC to efficiently verify automotive C code.

V. DISCUSSION AND CONCLUSION

Automation was a primary goal, as it enables automated regression verification and limits the effort for the verification engineer. The CBMC model checker and its mature ANSI-C support allowed to use our existing build system and largely unmodified code base. The ARXML component descriptions and the layered architecture of AUTOSAR made it possible to delimit the SWCs and automate the generation of a test harness and stubs that abstract the behaviour of the RTE.

We did, however, face challenges regarding automation, modeling the environment, and scalability. Unlike SWCs, CDDs are not standardized by AUTOSAR. They may use interfaces that are not available to standardized SWCs (e.g., to directly access peripherals). Consequently, the stubs for non-standardized interfaces specific to a CDD need to be generated manually. Moreover, even for SWCs, an overly abstract model of the RTE may lead to false positives. This can be addressed

by providing a more precise model of the RTE (requiring substantial insight into the details of the RTE) or by including actual RTE code. The latter approach, however, amounts to verifying the SWC in the *absence* of an environment.

As CBMC provides limited support for static analysis, we combined it with an upfront run of Frama-C in order to reduce the computational effort for the model checking – interfacing the tools required a non-trivial implementation effort.

Preliminary experiments showed that verifying multiple, interacting components reduces spurious bug reports. This, however, would require to take into account all execution schedules of the runnables, which we consider future work. Another future work is to reuse our verification efforts of the presented SWCs whenever a repeated analysis is necessary (i.e. when the implementation is changed or re-configured) by considering incremental verification techniques.

Overall, our conclusion and outlook is positive: despite all challenges and the engineering effort required to deploy CBMC to verify AUTOSAR components, we ultimately succeeded in checking non-trivial and realistic SWCs.

ACKNOWLEDGMENTS

This work was partially funded by the Vienna Science and Technology Fund (WWTF) under grant NXT19-006. The authors thank the anonymous reviewers for their valuable feedback and suggestions.

REFERENCES

- [1] ISO/TC 22/SC 32, “ISO/DIS 26262 Road vehicles – Functional safety,” International Organization for Standardization (ISO), Tech. Rep. 26262, 2018.
- [2] W. Steiner, “Formal methods in industrial dependable systems design - the TTTech example,” in *Formal Methods in Computer-Aided Design (FMCAD)*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, p. 8. [Online]. Available: <https://doi.org/10.23919/FMCAD.2017.8102232>
- [3] V. D’Silva, D. Kroening, and G. Weissenbacher, “A survey of automated techniques for formal software verification,” *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 7, 2008.
- [4] R. Jhala and R. Majumdar, “Software model checking,” *ACM Comput. Surv.*, vol. 41, no. 4, 2009.
- [5] A. Imparato, R. R. Maietta, S. Scala, and V. Vacca, “A comparative study of static analysis tools for AUTOSAR automotive software components development,” in *International Symposium on Software Reliability Engineering (ISSRE) Workshops*. IEEE, 2017.
- [6] E. M. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 2988. Springer, 2004.
- [7] T. Durand, “Model checking automotive software components,” Master’s thesis, TU Wien, August 2020.
- [8] “Software Component Template - AUTOSAR Rel.4.2.2,” Tech. Rep.
- [9] “System Template - AUTOSAR Rel.4.2.2,” Tech. Rep. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_TPS_SystemTemplate.pdf
- [10] “Specification of RTE - AUTOSAR Rel.4.2.2,” Tech. Rep.
- [11] P. Cuq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-C - A software analysis perspective,” in *Software Engineering and Formal Methods (SEFM)*, ser. LNCS, vol. 7504. Springer, 2012.
- [12] D. Bühler, “Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C.” Ph.D. dissertation, University of Rennes 1, France, 2017. [Online]. Available: <https://tel.archives-ouvertes.fr/tel-01664726>
- [13] “GitHub eliben/pycparser,” <https://github.com/eliben/pycparser>, accessed: 2021-05-17.
- [14] “lxml - XML and HTML with Python,” <https://lxml.de/>, accessed: 2021-05-17.
- [15] “GitHub cogu/autosar,” <https://github.com/cogu/autosar>, accessed: 2021-05-17.
- [16] D. Beyer, S. Löwe, and P. Wendler, “Reliable benchmarking: requirements and solutions,” *Int. J. Softw. Tools Technol. Transf.*, vol. 21, no. 1, 2019.
- [17] D. Beyer and M. E. Keremoglu, “Cpachecker: A tool for configurable software verification,” in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 6806. Springer, 2011, pp. 184–190. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_16
- [18] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski, “Ultimate automizer and the search for perfect interpolants - (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 10806. Springer, 2018, pp. 447–451. [Online]. Available: https://doi.org/10.1007/978-3-319-89963-3_30
- [19] M. Ahmed and M. Safar, “Formal verification of AUTOSAR watchdog manager module using symbolic execution,” in *International Conference on Microelectronics (ICM)*. IEEE, 2018.
- [20] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Operation Systems Design and Implementation (OSDI)*. USENIX Association, 2008.
- [21] Y. Sun, M. Brain, D. Kroening, A. Hawthorn, T. Wilson, F. Schanda, F. J. G. Jimenez, S. Daniel, C. Bryan, and I. Broster, “Functional requirements-based automated testing for avionics,” in *International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2017.
- [22] R. Mittag, “Entwicklung statischer Analysen für AUTOSAR Steuergerätesoftware,” Master’s thesis, TU Chemnitz, 2018.
- [23] P. Berger, J. Katoen, E. Ábrahám, M. T. B. Waez, and T. Rambow, “Verifying auto-generated C code from Simulink - an experience report in the automotive domain,” in *Symposium on Formal Methods (FM)*, ser. LNCS, vol. 10951. Springer, 2018.
- [24] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, “Incremental bounded model checking for embedded software,” *Formal Aspects Comput.*, vol. 29, no. 5, 2017.
- [25] L. Fang, T. Kitamura, T. B. N. Do, and H. Ohsaki, “Formal model-based test for AUTOSAR multicore RTOS,” in *International Conference on Software Testing, Verification and Validation (ICST)*, 2012.
- [26] L. Westhofen, “Verifying automotive C code using modern software model checkers,” Master’s thesis, RWTH Aachen University, 2019.

Automating System Configuration

Nestan Tsiskaridze^{ID}, Maxwell Strange^{ID}, Makai Mann^{ID}, Kavya Sreedhar^{ID}, Qiaoyi Liu^{ID},
Mark Horowitz^{ID}, Clark Barrett^{ID}

Stanford University, Stanford, CA 94305, USA

E-mail: {nestan, mstrange, makaim, skavya, joeyliu}@stanford.edu, horowitz@ee.stanford.edu, barrett@cs.stanford.edu

Abstract—The increasing complexity of modern configurable systems makes it critical to improve the level of automation in the process of system configuration. Such automation can also improve the agility of the development cycle, allowing for rapid and automated integration of decoupled workflows. In this paper, we present a new framework for automated configuration of systems representable as state machines. The framework leverages model checking and satisfiability modulo theories (SMT) and can be applied to any application domain representable using SMT formulas. Our approach can also be applied modularly, improving its scalability. Furthermore, we show how optimization can be used to produce configurations that are best according to some metric and also more likely to be understandable to humans. We showcase this framework and its flexibility by using it to configure a CGRA memory tile for various image processing applications.

I. INTRODUCTION

In systems engineering, the *system configuration* problem arises when systems are parameterized to increase their flexibility or functionality. It refers to the problem of choosing the appropriate parameter values for the context or application in which the system will be used. Most hardware and software systems, including hardware IPs, operating systems, networks, servers, and data centers, require some degree of configuration. The need for configuration also often arises when integrating decoupled parts of a system, including integrating software and hardware.

The difficulty of the system configuration problem has been gradually growing as systems increase in scale and complexity. In particular, in an effort to make designs more widely applicable and re-usable, there has been an increasing use of hardware that is configurable, not only at design time or setup time, but even during normal operation. Manual configuration of such systems is error-prone and may even be impossible, depending on how frequently the systems need to be reconfigured.

Automation of the configuration problem can also be beneficial during the system design process. In particular, it obviates the need for new hand-coded configuration files every time some configurable component changes. Increased automation of such steps supports a move towards more agile design processes. Agile approaches typically require the ability to rapidly and (largely) automatically integrate changing parts of a system while continuously maintaining correct end-to-end functionality. Having design blocks that are flexibly configurable aids this effort, as does the ability to automate the configuration.

A potential disadvantage of automated configuration is that it could lead to an increase in the opacity of the overall system. Hand-written configurations can be documented and explained to allow for easier understandability and maintainability. Thus, an additional goal when automating configuration should be to produce results that are comprehensible to humans and that can be easily reviewed and maintained.

In this paper, we present a general framework for automated system configuration. It provides a flexible approach for solving the configuration problem for systems composed of software, hardware, or both. The systems are modeled using transition systems, where transition formulas can use the full expressive power of SMT-LIB [1], the language used by satisfiability modulo theories (SMT) [2] solvers. The framework provides a systematic approach to facilitate fully automated or automation-guided system configuration. It is well-suited for both stand-alone designs and for designs with multiple configurable parts. For the latter, it is especially useful during system integration and rapid development.

The main contributions of this paper are:

- We introduce a “programming by example” approach for formalizing common input-output specifications. In an exact formulation of the configuration problem, the input-output specification would need to universally quantify over the input variables. Our approach avoids the need for quantifiers.
- We propose a new modular approach for configuration finding in a general SMT setting that makes use of abduction.
- We show how to leverage optimization to obtain human-readable configurations.
- We present a case study—automated configuration of a memory tile in the context of an agile hardware design project targeting image processing applications.

The remainder of the paper is organized as follows. Section II presents background and notation. Section III formalizes the configuration solving problem and introduces our framework, including some extensions and limitations. In Section IV, we show how optimization techniques can be integrated into the approach, both for the purpose of improving performance as well as for improving human readability, and we discuss a few additional extensions of the framework. In Section V we present a case study, giving the details of a specific system design and showing how our framework can be applied. Experimental results for this case study are

then reported in Section VI. We survey the related work in Section VII and conclude in Section VIII.

II. BACKGROUND

We assume the standard many-sorted first-order logic setting with the usual notions of signature, term, formula, 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, i.e., the *models* of \mathcal{T} . A Σ -formula φ is *satisfiable* (resp., *unsatisfiable*) in \mathcal{T} if it is satisfied by some (resp., no) interpretation in \mathbf{I} . We define $\models_{\mathcal{T}}$ over Σ -formulas: if φ and ψ are Σ -formulas, then $\varphi \models_{\mathcal{T}} \psi$ if all interpretations which satisfy φ also satisfy ψ . In this case, we also call φ an *abduct* of ψ under \mathcal{T} . For generality, we assume an arbitrary but fixed background theory \mathcal{T} (which could be a combination of theories) with signature Σ and an infinite set \mathcal{X} of variables. We will assume that all terms and formulas are Σ -terms and Σ -formulas whose free variables are in \mathcal{X} , that entailment is entailment modulo \mathcal{T} , and that interpretations are \mathcal{T} -interpretations that assign every variable in \mathcal{X} .

Given an interpretation \mathcal{I} , a variable assignment s over a set of variables V is a mapping that assigns each variable $v \in V$ of sort σ to an element of $\sigma^{\mathcal{I}}$, denoted v^s . The assignment over V induced by an interpretation \mathcal{I} (i.e., the assignment that maps each variable in V to its interpretation in \mathcal{I}) is denoted \mathcal{I}^V . The assignment s restricted to the domain $U \subseteq V$ is denoted by s^U . We write $\mathcal{I}[s]$ for the interpretation that is equivalent to \mathcal{I} except that each variable $v \in V$ is mapped to v^s . We write $f \circ g$ for functional composition, i.e., $f \circ g(x) = f(g(x))$.

Satisfiability Modulo Theories (SMT). Satisfiability Modulo Theories [2] is an extension of the Boolean satisfiability (SAT) problem to satisfiability in first-order theories. SMT solvers combine the Boolean reasoning of a SAT solver with specialized theory solvers to check satisfiability of many-sorted first-order logic formulas. Some examples of commonly supported theories are: fixed-width bit-vectors, uninterpreted functions, linear arithmetic, and arrays. In our case study, we utilize fixed-width bit-vectors for modeling a hardware design.

Symbolic Transition Systems.

A symbolic transition system (STS) \mathcal{S} is a tuple $\mathcal{S} := \langle V, I, T \rangle$, where V is a finite set of state variables (possibly of different sorts), $I(V)$ is a formula denoting the initial states of the system, and $T(V, V')$ is a formula expressing a transition relation, with V' defined as follows. Let *prime* be a bijection that maps each variable $v \in V$ to a new variable (not in V) v' of the same sort. V' is the codomain of *prime*.

A state s of \mathcal{S} is a variable assignment over V . A sequence of states is called a *path*. An *execution* of \mathcal{S} of length k is a pair $\langle \mathcal{I}, \pi \rangle$, where \mathcal{I} is an interpretation and $\pi := s_0, s_1, \dots, s_{k-1}$ is a *path* such that $\mathcal{I}[s_0] \models I(V)$ and $\mathcal{I}[s_i][s_{i+1} \circ \text{prime}^{-1}] \models T(V, V')$ for all $0 \leq i < k-1$.

Unrolling and Bounded Model Checking.

An *unrolling* of length k of a symbolic transition system is a formula that captures an execution of length k by creating copies of the transition relation. This is accomplished by

introducing fresh copies of every state variable for each state in the execution path. We use V_i to denote the set of variables obtained by replacing each variable $v \in V$ with a new variable called v_i of the same sort. We refer to these as *timed* variables. Given an STS \mathcal{S} , let $\text{unroll}(\mathcal{S}, k) = I(V_0) \wedge \bigwedge_{0 \leq i < k} T(V_i, V_{i+1})$.

Bounded model checking (BMC) [3] is an unrolling-based symbolic model checking approach. Let $P(V)$ be a formula representing a desired property of a symbolic transition system. BMC creates an unrolled transition system and adds an additional constraint that the property is violated at time k . The BMC formula at bound k is thus: $\text{unroll}(\mathcal{S}, k) \wedge \neg P(V_k)$. A typical approach for BMC starts with $k = 0$ and incrementally increases it if no counterexample is found at the current bound. A satisfiable BMC formula can easily be converted into an execution that violates the property.

Optimization. An *optimization problem* \mathcal{OP} is a tuple $\langle t, A, \preceq, \phi, \mathcal{O} \rangle$ where:

- t is an *objective term* to optimize of sort σ ;
- A is a set and \preceq is a total order over A .
- ϕ is a formula to satisfy; and
- $\mathcal{O} \in \{\min, \max\}$ is the optimization objective.

\mathcal{I} is a solution to \mathcal{OP} if $\sigma^{\mathcal{I}} = A$, $\mathcal{I} \models \phi$, and for any \mathcal{I}' , such that $\sigma^{\mathcal{I}'} = A$ and $\mathcal{I}' \models \phi$:

$$(\mathcal{O} = \min \rightarrow t^{\mathcal{I}} \preceq t^{\mathcal{I}'}) \wedge (\mathcal{O} = \max \rightarrow t^{\mathcal{I}'} \preceq t^{\mathcal{I}}).$$

A *multi-objective optimization problem* \mathcal{MOP} is a finite sequence of optimization problems $\{\mathcal{OP}_1, \dots, \mathcal{OP}_n\}$ over the same formula ϕ , where $\mathcal{OP}_i := \langle t_i, A_i, \preceq_i, \phi, \mathcal{O}_i \rangle$ and t_i is of sort σ_i for $i \in [1, n]$. \mathcal{I} is a solution to \mathcal{MOP} if $\sigma_i^{\mathcal{I}} = A_i$, $\mathcal{I} \models \phi$, and for any \mathcal{I}' , such that $\sigma_i^{\mathcal{I}'} = A_i$ and $\mathcal{I}' \models \phi$, either:

- $t_i^{\mathcal{I}} = t_i^{\mathcal{I}'}$ for all $i \in [1, n]$; or
- for some $j \in [1, n]$, $t_j^{\mathcal{I}} = t_j^{\mathcal{I}'}$ for all $i \in [1, j]$, and

$$(\mathcal{O}_j = \min \rightarrow t_j^{\mathcal{I}} \prec_j t_j^{\mathcal{I}'}) \wedge (\mathcal{O}_j = \max \rightarrow t_j^{\mathcal{I}'} \prec_j t_j^{\mathcal{I}}),$$

where \prec is the strict total order associated with \preceq .

III. CONFIGURATION SOLVING FRAMEWORK

In this section, we formalize the configuration problem and introduce our automated framework for solving it. We also describe how to improve scalability using a modular approach.

A. Problem Formalization

Suppose we have a configurable system that we want to use in a particular application context. We assume the application context can precisely define an input/output relationship that it expects the system to adhere to. The *configuration finding problem* is then: given a system S and an application-supplied input-output relationship P for S , find a configuration \mathcal{C} for S such that S satisfies P with configuration \mathcal{C} . In this paper, we assume that P specifies behavior for only a finite number of steps. The rationale is that for many configurable systems, a segment of a desired execution is sufficient to partially (or fully) determine what the configuration should be. This is the case for the systems we target and for the case study we

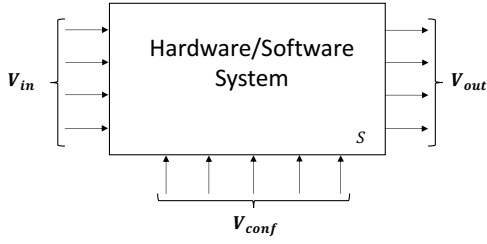


Fig. 1: Formal system model.

describe later. More general specifications are an important direction for future work.

Formally, a configuration problem \mathcal{CP} is a tuple $\langle \mathcal{S}, k, V_{in}, V_{out}, V_{conf}, P \rangle$ where:

- $\mathcal{S} := \langle V, I, T \rangle$ is a symbolic transition system representing a configurable system S , as in Figure 1;
- k is the number of transitions over which the input-output specification will be defined;
- $V_{in}, V_{out}, V_{conf}$ are three distinguished subsets of the state variables V of \mathcal{S} ; V_{in} contains *input* variables (input variables do not appear in $I(V)$, and their primed versions do not appear in T); V_{out} contains *output* variables; and $V_{conf} \neq \emptyset$ contains the *configuration* variables; pairwise intersections of these sets may either be empty or non-empty, and V may contain variables that are not in any of these sets; and
- P is an *input-output property*, or an *input-output specification*, a formula capturing an input-output relationship for k transitions: $P(V_{in}^0, \dots, V_{in}^{(k-1)}, V_{out}^0, \dots, V_{out}^k)$; in this paper, we use a “programming by example” property, specifying a set of exact values on input and output variables at each transition: $\bigwedge_{0 \leq i < k} V_{in}^i = c_{in}^i \wedge \bigwedge_{0 \leq i \leq k} V_{out}^i = c_{out}^i$. This approach works well on our case study (i.e. the configuration found for the given example generalizes to other inputs), and it avoids the need for universal quantification on the input variables. Handling other kinds of properties is an important direction for future work.

A *configuration* \mathcal{C} is defined as an assignment to the variables in V_{conf} .

In this paper, we assume the configuration variables V_{conf} remain unchanged once configured (a reasonable assumption for many systems, including the one in the case study we present in Section V). We enforce this by explicitly adding an additional *configuration constancy constraint*: $conf(V_{conf}, k) = \bigwedge_{0 \leq i < k} V_{conf}^{(i+1)} = V_{conf}^i$. The configuration finding problem then reduces to checking the satisfiability of the *configuration formula*:

$$\phi(\mathcal{CP}) = unroll(\mathcal{S}, k) \wedge conf(V_{conf}, k) \wedge P(V_{in}^0, \dots, V_{in}^{(k-1)}, V_{out}^0, \dots, V_{out}^k) \quad (1)$$

A configuration \mathcal{C} is *correct* for \mathcal{CP} if there exists an interpretation \mathcal{I} such that $\mathcal{I} \models \phi$ and $\mathcal{C} = \mathcal{I}^{V_{conf}}$.

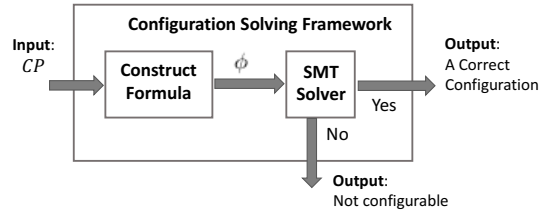


Fig. 2: Configuration solving framework (basic) scheme. \mathcal{CP} is a configuration problem. ϕ is a configuration formula.

Example 1. (simple ALU)

Let $\mathcal{S} := \langle \{x : int, a : int, cfg : Bool\}, x = 0, x' = ite(cfg, x + a, x - a) \rangle$ be a transition system in a configuration finding problem, where $V_{in} = \{a\}$, $V_{out} = \{x\}$, $V_{conf} = \{cfg\}$, and ite is the if-then-else operator. There are two ways to configure \mathcal{S} : as a system that always adds the current input to the current state, or as a system that always subtracts the current input from the current state. Let us consider two instances of an input-output relation for $k = 2$:

- 1) $P_1(a \ 0, a \ 1, x \ 0, x \ 1, x \ 2) = a \ 0 = 1 \wedge a \ 1 = 1 \wedge x \ 0 = 0 \wedge x \ 1 = 1 \wedge x \ 2 = 2$. We are interested in whether there exists a value of cfg which satisfies both the configuration constancy constraint (i.e., remains unchanged) and P_1 . To determine this, we check the satisfiability of $unroll(\mathcal{S}, 2) \wedge conf(cfg \ 0, cfg \ 1, cfg \ 2) \wedge P_1(a \ 0, a \ 1, x \ 0, x \ 1, x \ 2)$, which expands to:

$$\begin{aligned} x \ 0 &= 0 \wedge \\ x \ 1 &= ite(cfg \ 0, x \ 0 + a \ 0, x \ 0 - a \ 0) \wedge \\ x \ 2 &= ite(cfg \ 1, x \ 1 + a \ 1, x \ 1 - a \ 1) \wedge \\ cfg \ 1 &= cfg \ 0 \wedge cfg \ 2 = cfg \ 1 \wedge \\ a \ 0 &= 1 \wedge a \ 1 = 1 \wedge x \ 0 = 0 \wedge x \ 1 = 1 \wedge x \ 2 = 2 \end{aligned}$$

The formula is satisfiable when $cfg \ 0 = True$.

- 2) $P_2(a \ 0, a \ 1, x \ 0, x \ 1, x \ 2) = a \ 0 = 1 \wedge a \ 1 = 1 \wedge x \ 0 = 0 \wedge x \ 1 = 1 \wedge x \ 2 = 0$. For this case, the formula to be checked is:

$$\begin{aligned} x \ 0 &= 0 \wedge \\ x \ 1 &= ite(cfg \ 0, x \ 0 + a \ 0, x \ 0 - a \ 0) \wedge \\ x \ 2 &= ite(cfg \ 1, x \ 1 + a \ 1, x \ 1 - a \ 1) \wedge \\ cfg \ 1 &= cfg \ 0 \wedge cfg \ 2 = cfg \ 1 \wedge \\ a \ 0 &= 1 \wedge a \ 1 = 1 \wedge x \ 0 = 0 \wedge x \ 1 = 1 \wedge x \ 2 = 0 \end{aligned}$$

This formula is unsatisfiable, and thus there is no value of cfg that satisfies the desired property.

The framework for the basic scheme just outlined is shown in Figure 2. The input to the framework is a configuration problem. The framework constructs formula (1) and calls a solver to determine whether it is satisfiable. The output is either “not configurable” or the configuration \mathcal{C} .

There are two main sources of complexity that limit the scalability of the approach. The first is the complexity of the

Algorithm 1 Modular configuration finding.

Procedure SOLVEMODULAR

Input: $(\mathcal{CP}_1, \mathcal{CP}_2)$ a decomposition of \mathcal{CP} .

Output: a pair (r, \mathcal{C}) where if $r = \text{sat}$, then \mathcal{C} is a configuration of \mathcal{S}

```

1:  $\phi_1 := \text{MAKECP}(\mathcal{CP}_1)$ 
2:  $(r, \mathcal{I}_1) := \text{SOLVE}(\phi_1)$ 
3: if  $r = \text{sat}$  then
4:    $\phi_2 := \text{MAKECP}(\mathcal{CP}_2) \wedge \text{GETABDUCT}(\phi_1, \mathcal{I}_1)$ 
5:    $(r, \mathcal{I}) := \text{SOLVE}(\phi_2)$ 
6: end if
7: return  $(r, \mathcal{I}^{V_{\text{conf}}})$ 

```

design itself, and the second is the bound k required by P . To address design complexity, we propose designing for modular configuration, discussed in more detail in Section III-B below. Designing systems that can be configured using only small values of k is an interesting research challenge that we plan to investigate in future work.

Another way to improve scalability is by using design knowledge to strengthen the formula ϕ . For example, if a configuration variable must be within a specific range, then this can be added as a constraint. Any constraint expressible in the language supported by the backend SMT solver can be supported.

B. Modular Configuration

A natural remedy for design complexity is modular decomposition. Here, we explain a systematic approach for modular configuration, including conditions under which a full configuration can be recovered.

Given $\mathcal{CP} = \langle \mathcal{S}, k, V_{\text{in}}, V_{\text{out}}, V_{\text{conf}}, P \rangle$ with $\mathcal{S} = \langle V, I, T \rangle$, we say $(\mathcal{CP}_1, \mathcal{CP}_2)$ is a *decomposition* of \mathcal{CP} (where $\mathcal{CP}_i := \langle \mathcal{S}_i, k, V_{\text{in}}^i, V_{\text{out}}^i, V_{\text{conf}}^i, P_i \rangle$ and $\mathcal{S}_i := \langle V_i, I_i, T_i \rangle$ for $i = 1, 2$) if: (i) $T_1(V_1, V_1') \wedge T_2(V_2, V_2') \implies T(V, V')$; (ii) $I_1(V_1) \wedge I_2(V_2) \implies I(V)$; (iii) $P_1 \wedge P_2 \implies P$; and (iv) $V_{\text{conf}} \subseteq V_{\text{conf}}^1 \cup V_{\text{conf}}^2$.

We now describe a procedure SOLVEMODULAR, presented in Algorithm 1, which, given a decomposition $(\mathcal{CP}_1, \mathcal{CP}_2)$ of a configuration problem \mathcal{CP} , attempts to solve \mathcal{CP} by solving \mathcal{CP}_1 and \mathcal{CP}_2 . The call to MAKECP on line 1 constructs the configuration formula for \mathcal{CP}_1 . The call to SOLVE on line 2 invokes a solver to check the satisfiability of the configuration formula. If the formula is satisfiable, SOLVE returns a pair $(\text{sat}, \mathcal{I})$ where \mathcal{I} is a satisfying interpretation found by the solver. If the formula is unsatisfiable, SOLVE returns a pair $(\text{unsat}, \mathcal{I})$ where \mathcal{I} is an arbitrary interpretation. Line 4 creates the configuration formula for \mathcal{CP}_2 . The formula is additionally constrained to ensure that the solution for \mathcal{CP}_2 still satisfies ϕ_1 . The call to GETABDUCT returns a formula ψ such that $\psi \models_{\mathcal{T}} \phi_1$. The goal is to use the information in \mathcal{I}_1 to generate a simple formula for ψ . The approach we take is to find a set of sub-terms in ϕ_1 such that, if we constrain them to be equal to their values in \mathcal{I}_1 , this ensures that ϕ_1 is satisfied. In the worst case, we could constrain ϕ_1 itself to be equal to \top , which would effectively require solving all of ϕ_1 again at the same time as solving ϕ_2 . However, in practice, we can do much better. For example, it is often sufficient to let

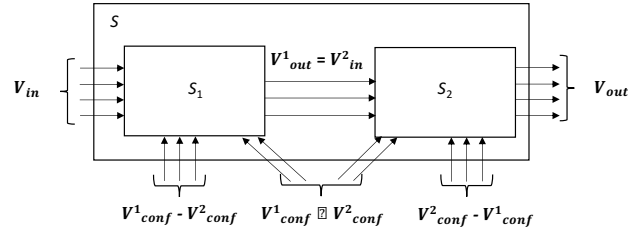


Fig. 3: Modular decomposition of system \mathcal{S} into systems \mathcal{S}_1 and \mathcal{S}_2 . V^1_t and $V^1_{c\ nf}$ are the output and the configuration variables of \mathcal{S}_1 . V^2_{in} and $V^2_{c\ nf}$ are the input and the configuration variables of \mathcal{S}_2 . $V_{c\ nf} \subseteq V^1_{c\ nf} \cup V^2_{c\ nf}$.

ψ be the formula that assigns the free variables in ϕ_1 to their model values from \mathcal{I}_1 .¹ If the second call to SOLVE succeeds, the result is a correct configuration for \mathcal{CP} .

Theorem III.1. (Soundness)

If $(\mathcal{CP}_1, \mathcal{CP}_2)$ is a decomposition of a configuration problem \mathcal{CP} , and SOLVEMODULAR($\mathcal{CP}_1, \mathcal{CP}_2$) returns a pair $(\text{sat}, \mathcal{C})$, then \mathcal{C} is a correct configuration of \mathcal{CP} .

Proof. Let SOLVEMODULAR return $(\text{sat}, \mathcal{I}^{V_{\text{conf}}})$. We prove that $\mathcal{I}^{V_{\text{conf}}}$ is a correct configuration of \mathcal{CP} . First, we notice that SOLVEMODULAR returns $r = \text{sat}$ iff both calls to SOLVE(ϕ_1) and SOLVE(ϕ_2) return $r = \text{sat}$. Let $(\text{sat}, \mathcal{I}_1)$ and $(\text{sat}, \mathcal{I})$ be the results of SOLVE(ϕ_1) and SOLVE(ϕ_2), respectively. Let $\psi = \text{GETABDUCT}(\phi_1, \mathcal{I}_1)$. From line 5, $\mathcal{I} \models \phi_2$. Thus, $\mathcal{I} \models \text{MAKECP}(\mathcal{CP}_2)$ and $\mathcal{I} \models \psi$. Since $\psi \models_{\mathcal{T}} \phi_1$, we also have $\mathcal{I} \models \phi_1$. Consequently, \mathcal{I} satisfies: $I_1, T_1(V_1 - i, V_1 - (i + 1))$ for $i \in [0, k - 1]$, $\text{conf}(V^1_{\text{conf}}, k)$, and P_1 . Furthermore, \mathcal{I} satisfies: $I_2, T_2(V_2 - i, V_2 - (i + 1))$ for $i \in [0, k - 1]$, $\text{conf}(V^2_{\text{conf}}, k)$, and P_2 . By the definition of decomposition, then, \mathcal{I} satisfies $I(V), T(V - i, V - (i + 1))$ for $i \in [0, k - 1]$, and P . Finally, from $\mathcal{I} \models \text{conf}(V^1_{\text{conf}}, k)$, $\mathcal{I} \models \text{conf}(V^2_{\text{conf}}, k)$, and condition (iv) of the definition of decomposition ($V_{\text{conf}} \subseteq V^1_{\text{conf}} \cup V^2_{\text{conf}}$), it follows that $\mathcal{I} \models \text{conf}(V_{\text{conf}}, k)$. Thus, \mathcal{I} satisfies the configuration formula of \mathcal{CP} . Therefore, $\mathcal{C} := \mathcal{I}^{V_{\text{conf}}}$ is a correct configuration of \mathcal{CP} . \square

If SOLVEMODULAR returns $r = \text{unsat}$, this does not (in general) imply that \mathcal{CP} is unconfigurable. Rather, it may be that the particular decomposition fails, or even that the particular solution found for \mathcal{CP}_1 is at fault (and another solution would have succeeded).

However, in practice, we have found that the algorithm works well when the decomposition separates a module into two largely independent parts. An example is shown in Figure 3. Here, the two submodules share only a subset of the configuration variables as well as an interface where outputs of the first module flow into inputs of the second module.

¹See the appendix of an extended version of this paper for details on when and why this works [4]. Investigating other possible implementations for GETABDUCT is an interesting direction for future work.

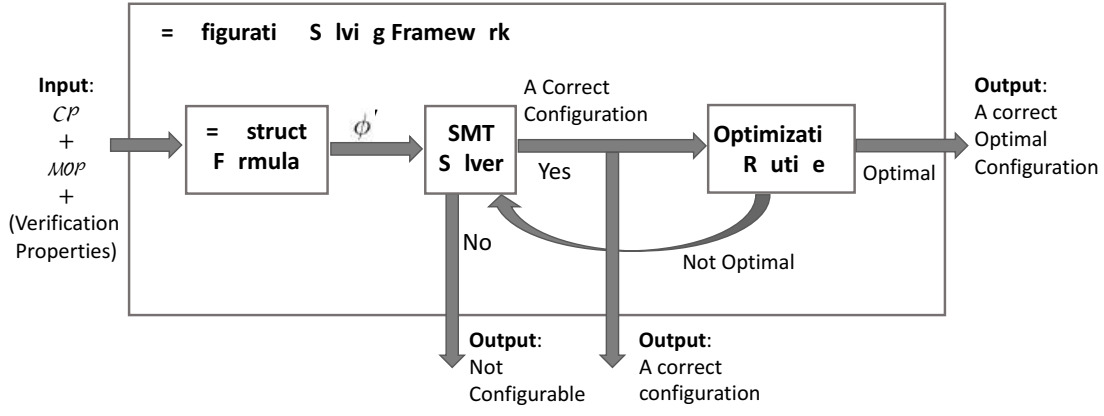


Fig. 4: Optimization-assisted configuration framework. The input is a configuration problem with optional optimization and verification objectives. The framework can return: (i) a non-optimal but correct configuration, or (ii) an optimal and correct configuration, or (iii) *unsat*. ϕ' is a conjunction of the configuration formula ϕ and the optional verification properties.

IV. OPTIMIZATION-ASSISTED CONFIGURATION

A solver can return an unnatural or non-intuitive configuration, complicating the ability of users to understand or maintain the configuration.

We observe that users tend to prefer the simplest configurations, where the notion of simplest corresponds to minimizing some metric when finding solutions. To this end, we show how to extend our framework with optimization goals.

Figure 4 depicts our configuration framework extended with support for multi-objective optimization. There are various ways to combine optimization with configuration solving; we depict one approach using iteration. One instance of this approach works as follows: first a solution is found and the value of the objective term is calculated; then the search space is systematically explored by iteratively constraining the value to be better than the current best value; when no better value can be found, the optimal value has been discovered. There are many different kinds of optimizations that fit this general framework. We present several useful examples in the context of the case study in Section V.

Further extensions. Figure 4 also includes an extension to support combining configuration-finding with verification. In this scheme, any invariants that the system should obey are conjoined to the configuration formula. This ensures that any configuration found satisfies the invariant up to bound k . To check that an invariant holds for all reachable states requires a separate run of an unbounded model checker.

Finding the configuration itself using unbounded model checking is an interesting direction for future work. A significant challenge is that this requires writing the input-output property as a single state formula, which may be much harder than writing it as a bounded set of input, output pairs (in much the same way that loop invariants are difficult to come up with in software). If the input-output property can be written as a state formula P , it may be possible to utilize invariant synthesis techniques by seeking to synthesize an invariant of the form: $\bigwedge_i (V_{\text{conf}}^i = C^i) \implies P$, where the

left-hand side of the implication contains all configuration variables $V_{\text{conf}}^i \in V_{\text{conf}}$, and each C^i is a constant value to be synthesized.

V. CASE STUDY

We present a case study with a course-grained reconfigurable architecture (CGRA) design developed in the Agile Hardware Center at Stanford University [5]. Reconfigurable architectures are appealing because they offer the high performance of hardware with software-like flexibility. CGRAs in particular use sophisticated reconfigurable elements with the aim of narrowing the performance gap with custom ASICs [6].

However, configuring a CGRA is challenging, typically requiring manual effort by an experienced engineer who fully understands the application and the design. To the best of our knowledge, ours is the first framework that finds correct CGRA configurations fully automatically.

In this paper, we focus on configuring a *memory tile* of the CGRA for image processing applications. In these applications data is streamed into the memory tile and must be reordered in various ways before being streamed out. Only the timing and order of the data are changed; the data itself remains the same. Below, we first describe the memory tile design, then present some specific applications, and then explain how we automate configuration of the design for these applications.

A. CGRA Memory Tile Design

The memory tile is a non-trivial design (34998 FF and 164696 gates). Figure 5 shows its architecture. It contains three types of units: *memories*, *addressors*, and *accessors*. Addressors and accessors are reconfigurable units. The accessors control *when* to write or read. The addressors control *where* to write or read. There are three memory modules: an *aggregator* module (AGG), a *static random-access memory* module (SRAM), and a *transpose buffer* module (TB). Each module has an *input accessor* and an *input addressor* associated with it for writes, and an *output accessor* and an *output addressor* for reads. The modules are chained: outputs of AGG are inputs

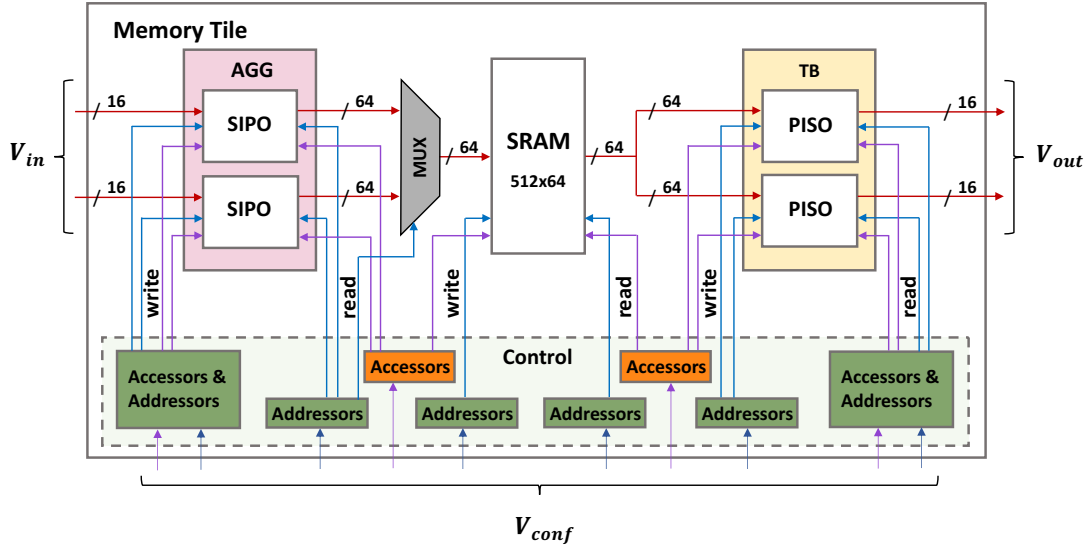


Fig. 5: Memory tile architecture. All accessors and addressors are included in the *control* box. Red arrows represent data flow. Blue and purple arrows represent addressor and accessor control signals, respectively. Green boxes are local to a single module. Orange boxes are shared between modules. V_{conf} consists of all accessor and addressor configuration variables.

Procedure AFFINESEQUENCE

Input: dim : a value indicating the number of nested loops,
 $ranges[dim]$: an array of loop bounds, one for each loop,
 $strides[dim]$: an array of strides, one for each loop,
 $offset$: the offset for the address computation
Output: $vals[\Pi_i ranges[i]]$: a set of output addresses
1: var $c[dim]$; ▷ Index variables for each loop
2: var $i := 0$;
3: **for** $c[dim - 1]$ **in** $[0, ranges[dim - 1]]$ **do**
4: ...
5: **for** $c[0]$ **in** $[0, ranges[0]]$ **do**
6: $vals[i] := \prod_{j=0}^{dim-1} c[j] * strides[j] + offset$;
7: $i := i + 1$;
8: **end for**
9: **end for**

Fig. 6: Affine sequence generator using nested loops.

to SRAM, and outputs of SRAM are inputs to TB. Accessors are *shared* between each pair of connected memory modules. Shared accessors act as *schedule generators* for each memory connection. They specify when the data should be transferred and set any required delays between when the data is produced and consumed. Addressors are unique for each module.

The addressors and accessors in the memory tile make use of affine sequence generators to generate sequences of values for reading and writing. Figure 6 shows pseudocode for an affine sequence generator. It takes as input a number dim of loops, an array $ranges$ with bounds for each loop, an array $strides$ with strides for each loop, and $offset$ which is a base value. It then computes a sequence of outputs, $vals$, by running dim nested loops, and computing the sum of the offset and the product of each stride with its loop index in the innermost loop. Each of the inputs to the procedure corresponds to a configuration register in the hardware.

While each addressor and accessor contains an affine se-

quence generator, they differ in how they interpret $vals$. For an addressor, $vals$ contains raw addresses sent to a memory (for either reading or writing). For an accessor, $vals$ contains clock cycle counts that are compared to a running cycle counter to determine when to read or write. Note that an (accessor, addressor) pair should have the same values for their dim and $ranges$ variables to ensure that they produce the same number of values. There are 4 accessors (including 2 shared with SRAM) and 4 addressors for AGG (1 for each memory port). TB has 4 accessors (including 2 shared with SRAM) and 4 addressors (1 for each memory port). SRAM has 2 addressors, and shares 2 accessors with AGG and 2 accessors with TB.

The memory tile processes 16-bit words. However, it uses a 512x64-bit SRAM which stores four 16-bit words at each address. The rationale for this design is to emulate a multi-ported SRAM while minimizing the energy consumption per memory access [7]. To match the data width at the SRAM interface, AGG and TB implement width converters. AGG implements a *serial-in to parallel-out* (SIPO) converter—serial data is loaded, one 16-bit word at a time, and these are packed into 64-bit outputs. TB implements a *parallel-in to serial-out* (PISO) converter—parallel data is loaded into the PISO as a 64-bit word and is shifted out of the PISO serially, one 16-bit word at a time. The memory tile uses a 2-input and 2-output port architecture to support more throughput. Thus, AGG and TB contain two SIPOs and two PISOs, respectively.

B. Stencil Applications

We consider a common class of image-processing techniques called *stencils*. Stencil computations usually consist of a multi-stage pipeline, where each stage is a dense linear algebra computation in a local region. So-called *push memories* are

inserted between computation units, whose job is to orchestrate the order and the timing of the data explicitly [8]. We explore configuring memory tiles as push memories for four stencil applications:

- *Identity*. The identity stencil simply streams the input back out in the same order. It is useful as a baseline test and also can be used to implement a fixed delay on a stream.
- *3x3 Convolution*. This stencil is used in a variety of image processing applications [9] (e.g., to blur images). It multiplies a 3x3 sliding image window by a 3x3 kernel of constant values.
- *Cascade*. This application implements a pipeline with two convolution kernels executed in sequence. The Cascade application requires configuration of two memory tiles, denoted by *conv* and *hw*.
- *Harris*. Harris is a corner detection algorithm that can be used to infer image features [10]. It extracts the gradients of an image in different orientations and combines this information using multiple convolutions. This is the most complex of our applications, requiring the configuration of five different memory tiles, which we denote as *cim*, *lxx*, *lxy*, *lyy*, and *pad*.

C. Automating the Memory Tile Configuration

We decompose the memory tile into three sub-modules (for scalability), following the approach shown in Figure 3. The first sub-module includes AGG, its input/output accessor/addressor modules, and the MUX (1372 FF, 19676 gates). The second sub-module includes SRAM, both AGG read accessors, and both TB write accessors (33712 FF, 150750 gates). The third sub-module includes TB and its input/output accessor/addressor modules (1126 FF, 18538 gates). Shared accessors contain the shared configuration variables, whose values are propagated to the next module during modular configuration.

In order to configure each module in the memory tile, we look at the transition system defined by its memory and its accessors and addressors. We then use the “programming by example” approach described above. We specify the input-output property P as a sequence of distinct input values (e.g., 1,2,3,...), paired with the corresponding application-specific desired output sequence based on those values. We then solve for the configuration variables as described in Section III-A above.

As mentioned in Section IV, it is important to generate configurations that can easily be read and understood. Working together with the designers, we devised a set of optimization objectives that greatly improve the readability of memory tile configurations. We explain these next. We apply the framework of Figure 4 to configure and optimize each module separately.

Objective 1: we first minimize the dim variables in the module, since this corresponds to using fewer nested loops and fewer loop counters, resulting in simpler solutions in general. We prioritize minimizing dim variables controlling writes over those controlling reads, as lower write complexity

leads to lower read complexity anyway. We formalize this as the following multi-objective optimization problem:

$$\begin{aligned} \mathcal{MOP}_1 &:= \{\mathcal{OP}_1, \mathcal{OP}_w^1, \dots, \mathcal{OP}_w^{d_w}, \mathcal{OP}_r^1, \dots, \mathcal{OP}_r^{d_r}\} : \\ \mathcal{OP}_1 &:= \langle \sum_i dim_i, A_{BV}, \preceq_{BV}, \phi, min \rangle \text{ for } i \in [1, d], \\ \mathcal{OP}_w^i &:= \langle dim_w^i, A_{BV}, \preceq_{BV}, \phi, min \rangle \text{ for } i \in [1, d_w] \\ \mathcal{OP}_r^i &:= \langle dim_r^i, A_{BV}, \preceq_{BV}, \phi, min \rangle \text{ for } i \in [1, d_r] \end{aligned}$$

Here, A_{BV} is the domain of bit-vectors (i.e., unsigned machine integers), \preceq_{BV} is the usual total order on bit-vector values, d is the number of affine sequence generators in the module, and dim_i for $i \in [1, d]$ are all of the dim variables in the module. These are further partitioned into write dimensionality variables dim_w^i , $i \in [1, d_w]$, and read dimensionality variables, dim_r^i , $i \in [1, d_r]$, with $d_w + d_r = d$. ϕ is the configuration formula.

Objective 2: we minimize the products of the range configuration variables in each loop-nest structure. The objective term corresponds to the aggregate number of reads or writes that occur to a particular memory. By minimizing this number, we eliminate unnecessary reads and writes to the memory. Formally, the optimization problem is:

$$\mathcal{OP}_2 := \langle \sum_{i=0}^{d-1} \prod_{j=0}^{dim_i-1} ranges_i[j], A_{BV}, \preceq_{BV}, \phi, min \rangle$$

Objective 3: we minimize stride variables to avoid generating configurations using unnecessarily large addresses.

Many different sets of values for strides could produce the same *vals* stream in the end, so by choosing the smallest values, we hope to generate the simplest solution. The optimization problem simply minimizes the sum of all stride variables in the module:

$$\mathcal{OP}_3 := \langle \sum_i strides_i, A_{BV}, \preceq_{BV}, \phi, min \rangle.$$

Objective 4: we also minimize *offset* configuration variables in addressor modules. For addressor modules, minimizing the *offset* addressor variable prevents unnecessary offsets, improving the readability of the generated configuration. Note that values of *offset* variables in the accessors are fixed by the application. The corresponding problem is as follows, minimizing the sum of all addressor *offset* variables in the module:

$$\mathcal{OP}_4 := \langle \sum_i offset_i, A_{BV}, \preceq_{BV}, \phi, min \rangle.$$

Combined objective: the combined optimization query includes all four objectives and captures the full set of optimization objectives for each module:

$$\mathcal{MOP}_{\mathcal{H}} := \{\mathcal{MOP}_1, \mathcal{OP}_2, \mathcal{OP}_3, \mathcal{OP}_4\}.$$

We solve and prioritize \mathcal{MOP}_1 by iteratively increasing the bound on the sum $\sum_i dim_i$, and for each bound, trying all possible assignments to the variables, in the order specified by \mathcal{MOP}_1 . Note that this approach does not directly fit the scheme described in Figure 4, since it does not require finding a first solution that is iteratively improved. Instead, it

iteratively widens the search space until the first solution is found.

For the other objectives, we use a branch-and-bound algorithm. First, a solution is found, and the value of the term is calculated; then, the solution space is explored systematically, by iteratively constraining the value of the objective term to be better than the current best value. Each optimal solution is propagated to the next optimization objective as a constraint.

VI. EVALUATION

Implementation. We have implemented our framework using Pono [11], an open-source SMT-based model checker. Pono is built on Smt-Switch [12], a generic C++ API for interacting with SMT solvers. Pono provides infrastructure for reading in, unrolling, and otherwise manipulating transition systems. We use Boolector [13] as the underlying SMT solver. We convert the memory tile design in our case study from a SystemVerilog representation to its equivalent representation in the Btor2 format [13], which is accepted by Pono. We use Yosys [14], a Verilog synthesis suite, to do the translation. The experimental code is available at <https://github.com/StanfordAHA/Configuration/>.

Experimental Results. We evaluate our configuration-finding framework using the memory tile design and the four stencil applications described in Section V. For each application, we generated benchmarks for various input image sizes, from 16x16 to 60x60. For applications that require more than one memory tile (i.e., cascade and harris), we choose one representative configuration problem: conv for Cascade and lxx for Harris (more results appear in the appendix of an extended version of this paper [4]). The number of transitions required for each configuration problem is based on the number of clock cycles it takes to process an image of a given size for a given application.

For each benchmark, we first run the basic algorithm described in Section III, which finds the first satisfying configuration. We try both with and without the modular approach described in Section III-B. We then run our optimization-assisted configuration algorithm (using only the modular approach) as described in Section IV. We run our experiments on a 2x Intel Xeon E5-2620 v4 @ 2.10GHz 8-core 128GB computer. Timeout is set to 4000 seconds. Memory limit is 100 GB.

The results are shown in Figure 7. Each chart shows results for both the basic algorithm (First Configuration) and the optimization-assisted algorithm (Optimal Configuration). Within each of these categories, up to five different results are shown for each image size: *top* is the time required to configure the entire design, monolithically; *agg*, *tb*, and *sram* refer to the time required to configure each of the sub-modules independently; and *sram_agg_tb* is the time required to configure the SRAM module after first configuring AGG and TB (this is the most efficient order for these modules) and then propagating the shared configurations from those modules as described in Figure 3. Note that in the modular

approach, AGG and TB are configured independently; thus, the configuration can be performed in parallel, and the total design configuration time is the sum of *sram_agg_tb* and the maximum of *agg* and *tb*. Timeouts are represented by full bars (up to the timeout limit), and memory outs are represented by omitting the bar completely. We also omit the bar for *sram_agg_tb* if either AGG or TB is not solved within the given time-memory budget. We make several observations about the results below.

Modular Approach. As the experiments show, the full memory tile is too large to solve within the given time-memory budget—it times out for all image sizes. However, by using the modular approach, we are able to configure the design for all applications for reasonably useful image sizes. For the Identity Stream, we can configure for all image sizes (with unroll depths up to 3601) relatively easily using the modular approach. Other applications are more challenging, but we are still able to scale up to images of size 40x40 (and unroll depth up to 1939 clock cycles).

We also observe that the AGG and TB modules take comparable time for the Identity Stream, but for other applications, configuration of the TB module is more challenging. This can be explained as follows. AGG and TB are both two-port designs, comparable in size and complexity. But for all applications, AGG can be configured by exploiting only a single port, while only the Identity Stream allows a single-port configuration of TB. Thus, we quickly find a simple configuration for TB with the Identity Stream, but no comparatively simple configuration exists for the other applications.

Optimal Configurations. The right-hand side of each chart shows the results of running our optimization-assisted configuration algorithm for each application. There are several interesting observations. First of all, for the AGG and TB modules, finding optimal configurations is generally more expensive. However, once these optimal configurations are found, it is often easier to find the corresponding SRAM configuration, suggesting that optimal configurations may help improve later stages of modular configuration. The total configuration time with optimization is generally comparable to or only slightly worse than the time required to configure without optimization. Given the value of optimal configurations in terms of simplicity and readability, these results suggest that modular configuration with optimization may be the best strategy in practice.

VII. RELATED WORK

The problem of system configuration has been studied in various formulations and domains, such as software tool configuration, hardware configuration, network configuration, distributed application configuration, and deployment strategies. In one research stream, the configuration problem is to select and arrange a set of components from a given set of assets in order to construct an overall system with a desired specification [15]–[18]. Other formulations take as input a configuration database, including configuration variables, and desired requirements to be met [19], [20]. The task is to find

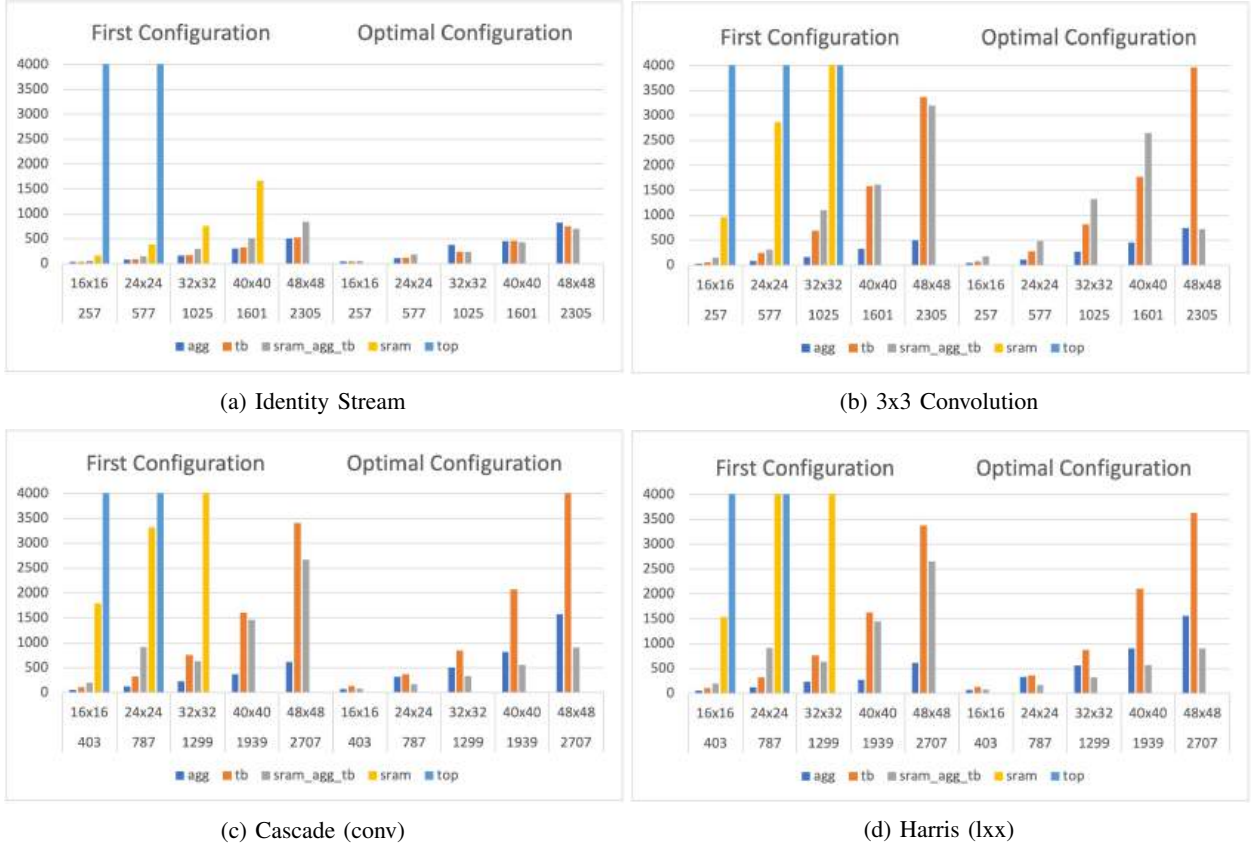


Fig. 7: Horizontal axis shows image sizes and number of clock cycles required for processing. Vertical axis shows time in seconds.

values for the configuration variables which instantiate the database so that it meets the requested requirement. The work whose problem definition is closest to ours is [21], which also uses transition systems. The authors define a configuration as an initial state of a transition system, which is very similar to our notion of configuration variables.

Constraint solving has been explored in various ways for automating system configuration. Efforts have been made to design declarative, constraint-based, object-oriented languages and policy-based tools to configure systems as well as to validate configurations [19], [22]–[24]. Early approaches were based on constraint satisfaction and constraint logic programming [18], [25], [26]. More recent approaches utilize SAT and SMT solvers [17], [19], [27], and counterexample-guided inductive synthesis and relational model finding [21], [28] for dynamic configuration. However, the way these approaches reduce configuration problems to constraint satisfaction problems is significantly different from our approach using input/output examples and unrolling.

More significantly, our work differs in its use of modularity and optimization to improve scalability and understandability. Some automated configuration efforts do employ optimization (e.g., [29]), but with a different goal, namely to configure a system in a way that maximizes its performance.

VIII. CONCLUSION

We proposed a new approach for automatically configuring systems representable as transition systems. Key contributions of our approach include its ability to leverage modularity and its use of optimization. Optimal configurations are more human-understandable, and both modularity and optimization can improve scalability. We demonstrated these claims with a case study using a CGRA memory tile.

Future directions for this work include incorporating unbounded model checking, applying the framework to a wider variety of designs, exploring modularity for more sophisticated theories, and finding provably correct configurations for applications with repeating input/output patterns.

ACKNOWLEDGMENTS


This work was funded in part by the Stanford Agile Hardware Center and by the Defence Advanced Research Projects Agency under grant number FA8650-18-2-7854.


REFERENCES

- [1] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” www.SMT-LIB.org, 2016.
- [2] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 825–885.

- [3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 1579. Springer, 1999, pp. 193–207.
- [4] N. Tsiskaridze, M. Strange, M. Mann, K. Sreedhar, Q. Liu, M. Horowitz, and C. Barrett, "Automating system configuration," 2021. [Online]. Available: <https://arxiv.org/abs/2108.05987>
- [5] "Aha! agile hardware project at stanford university," <https://aha.stanford.edu/>.
- [6] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, no. 6, Oct. 2019. [Online]. Available: <https://doi.org/10.1145/3357375>
- [7] A. Vasilyev, "Evaluating spatially programmable architecture for imaging and vision applications," Ph.D. dissertation, Stanford University, 2019.
- [8] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 137–151.
- [9] R. Chandel and G. Gupta, "Image filtering algorithms and techniques: A review," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 10, 2013.
- [10] C. G. Harris, M. Stephens *et al.*, "A combined corner and edge detector," in *Alvey vision conference*, vol. 15, no. 50. Citeseer, 1988, pp. 10–5244.
- [11] 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 *CAV*, ser. Lecture Notes in Computer Science. Springer, 2021.
- [12] M. Mann, A. Wilson, Y. Zohar, L. Stuntz, A. Irfan, K. Brown, C. Donovan, A. Guman, C. Tinelli, and C. W. Barrett, "Smt-Switch: A Solver-agnostic C++ API for SMT Solving," in *International Conference on Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science. Springer, 2021.
- [13] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , btor2mc 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*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 587–595. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_32
- [14] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.
- [15] J. P. McDermott, "R1: A rule-based configurer of computer systems," *Artif. Intell.*, vol. 19, no. 1, pp. 39–88, 1982. [Online]. Available: [https://doi.org/10.1016/0004-3702\(82\)90021-2](https://doi.org/10.1016/0004-3702(82)90021-2)
- [16] M. A. Mansor, M. Kasihmuddin, and S. Sathasivam, "Vlsi circuit configuration using satisfiability logic in hopfield network," *International Journal of Intelligent Systems and Applications*, vol. 8, pp. 22–29, 2016.
- [17] R. Michel, A. Hubaux, V. Ganesh, and P. Heymans, "An smt-based approach to automated configuration," in *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, ser. EPTC Series in Computing, P. Fontaine and A. Goel, Eds., vol. 20. EasyChair, 2012, pp. 109–119. [Online]. Available: <https://easychair.org/publications/paper/bKGs>
- [18] D. Sabin and E. C. Freuder, "Configuration as composite constraint satisfaction," 1996.
- [19] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative infrastructure configuration synthesis and debugging," *J. Netw. Syst. Manag.*, vol. 16, no. 3, pp. 235–258, 2008. [Online]. Available: <https://doi.org/10.1007/s10922-008-9108-y>
- [20] S. Narain, "Network configuration management via model finding," in *LISA*, 2005.
- [21] T. Nelson, N. Danas, T. Giannakopoulos, and S. Krishnamurthi, "Synthesizing mutable configurations: Setting up systems for success," in *34th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 81–85. [Online]. Available: <https://doi.org/10.1109/ASEW.2019.00034>
- [22] J. Hewson, "Constraint-based specifications for system configuration," 2013.
- [23] L. Ramshaw, A. Sahai, J. Saxe, and S. Singhal, "Cauldron: A policy-based design tool," vol. 2006, 07 2006, pp. 10 pp.–.
- [24] J. Hewson, "Constraint-based specifications for system configuration," Ph.D. dissertation, 11 2013.
- [25] N. Sharma and R. Colomb, "Mechanising shared configuration and diagnosis theories through constraint logic programming," *The Journal of Logic Programming*, vol. 37, no. 1, pp. 255–283, 1998. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0743106698100109>
- [26] J. Tiihonen, M. Heiskala, A. Anderson, and T. Soininen, "Wecotin—a practical logic-based sales configurator," *AI Communications*, vol. 26, no. 1, pp. 99–131, 2013.
- [27] S. Peter and T. Givargis, "Component-based synthesis of embedded systems using satisfiability modulo theories," *ACM Trans. Design Autom. Electr. Syst.*, vol. 20, no. 4, pp. 49:1–49:27, 2015. [Online]. Available: <https://doi.org/10.1145/2746235>
- [28] A. Wagner, "Where to begin? synthesizing initial configurations for cellular automata," 2020.
- [29] J. A. Hewson, P. Anderson, and A. D. Gordon, "A declarative approach to automated configuration," in *Strategies, Tools , and Techniques: Proceedings of the 26th Large Installation System Administration Conference, LISA 2012, San Diego, CA, USA, December 9-14, 2012*, C. Rowland, Ed. USENIX Association, 2012, pp. 51–66. [Online]. Available: <https://www.usenix.org/conference/lisa12/technical-sessions/presentation/hewson>

Towards an Automatic Proof of Lamport's Paxos

Aman Goel 
University of Michigan, Ann Arbor
amangoel@umich.edu

Karem A. Sakallah 
University of Michigan, Ann Arbor
karem@umich.edu

Abstract—Lamport's celebrated Paxos consensus protocol is generally viewed as a complex hard-to-understand algorithm. Notwithstanding its complexity, in this paper, we take a step towards automatically proving the safety of Paxos by taking advantage of three structural features in its specification: *spatial regularity* in its unordered domains, *temporal regularity* in its totally-ordered domain, and its *hierarchical composition*. By carefully integrating these structural features in IC3PO, a novel model checking algorithm, we were able to infer an inductive invariant that identically matches the human-written one previously derived with significant manual effort using interactive theorem proving. While various attempts have been made to verify different versions of Paxos, to the best of our knowledge, this is the first demonstration of an automatically-inferred inductive invariant for Lamport's original Paxos specification. We note that these structural features are not specific to Paxos and that IC3PO can serve as an automatic general-purpose protocol verification tool.

Index Terms—Distributed protocols, incremental induction, inductive invariant, invariant inference, model checking, Paxos.

I. INTRODUCTION

In this paper, we focus on proving the *safety* of distributed protocols like Paxos [1], [2] which form the basis for implementing many efficient and highly fault-tolerant distributed services [3]–[5]. Developed by Lamport, the Paxos consensus protocol allows a set of processes to communicate with each other by exchanging messages and reach agreement on a single value. Verifying the correctness of such a concurrent system requires the derivation of a *quantified inductive invariant* that, together with the protocol specification, acts as an inductive proof of its safety under all possible system behaviors.

Several manual or semi-automatic verification techniques based on interactive theorem proving [6]–[9] have been proposed to derive a safety proof for Paxos. Chand et al. [10] formally verified the TLA+ [11] specification of Paxos by manually deriving a proof using the TLAPS proof assistant [7]. Padon et al. [12] used the Ivy [13] verifier, which requires a user to manually refine automatically-generated counterexamples-to-induction, to obtain an inductive invariant for a simplified version of Paxos in the decidable EPR fragment [14] of first-order logic. The approaches in [15]–[19] are examples of manually-derived *refinement proofs* [20]–[23] that show how a low-level implementation refines a high-level specification. All these methods, however, require a detailed understanding of the intricate inner workings of the protocol and entail significant manual effort to guide proof development.

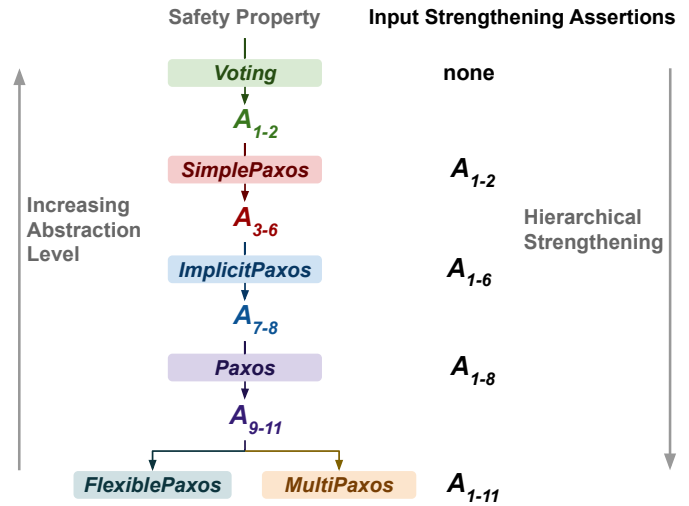


Fig. 1: Hierarchical strengthening of Paxos and its variants. Each level uses all strengthening assertions above that level as input, and outputs the required remaining assertions, altogether inferring the inductive invariant at each level.

In contrast, we propose an approach, implemented in the IC3PO protocol verifier, to *automatically infer the required inductive invariant* for an unbounded distributed protocol by adding three simple extensions to the finite-domain IC3/PDR [24], [25] incremental induction algorithm for model checking [26]. *Symmetry boosting*, introduced in [27], takes advantage of a protocol's *spatial* regularity to automatically infer quantified strengthening assertions that reflect the protocol's structural symmetries. This paper describes *range boosting* and *hierarchical strengthening* which take advantage, respectively, of a protocol's *temporal* regularity and hierarchical structure, and demonstrates how IC3PO was used to automatically obtain an inductive invariant for Paxos using the four-level hierarchy shown in Figure 1.

Our main contributions are:

- A *range boosting* technique that extends incremental induction to utilize the *temporal regularity* in totally-ordered domains, and thus, enables automatic invariant inference for protocols with even *infinite-state* processes.
- A *hierarchical strengthening* approach to derive the required inductive invariant in a top-down step-wise procedure for hierarchically-specified distributed protocols through incremental induction extended with symmetry and range boosting, by automatically verifying high-level abstractions first and using invariants of these higher-

level abstractions as *strengthening assertions* to derive the inductive invariant for the detailed lower-level protocol.

- Safety verification of *Lamport’s Paxos algorithm*, both single- and multi-decree Paxos, through the derivation of a compact, human-readable inductive proof that is automatically inferred using IC3PO, resulting in a drastic reduction in verification effort compared to previous approaches [16], [28], [29].

The paper is structured as follows: §II presents preliminaries. §III and §IV describe range boosting and hierarchical strengthening. §V details the four-level hierarchy we used to prove Paxos and §VI is a record of the IC3PO run showing the actual assertions it inferred at each level of the hierarchy. §VII discusses some of the features and interesting details on this automatically-generated proof. Experimental comparisons with other approaches are provided in §VIII and the paper concludes with a brief survey of related work in §IX and a discussion of future directions in §X.

II. PRELIMINARIES

A. Notation

We will use *Init*, *Next*, and *Safet* to denote the quantified formulas that specify, respectively, a protocol’s initial states, its transition relation, and the safety property that is required to hold on all reachable states. We use primes (e.g., φ') to represent a formula after a single transition step. The notation $V!A$ (resp. $S!A$, $I!A$, and $P!A$) means that assertion A was inferred by IC3PO for the *Voting* (resp. *SimplePaxos*, *ImplicitPaxos*, and *Paxos*) protocol.

As an example, consider a protocol \mathcal{P} with two sorts, a symmetric sort *aSort* and a totally-ordered sort *bSort*, along with relations $p(\text{aSort}, \text{bSort})$ and $q(\text{bSort})$ defined on these sorts. Viewed as a parameterized system $\mathcal{P}(\text{aSort}, \text{bSort})$, we can specify its finite instance $\mathcal{P}(3, 4)$ as:

$$\begin{aligned} \mathcal{P}(3, 4) : \quad & \text{aSort}_3 \triangleq \{a_1, a_2, a_3\} \\ & \text{bSort}_4 \triangleq [b_{\min}, b_1, b_2, b_{\max}] \end{aligned} \quad (1)$$

where aSort_3 represents the finite symmetric sort of this instance defined as a set of arbitrarily-named distinct constants, while the finite totally-ordered sort bSort_4 is composed of a list of ordered constants, i.e., $b_{\min} < b_1 < b_2 < b_{\max}$. This instance can be encoded using twelve p and four q BOOLEAN state variables. A *state* of this instance corresponds to a complete assignment to these 16 state variables, with a total state-space size of 2^{16} . We will use $\widehat{\text{Next}}$ instead of *Next* to denote the transition relation of the finite instance.

B. Clause Boosting and Quantifier Inference

The basic framework for inferring the quantified assertions required to prove protocol safety is described in [27]. It extends the finite IC3/PDR incremental induction algorithm by *boosting* its clause learning during the 1-step backward reachability checks performed through Satisfiability Modulo Theories (SMT) [30] solving. Specifically, a clause φ is learned in (and refines) frame F_i if the 1-step query $\psi_i :=$

$F_{i-1} \wedge \widehat{\text{Next}} \wedge [\neg\varphi']$ is unsatisfiable. This means that cube $\neg\varphi$ in frame F_i is unreachable from frame F_{i-1} . Boosting refers to: a) “growing” φ to a set of clauses that also satisfy this *unreachability constraint* from frame F_{i-1} , and b) refining the frame F_i with the entire clause set instead of just φ . Such boosting accelerates the convergence of incremental induction but, more importantly, makes it possible, under some regularity assumptions, to represent this set of clauses by a *single logically-equivalent quantified clause* Φ and is the key to generalizing the results of such finite analysis to unbounded domains.

C. Symmetric Boosting and Quantifier Inference

Protocols that are strictly specified in terms of symmetric sorts can be characterized as having *spatial* regularity. For example, the constants in a sort representing a finite set of k identical processes are essentially indistinguishable *replicas* that can be permuted arbitrarily without changing the protocol behavior. A learned clause φ parameterized by the constants of such a sort can be boosted by permuting its constants in all possible $k!$ ways yielding a set of symmetrically-equivalent clauses, i.e., its symmetry *orbit* φ^{Sym_k} under the full symmetric group S_{m_k} . By construction, all clauses in φ ’s orbit automatically satisfy the unreachability constraint without the need to perform additional 1-step queries. Furthermore, the quantified clause Φ that encodes φ ’s orbit is algorithmically constructed by a syntactic analysis of φ ’s structure, and can involve complex universal and existential quantifier alternations over both state and non-state (auxiliary) variables. The reader is referred to [27], [31] for the complete details of the connection between symmetry and quantification and the procedure for quantifier inference.

D. Finite Convergence

When a boosted finite incremental induction run terminates, it either produces a finite counterexample demonstrating that the specified safety property fails, or produces a set of quantified assertions A_1, \dots, A_n that yield the inductive invariant $\text{inv} = \text{Safet} \wedge A_1 \wedge \dots \wedge A_n$ proving safety for the given finite size. At this point, an algorithmic *finite convergence* procedure is invoked to check if the current instance size has captured all possible protocol behaviors and, if not, to systematically increase the finite instance size until protocol behavior saturates and the cutoff size is reached [32]–[36].

III. RANGE BOOSTING

Clause boosting is not limited to clauses that are parameterized by the constants of symmetric sorts, and can be extended to clauses whose literals depend on the constants of totally-ordered sorts such as ballot, round, epoch, etc., that are used to model the temporal order of events in a distributed protocol. However, the boosting procedure for such clauses differs from symmetric boosting in two ways: a) the ordering relation between totally-ordered constants must be explicitly preserved, and b) adherence of a boosted clause to the unreachability

constraint is not guaranteed and must be explicitly checked with a 1-step backward reachability query.

We extended IC3PO with a *range boosting* procedure that complements its symmetry boosting mechanism, allowing it to transparently handle protocols with both symmetric and totally-ordered sorts.

Let φ be a clause that is parameterized by totally-ordered constants and let $\varphi^{Ordered}$ denote those variants of φ that are obtained by ordering-compliant permutations of its constants. Clause φ is boosted by making 1-step backward reachability queries on $\varphi^{Ordered}$ to identify its *safe* subset φ^{Safe} , i.e., those variants that satisfy the unreachability constraint.

For example, consider the following clause φ_1 defined on the finite instance $\mathcal{P}(3, 4)$ from (1):

$$\varphi_1 = p(a_1, b_1) \vee q(b_2) \quad (2)$$

Since φ_1 contains two ordered constants (b_1, b_2) , it has six ordering-compliant variants (b_{\min}, b_1) , (b_{\min}, b_2) , (b_{\min}, b_{\max}) , (b_1, b_2) , (b_1, b_{\max}) , and (b_2, b_{\max}) . However only three of these variants end up satisfying the unreachability constraint yielding the following safe subset of $\varphi_1^{Ordered}$:

$$\begin{aligned} \varphi_1^{Safe} = & [p(a_1, b_1) \vee q(b_2)] \wedge \\ & [p(a_1, b_1) \vee q(b_{\max})] \wedge \\ & [p(a_1, b_2) \vee q(b_{\max})] \end{aligned} \quad (3)$$

The inferred quantified clause that encodes these three clauses is now constructed using two universally-quantified variables $X_1, X_2 \in \text{bSort}_4$ that replace b_1 and b_2 in φ_1 and expressed as an implication whose antecedent specifies a constraint over the ordered “range” $b_{\min} < X_1 < X_2$ that must be satisfied by the quantified variables:

$$\begin{aligned} \Phi_1 = & \forall X_1, X_2 \in \text{bSort}_4 : \\ & (b_{\min} < X_1) \wedge (X_1 < X_2) \rightarrow [p(a_1, X_1) \vee q(X_2)] \end{aligned} \quad (4)$$

In general, a clause that is parameterized by k constants from a totally-ordered domain whose size is greater than k can be range-boosted and encoded by a universally-quantified predicate with k variables which is expressed as an implication whose antecedent is a range constraint that evaluates to true for just those combinations of the k variables that correspond to safe variants of φ .

This procedure extends easily to the case of multiple totally-ordered domains as well, allowing range boosting to be performed independently for each such domain in *any* order since constants from different domains do not interfere with each other.

IV. HIERARCHICAL STRENGTHENING

As advocated in [37], hierarchical structuring is an effective way to manage complexity during manual proof development. It can also be easily incorporated in the IC3PO style of invariant generation based on symmetry and range boosting.

Given a low-level specification L that implements a high-level specification H , i.e., $L \prec H$, hierarchical strengthen-

ing starts by automatically deriving strengthening assertions $H!A^H$ that, together with the safety property $H!Safet$, proves the safety of H . It then maps and propagates $H!A^H$ to L , denoted as $L!A^H$, and proceeds to prove the strengthened property $L!Safet \wedge L!A^H$ in L by deriving any additional assertions $L!A^L$ needed to establish the safety of L . The underlying assumption in this procedure is that proving H is much easier than proving L directly, and that any assertions derived to prove H are also applicable, with suitable mapping, to L . The final inductive invariant that proves L will, thus, have the form $L!inv = (L!Safet \wedge L!A^H) \wedge L!A^L$ which can be interpreted as reducing the complexity of L ’s proof by strengthening its safety property with assertions derived for H .

Such strengthening can be extended to a k -level hierarchy $H \prec M_1 \prec \dots \prec M_{k-2} \prec L$, where M_1 to M_{k-2} are suitably-defined intermediate levels between H and L . This, in turn, allows single-level automatic verification techniques based on incremental induction, like IC3PO, to scale to complex protocols like *Paxos*, by step-wise verifying higher-level abstractions first and using their auto-generated proofs to incrementally build the proof for the lower-level protocol.

V. HIERARCHICAL SPECIFICATION OF PAXOS

This section describes in detail the multi-level hierarchical structure of the Paxos protocol, as shown earlier in Figure 1.

A. Lamport’s Voting Protocol

Figure 2 presents the TLA+ [11] description¹ of the *Voting* protocol [38], which is a very high-level abstraction of *Paxos* that formalizes the way Lamport first thought about the Paxos consensus algorithm without getting distracted by details introduced by having the processes communicate by messages. *Voting* has three unordered sorts named *value*, *acceptor* and *quorum*, and a totally-ordered sort named *ballot*. The protocol has two state symbols, *votes* and *maxBal* defined on these sorts that serve as the protocol’s state variables. *votes*(a, b, v) is true iff an acceptor a has voted for value v in ballot number b . *maxBal*(a) returns a ballot number such that acceptor a will never cast any further vote in a ballot numbered less than *maxBal*(a). The global axiom (line 5) defines the elements of the *quorum* sort to be subsets of the *acceptor* sort and restricts them further by requiring them to be pair-wise non-disjoint. Lines 6-9 specify definitions *chosenAt*, *chosen*, *showsSafeAt*, and *isSafeAt*, which serve as auxiliary non-state variables. Protocol transitions are specified by the actions *IncreaseMaxBal* and *VoteFor* (lines 10-11), and lines 12-14 specify the protocol’s initial states, transition relation, and safety property.

¹Lamport’s TLA+ encoding uses sets to denote variables. For example in [38], *votes*[a] represents the set of votes cast by acceptor a . Throughout this paper, we use an equivalent representation based on relations/functions to enable encoding for SMT solving. $\langle b, v \rangle \in \text{votes}[a]$ is equivalently encoded in relational form as *votes*(a, b, v) = \top .

MODULE Voting	
1	CONSTANTS value, acceptor, quorum
2	ballot $\triangleq \text{Nat} \cup \{-1\}$
3	VARIABLES votes, maxBal
4	votes $\in (\text{acceptor} \times \text{ballot} \times \text{value}) \rightarrow \text{BOOLEAN}$ maxBal $\in \text{acceptor} \rightarrow \text{ballot}$
5	ASSUME $\wedge \forall Q \in \text{quorum} : Q \subseteq \text{acceptor}$ $\wedge \forall Q_1, Q_2 \in \text{quorum} : Q_1 \cap Q_2 \neq \{\}$
6	chosenAt(b, v) $\triangleq \exists Q \in \text{quorum} : \forall A \in Q : \text{votes}(A, b, v)$
7	chosen(v) $\triangleq \exists B \in \text{ballot} : \text{chosenAt}(B, v)$
8	showsSafeAt(q, b, v) \triangleq $\wedge \forall A \in q : \text{maxBal}(A) \geq b$ $\wedge \exists C \in \text{ballot} :$ $\wedge (C < b)$ $\wedge (C \neq -1) \rightarrow \exists A \in q : \text{votes}(A, C, v)$ $\wedge \forall D \in \text{ballot} :$ $(C < D < b) \rightarrow$ $\forall A \in Q : \forall V \in \text{value} : \neg \text{votes}(A, D, V)$
9	isSafeAt(b, v) $\triangleq \exists Q \in \text{quorum} : \text{showsSafeAt}(Q, b, v)$
10	IncreaseMaxBal(a, b) \triangleq $\wedge b \neq -1 \wedge b > \text{maxBal}(a)$ $\wedge \text{maxBal}' = [\text{maxBal} \text{ EXCEPT } ![a] = b]$ $\wedge \text{UNCHANGED votes}$
11	VoteFor(a, b, v) \triangleq $\wedge b \neq -1 \wedge \text{maxBal}(a) \leq b$ $\wedge \forall V \in \text{value} : \neg \text{votes}(a, b, V)$ $\wedge \forall C \in \text{acceptor} :$ $(C \neq a) \rightarrow$ $\forall V \in \text{value} : \text{votes}(C, b, V) \rightarrow (V = v)$ $\wedge \text{isSafeAt}(b, v)$ $\wedge \text{votes}' = [\text{votes} \text{ EXCEPT } ![a, b, v] = \top]$ $\wedge \text{maxBal}' = [\text{maxBal} \text{ EXCEPT } ![a] = b]$
12	Init \triangleq $\wedge \forall A \in \text{acceptor} : B \in \text{ballot} : V \in \text{value} : \neg \text{votes}(A, B, V)$ $\wedge \forall A \in \text{acceptor} : \text{maxBal}(A) = -1$
13	Next $\triangleq \exists A \in \text{acceptor}, B \in \text{ballot}, V \in \text{value} :$ IncreaseMaxBal(A, B) \vee VoteFor(A, B, V)
14	Safety $\triangleq \forall V_1, V_2 \in \text{value} : \text{chosen}(V_1) \wedge \text{chosen}(V_2) \rightarrow V_1 = V_2$

Fig. 2: Lamport's Voting protocol in pretty-printed TLA+

Viewed as a parameterized system, the template of the *Voting* protocol is *Voting*(value, acceptor, quorum, ballot). Its finite instance:

Voting(2, 3, 3, 4) :

value₂ $\triangleq \{v_1, v_2\}$
acceptor₃ $\triangleq \{a_1, a_2, a_3\}$
quorum₃ $\triangleq \{q_{12} : \{a_1, a_2\}, q_{13} : \{a_1, a_3\}, q_{23} : \{a_2, a_3\}\}$
ballot₄ $\triangleq [b_{\min}, b_1, b_2, b_{\max}]$

has three finite symmetric sorts named value₂, acceptor₃ and quorum₃, defined as sets of arbitrarily-named distinct constants, while the finite totally-ordered sort ballot₄ is composed of a list of ordered constants, i.e., $b_{\min} < b_1 < b_2 < b_{\max}$, where $b_{\min} = -1$ since -1 is the “minimum” ballot number. The constants of the quorum₃ sort are subsets of the acceptor₃ sort and are named to reflect

MODULE Paxos	
1	CONSTANTS value, acceptor, quorum
2	ballot $\triangleq \text{Nat} \cup \{-1\}$
3	VARIABLES msg1a, msg1b, msg2a, msg2b, maxBal maxVBal, maxVal
4	msg1a $\in \text{ballot} \rightarrow \text{BOOLEAN}$ msg1b $\in (\text{acceptor} \times \text{ballot} \times \text{ballot} \times \text{value}) \rightarrow \text{BOOLEAN}$ msg2a $\in (\text{ballot} \times \text{value}) \rightarrow \text{BOOLEAN}$ msg2b $\in (\text{acceptor} \times \text{ballot} \times \text{value}) \rightarrow \text{BOOLEAN}$ maxBal $\in \text{acceptor} \rightarrow \text{ballot}$ maxVBal $\in \text{acceptor} \rightarrow \text{ballot}$ maxVal $\in \text{acceptor} \rightarrow \text{value}$ none $\in \text{value}$
5	ASSUME $\wedge \forall Q \in \text{quorum} : Q \subseteq \text{acceptor}$ $\wedge \forall Q_1, Q_2 \in \text{quorum} : Q_1 \cap Q_2 \neq \{\}$
6	chosenAt(b, v) $\triangleq \exists Q \in \text{quorum} : \forall A \in Q : \text{msg2b}(A, b, v)$
7	chosen(v) $\triangleq \exists B \in \text{ballot} : \text{chosenAt}(B, v)$
8	showsSafeAtPaxos(q, b, v) \triangleq $\wedge \forall A \in q : \exists M_b \in \text{ballot} : \exists M \in \text{value} : \text{msg1b}(A, b, M_b, M)$ $\wedge \forall A \in \text{acceptor} : \forall M_b \in \text{ballot} : \forall M \in \text{value} :$ $\neg (A \in q \wedge \text{msg1b}(A, b, M_b, M) \wedge (M_b \neq -1))$ $\vee \exists M_b \in \text{ballot} :$ $\wedge \exists A \in q : \text{msg1b}(A, b, M_b, v) \wedge (M_b \neq -1)$ $\wedge \forall A \in q : \forall M_{b2} \in \text{ballot} : \forall M_2 \in \text{value} :$ $\text{msg1b}(A, b, M_{b2}, M_2) \wedge (M_{b2} \neq -1) \rightarrow M_{b2} \leq M_b$
9	isSafeAtPaxos(b, v) $\triangleq \exists Q \in \text{quorum} : \text{showsSafeAtPaxos}(Q, b, v)$
10	Phase1a(b) \triangleq $\wedge b \neq -1$ $\wedge \text{msg1a}' = [\text{msg1a} \text{ EXCEPT } ![b] = \top]$ $\wedge \text{UNCHANGED msg1b, msg2a, msg2b, maxBal, maxVBal, maxVal}$
11	Phase1b(a, b) \triangleq $\wedge b \neq -1 \wedge \text{msg1a}(b) \wedge b > \text{maxBal}(a)$ $\wedge \text{maxBal}' = [\text{maxBal} \text{ EXCEPT } ![a] = b]$ $\wedge \text{msg1b}' = [\text{msg1b} \text{ EXCEPT } ![a, b, \text{maxVBal}(a), \text{maxVal}(a)] = \top]$ $\wedge \text{UNCHANGED msg1a, msg2a, msg2b, maxVBal, maxVal}$
12	Phase2a(b, v) \triangleq $\wedge b \neq -1 \wedge v \neq \text{none} \wedge \neg (\exists V \in \text{value} : \text{msg2a}(b, V))$ $\wedge \text{isSafeAtPaxos}(b, v)$ $\wedge \text{msg2a}' = [\text{msg2a} \text{ EXCEPT } ![b, v] = \top]$ $\wedge \text{UNCHANGED msg1a, msg1b, msg2b, maxBal, maxVBal, maxVal}$
13	Phase2b(a, b, v) \triangleq $\wedge b \neq -1 \wedge v \neq \text{none} \wedge \text{msg2a}(b, v) \wedge b \geq \text{maxBal}(a)$ $\wedge \text{maxBal}' = [\text{maxBal} \text{ EXCEPT } ![a] = b]$ $\wedge \text{maxVBal}' = [\text{maxVBal} \text{ EXCEPT } ![a] = b]$ $\wedge \text{maxVal}' = [\text{maxVal} \text{ EXCEPT } ![a] = v]$ $\wedge \text{msg2b}' = [\text{msg2b} \text{ EXCEPT } ![a, b, v] = \top]$ $\wedge \text{UNCHANGED msg1a, msg1b, msg2a}$
14	Init $\triangleq \forall A \in \text{acceptor} : B \in \text{ballot} :$ $\wedge \neg \text{msg1a}(B)$ $\wedge \forall M_b \in \text{ballot} : M \in \text{value} : \neg \text{msg1b}(A, B, M_b, M)$ $\wedge \forall V \in \text{value} : \neg \text{msg2a}(B, V) \wedge \neg \text{msg2b}(A, B, V)$ $\wedge \text{maxBal}(A) = -1$ $\wedge \text{maxVBal}(A) = -1 \wedge \text{maxVal}(A) = \text{none}$
15	Next $\triangleq \exists A \in \text{acceptor} : B \in \text{ballot} : V \in \text{value} :$ $\vee \text{Phase1a}(B) \vee \text{Phase1b}(A, B)$ $\vee \text{Phase2a}(B, V) \vee \text{Phase2b}(A, B, V)$
16	Safety $\triangleq \forall V_1, V_2 \in \text{value} : \text{chosen}(V_1) \wedge \text{chosen}(V_2) \rightarrow V_1 = V_2$

Fig. 3: Lamport's Paxos protocol in pretty-printed TLA+

their symmetric dependence on the acceptor_3 sort. This instance has 24 votes state variables that return a BOOLEAN and 3 maxBal state variables that return a ballot number in ballot_4 . A *state* of this instance corresponds to a complete assignment to these 27 state variables.

B. Lamport's Paxos Protocol

Figure 3 presents the TLA+ description of Lamport's *Paxos* protocol [39], which is a specification of the Paxos consensus algorithm [1], [2]. *Paxos* implements *Voting* through the refinement mapping $[\text{votes} \leftarrow \text{msg2b}, \text{maxBal} \leftarrow \text{maxBal}]$, where acceptors now communicate with each other through distributed message passing. State variables msg1a , msg1b , msg2a , and msg2b are used to model the set of different messages that can be sent in the protocol, corresponding to actions *Phase1a*, *Phase1b*, *Phase2a*, and *Phase2b* respectively. The pair $\langle \text{maxVbal}(a), \text{maxVal}(a) \rangle$ is the vote with the largest ballot number cast by acceptor a . The ballot b leader can send a $\text{msg1a}(b)$ by performing the action *Phase1a*(b). *Phase1b*(a, b) implements the *IncreaseMaxBal*(a, b) action from *Voting*, where after receiving $\text{msg1a}(b)$, acceptor a sends msg1b to the ballot b leader containing the values of $\text{maxVbal}(a)$ and $\text{maxVal}(a)$. In the *Phase2a*(b, v) action, the ballot b leader sends msg2a asking the acceptors to vote for a value v that is safe at ballot number b . Its enabling condition $\text{isSafeAtPaxos}(b, v)$ checks the enabling condition $\text{isSafeAt}(b, v)$ from *Voting*. *Phase2b* implements the *VoteFor* action in *Voting*, and enables acceptor a to vote for value v in ballot number b . We refer the reader to [40] for a detailed explanation to understand the internals of *Paxos*.

Represented as a parameterized system $\text{Paxos}(\text{value}, \text{acceptor}, \text{quorum}, \text{ballot})$, its finite instance $\text{Paxos}(2, 3, 3, 4)$ has 132 BOOLEAN state variables, 6 state variables that return a ballot number in ballot_4 , and 3 state variables that return a value in value_2 .

C. Intermediate Levels between Voting and Paxos

We introduced two intermediate levels, *SimplePaxos* and *ImplicitPaxos*, between *Voting* and *Paxos*. These intermediate levels are abstractions of *Paxos*, inspired from the already-existing literature [12], [41]–[44]. *ImplicitPaxos* is inspired from the specification of Generalized Paxos by Lamport [41] and uses a commonly-used encoding transformation, as utilized in [12], [43], [44]. Instead of explicitly keeping a track of $\text{maxVbal}(a)$ and $\text{maxVal}(a)$, *ImplicitPaxos* abstracts them away and implicitly computes their respective values using the history of all votes cast by the acceptor a , i.e., using the history of msg2b from acceptor a , by modifying the *Phase1b*(a, b) action (line 11 in Figure 3) to as shown in Figure 4.

SimplePaxos further simplifies *ImplicitPaxos* and eliminates tracking of the maximum ballot (and the corresponding value) in which an acceptor voted from msg1b completely, i.e., the last two arguments of msg1b are abstracted away. Instead, the history of all votes cast is used to describe how new votes are cast. This is done by replacing the definition

```

MODULE ImplicitPaxos
11 Phase1b(a, b)  $\triangleq$ 
   $\wedge b \neq -1 \wedge \text{msg1a}(b) \wedge b > \text{maxBal}(a)$ 
   $\wedge \text{maxBal}' = [\text{maxBal} \text{ EXCEPT } ![a] = b]$ 
   $\wedge \exists M_b \in \text{ballot} : \exists M \in \text{value} :$ 
     $\wedge \vee \wedge (M_b = -1)$ 
       $\wedge \forall B \in \text{ballot} : \forall V \in \text{value} : \neg \text{msg2b}(a, B, V)$ 
     $\vee \wedge (M_b \neq -1) \wedge \text{msg2b}(a, M_b, M)$ 
       $\wedge \forall B \in \text{ballot} : \forall V \in \text{value} :$ 
         $\text{msg2b}(a, B, V) \rightarrow B \leq M_b$ 
   $\wedge \text{msg1b}' = [\text{msg1b} \text{ EXCEPT } ![a, b, M_b, M] = \top]$ 
   $\wedge \text{UNCHANGED } \text{msg1a}, \text{msg2a}, \text{msg2b}$ 

```

Fig. 4: Modifications in *ImplicitPaxos* compared to *Paxos*

```

MODULE SimplePaxos
8 showsSafeAtSimplePaxos(q, b, v)  $\triangleq$ 
   $\wedge \forall A \in q : \text{msg1b}(A, b)$ 
   $\wedge \forall A \in \text{acceptor} : \forall M_b \in \text{ballot} : \forall M \in \text{value} :$ 
     $\neg (A \in q \wedge \text{msg1b}(A, b) \wedge \text{msg2b}(A, M_b, M))$ 
   $\vee \exists M_b \in \text{ballot} :$ 
     $\wedge \exists A \in q : \text{msg1b}(A, b) \wedge \text{msg2b}(A, M_b, v)$ 
     $\wedge \forall A \in q : \forall M_{b2} \in \text{ballot} : \forall M_2 \in \text{value} :$ 
       $\text{msg1b}(A, b) \wedge \text{msg2b}(A, M_{b2}, M_2) \rightarrow M_{b2} \leq M_b$ 

```

Fig. 5: Modifications in *SimplePaxos* compared to *ImplicitPaxos*

showsSafeAtPaxos (line 8 in Figure 3) with its simplified form, expressed using msg2b as shown in Figure 5.

VI. HIERARCHICAL VERIFICATION OF PAXOS

Using the 4-level hierarchy $\text{Paxos} \prec \text{ImplicitPaxos} \prec \text{SimplePaxos} \prec \text{Voting}$, this section is a “log” of how IC3PO automatically derived the required strengthening assertions that established the safety of *Paxos*.

A. Proving Voting

Using instance $\text{Voting}(2, 3, 3, 4)$, IC3PO proved the safety of *Voting* by automatically deriving the inductive invariant $V!_{\text{inv}} \triangleq V!_{\text{Safe}} \wedge V!_{A_1} \wedge V!_{A_2}$ where

$$V!_{A_1} = \forall A \in \text{acceptor}, B \in \text{ballot}, V \in \text{value} :$$

$$\text{votes}(A, B, V) \rightarrow \text{isSafeA}(B, V)$$

$$V!_{A_2} = \forall A \in \text{acceptor}, B \in \text{ballot}, V_1, V_2 \in \text{value} :$$

$$\text{chosenA}(B, V_1) \wedge \text{votes}(A, B, V_2) \rightarrow (V_1 = V_2)$$

In words, these two strengthening assertions mean:

- A_1 : If an acceptor voted for value V in ballot number B , then V is safe at B .
- A_2 : If value V_1 is chosen at ballot B , then no acceptor can vote for a value different than V_1 in B .

B. Proving SimplePaxos

Using the refinement mapping $[\text{votes} \leftarrow \text{msg2b}, \text{maxBal} \leftarrow \text{maxBal}]$, IC3PO transformed $V!_{A_1}$ and $V!_{A_2}$ to the follow-

ing corresponding versions for *SimplePaxos*:

$$\begin{aligned}
S!A_1 &= \forall A \in \text{acceptor}, B \in \text{ballot}, V \in \text{value} : \\
&\quad \text{msg2b}(A, B, V) \rightarrow \text{isSafeA}(B, V) \\
S!A_2 &= \forall A \in \text{acceptor}, B \in \text{ballot}, V_1, V_2 \in \text{value} : \\
&\quad \text{chosenA}(B, V_1) \wedge \text{msg2b}(A, B, V_2) \rightarrow (V_1 = V_2)
\end{aligned}$$

These two assertions, passed down from the proof of *Voting*, represented a strengthening of the safety property of *SimplePaxos* that allowed IC3PO to prove it with the inductive invariant $S!inv \triangleq S!Safet \wedge \bigwedge_{1 \leq i \leq 6} S!A_i$ where

$$\begin{aligned}
S!A_3 &= \forall B \in \text{ballot}, V \in \text{value} : \\
&\quad \text{msg2a}(B, V) \rightarrow \text{isSafeA}(B, V) \\
S!A_4 &= \forall B \in \text{ballot}, V_1, V_2 \in \text{value} : \\
&\quad \text{msg2a}(B, V_1) \wedge \text{msg2a}(B, V_2) \rightarrow (V_1 = V_2) \\
S!A_5 &= \forall A \in \text{acceptor}, B \in \text{ballot}, V \in \text{value} : \\
&\quad \text{msg2b}(A, B, V) \rightarrow \text{msg2a}(B, V) \\
S!A_6 &= \forall A \in \text{acceptor}, B \in \text{ballot} : \\
&\quad \text{msg1b}(A, B) \rightarrow \text{maxBal}(A) \geq B
\end{aligned}$$

are four additional automatically-generated strengthening assertions that express the following facts about *SimplePaxos*:

- A_3 : If ballot B leader sends a $2a$ message for value V , then V is safe at B .
- A_4 : A ballot leader can send $2a$ messages only for a unique value.
- A_5 : If an acceptor voted for a value in ballot number B , then there is a $2a$ message for that value at B .
- A_6 : If an acceptor has sent a $1b$ message at a ballot number B , then its maxBal is at least as high as B .

C. Proving ImplicitPaxos

All variables from *SimplePaxos* refine to *ImplicitPaxos* as is, except for msg1b that adds explicit tracking of the maximum vote voted by an acceptor in *ImplicitPaxos*. Assertions $S!A_1$ to $S!A_5$ map to $I!A_1$ to $I!A_5$ in *ImplicitPaxos* as is, while $S!A_6$ maps as:

$$\begin{aligned}
I!A_6 &= \forall A \in \text{acceptor}, B, B_{\text{max}} \in \text{ballot}, V_{\text{max}} \in \text{value} : \\
&\quad \text{msg1b}(A, B, B_{\text{max}}, V_{\text{max}}) \rightarrow \text{maxBal}(A) \geq B
\end{aligned}$$

These six assertions, passed down from the proof of *SimplePaxos*, represented a strengthening of the safety property of *ImplicitPaxos* that allowed IC3PO to prove it with the inductive invariant $I!inv \triangleq I!Safet \wedge \bigwedge_{1 \leq i \leq 8} I!A_i$ where

$$\begin{aligned}
I!A_7 &= \forall A \in \text{acceptor}, B, B_{\text{max}} \in \text{ballot}, V_{\text{max}} \in \text{value} : \\
&\quad [(B > -1) \wedge (B_{\text{max}} > -1) \wedge \text{msg1b}(A, B, B_{\text{max}}, V_{\text{max}})] \\
&\quad \rightarrow \text{msg2b}(A, B_{\text{max}}, V_{\text{max}}) \\
I!A_8 &= \forall A \in \text{acceptor}, B, B_{\text{mid}}, B_{\text{max}} \in \text{ballot}, \\
&\quad V, V_{\text{max}} \in \text{value} : \\
&\quad [(B > B_{\text{mid}}) \wedge (B_{\text{mid}} > B_{\text{max}}) \wedge \text{msg1b}(A, B, B_{\text{max}}, V_{\text{max}})] \\
&\quad \rightarrow \neg \text{msg2b}(A, B_{\text{mid}}, V)
\end{aligned}$$

are two additional automatically-generated strengthening assertions that express the following facts about *ImplicitPaxos*:

- A : If an acceptor issued a $1b$ message at ballot number B with the maximum vote $\langle B_{\text{max}}, V_{\text{max}} \rangle$, and both B and B_{max} are higher than -1 , then the acceptor has voted for value V_{max} in ballot B_{max} .
- A_8 : If an acceptor issued a $1b$ message at ballot number B with the maximum vote $\langle B_{\text{max}}, V_{\text{max}} \rangle$, then the acceptor cannot have voted in any ballot number strictly between B_{max} and B .

D. Proving Paxos

All variables from *ImplicitPaxos* refine to *Paxos* trivially, mapping $I!A_1, \dots, I!A_8$ to $P!A_1, \dots, P!A_6$ in *Paxos* as is. These eight assertions, passed down from the proof of *ImplicitPaxos*, represented a strengthening of the safety property of *Paxos* that allowed IC3PO to prove it with the inductive invariant $P!inv \triangleq P!Safet \wedge \bigwedge_{1 \leq i \leq 11} P!A_i$ where

$$\begin{aligned}
P!A_9 &= \forall A \in \text{acceptor} : \text{maxVbal}(A) \leq \text{maxBal}(A) \\
P!A_{10} &= \forall A \in \text{acceptor}, B \in \text{ballot}, V \in \text{value} : \\
&\quad \text{msg2b}(A, B, V) \rightarrow \text{maxVbal}(A) \geq B \\
P!A_{11} &= \forall A \in \text{acceptor} : \text{maxVbal}(A) > -1 \\
&\quad \rightarrow \text{msg2b}(A, \text{maxVbal}(A), \text{maxVal}(A))
\end{aligned}$$

are three additional automatically-generated strengthening assertions that express the following facts about *Paxos*:

- A_9 : maxVbal of an acceptor is less than or equal to its maxBal .
- A_{10} : If an acceptor voted in a ballot number B , then its maxVbal is at least as high as B .
- A_{11} : If acceptor A has its maxVbal higher than -1 , then A has already cast a vote $\langle \text{maxVbal}(A), \text{maxVal}(A) \rangle$.

VII. DISCUSSION

This section provides a discussion about certain key points and features about the *Paxos* proof from Section VI.

A. Comparison against Human-written Invariants

Optionally, the inductive invariant $P!inv$ can be minimized to derive a subsumption-free and closed set of invariants, which removes A_1 and A_2 that are subsumed by the conjunction $A_3 \wedge A_4 \wedge A_5$. After this minimization, the inductive invariant of *Paxos* matches identically with the manually-written and TLAPS-checked inductive invariant from [28], guaranteeing its correctness. Similarly, the inductive invariant of *Voting*, i.e., $V!inv$, matches directly with the manually-written and TLAPS-checked inductive invariant from [45].

B. Benefits of Range Boosting

Assertions A_6 to A_{11} express conditions defined over ordered ranges in the *infinite* totally-ordered ballot domain. Inferring such invariants automatically through IC3PO becomes possible through range boosting (Section III), that extends incremental induction with the knowledge of *temporal regularity* over totally-ordered domains by learning quantified clauses over ordered ranges.

C. Protocol's Formula Structure

Note that A_1 to A_3 use definitions *isSafeAt* and *chosenAt*, which implicitly enables IC3PO to incorporate learning with complex quantifier alternations. Inspired from previous works on the importance of using derived/ghost variables [36], [46], [47], IC3PO utilizes the *formula structure* of the protocol's transition relation in a unique manner, by incorporating *definitions* in the protocol specification as auxiliary non-state variables during reachability analysis, described in detail in [27]. This provides a simple and inexpensive procedure to incorporate clause learning with complex quantifier alternations.

D. Decidability

Protocol specifications at each of the four levels include quantifier alternation cycles that make unbounded SMT reasoning fall into the undecidable fragment of first-order logic. Unsurprisingly, previous works that rely on unbounded SMT reasoning, like SWISS [48], fol-ic3 [49], DistAI [50], I4 [51], and UPDR [52], struggle with verifying Lamport's Paxos. IC3PO, on the other hand, performs incremental induction and finite convergence over finite protocol instances using finite-domain reasoning that is always decidable.

E. Why a Four-Level Hierarchy?

The original Paxos specification is composed of a two-level hierarchy *Paxos* \prec *Voting*. Given the two strengthening assertions A_1 and A_2 from *Voting*, inferring the remaining nine assertions for *Paxos* directly in one step of hierarchical strengthening is difficult, since these two specifications are too far apart to be proved directly. IC3PO struggled with the large state space of *Paxos* and learnt too many weak clauses involving *msg1b*, *maxVBal* and *maxVal*, eventually running out of memory due to invariant inference getting confused with several counterexamples-to-induction. Table I compares the state-space size of protocol instances at each of the four hierarchical levels. Even though 2^{14} is not huge, especially with respect to hardware verification problems [53]–[55], *Paxos* has a dense state-transition graph where state-transitions are tightly coupled with high in- and out- degree, making the problem difficult for automatic invariant inference with incremental induction based model checking.

Adding *ImplicitPaxos* reduced the complexity in *Paxos* by abstracting away *maxVBal* and *maxVal*. Still, scalability remained a challenge due to *msg1b*, that contributed to 96 out of 147 state bits in *Paxos*(2, 3, 3, 4). Adding another level, i.e., *SimplePaxos*, removed 84 out of these 96 state bits by abstracting away explicit tracking of the maximum vote of

an acceptor from *msg1b*. When compared against *Paxos*, *SimplePaxos* is significantly simpler, with a total state-space size to be just 2^{54} for its finite instance *SimplePaxos*(2, 3, 3, 4), which led IC3PO to successfully prove *Paxos* automatically using the four-level hierarchy.

F. Extension to MultiPaxos and FlexiblePaxos

Till now, by *Paxos* we meant *single-decree* Paxos which is the core consensus algorithm underlying the complete Paxos state-machine replication protocol [1], [2], commonly referred to as *MultiPaxos* [43]. In *MultiPaxos*, a sequence of instances execute *single-decree Paxos* such that the value chosen in the i^{th} instance becomes the i^{th} command executed by the replicated state machine. Additionally, if the leader is relatively stable, *Phase1* becomes unnecessary and is skipped, reducing the failure-free message delay from 4 delays to 2 delays.

Mapping each of the assertions A_1, \dots, A_{11} to *MultiPaxos* is trivial, and simply adds the corresponding instance as an additional universally-quantified argument, e.g., A_{11} maps as:

$$\begin{aligned} M!A_{11} = & \forall A \in \text{acceptor}, I \in \text{instances} : \\ & \text{maxVBal}(A, I) > -1 \\ & \rightarrow \text{msg2b}(A, I, \text{maxVBal}(A, I), \text{maxVal}(A, I)) \end{aligned}$$

Unsurprisingly, the 11 strengthening assertions, passed down from the proof of *Paxos*, together with the safety property of *MultiPaxos*, allowed IC3PO to trivially prove it with no additional strengthening assertions needed, meaning $M!Safet \wedge \bigwedge_{1 \leq i \leq 11} M!A_i$ is already an inductive invariant of *MultiPaxos*. As described in previous works [1], [2], [6], [10], the crux of proving the safety of *MultiPaxos* is based on proving *single-decree Paxos* since each consensus instance participates independently without any interference from other instances. Our experiments validated this further.

Similarly, we also tried another Paxos variant called *FlexiblePaxos* [56], which also verifies trivially with the same inductive invariant, i.e., with no additional strengthening assertions needed.

VIII. EXPERIMENTS

IC3PO [57] currently accepts protocol descriptions in the Ivy language [13] and uses the Ivy compiler to extract a logical formulation of the protocol in a SMT-LIB [30] compatible format. To get an idea on the effectiveness of hierarchical strengthening, we also evaluated automatically deriving inductive proofs for EPR variants of Paxos from [12] without any hierarchical strengthening. These specifications describe Paxos in the EPR fragment [14] of first-order logic and also incorporate simplifications equivalent to the ones described for *SimplePaxos* in Section V-C. We performed a detailed comparison against other state-of-the-art techniques for automatically verifying distributed protocols:

- SWISS [48] uses SMT solving to derive an inductive invariant by performing an enumerative search in an optimized and bounded invariant search space.
- fol-ic3 [49], implemented in *mypyvy* [58], extends IC3 with a separators-based technique that performs enumer-

Finite Instance	State-space Size
<i>Voting</i> (2, 3, 3, 4)	2^{30}
<i>SimplePaxos</i> (2, 3, 3, 4)	2^{54}
<i>ImplicitPaxos</i> (2, 3, 3, 4)	2^{138}
<i>Paxos</i> (2, 3, 3, 4)	2^{14}

TABLE I: State-space size for finite instances with 2 value, 3 acceptor, 3 quorum, and 4 ballot

		Time (seconds)							Inv		SMT	
Protocol		S.A.	IC3PO	SWISS	fol-ic3	DistAI	I4	UPDR	IC3PO	Human	IC3PO	I4
EPR	epr-paxos	∅	568	15950*	timeout	error	memout	timeout	6	11	5680	1701556
	epr-flexible_paxos	∅	561	18232*	timeout	error	memout	failure	6	11	1509	1761504
	epr-multi_paxos	∅	timeout	timeout	timeout	error	memout	timeout	—	12	—	1902621
	<i>Voting</i>	∅	64	timeout	timeout	error	memout	timeout	3	3	1057	1714170
ORIGINAL	<i>SimplePaxos</i>	A_{1-2}	51	timeout	timeout	error	failure	timeout	5	5	618	158470
	<i>ImplicitPaxos</i>	A_{1-6}	2008	timeout	timeout	error	failure	timeout	7	7	18329	69715
	<i>Paxos</i>	A_{1-8}	98	timeout	timeout	error	failure	timeout	10	10	668	76030
	<i>MultiPaxos</i>	A_{1-11}	340	timeout	timeout	error	timeout	timeout	10	10	161	—
	<i>FlexiblePaxos</i>	A_{1-11}	1408	timeout	timeout	error	failure	timeout	10	10	161	6983

TABLE II: Comparison of IC3PO against other state-of-the-art verifiers

ORIGINAL problems employ hierarchical strengthening (as detailed in Section VI), while EPR problems do not.

Column 2 (labeled S.A.) lists strengthening assertions added through hierarchical strengthening to the safety property (\emptyset means none).

Columns 3-8 (labeled Time) compare the runtime in seconds. For failed SWISS runs, we include the runtime from [48] (indicated with *). Columns 9-10 (labeled Inv) compare number of assertions in the inductive invariant between IC3PO (with subsumption checking and minimization) and human-written proofs.

Columns 11-12 (labeled SMT) compare total number of SMT queries made by IC3PO versus I4 (until failure for unsuccessful runs).

ative search for a quantified separator in the space of bounded mixed quantifier prefixes.

- DistAI [50] performs data-driven invariant learning by enumerating over possible invariants derived from simulating a protocol at different instance sizes, followed by iteratively refining and checking candidate invariants.
- I4 [51], [59] performs finite-domain IC3 (without accounting for regularity) using the AVR model checker [55], [60], followed by iteratively generalizing and checking the inductive invariant produced by AVR.
- UPDR, from the *mypyvy* [58] framework, implements PDR[✓]/UPDR [61] for verifying distributed protocols.

All experiments were performed on an Intel (R) Xeon CPU (X5670). For each run, we used a 5-hour timeout and a 32 GB memory limit. All tools were executed in their respective default configurations. We used Z3 [62] version 4.8.10, Yices 2 [63] version 2.6.2, and CVC4 [64] version 1.8.

A. Results

Table II summarizes the experimental results. EPR variants were run without any hierarchical strengthening. For ORIGINAL problems, we employed hierarchical strengthening using each tool to verify Lamport’s original Paxos specification (and its variants) through higher-level strengthening assertions that were automatically generated from IC3PO (as detailed in Section VI). Note that ORIGINAL problems include quantifier-alternation cycles that make unbounded SMT reasoning fall into the undecidable fragment of first-order logic.

IC3PO emerges as the only successful technique that verifies Lamport’s Paxos and its variants, and automatically infers the required inductive invariants efficiently. Unsurprisingly, none of the other tools (i.e., SWISS, fol-ic3, DistAI, I4 and UPDR) were able to solve ORIGINAL problems since each of these tools rely on unbounded SMT reasoning and struggle on problems that fall outside the decidable EPR fragment of first-order logic.

B. Discussion

Effect of hierarchical strengthening: Comparing EPR versus ORIGINAL shows the advantages offered by hierarchical strengthening. Even though IC3PO was able to automatically verify EPR versions of single-decree Paxos and flexible Paxos from [12], none of the tools were able to automatically verify the EPR version of multi-decree Paxos. ORIGINAL variants, on the other hand, employed hierarchical strengthening which allowed IC3PO to verify Lamport’s Paxos automatically and efficiently by using the protocol’s hierarchical structure.

Comparison against other verifiers: DistAI failed on all problems due to unsupported constructs and parsing errors. I4 and UPDR (as well as DistAI) are limited to generating only universally-quantified invariants over state variables, and hence, were unable to solve any problem. While both IC3PO and I4 use incremental induction over a finite protocol instance, the number of SMT queries made by I4 grows drastically, indicating the benefits offered by symmetry and range boosting employed in IC3PO. fol-ic3 also fails on all problems, showing limited scalability of its enumeration-based separators technique operating directly in the unbounded domain. For SWISS, we weren’t able to replicate results for EPR problems as reported in [48] using our experimental setup. Nevertheless, SWISS showed limited capabilities for solving ORIGINAL problems.

Comparison against human-written invariants: As evident from A_1 to A_{11} in Section VI, IC3PO generated concise, human-readable inductive invariants. In fact, every invariant of Paxos written manually by Lamport et al. (as detailed in [28], [39]) had a corresponding equivalent invariant in the inductive proof automatically generated with IC3PO. In contrast, deriving such invariants manually, even in the presence of a hierarchical structure, is a tedious and error-prone process that demands deep domain expertise [12], [16], [28], [29].

Overall, the evaluation confirms our main hypothesis, that it is possible to utilize the regularity and hierarchical structure in complex distributed protocols, like in Paxos, to scale automatic verification beyond the current state-of-the-art.

IX. RELATED WORK

Introduced by Lamport, TLA+ is a widely-adopted language for the specification and verification of distributed protocols [65], [66]. The TLA+ toolbox [67] provides the TLC model checker, which is primarily used as a debugging tool for verifying small finite protocol instances [68], and not as a tool for inferring inductive invariants. The TLAPS proof assistant [7], [8] allows checking proofs manually written in TLA+, and has been used to verify several distributed protocols, including variants of Paxos [10], [15].

The derivation of inductive invariants for distributed protocols continues to be mostly carried out through refinement proofs using interactive theorem proving [13], [16], [17], [19], [69]–[72], which demands significant manual effort and profound domain expertise. The first attempts at automatically deriving quantified invariants were reported in [32], [33], using *invisible invariants*. The intuition underlying this method was the assumption that the system is “sufficiently symmetric,” and that its behavior can be captured by any m -subset of its processes as a universally-quantified invariant. However, universally-quantified invariants are not guaranteed to be inductive or to imply the safety property. Spatial regularity was further explored in [73]–[78] to reduce the verification of an n -process system to that of a *quotient* system at a small *cutoff* size.

Notwithstanding the undecidability result of Apt and Kozen [79], many efforts to automatically infer quantified inductive invariants have been reported with the pace increasing in recent years [48], [50]–[52], [80]–[82]. Verification of parameterized systems is further explored in [83]–[87]. However, unlike IC3PO, these methods generally do not scale to complex protocols like Lamport’s Paxos, since these methods rely heavily on unbounded reasoning and are limited to specifications in the EPR fragment of first-order logic.

Our technique builds on these works, with the capability to automatically infer the required quantified inductive invariant using the latest advancements in model checking, by extending our recent work [27] on symmetry boosting and finite convergence with range boosting and hierarchical strengthening.

X. CONCLUSIONS & FUTURE WORK

We proposed *range boosting*, a novel technique that extends the incremental induction algorithm to utilize the temporal regularity in distributed protocols through quantified reasoning over ordered ranges. We also presented *hierarchical strengthening*, a simple technique that utilizes the hierarchical structure of protocol specifications to enable automatic verification of complex distributed protocols with high scalability. Given the four-level hierarchy of the Paxos specification, we showed that these techniques, coupled with our recent work on symmetry boosting and finite convergence, provide, to our knowledge, the first demonstration of an automatically-inferred inductive invariant for the original Lamport’s Paxos algorithm.

While introducing *SimplePaxos* and *ImplicitPaxos* to get the four-level Paxos hierarchy was quite easy, these intermediate levels were still added manually. It is appealing to explore

counterexample-guided abstraction-refinement (CEGAR) techniques [88], [89] to automatically identify these intermediate levels whenever needed to overcome complexity. Specifically, investigating how to leverage clause learning feedback from incomplete runs to identify bottlenecks in proof inference and utilizing this information to automatically abstract away irrelevant details from the low-level protocol can help in making the complete procedure automatic end-to-end. We leave this investigation as future work.

Exploring inference with existential quantifiers in range boosting can also be an interesting future direction, though intuitively, existential quantification over temporal behaviors looks unnecessary for proving safety properties. Future work also includes automatically inferring inductive proofs for other distributed protocols, such as Byzantine Paxos [15], Raft [90], etc., and exploring the verification of consensus algorithms in blockchain applications.

DATA AVAILABILITY STATEMENT AND ACKNOWLEDGMENTS

The software and data sets generated and analyzed during the current study, including all experimental data, evaluation scripts, and IC3PO source code are available at <https://github.com/aman-goel/fmcd2021exp>.

We thank Leslie Lamport for the TLA+ video course [91], which shaped several ideas presented in this paper. We thank the developers of TLA+ [92], [93], Yices [63], Z3 [62], pySMT [94], and Ivy [13] for making their tools openly available. We also thank the reviewers for their valuable comments.

REFERENCES

- [1] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, p. 133–169, May 1998. [Online]. Available: <https://doi.org/10.1145/279227.279229>
- [2] —, “Paxos made simple,” pp. 51–58, December 2001. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>
- [3] M. Burrows, “The chubby lock service for loosely-coupled distributed systems,” in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 335–350.
- [4] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: An engineering perspective,” in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 398–407. [Online]. Available: <https://doi.org/10.1145/1281100.1281103>
- [5] M. Isard, “Autopilot: Automatic data center management,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, p. 60–67, Apr. 2007. [Online]. Available: <https://doi.org/10.1145/1243418.1243426>
- [6] R. De Prisco, B. Lampson, and N. Lynch, “Revisiting the paxos algorithm,” *Theoretical Computer Science*, vol. 243, no. 1-2, pp. 35–91, 2000.
- [7] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, “The tla+ proof system: Building a heterogeneous verification platform,” in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2010, pp. 44–44.
- [8] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, “Tla+ proofs,” in *FM 2012: Formal Methods*, D. Gianakopoulou and D. Méry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 147–154.
- [9] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

- [10] S. Chand, Y. A. Liu, and S. D. Stoller, “Formal verification of multi-paxos for distributed consensus,” in *International Symposium on Formal Methods*. Springer, 2016, pp. 119–136.
- [11] L. Lamport, *Specifying Systems*. Addison-Wesley Boston, 2002, vol. 388.
- [12] O. Padon, G. Losa, M. Sagiv, and S. Shoham, “Paxos made epr: decidable reasoning about distributed protocols,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 108:1–108:31, 2017.
- [13] 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.
- [14] R. Piskac, L. de Moura, and N. Bjørner, “Deciding effectively propositional logic using dpll and substitution sets,” *Journal of Automated Reasoning*, vol. 44, no. 4, pp. 401–424, 2010.
- [15] L. Lamport, “Byzantizing paxos by refinement,” in *International Symposium on Distributed Computing*. Springer, 2011, pp. 211–224.
- [16] 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.
- [17] 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>
- [18] S. Merz, “Formal specification and verification,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 103–129.
- [19] B. Kragl, S. Qadeer, and T. A. Henzinger, “Refinement for structured concurrent programs,” in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 275–298.
- [20] M. Abadi and L. Lamport, “The existence of refinement mappings,” *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.
- [21] L. Lamport, “The temporal logic of actions,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.
- [22] —, “Refinement in state-based formalisms,” *Digital Equipment Corporation*, 1996.
- [23] S. J. Garland and N. A. Lynch, “Using i/o automata for developing distributed systems,” *Foundations of component-based systems*, vol. 13, no. 285-312, pp. 5–2, 2000.
- [24] A. R. Bradley, “SAT-Based Model Checking without Unrolling,” in *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, ser. VMCAI’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 70–87. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1946284.1946291>
- [25] N. Een, A. Mishchenko, and R. Brayton, “Efficient Implementation of Property Directed Reachability,” in *Formal Methods in Computer Aided Design (FMCAD’11)*, Oct. 2011, pp. 125 – 134.
- [26] E. M. Clarke, E. A. Emerson, and J. Sifakis, “Model checking: algorithmic verification and debugging,” *Communications of the ACM*, vol. 52, no. 11, pp. 74–84, 2009.
- [27] A. Goel and K. Sakallah, “On symmetry and quantification: A new approach to verify distributed protocols,” in *NASA Formal Methods*, A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez, Eds. Cham: Springer International Publishing, 2021, pp. 131–150. [Online]. Available: https://doi.org/10.1007/978-3-030-76384-8_9
- [28] D. Doligez, L. Lamport, and S. Merz, “A TLA+ specification of the Paxos consensus algorithm and a TLAPS-checked proof of its correctness,” <https://github.com/tlaplus/tlapm/blob/master/examples/paxos/Paxos.tla>.
- [29] 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 *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 662–677.
- [30] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” www.SMT-LIB.org, 2016.
- [31] A. Goel and K. A. Sakallah, “On symmetry and quantification: A new approach to verify distributed protocols,” *CoRR*, vol. abs/2103.14831, 2021. [Online]. Available: <https://arxiv.org/abs/2103.14831>
- [32] 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.
- [33] 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.
- [34] L. Zuck and A. Pnueli, “Model checking and abstraction to the aid of parameterized systems (a survey),” *Computer Languages, Systems & Structures*, vol. 30, no. 3–4, pp. 139–169, 2004.
- [35] I. Balaban, Y. Fang, A. Pnueli, and L. D. Zuck, “Iiv: An invisible invariant verifier,” in *International Conference on Computer Aided Verification*. Springer, 2005, pp. 408–412.
- [36] K. S. Namjoshi, “Symmetry and completeness in the analysis of parameterized systems,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2007, pp. 299–313.
- [37] L. Lamport, “How to write a proof,” *The American mathematical monthly*, vol. 102, no. 7, pp. 600–608, 1995.
- [38] —, “A TLA+ specification of the Voting algorithm from Leslie Lamport’s lectures titled: The Paxos Algorithm - or How to Win a Turing Award.” <https://github.com/tlaplus/Examples/blob/master/specifications/PaxosHowToWinATuringAward/Voting.tla>, 2019.
- [39] —, “A TLA+ specification of the Paxos Consensus algorithm from Leslie Lamport’s lectures titled: The Paxos Algorithm - or How to Win a Turing Award.” <https://github.com/tlaplus/Examples/blob/master/specifications/PaxosHowToWinATuringAward/Paxos.tla>, 2019.
- [40] —, “The Paxos Algorithm - or How to Win a Turing Award.” <https://lamport.azurewebsites.net/tla/paxos-algorithm.html?back-link=more-stuff.html>, 2019.
- [41] —, “Generalized consensus and paxos,” Tech. Rep. MSR-TR-2005-33, March 2005. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/>
- [42] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, “Making fast consensus generally faster,” in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 156–167.
- [43] “A TLA+ specification of the MultiPaxos algorithm.” <https://github.com/tlaplus/Examples/tree/master/specifications/MultiPaxos>.
- [44] G. Losa, “Paxos consensus protocol in Ivy.” <https://github.com/nano-o/ivy-proofs/blob/master/paxos/paxos.ivy>.
- [45] L. Lamport and S. Merz, “A TLA+ specification of the Voting algorithm and a TLAPS-checked proof of its correctness,” <https://github.com/tlaplus/tlapm/blob/master/examples/ByzPaxos/VoteProof.tla>.
- [46] L. Lamport, “Proving the correctness of multiprocess programs,” *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977.
- [47] S. Owicki and D. Gries, “Verifying properties of parallel programs: An axiomatic approach,” *Communications of the ACM*, vol. 19, no. 5, pp. 279–285, 1976.
- [48] 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>
- [49] 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://github.com/wilcoxjay/myppyvy/tree/pldi20-artifact>
- [50] 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.
- [51] 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 *Proceedings of the 27th Symposium on Operating Systems Principles*. ACM, 2019.
- [52] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzy, and S. Shoham, “Property-directed inference of universal invariants or proving their absence,” *Journal of the ACM (JACM)*, vol. 64, no. 1, pp. 1–33, 2017. [Online]. Available: <https://bitbucket.org/tausigplan/updr-distrib/src/master/>

- [53] A. Biere, N. Froleyks, and M. Preiner, “Hardware model checking competition (HWMCC) 2020,” <http://fmv.jku.at/hwmcc20>.
- [54] A. Goel and K. Sakallah, “Empirical evaluation of ic3-based model checking techniques on verilog rtl designs,” in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 618–621.
- [55] 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.
- [56] H. Howard, D. Malkhi, and A. Spiegelman, “Flexible paxos: Quorum intersection revisited,” *CoRR*, vol. abs/1608.06696, 2016. [Online]. Available: <http://arxiv.org/abs/1608.06696>
- [57] A. Goel and K. A. Sakallah, “IC3PO: IC3 for Proving Protocol Properties,” <https://github.com/aman-goel/ic3po>.
- [58] “mypyvy on GitHub,” <https://github.com/wilcoxjay/mypyvy>.
- [59] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, “Towards automatic inference of inductive invariants,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 2019, pp. 30–36.
- [60] A. Goel and K. Sakallah, “AVR: Abstractly Verifying Reachability,” <http://www.github.com/aman-goel/avr>.
- [61] A. Karbyshev, N. Bjørner, 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>
- [62] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [63] B. Dutertre, “Yices 2.2,” in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 737–744.
- [64] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV ’11)*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, Jul. 2011, pp. 171–177. snowbird, Utah. [Online]. Available: <http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf>
- [65] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardouff, “How amazon web services uses formal methods,” *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015.
- [66] R. Beers, “Pre-RTL formal verification: an intel experience,” in *Proceedings of the 45th annual Design Automation Conference*, 2008, pp. 806–811.
- [67] “The TLA+ Toolbox,” <https://lamport.azurewebsites.net/tla/toolbox.html>.
- [68] Y. Yu, P. Manolios, and L. Lamport, “Model checking tla+ specifications,” in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 1999, pp. 54–66.
- [69] 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.
- [70] C. Drăgoi, T. A. Henzinger, and D. Zufferey, “Psync: a partially synchronous language for fault-tolerant distributed algorithms,” *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 400–415, 2016.
- [71] 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.
- [72] K. v. Gleissenthall, R. G. Kici, A. Bakst, D. Stefan, and R. Jhala, “Pre-tend synchrony: synchronous verification of asynchronous distributed programs,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.
- [73] C. N. Ip and D. L. Dill, “Better verification through symmetry,” in *Computer Hardware Description Languages and their Applications*. Elsevier, 1993, pp. 97–111.
- [74] 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>
- [75] E. M. Clarke, T. Filkorn, and S. Jha, “Exploiting symmetry in temporal logic model checking,” in *International Conference on Computer Aided Verification*. Springer, 1993, pp. 450–462.
- [76] E. A. Emerson and K. S. Namjoshi, “Reasoning about rings,” in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 85–94.
- [77] E. A. Emerson and A. P. Sistla, “Symmetry and model checking,” *Formal methods in system design*, vol. 9, no. 1-2, pp. 105–131, 1996.
- [78] A. P. Sistla, V. Gyuris, and E. A. Emerson, “Smc: a symmetry-based model checker for verification of safety and liveness properties,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 2, pp. 133–166, 2000.
- [79] 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.
- [80] A. Gurfinkel, S. Shoham, and Y. Vizel, “Quantifiers on demand,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2018, pp. 248–266.
- [81] 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.
- [82] 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>
- [83] 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.
- [84] 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.
- [85] 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.
- [86] P. Abdulla, F. Haziza, and L. Holik, “Parameterized verification through view abstraction,” *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 5, pp. 495–516, 2016.
- [87] M. Dooley and F. Somenzi, “Proving parameterized systems safe by generalizing clausal proofs of small instances,” in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 292–309.
- [88] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-Guided Abstraction Refinement,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. Emerson and A. Sistla, Eds. Springer Berlin / Heidelberg, 2000, vol. 1855, pp. 154–169, 10.1007/10722167_15. [Online]. Available: http://dx.doi.org/10.1007/10722167_15
- [89] —, “Counterexample-Guided Abstraction Refinement for Symbolic Model Checking,” *J. ACM*, vol. 50, pp. 752–794, September 2003. [Online]. Available: <http://doi.acm.org.proxy.lib.umich.edu/10.1145/876638.876643>
- [90] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 { USENIX } Annual Technical Conference ({ USENIX } { ATC } 14)*, 2014, pp. 305–319.
- [91] L. Lamport, “The TLA+ Video Course,” <https://lamport.azurewebsites.net/video/videos.html>.
- [92] M. A. Kupke, L. Lamport, and D. Ricketts, “The tla+ toolbox,” *Electronic Proceedings in Theoretical Computer Science*, vol. 310, p. 50–62, Dec 2019. [Online]. Available: <http://dx.doi.org/10.4204/EPTCS.310.6>
- [93] “TLA+ on GitHub,” <https://github.com/tlaplus>.
- [94] M. Gario and A. Micheli, “Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms,” in *SMT workshop*, vol. 2015, 2015.

Refinement-Based Verification of Device-to-Device Information Flow

Ning Dong¹, Roberto Guanciale², Mads Dam³
KTH Royal Institute of Technology

Abstract—I/O devices are the critical components that allow a computing system to communicate with the external environment. From the perspective of a device, interactions can be divided into two parts, with the processor (mainly memory operations by the driver) and through the communication medium with external devices. In this paper, we present an abstract model of I/O devices and their drivers to describe the expected results of their execution, where the communication between devices is made explicit and the device-to-device information flow is analyzed. In order to handle general I/O functionalities, both half-duplex (transmission and reception) and full-duplex (sending and receiving simultaneously) data transmissions are considered. We propose a refinement-based approach that concretizes a correct-by-construction abstract model into an actual hardware device and its driver. As an example, we formalize the Serial Peripheral Interface (SPI) with a driver. In the HOL4 interactive theorem prover, we verified the refinement between these models by establishing a weak bisimulation. We show how this result can be used to establish both functional correctness and information flow security for both single devices and when devices are connected in an end-to-end fashion.

Index Terms—Formal verification, Refinement, Serial interface, Device driver, Interactive theorem prover, Information flow

I. INTRODUCTION

I/O devices are indispensable components for interactions with the external environment (e.g., print documents, transmit data, and receive user’s commands). Their proper operation is critical for trustworthiness: Poorly written device drivers are the predominant reason for operating system crashes [1]–[3], and devices themselves can be vulnerable to side-channel attacks [4], [5].

Existing work [6]–[10] mostly focuses on the verification of functional properties of device drivers, by analyzing the interactions between the controlling software and the I/O device. In this paper, we present a verification approach that includes inter-device communication. This allows to establish end-to-end information flow properties, for example to guarantee the absence of side channels.

Our strategy is based on refinement. First we define a formal “concrete” model of a specific I/O device, which formalizes the device behavior that is observable by the controlling software and other external devices, and a model of its device driver. The combination of these two models provides a software/hardware subsystem that can interact with other software

components and external devices. We then define an abstract model of this subsystem, which is independent of the actual device and provides a general blueprint of the subsystem’s desired behavior and information flows. The goal is that this abstract model should provide a functionality that is correct and secure by construction, similar to ideal models used in cryptography. Our refinement establishes a weak bisimulation between the concrete and abstract systems.

There are three main benefits of this approach:

- Bisimulation allows to transfer both functional properties and information flow properties (e.g., progress-sensitive noninterference [11]) of the abstract model to the concrete one.
- The same abstract model can be refined by models for different I/O devices.
- The compositionality of bisimulation allows to preserve the verified properties when we compose the subsystem with other components: e.g., we can compose the subsystem with the other software or subsystems to show inter-host properties.

We choose the Serial Peripheral Interface (SPI) as the demonstrating example, and we provide the formal model of a specific device, the Texas Instruments McSPI device used in the AM335x family of processors [12], and its driver. The Serial Peripheral Interface is a synchronous protocol for serial communication that is mainly used in embedded devices. The protocol was first introduced in the late 1970s by Motorola and has become popular because of its simplicity and speed [13]. SPI devices support both half-duplex and full-duplex data transmissions, where the latter is used to improve performance by simultaneously sending and receiving data with external devices. Although full-duplex is effective in practice, this is to our knowledge the first example of verification in the literature of a full-duplex communication device, cf. [6]–[10].

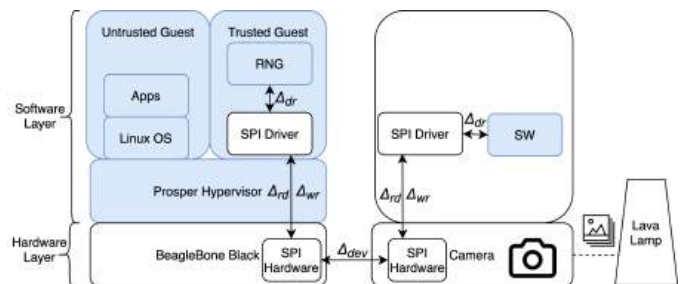


Fig. 1. The architecture of a random number generator

This work has been supported by the TrustFull project funded by the Swedish Foundation for Strategic Research. Ning Dong is supported by the China Scholarship Council for his doctoral studies.

We use the refinement to establish several interesting properties of the system: (1) The driver never leads the device to enter a configuration that is undocumented by the hardware specification; (2) Two interconnected SPI subsystems correctly and securely exchange data when they are activated by their controlling software; (3) Communications (driver-to-device and device-to-device) provide progress-sensitive noninterference at both concrete and abstract levels. The latter is established by a notion of contextual indistinguishability derived from the weak bisimulation.

To demonstrate our results, we developed the demonstrator of Figure 1. We use a BeagleBone Black running the verified Prosper hypervisor [14] together with an Arducam Shield Mini 2MP Plus camera to capture a physical source of randomness for, in our case, the Verificatum e-voting system [15]. The two devices communicate using SPI. The verification allowed us to slim down the driver by removing some unnecessary device register operations. The driver model is a direct manual translation of the driver binary. Formalization of this step is left as future work. In section X, we discuss our approach to automate this step by establishing a bisimulation between the driver model and its binary.

All proofs and models have been formalized in the HOL4 interactive theorem prover [16], which supports specification and proof in classical higher-order logic. For full definitions and proofs, we refer the reader to <https://github.com/kth-step/sw-spi-cam-model/releases/tag/fmcd>.

II. BACKGROUND

In this work, we model one of the devices of BeagleBone Black. This is a widely used development board with multiple peripherals, including SPI, I2C, UART, etc. The board has a TI AM335x processor [12] that uses the 32-bit ARMv7 instruction set architecture.

We focus on the SPI subsystem. Figure 2 shows the basic components involved in the SPI protocol: hardware connection, a controller, and a peripheral. In full-duplex mode, SPI permits to transmit and receive data simultaneously on separate data lines, SDI (Serial Data In) and SDO (Serial Data Out). The SPI controller uses the serial clock (SCK) line to maintain synchronization with the peripheral device. During each SPI clock cycle, from the controller’s perspective, one bit is transmitted from the controller to the peripheral on the SDO line, while the peripheral sends one bit to the controller on the SDI line. In half-duplex SPI transmissions, only one data line is used depending on the controller settings. In transmission-only mode, only the SDO data line is used, and vice versa for reception-only. The controller uses the chip select (CS) line to choose the desired communicating peripheral when multiple peripherals are connected. In this paper, we consider only the single peripheral case; extension to multiple peripherals is straightforward.

Bit transmission on the SDO/SDI lines is governed by the controller clock signal SCK, depending on configuration (clock polarity and edge settings). The SPI protocol can transmit messages of normally up to 16 bits, and delegates all error

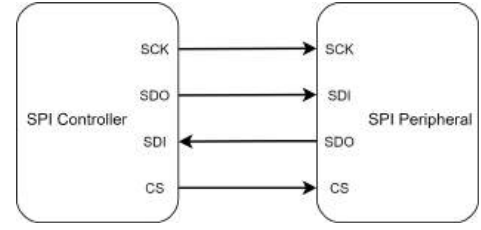


Fig. 2. Basic SPI connection: a controller and peripheral

detection, flow control, and application adaptation to higher-layer protocols. A driver can interact with the SPI hardware by register polling, interrupts, and Direct Memory Access (DMA). In this work, we rely on polling only. The following registers of the BeagleBone SPI controller are the ones used for polling:

- 1) The CP (controller/peripheral) bit of the MC (module control) register configures the SPI hardware as a controller (CP = 0) or a peripheral (CP = 1).
- 2) The channel configuration register (CCF) maintains the configuration of the communication channel. For instance, the TRM (transmit/receive modes) 2-bits of the CCF register controls the half and full-duplex modes: the values 0, 1, and 2 represent full-duplex, receive-only, and transmit-only respectively. The WL (word length) 5-bits configures the word length of the transmitted and received data. In our case, the driver fixes the WL bits to 7, which means the SPI word is 8-bits long, as all models transmit and receive bitwise data.
- 3) The TX0 (transmit buffer) register contains the data to transmit. The RX0 (receive buffer) register contains the received message bits.
- 4) The CST (channel 0 status) register is a read-only register and provides information about the status of TX0 and RX0 registers. The TXS (transmitter register status) bit of the CST register indicates if the TX0 register is empty: its value is 1 when the TX0 register is empty and can be written with the next word to transmit, and is 0 when the TX0 register is full and should not be overwritten. Analogously, the RXS (receiver register status) bit of the same register indicates the status of the RX0 register: its value is 1 when the RX0 register is full when data in the RX0 register is ready to be fetched and 0 when RX0 is empty.

III. ARCHITECTURAL MODEL

We model devices and drivers as labelled transition systems (LTS) in the style of CCS [17], modelling the interaction between software and driver, driver and device, as well as between devices (through signals “on the wire”) by the simultaneous occurrence of an action α and its dual $\bar{\alpha}$, where $\alpha, \bar{\alpha} \in \Delta_{wr} \cup \Delta_{rd} \cup \Delta_{dev} \cup \Delta_{dr}$. The top components of Figure 3 summarize the interfaces among models. Here, Δ_{wr} is the set of write operations by the CPU, which is represented by the action $wt\ a\ v$ for writing a byte v to the register with the memory-mapped address a , and the dual action $\bar{wt}\ a\ v$

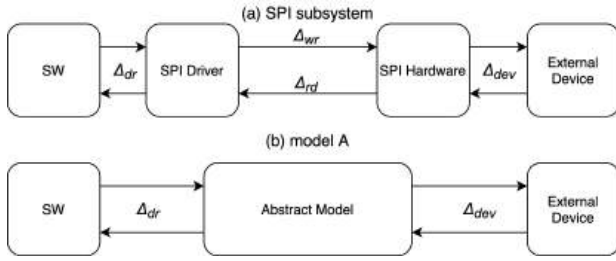


Fig. 3. The model architecture of SPI subsystem and abstract model

that is the corresponding action of the device. Similarly, Δ_{rd} is the set of read operations by the CPU which is represented by the action $rd\ a\ v$ for reading v from the register mapped at address a , and the dual action $\overline{rd\ a\ v}$. Representing this interaction as a CCS-style synchronous rendez-vous allows to reflect the potential side effects of register accesses on the SPI hardware. In the terminology of π -calculus [18], we use the “early” semantics. For instance, the reading of a memory-mapped register by the CPU non-deterministically spawns one transition for every possible resulting value.

The device model uses four additional types of action to model device-to-device interactions on the wire. The convention needs to take controller/peripheral asymmetry into account. For transmission-only mode the controller uses $\overline{tx\ v}$ to send a byte v over the wire, and in reception-only mode $tx\ v$ to receive a byte from the wire. For synchronous transfer of the (controller) byte v and (peripheral) byte v' , the controller uses $xfer\ v\ v'$. The peripheral uses the dual actions, i.e., $\overline{tx\ v}$ ($\overline{tx\ v}$) for reception (transmission) and always $xfer\ v\ v'$ for synchronous transfer. Let $\Delta_{dev} = \{tx\ v, \overline{tx\ v}, xfer\ v\ v', \overline{xfer\ v\ v'} \mid bytes\ v, v'\}$. Finally, the driver uses four additional actions to model invocations of the driver API by application SW and one additional action for returning control and result to SW (collected by Δ_{dr}).

The SPI subsystem consists of the SPI hardware running in parallel with its device driver with internal communication channels (e.g., $rd\ a\ v$), made inaccessible to the external world. In CCS parlance this is $(d|s) \setminus (\Delta_{wt} \cup \Delta_{rd})$, where d and s are states of the driver and hardware, respectively.

IV. SPI HARDWARE MODEL

The state of the SPI hardware is represented by a tuple $s = (regs, sreg, c)$. Here, $regs$ is a function mapping addresses of memory-mapped registers to words, and $sreg$ represents the internal hardware-controlled shift register for data transmission and reception. The component c captures the control state of the device and is used to track the progress of its four functionalities: initialization, transmission, reception, and full-duplex synchronous transfer.

With the exception of register RX0, register reads are side-effect free and simply communicate the current value of the register: i.e., for every state s , $s \xrightarrow{rd\ a\ s.reg(a)} s$. Transitions that model register writes (i.e., $s \xrightarrow{\overline{wt\ a\ v}} s'$) have side effects and are modeled by early instantiating all possible received values. Since many register updates are not atomic and require

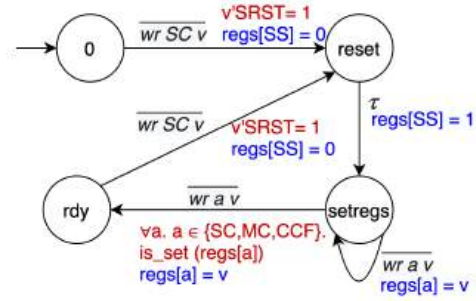


Fig. 4. SPI hardware initialization automaton

some time to take effect (e.g., writing into the transmission register does not automatically transfer the byte on the wire), transitions $s \xrightarrow{\overline{wt\ a\ v}} s'$ are usually followed by a silent transition $s' \xrightarrow{\tau} s''$, which is the system internal transition that applies the visible side effects.

A special error state \perp is entered under the following conditions:

- 1) The hardware receives read or write requests that violate the SPI specification [12] (e.g., the RX0 register is read when its value is indeterminate);
- 2) The hardware attempts an operation that is not allowed by the specification (e.g., to update the shift register before the initialization is completed);
- 3) An operation is not supported by the formal model, for instance, accessing control registers beyond the single channel modelled here.

The behavior of transitions that have side effects can be represented by an automaton, which is split into four sub-automata for the four device functionalities.

1) *Initialization*: Figure 4 shows the hardware initialization automaton, where the black, red, and blue annotations describe the label, enabling conditions and side effects of transitions respectively. Note that we have omitted all transitions that lead to \perp in Figure 4, which applies to the following figures as well. The initialization is activated when the value 1 is written to the SRST (software reset) bit of the SC (system configuration) register. The τ transition exiting state *reset* models the hardware completion of the reset operation and sets the SS (system status) register to 1. This register can be used by a driver to detect when the reset process is finished. In state *setregs*, the device awaits the set up of the hardware configuration, which is achieved by writing the SC, MC, and CCF registers. This step is necessary before starting data transmissions because the SPI hardware needs basic parameters, like the CP bit of the MC register and the WL bits of the CCF register. If one of these register updates sets a value that does not conform with the specification (e.g., the value of WL bits should no less than 3), then the model enters the state \perp . Once all required registers have been written, the model enters the ready state *rdy*. Now the SPI can be utilized for data transmissions or be reinitialized.

2) *Synchronous transfer*: Figure 5 depicts the synchronous transfer sub-automaton. From the ready state, the synchronous

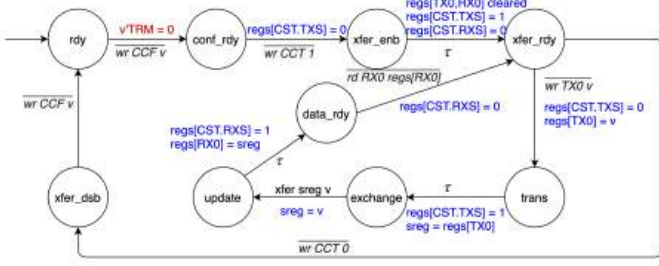


Fig. 5. SPI hardware synchronous transfer automaton

transfer is activated when the TRM bits of the CCF register are set to 0. Then, updating CCT with 1 activates the state *xfer_enb* and clears the TXS bit. The following silent transition makes the side effect of enabling the channel visible: the registers TX0 and RX0 are cleared, and the TXS and RXS bits are set to 1 and 0 respectively. From *xfer_rdy*, once the message *v* to transmit is written to TX0, the TXS bit is cleared. The following silent transition transfers the data from the TX0 register to the shift register and the TXS bit is set internally. The device will now synchronize with an external SPI device, simultaneously transmitting the shift register and receiving one byte *v*, which is copied into the shift register. The following silent transition makes the communication visible to the driver, by copying the shift register to RX0 and setting the RXS bit. Finally, from the state *data_rdy*, the received data can be fetched by reading the RX0 register. This also resets the RXS bit. The transmission process is repeated until the channel is disabled by writing 0 to the CCT register in the state *xfer_rdy* and then resetting the CCF register to its original value.

As mentioned before, from the diagram in Figure 5, we have omitted all transitions that lead to \perp . This happens, for instance, if TX0 is written before the TXS bit is set or when the model is in the state *data_rdy*, or if RX0 is read while RXS is not set.

3) *Transmission and reception*: The structure of the half-duplex automata for transmission and reception is similar to the synchronous transfer automaton. However, there are some notable differences:

- 1) The transmission and reception automatons are activated by setting the TRM bits to 1, resp. 2 for receive-only, resp. transmit-only mode.
- 2) In transmission mode, the transmission automaton will not receive data from the external device, which means the RXS bit remains unchanged. The EOT (end-of-transfer status) bit of the CST register is used to indicate the end of transmission. The EOT bit is cleared when *sreg* is updated with the output data, and it is set when the data is transmitted to the external device. In this way, a driver can check the EOT bit rather than the RXS bit when applying the transmit-only mode.
- 3) After the channel is enabled for the receive-only mode in the reception automaton, the hardware first receives the external data and then uploads it to the RX0 register. Therefore, unlike the synchronous transfer automaton,

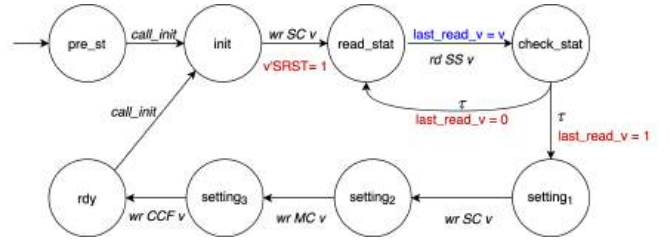


Fig. 6. Driver initialization automaton

the TX0 register should not be used. A correct driver should wait for the hardware until the received data is ready through reading the RXS bit. The TXS and EOT bits are not applied in the reception automaton.

V. SPI DRIVER MODEL

The driver model is a direct manual translation of the real SPI driver binary and interacts with the hardware model using operations on the device registers. The model exposes all accesses to memory-mapped registers that are performed by the actual driver.

The driver state is a tuple $d = (b_1, b_2, idx, last_read_v, c)$. Here, b_1 is the transmit, and b_2 the receive buffer. The variable *idx* points to the next byte in b_1 to be transmitted. The byte *last_read_v* is the last returned value from the hardware, used for the driver's internal operations. The last component *c* is the driver's control state. We define sub-automata corresponding to each of the four device functionalities.

1) *Driver initialization*: Figure 6 shows the driver initialization automaton. The automaton is invoked by an external call to the driver initialization function, represented here by the action *call_init*. In state *init*, the automaton writes the SC register to reset the hardware. Then the automaton reads the SS register and updates the *d.last_read_v* with the returned value. In the state *check_stat*, the automaton checks the fetched value to determine if the hardware finished the reset process. If the value is 1, the automaton enters the state *setting1*, otherwise it returns to the previous state and repeats this loop. Finally, the automaton enters the ready state by setting several registers in order (SC, MC, and CCF), indicating that the driver model is prepared to process function calls for data transmissions and reinitialization.

2) *Driver synchronous transfer*: The driver synchronous transfer automaton is shown in Figure 7. With the driver in state *rdy*, the automaton is invoked by action *call_xfer* with a buffer b_1 copied to the driver's internal output buffer ($d.b_1$). Before starting data transmission, the automaton first prepares the necessary settings for the hardware by writing the CCF and CCT registers. Notice that CCF is read prior to writing in order to maintain other channel configurations (e.g., transmission speed). At this point, the automaton loops reading the CST register and checking the TXS bit, as long as the value of TXS is 0. Once the value 1 is read, the automaton enters the state *write_data*. The following step writes the TX0 register with one byte data that is sent to the external device, leading to the state *read_rxs*. Hereafter, the automaton repeatedly reads the

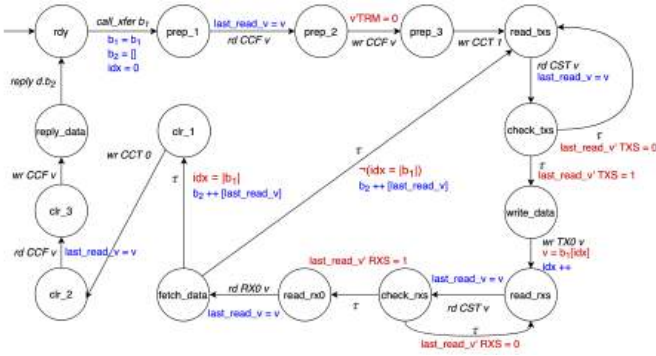


Fig. 7. Driver synchronous transfer automaton

CST register as before but checks the RXS bit rather than the TXS bit, which indicates the hardware transmission is finished and the received data is available in the RX0 register. If the RXS bit is 1, then the automaton in the state *read_rxs* issues a read request to the RX0 register. Next, the automaton can fetch the received data and check if all bytes in the output buffer are transmitted. If there are more bytes to transmit, the automaton returns to the state *read_txs* and repeats the process. Otherwise, the automaton clears the CCT register and the CCF register to their initial values. Finally, the driver replies the received data (*d.b2*) to the program that invoked the driver by using the label *reply* and returns to the ready state.

The driver's transmission and reception automata are similar and left out.

VI. ABSTRACT SPI SUBSYSTEM SPECIFICATION

In this section, we present an abstract specification of the combined device and driver subsystem. The model has the same interface as the concrete SPI subsystem (see Figure 3 (b)) and describes the visible effects of the four functionalities (i.e., initialization, full-duplex synchronous transfer, transmission, and reception) while ignoring all internal states of the SPI hardware and the memory-mapped device registers. The state of the abstract model is a pair, $a = (t, c)$. The component $t = (b_1, b_2, idx, v)$ is the data state, which contains the output and input buffers b_1 and b_2 , the index of the next byte to be transmitted idx , and the received byte v . The component c is the control state of the abstract model.

The abstract initialization and synchronous transfer automata in Figure 8 are largely self-explanatory. The control structure is the obvious one with bytes in the transmit buffer $a.t.b_1$ being sent one by one and received bytes getting stored in $a.t.b_2$. Note also that once in the ready state reinitialization must remain enabled.

VII. REFINEMENT

The refinement is established by exhibiting a weak bisimulation [19]. This approach is useful to allow multiple levels of concretizations and abstractions through transitivity and compositionality (under parallel) of the corresponding equivalence.

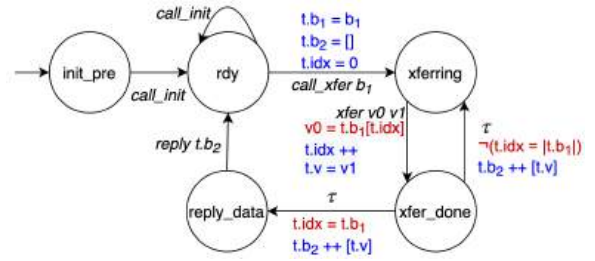


Fig. 8. Abstract initialization and synchronous transfer automata

Below we use $p \xrightarrow{\tau^*(a)}_1 p'$ to indicate an arbitrary number of τ transitions, optionally followed by an a transition.

Definition VII.1 (Weak bisimulation). *Given two transition systems (S, \rightarrow_1) and (T, \rightarrow_2) , a binary relation $R \subset S \times T$ is a weak simulation if for every $(p, q) \in R$:*

- If $p \xrightarrow{a}_1 p'$ then $q \xrightarrow{\tau^*a}_2 q'$ for some q' s.t. $(p', q') \in R$.
- If $p \xrightarrow{\tau}_1 p'$ then $q \xrightarrow{\tau^*}_2 q'$ for some q' s.t. $(p', q') \in R$.

The relation R is a weak bisimulation if both R and R^{-1} are weak simulations. In the following, we write $S \sim_R T$ when R is a weak bisimulation, and $S \sim T$ if there exists R such that $S \sim_R T$.

Our weak bisimulation definition is slightly different from the standard definition that allows arbitrary τ transitions after the observation a (e.g., $q \xrightarrow{\tau^*a\tau^*}_2 q'$). It is easy to show that our definition entails the standard one.

Weak bisimulation is transitive and compositional:

Theorem VII.1. *If $S \sim_{R_1} T$ and $T \sim_{R_2} U$ then $S \sim_{R_1 \circ R_2} U$, where $p (R_1 \circ R_2) q \Leftrightarrow \exists r. p R_1 r \wedge r R_2 q$*

Theorem VII.2. *If $S \sim_R T$ then $S|U \sim_{R'} T|U$, where $p|r R' q|r \Leftrightarrow p R q$.*

A. An intermediate model

In order to show a weak bisimulation between the SPI subsystem and the abstract model A , we introduce an intermediate model B . The intermediate model, still abstracting from memory operations, has the states $b = (t, sreg, c)$ with the control state c as in the abstract model, and with t of the shape (b_1, b_2, idx) , i.e., as t , but not including the received byte v , which is instead represented in an explicit shift register *sreg*, as in the SPI hardware model. Figure 9 shows on the top the full-duplex synchronous transfer automaton of the B model, and on the bottom demonstrates in part the weakly bisimilar control states in blue of the SPI subsystem under a relation R_1 . For example, the control state *update* of the B model is weak bisimilar with two states of the SPI subsystem, (*check_rxs|update*) and (*read_rxs|update*) (driver and hardware's control states respectively). The control state (*check_rxs|update*) is reached from the (*read_rxs|update*) by reading the CST register, which is omitted in the B model. The τ transitions between two control states that are weakly bisimilar with the same abstract state are also ignored. In our example, if the RXS bit is 0 when the SPI hardware

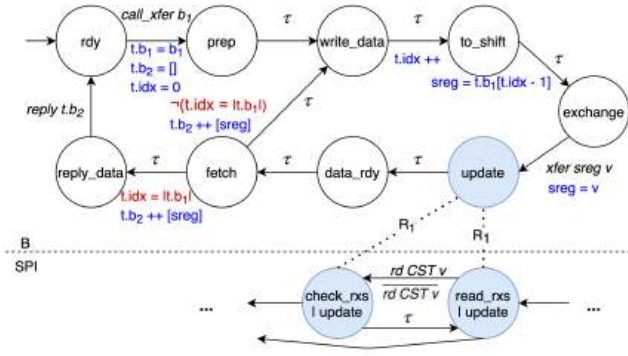


Fig. 9. Model B synchronous transfer automaton and part weak bisimulation

is in the control state *update*, the driver will return to the previous state by internally checking the fetched value. This stepwise approach makes it much easier to build the desired bisimulation relation.

B. Weak bisimilarity of the abstract and SPI models

The following two lemmas show the weak bisimilarity of B and SPI models, A and B models respectively.

1) Weak bisimilarity of the intermediate and SPI models:

We define a relation R_1 for the B and SPI models, which matches their control states as indicated in Figure 9 and requires the equivalence of data buffers and records, shift registers, etc. In addition, the relation R_1 requires that if b is not in the error state then neither are the driver and hardware models, and vice versa.

Lemma VII.1. $(d|s) \setminus \{\Delta_{wr} \cup \Delta_{rd}\} \sim_{R_1} b$

Proof: The two models have the same four functionalities, and the state transitions of the two models can be divided into the corresponding four sub-automata. We comment on the full-duplex synchronous transfer automaton, since the transmission and reception are similar and the initialization is straightforward. There are four kinds of transitions in this automaton for both models: *call_xfer buf*, *xfer v v'*, τ and *reply buf'*.

- *call_xfer buf*: The main point is to guarantee that the driver model performs the buffer copy and clears the internal received buffer as prescribed by the intermediate model.
- *xfer v v'*: When the two models are in the control state *exchange*, *xfer v v'* is used to exchange single bytes v , v' with the external device. In order to guarantee weak bisimilarity, the driver must guarantee to write the value v to the TX0 register.
- τ : The major concern is to show the equivalence of data buffers, index and shift registers of the two models. There are three critical requirements that the driver should adhere to, otherwise the hardware model enters the error state and the weak bisimulation condition is violated.

- 1) The driver should delay writing the TX0 register until the TXS bit is 1, because the value 0 of TXS bit means the TX0 register is not ready to be written.

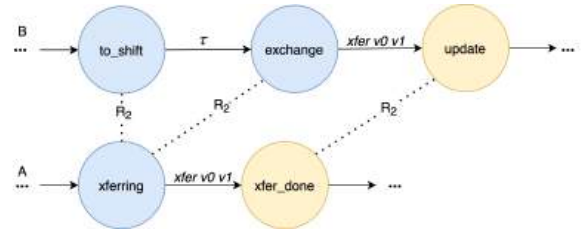


Fig. 10. Weak bisimulation example of the A and B models

This also means the driver should not immediately write the next byte after legally writing the TX0 register.

- 2) The driver should wait for the RXS bit to become 1 before reading the RX0 register. Otherwise, the RX0 register may not contain the received data.
- 3) To avoid error situations, the driver should read the CCF register before writing in order to keep the necessary channel configurations unchanged, such as WL bits.

- *reply buf'*: When replying, the driver must ensure that the data in *buf'* is identical to the bytes read from the device.

2) *Weak bisimilarity of the abstract and intermediate models:* The relation R_2 is defined in a similar way for the abstract and intermediate models. Figure 10 shows the relation for a part of the synchronous transfer automata of the two models, where weakly bisimilar control states are coloured identically. This relation basically matches control states under the requirement that buffers and records remain unchanged. The bisimulation condition forces input and output data of the two models to be the same.

Lemma VII.2. $b \sim_{R_2} a$

Proof: Same methodology as for Lemma VII.1. ■

From Theorem VII.1, Lemma VII.1 and Lemma VII.2, it directly follows that there is a relation R_3 for the abstract and SPI models:

Theorem VII.3. $(d|s) \setminus \{\Delta_{wr} \cup \Delta_{rd}\} \sim_{R_3} a$ where $R_3 = R_1 \circ R_2$

VIII. SYSTEM PROPERTIES

In order to demonstrate the functional properties of the system, we verify three theorems for the abstract model. These theorems transfer easily to the concrete models using the bisimulation results of Section VII. Additionally, we show that the abstract (SPI subsystem) model never enters the error state.

The functional correctness of full-duplex synchronous transfer should show that buffers are exchanged correctly between two devices. To show this property, we define the process $G(a_0, a_1) = (a_0 | (a_1 \{xfer v v' / xfer v' v\})) \setminus \Delta_{dev}$, which composes the abstract model of an SPI subsystem with a “dual” paired device: if one controller device uses *xfer v v'* to transmit and receive data, the peripheral device uses the dual

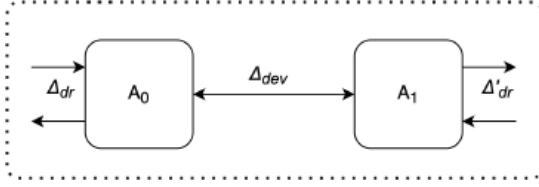


Fig. 11. Composition of two devices

label to synchronize. Figure 11 depicts the composition of two devices.

Theorem VIII.1 shows the functional correctness of the full-duplex synchronous transfer. Notice that buffers must have the same length, otherwise the larger buffer cannot be transmitted in its entirety.

Theorem VIII.1. *If $0 < |b_0| = |b_1|$, $(t_0, rdy) \xrightarrow{\text{call_xfer } b_0} a_0$, and $(t_1, rdy) \xrightarrow{\text{call_xfer } b_1} a_1$, then $\exists n a'_0 a'_1 a''_0 a''_1. G(a_0, a_1) (\xrightarrow{\tau})^n G(a'_0, a'_1) \wedge a'_0 \xrightarrow{\text{reply } b_1} a''_0 \wedge a'_1 \xrightarrow{\text{reply } b_0} a''_1$*

Proof: We show that the first byte can be exchanged correctly and then complete the proof by induction. ■

An analogous theorem shows the correctness of transmission/reception. In this case, l , the number of bytes to be received, should be greater than or equal to the length of the data buffer b_0 , otherwise extra data of the buffer will be lost.

Theorem VIII.2. *If $0 < |b_0| \leq l$, $(t_0, rdy) \xrightarrow{\text{call_tx } b_0} a_0$, and $(t_1, rdy) \xrightarrow{\text{call_rx } l} a_1$, then $\exists n a'_0 a'_1 a''_1. G(a_0, a_1) (\xrightarrow{\tau})^n G(a'_0, a'_1) \wedge a'_1 \xrightarrow{\text{reply } b_0} a''_1$*

Finally, we show that the abstract model can never enter an erroneous state. The bisimulation transfers this property to the SPI hardware and the driver:

Theorem VIII.3. *If $c \neq \perp$ and $(t, c) \rightarrow (t', c')$, then $c' \neq \perp$*

IX. INFORMATION FLOW SECURITY

Formal device and driver verification projects have generally focused on functional correctness [6]–[10]. However, the device driver can possibly leak sensitive information and therefore, for critical applications, information flow analysis is needed. One of the main benefits of establishing weak bisimulation instead of a simulation is that the former guarantees that two systems have the same information flows (up to channels that are not modeled here, like timing). We show that weak bisimilarity is sufficient to capture progress-sensitive noninterference (PSNI), in the sense of Hedin and Sabelfeld [11]. Let E be the set of transition labels of the system under consideration. In our case, we may consider a system as in Figure 11 with $E = \Delta_{dr} \cup \Delta'_{dr}$, where Δ_{dr} and Δ'_{dr} are distinct driver interfaces that are both high, since the interfaces are used to communicate sensitive data. We assume a context C that is allowed to interact with the system using any label in E . This context is additionally equipped with a public, distinguished interface of labels P that the context can use to receive and produce publicly observable stimuli. Then, any observations using labels in P that can cause the abstract and

concrete models to be distinguished must be due to C being able to bring the two systems to states that C can distinguish. Of course, if the two systems are weakly bisimilar, this is in fact not possible, motivating the following definition.

Definition IX.1 (Contextual indistinguishability). *Two states s_1 and s_2 are contextually indistinguishable, $s_1 \approx s_2$, if for every context C , $(s_1 \mid C) \setminus E \sim (s_2 \mid C) \setminus E$.*

We use the term contextual indistinguishability instead of contextual equivalence, as the former considers only contexts of very specific shapes. It is not the case that contextual indistinguishability implies contextual equivalence in general, as the latter is a congruence, specifically under CCS sum, which the former is not. However, weak bisimulation is a congruence under parallel composition and restriction. Thus, if s_1 and s_2 are weakly bisimilar, then they are also contextually indistinguishable. The converse implication, of course, does not hold. It also follows directly that \approx is transitive.

The concept of contextual indistinguishability is related to Focardi et al.'s nondeducibility of composition (NDC) [20], which in our setting would be the condition $(s \mid C) \setminus H \sim s \setminus H$ on s , where H represents the high labels and C is restricted to interact using only H . However, it is not clear how to adapt the NDC condition to our refinement-based setting, and also, in contrast to contextual indistinguishability, the NDC condition is not able to accommodate systems such as ours that obtain low observability only through the use of the context.

For the definition of PSNI, a run π is any sequence of transitions starting from an initial state. Such a run is *complete* if it cannot be extended, i.e., it is either unbounded or ends in a final state. For a run π , we let $O(\pi)$ be the list of public labels in π . We can now define PSNI adapted to our setting of reactive systems as follows:

Definition IX.2 (PSNI). *Two states s_1 and s_2 are PSNI, if for every complete run π_1 starting from s_1 , there exists a complete run π_2 starting from s_2 such that $O(\pi_1) = O(\pi_2)$, and vice versa.*

The definition can be seen to be equivalent to the one in [11], or in terms of termination only, with the notion of weak termination-sensitive noninterference of [21]¹.

Contextual indistinguishability is a sufficient condition for PSNI, because it guarantees the existence of traces for two transition systems with the same observable labels.

Theorem IX.1. *If $s \approx t$, then s and t are PSNI.*

If s and t are not PSNI, then we find a complete run π_1 from s such that all complete runs π_2 starting from t have different low observations from π_1 . Clearly, this allows a context c using labels in $L \cup H$ to steer s , possibly nondeterministically, into a state s' that cannot be matched by t , in the sense of weak bisimilarity. Here L represents low labels.

¹In fact, at our low level of modelling, with weak bisimulation, the adversary does not have any model-external means (such as exhausting the memory) at its disposal to prevent progress. Hence our account is also strongly termination-sensitive in the terminology of [21].

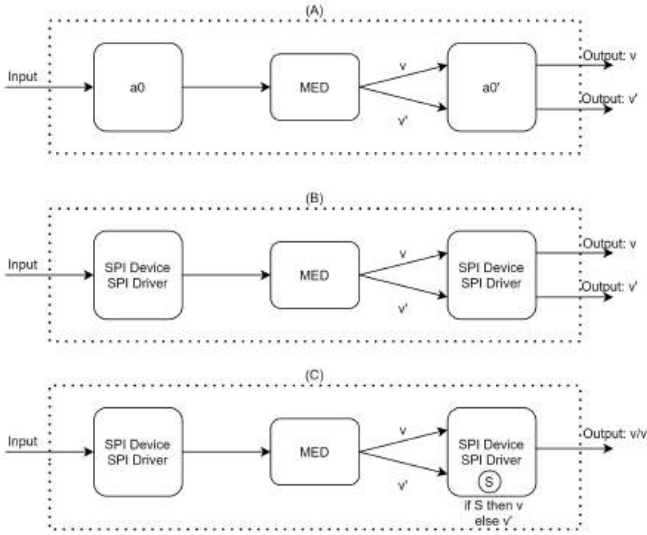


Fig. 12. Information flow security example

We can also show that PSNI transfers under \approx :

Theorem IX.2. *Suppose $s \approx s'$ and s' and t are PSNI. Then s and t are PSNI.*

We cannot in general replace weak bisimulation by the corresponding notion of simulation in the definition of contextual indistinguishability. A device driver may leak a sensitive boolean s by either terminating execution conditionally on s or by entering a diverging loop (e.g., `while (s) {}`), but still be (weakly) simulated by the abstract model. In this case, an external attacker may discover the value of the secret boolean by observing the impossibility of transmission of a buffer.

Also, establishing bisimulation allows to compose the system with non-deterministic components safely. For instance, we can introduce a faulty communication medium (*MED*) between two devices that can indeterminately deliver wrong values. Figure 12 (A) represents the abstract model where two abstract devices (our *A* model) are connected through the given medium. As a result of the medium, the final output of the abstract model is non-deterministically v or v' . The compositionality of the weak bisimulation guarantees that in the system where the two concrete SPI subsystems are interconnected by the same medium (see Figure 12 (B)), the final output is also non-deterministically v or v' : the system has the same information flows. On the other hand, the system (Figure 12 (C)), where the receiving device driver decides the value according to a secret value, leaks a secret value via the final output. This model cannot be validated using contextual indistinguishability, but it can be when weak bisimulation is replaced by a corresponding notion of weak simulation.

X. APPLICATION: SECURING A RANDOM NUMBER GENERATOR USING SPI

As a demonstrating application, we developed a secure random number generator (RNG) that relies on the SPI hardware for sourcing entropy. The architecture of the system is depicted

in Figure 1. The blue components are the software components not including the SPI driver(s). The SPI driver interacts with the SPI hardware through operations on memory-mapped registers (Δ_{rd} and Δ_{wr}). We use a BeagleBone Black to connect with an Arducam Shield Mini 2MP Plus camera through SPI. The RNG captures images of the floating material in a lava lamp. This has been shown to be a good source of physical randomness [22], [23].

In order to prevent vulnerabilities of other software affecting the RNG, we develop a bare-metal application that integrates the SPI driver and that is executed on top of the Prosper hypervisor [14]. This is a hypervisor for ARMv7-A processors that provides provable separation between different guests and can be configured to grant accesses to the SPI registers to a dedicated partition only, running our driver. This allows an untrusted partitioned Linux guest (such as in our case, the Verificatum e-voting application [15]) to harden the built-in Linux RNG with physical randomness through a hypercall interface provided by the hypervisor with strong end-to-end security guarantees. In this scenario, the SPI subsystem plays an important role. Additionally to failing to function, a faulty device driver may reduce the entropy of the system by simply returning predictable buffers or it could communicate, directly or indirectly, internal data to the external device. Formal verification of the driver model allows us to rule out these problems. Moreover, it helped to identify redundant operations of the driver. For example, the initial version (extracted from the u-boot library) sets up the WL bits of the CCF register whenever the transmission functions are used, however it is enough to set them once in the initialization function.

In order to guarantee the absence of vulnerabilities at the code level, the refinement should be pushed down to the binary code of the device driver. We extract the driver model by manual inspection of the driver binary. This step has yet to be formalized. We don't view this as a major weakness, however, given that the memory-mapped registers use uncached memory only. We have experimented with the usage of the binary analysis tool HolBA [24] for verifying weak bisimilarity of the driver's assembly code and the driver model. The weak bisimulation relates fragments of binary instructions (i.e., program counter addresses) to a state of the driver's automaton. Each fragment has a single entry point, and either (1) consists of one single instruction accessing a device register or (2) does not access the device. In the former case, the instruction directly corresponds to a transition of the driver model. In the latter case, the fragment corresponds to a finite sequence of silent transitions. We then translate the relation into pre/post conditions for the fragments, which can be analyzed via HolBA weakest precondition tool and a Satisfiability Modulo Theories (SMT) solver.

XI. RELATED WORK

Some previous work has applied the bisimulation methodology for verification in a theorem prover context [25], [26]. For example, Röckl et al. [25] verified the correctness of several communication protocols by proving weak bisimilarity. We

prove the equivalence of the abstract and SPI models using the same approach.

Several projects of formal verification of low-level software have focused on the operating system (OS), like seL4 [27] and CertiKOS [28]. However, the functional correctness of device drivers usually is not considered. For example, the seL4 microkernel [27] only guarantees the isolation of device drivers located in the user space, where the correctness of drivers is ignored. CertiKOS [28] initially did not verify the drivers as well. Based on CertiKOS, Chen et al. [10] developed a verified interruptible operating system with device drivers. They proposed a general device model with several instantiations and a realistic formal model of device interrupts. Although their device model has similarities with the one presented here, there are notable differences:

- 1) Their device model only contains events that can be observed by the CPU and ignores events that the external environments can observe. Our models consider device-to-device operations and properties (e.g., data transmissions);
- 2) Their device model covers only half-duplex communication (e.g., sending and receiving data over the UART port), while we also model full-duplex data transmission in both the abstract and concrete models;
- 3) In their case, device drivers are implemented inside the OS kernel and each device driver is treated as running independently on its own logical CPU. This requires a different isolation property of the OS kernel to guarantee the separation between different devices and the kernel, which is not provided by most OS kernels. Here, we describe the device driver as a normal process that can be embedded either inside or outside of the OS kernel.

Other previous work on verifying the functional correctness of device drivers studied various I/O devices, like UART [7], hard disk [8], and USB OHCI [6]. In their work, there is no abstract I/O device model to represent the general behaviours of different I/O devices, and it is too restrictive to extend their work on other hardware devices. Duan et al. [9] proposed an abstract device model that is plugged into the formal model of ARMv4 instruction set architecture and later extended it to support interrupts with respect to the ARMv7 architecture [29]. However, the device state is merged into the machine state in their model, which requires to carefully handle the interleavings between the execution of the device and processor. Because of the complexity, it is difficult to apply their model to verify I/O devices.

XII. CONCLUSION AND FUTURE WORK

We modeled and verified an SPI subsystem that consists of the device hardware and its driver. The verification establishes a weak bisimulation between this model and an abstract specification, which is used to transfer functional and information flow properties of the abstract model to the concrete one.

Our methodology can be reused to verify other SPI subsystems by establishing a refinement with the abstract model

presented in this paper. There are some valuable lessons we have learned from this project:

- 1) Reading the hardware technical reference manual is not sufficient to understand the usage of real hardware. For instance, the order of some operations is unclear. Since the concrete hardware design is usually unavailable, lots of experiments are needed to properly account for the actual functionalities of different I/O registers.
- 2) The abstract model must capture the intended information channels. For example, our initial driver model did not have the *reply* label. It prevents the indented leakage of the received bytes to the software invoking the driver and makes it impossible to establish a refinement with the actual implementation.
- 3) It is usually inconvenient to build an abstraction of the device without taking the driver into account. Indeed the very purpose of the driver is to provide a tractable and efficient abstraction of the generally highly configurable hardware. This turns out to be useful not only for programming but also for verification.

In order to complete the binary verification of the device driver, we plan to follow the strategy of Section X, which establishes a bisimulation between the SPI driver model and its binary code using contract-based verification of the HolBA platform [24]. Moreover, we are planning to address two limitations of the current models: The absence of DMA and interrupts. While these can be encoded via explicit synchronizations processor/device-memory or processor-device, we think that explicit treatment of these features can simplify models and proofs [30]. Currently, our models are shallowly embedded in HOL4. This allows us to partially automate our proof via the HOL4 standard tactics. For example, large parts of the proof search are fully automated using METIS_TAC. Our work can give insight for deeply embedding the models in HOL4. This can provide a general framework for modeling multiple types of I/O devices and increase automation by implementing decision procedures for checking bisimilarity.

Finally, our information flow analysis does not deal properly with side channels. How to do this is an open challenge, even for uncached memory, as here. For instance, precisely modelling timing is infeasible for real systems since we do not have accurate timing information of the underlying hardware. A more successful strategy consists in defining abstract leakage models in the form of observations (e.g., accessed memory addresses affect caches that in turn affect the timing) and preventing timing side channels by proving observational equivalence. We are currently working on validating [31] such models and defining methodologies to handle different side channels at each refinement step [32].

REFERENCES

- [1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating systems errors,” in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 73–88.
- [2] A. Ganapathi, V. Ganapathi, and D. A. Patterson, “Windows XP kernel crash analysis,” in *LISA*, vol. 6, 2006, pp. 49–159.

- [3] V. Orgovan and M. Tricker, "An introduction to driver quality," in *Microsoft Windows Hardware Engineering Conf*, 2003.
- [4] J.-M. Schmidt, T. Plos, M. Kirschbaum, M. Hutter, M. Medwed, and C. Herbst, "Side-channel leakage across borders," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2010, pp. 36–48.
- [5] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected I/O operations in AMD's secure encrypted virtualization," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1257–1272.
- [6] D. Monniaux, "Verification of device drivers and intelligent controllers: a case study," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007, pp. 30–36.
- [7] E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev, "Formal device and programming model for a serial interface," in *Proceedings, 4th International Verification Workshop (VERIFY)*, Bremen, Germany, vol. 259, 2007, pp. 4–20.
- [8] E. Alkassar and M. A. Hillebrand, "Formal functional verification of device drivers," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2008, pp. 225–239.
- [9] J. Duan and J. Regehr, "Correctness proofs for device drivers in embedded systems," in *SSV*, 2010.
- [10] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu, "Toward compositional verification of interruptible OS kernels and device drivers," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 431–447.
- [11] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software safety and security*. IOS Press, 2012, pp. 319–347.
- [12] *AM335x and AMIC110 Sitara Processors Technical Reference Manual*. Texas Instruments, 2019. [Online]. Available: <https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf>
- [13] S. Choudhury, G. Singh, and R. Mehra, "Design and verification serial peripheral interface (SPI) protocol for low power applications," *International Journal of Innovative Research in Science, Engineering and Tegnology*, pp. 16 750–16 758, 2014.
- [14] R. Guanciale, H. Nemat, M. Dam, and C. Baumann, "Provably secure memory isolation for Linux on ARM," *Journal of Computer Security*, vol. 24, no. 6, pp. 793–837, 2016.
- [15] "Open Verificatum project." [Online]. Available: <http://verificatum.org/>
- [16] K. Slind and M. Norrish, "A brief overview of HOL4," in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 28–32.
- [17] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. Springer, 1980, vol. 92. [Online]. Available: <https://doi.org/10.1007/3-540-10235-3>
- [18] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Inf. Comput.*, vol. 100, no. 1, pp. 1–40, 1992. [Online]. Available: [https://doi.org/10.1016/0890-5401\(92\)90008-4](https://doi.org/10.1016/0890-5401(92)90008-4)
- [19] R. Milner, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989.
- [20] R. Focardi, R. Gorrieri, and F. Martinelli, "Non interference for the analysis of cryptographic protocols," in *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000*, ser. Lecture Notes in Computer Science, vol. 1853. Springer, 2000, pp. 354–372.
- [21] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing-and termination-sensitive secure information flow: Exploring a new approach," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 413–428.
- [22] L. C. Noll, R. G. Mende, and S. Sisodiya, "Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system," Mar. 24 1998, US Patent 5,732,138.
- [23] J. Liebow-Feaser, "Lavarand in production: The nitty-gritty technical details," Apr. 2021. [Online]. Available: <https://blog.cloudflare.com/lavarand-in-production-the-nitty-gritty-technical-details/>
- [24] A. Lindner, R. Guanciale, and R. Metere, "Trabin: trustworthy analyses of binaries," *Science of Computer Programming*, vol. 174, pp. 72–89, 2019.
- [25] C. Röckl and J. Esparza, "Proof-checking protocols using bisimulations," in *International Conference on Concurrency Theory*. Springer, 1999, pp. 525–540.
- [26] P. Manolios, K. Namjoshi, and R. Sumners, "Linking theorem proving and model-checking with well-founded bisimulation," in *International Conference on Computer Aided Verification*. Springer, 1999, pp. 369–379.
- [27] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.
- [28] R. Gu, J. Koenig, T. Ramanananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 595–608, 2015.
- [29] J. Duan, *Formal verification of device drivers in embedded systems*. The University of Utah, 2013.
- [30] O. Schwarz and M. Dam, "Formal verification of secure user mode device execution with DMA," in *Haifa Verification Conference*. Springer, 2014, pp. 236–251.
- [31] H. Nemat, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs, "Validation of abstract side-channel models for computer architectures," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 225–248.
- [32] C. Baumann, M. Dam, R. Guanciale, and H. Nemat, "On compositional information flow aware refinement," in *IEEE Computer Security Foundations Symposium*, 2021.

Celestial: A Smart Contracts Verification Framework

Samvid Dharanikota*
Microsoft Research India
Bangalore, India
samvid.dharani@gmail.com

Suvam Mukherjee*
Microsoft Corporation
Redmond, USA
sumukherjee@microsoft.com

Chandrika Bhardwaj#
Goldman Sachs
Bangalore, India
chandrika.bhardwaj@gs.com

Aseem Rastogi
Microsoft Research India
Bangalore, India
aseemr@microsoft.com

Akash Lal
Microsoft Research India
Bangalore, India
akashl@microsoft.com

Abstract—We present CELESTIAL, a framework for formally verifying smart contracts written in the Solidity language for the Ethereum blockchain. CELESTIAL allows programmers to write expressive functional specifications for their contracts. It translates the contracts and the specifications to F^* to formally verify, against an F^* model of the blockchain semantics, that the contracts meet their specifications. Once the verification succeeds, CELESTIAL performs an erasure of the specifications to generate Solidity code for execution on the Ethereum blockchain. We use CELESTIAL to verify several real-world smart contracts from different application domains. Our experience shows that CELESTIAL is a valuable tool for writing high-assurance smart contracts.

Index Terms—Smart contracts, Blockchain, Reliability, Testing

I. INTRODUCTION

Smart contracts are programs that enforce agreements between parties transacting over a blockchain. Till date, more than a million smart contracts have been deployed on the Ethereum blockchain with applications such as digital wallets, tokens, auctions, and games, holding digital assests worth over \$200 billion [19].

The most popular language for smart contract development is Solidity [20]. Solidity contracts are compiled to Ethereum Virtual Machine (EVM) bytecode for execution on the blockchain. Unfortunately, Solidity has obscure operational semantics understood only partially by most programmers. This often leaves vulnerabilities in the smart contracts. Repeated high-profile attacks (e.g. TheDAO [17] and ParityWallet [18] attacks) orchestrated around these vulnerabilities have resulted in financial losses running into millions of dollars. Worse, smart contracts are “burned” into the blockchain on deployment, which does not allow subsequent patches to fix the vulnerabilities. As a result, it is necessary to ensure correctness at the time of deployment.

Smart contracts are relatively small pieces of code with simple data-structures [29]. All these qualities combined—their critical nature, immutability after deployment, and small

size—make smart contracts a good fit for formal verification. The challenge, however, is to lower the formal verification entry barrier for smart contracts developers.

Towards that goal, we present CELESTIAL[§], an open-source framework for developing formally verified smart contracts. CELESTIAL allows programmers to annotate their Solidity contracts with Hoare-style specifications [32] capturing functional correctness properties. The contracts and the specifications are translated to F^* [45], which in an *automated manner*, proves that the contracts meet their specifications. Once F^* returns a verified verdict, CELESTIAL erases the specifications from the input contracts, and emits Solidity code that can be deployed and executed on the Ethereum blockchain. By using Solidity as the source language, and providing fully-automated verification, CELESTIAL ensures a low entry barrier for smart contract developers.

F^* is a proof assistant and program verifier with a fully dependent type system. We find it suitable for smart contract verification for several reasons. First, it provides SMT-based automation which, as we show empirically, suffices for fully-automated verification of real-world smart contracts. Second, F^* supports user-defined effects, allowing us to work in a custom state and exception effect [21] modeling the blockchain semantics. Finally, F^* supports expressive higher-order specifications, though we use its first-order subset with quantifiers and arithmetic (adding our own libraries for arrays and maps).

We evaluate CELESTIAL by verifying several real-world Solidity smart contracts that together currently hold millions of dollars of financial assets. The contracts span different application domains including tokens, wallets, and a governance protocol for Azure Blockchain. We studied the contracts (and in some cases, discussed with the developers) to design their specifications and formally verified that the contracts meet those specifications. In the process, we uncovered bugs in some cases (e.g. missing overflow checks), manifesting as F^* verification failure. Once we fixed those bugs (e.g. by adding runtime checks), F^* was able to successfully verify the contracts in all the cases. The overhead of any additional

*Equal contribution

#Work done during an internship at Microsoft Research India.

[§]<https://github.com/microsoft/verisol/tree/celestial/Celestial>

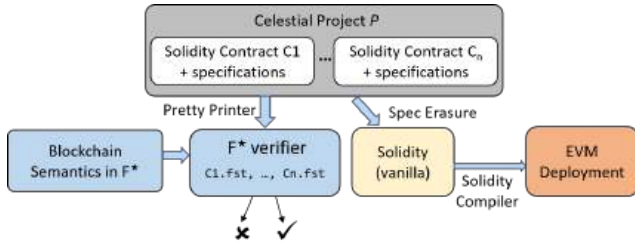


Fig. 1: Architecture of the CELESTIAL framework.

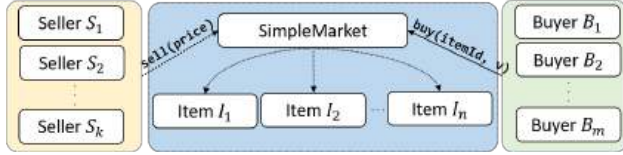


Fig. 2: A simple blockchain based e-commerce application.

instrumentation, which was required for correctness, was at most 20% in terms of gas consumption.

Summarizing our main contributions:

- 1) We present CELESTIAL, a framework for developing verified Solidity smart contracts. CELESTIAL allows annotation of Solidity contracts with specifications, and verifies them, in an automated manner, using F*.
- 2) We evaluate CELESTIAL by verifying functional correctness of several real-world, high-valued smart contracts.

II. OVERVIEW

The high-level architecture of CELESTIAL is outlined in Figure 1. A CELESTIAL project is a set of contracts (e.g. C1, C2, etc. in the figure) written in Solidity. These contracts may be annotated with functional specifications encoding properties of interest. CELESTIAL provides two kinds of translations for these contracts. The first one translates the contracts and their specifications to F* [45], a dependently-typed functional programming language designed for program verification. F*, using a model of the blockchain semantics (Section III), verifies that the contracts meet their specifications. A second translation simply erases all specifications to emit vanilla Solidity contracts. In this section, we use a simple application (Section II-A) to describe the specification language of CELESTIAL (Section II-B). We discuss the verification scope and limitations of the framework later in Section II-C.

A. SIMPLEMART

Consider a simple blockchain-based e-commerce application SIMPLEMART from Figure 2. The application contains a SimpleMarket contract (Listing 1) which interacts with one or more buyers and sellers that may either be smart contracts themselves or externally-owned accounts. A seller registers an item for sale by invoking the `sell` method of SimpleMarket, with the price as argument. In response, SimpleMarket creates an instance of the Item contract, which holds metadata about the new item available for sale. It

```

1 contract SimpleMarket {
2   mapping(address => uint) sellerCredits;
3   mapping(address => Item) itemsToSell;
4   uint totalCredits;
5   event eNewItem (address, address);
6   event eItemSold (address, address);
7
8   function sell (uint price) public
9     returns (address itemId) {
10    Item item = new Item(address(this), msg.sender, price);
11    itemId = address(item);
12    itemsToSell[address(item)] = item;
13    emit eNewItem(msg.sender, itemId);
14  }
15  function buy (address itemId) public payable
16    returns (address seller) {
17    Item item = itemsToSell[itemId];
18    if (item == null) { revert ("No such item"); }
19    if (msg.value != item.getPrice()) {
20      { revert ("Incorrect price"); }
21    }
22    seller = item.getSeller();
23    totalCredits = safe_add (totalCredits, msg.value);
24    sellerCredits[seller] =
25      sellerCredits[seller] + msg.value;
26    delete (itemsToSell[itemId]);
27    emit eItemSold(msg.sender, itemId);
28  }
29  function withdraw (uint amount) public {
30    if (sellerCredits[msg.sender] >= amount) {
31      msg.sender.transfer(amount);
32      sellerCredits[msg.sender] -= amount;
33      totalCredits -= amount;
34    } else { revert ("Insufficient balance"); }
35  }

```

Listing 1: The SimpleMarket Solidity contract

also emits an event (`eNewItem`) informing the seller about the identity (in this case, the address) of the new item. A buyer may purchase an item by invoking the `buy` method of SimpleMarket, passing the item address as an argument, along with the ether amount matching the item price. If the item has not been sold already, SimpleMarket records the sale in its state, which involves adding the ether towards the total sales proceeds for the respective seller and marking the item as being sold. The seller may then withdraw the ether from SimpleMarket via the `withdraw` method.

Functional correctness of the `buy` method requires that if a buyer initiates `buy` with a valid item and price, then the item is sold and the seller sales proceeds are credited, leaving all other sellers' proceeds unchanged. In addition, we would also like to verify that the call does not result in arithmetic overflow of the seller's proceeds because this can result in honest sellers losing their credits.

B. Specification Language

Listing 2 shows excerpts of the CELESTIAL versions of Item and SimpleMarket contracts. The general form of a CELESTIAL contract is shown in Listing 3. These annotations are Hoare-style specifications, similar to languages like Dafny [36]. The specifications are written over the contract fields, function arguments, as well as implicit variables such as `balance` (the contract balance), `value` (ether value in a payable method), and `log` (the transaction event log, formally modeled as a list of events). Our specifications cover the full power of first-order reasoning with quantifiers, along with

```

1 contract Item {
2   address seller; uint price; address market;
3   function getSeller () returns (address s)
4     modifies []
5     post (s == seller)
6   { return seller; }
7   // other methods
8 }
9 contract SimpleMarketplace {
10  // contract fields
11  ...
12  invariant balanceAndSellerCredits {
13    balance == totalCredits &&
14    totalCredits >= sum_mapping (sellerCredits)
15  }
16  function buy (address itemId) public
17    returns (address seller)
18    modifies [sellerCredits, totalCredits, itemsToSell,
19             log]
20    tx_reverts !(itemId in itemsToSell)
21              || msg.value != itemsToSell[itemId].price
22              || msg.value + totalCredits > uint_max
23    post (!(itemId in itemsToSell)
24          && sellerCredits == old(sellerCredits)[
25            seller => old(sellerCredits)[seller] + msg.
26              value]
27          && log == (eItemSold, msg.sender, itemId)::old(
28            log))
29  { // implementation of the buy function }
30 }

```

Listing 2: Item and SimpleMarket CELESTIAL contracts

```

1 contract A {
2   uint x, y; // fields, as usual
3
4   invariant {  $\phi_1$  } // contract-level invariant
5
6   function foo () public
7     modifies [x] // fields that are modified
8     tx_reverts  $\phi_2$  // revert condition (under-specified)
9     pre  $\phi_3$  // precondition
10    post  $\phi$  // postcondition
11    { s } // Solidity implementation
12 }

```

Listing 3: A representative CELESTIAL contract

theories for arithmetic (both modular and non-modular), arrays and maps. We provide programmers the ability to write pure functions that can be invoked only from specifications, not Solidity methods, to enable code reuse. We now explain the individual elements of CELESTIAL specifications.

a) Contract invariant: Contract invariant is a predicate on the state of the contract (i.e. its field values) that is expected to be valid at the boundaries of its public methods. When verifying a contract, the invariant is added to the pre- and postconditions of every public method. All contract fields in a CELESTIAL contract are necessarily private (see Section II-C). Additionally, CELESTIAL ensures that all its contracts are *external callback free* (Section IV) to disallow re-entrancy based attacks from external contracts. Hence, it is safe to assume the invariant at the beginning of public methods. Constructors are special; they only guarantee invariant in their postcondition but don’t assume it as a precondition. For example, the invariant on line 12 in Listing 2 specifies that the contract’s balance equals or exceeds the total proceeds from sales which has not been already claimed by the respective sellers (sum_mapping is a library function for summing values

in an int-valued map).

b) Field updates: The modifies clause specifies contract fields that a method can update. The getSeller method in Item has an empty modifies clause (line 4 in Listing 2), which specifies that the function may read the state of the contract, but cannot make any updates.

c) Pre- and postconditions: Preconditions (pre) are properties that hold at the beginning of a method execution. Public methods must have a trivial precondition true because they can be invoked by the untrusted external world. Postconditions (post) are properties that hold when the method terminates successfully (without reverting). The postconditions may refer to field values at the beginning of the method using the old keyword. For example, the condition in line 23 in Listing 2 specifies that the final sellerCredits is the original sellerCredits map with only the seller key updated.

d) Revert conditions: tx_reverts under-specifies the conditions under which a method reverts, i.e. if tx_reverts holds at the beginning of a method, the method will definitely revert. For example, the buy function definitely reverts if the buyer invokes it with an item which is not available for sale, or the buyer provides ether which does not match the item price, or the totalCredits overflows. This is captured in the specification in line 19. Not specifying tx_reverts is equivalent to tx_reverts(false).

e) Safe Arithmetic: In Solidity, arithmetic operations may silently over- or underflow, whereas division by 0 results in reverts. CELESTIAL, when translating to F*, adds assertions before every arithmetic operation which check for no over- and underflows, and division by 0. The programmer must add specifications or runtime checks to allow the verifier to prove the safety of the arithmetic operations. CELESTIAL also provides a safe arithmetic library with built-in runtime checks (safe_add operation in line 22 of Listing 1).

To summarize, we have expressed the following properties of the buy method. The revert condition specifies that the method reverts when the item is not present or the ether sent by the buyer does not match the item price. The method also reverts when totalCredits overflows. Since an invariant of the contract is that totalCredits is greater than the sum of pending credits of all the sellers, when totalCredits does not overflow, individual seller credits also don’t overflow. Finally, line 23 in Listing 2 specifies that only the item seller’s credits are incremented by price of the item, while credits for all other sellers remain same.

C. Verification Scope and Limitations

a) Threat model: All contracts and user accounts that are not part of a CELESTIAL project P are treated as the *external world* for P. The external world is free to initiate arbitrary transactions by calling public methods of P with arbitrary arguments. The external world, however, cannot directly access the private fields and methods of P.

b) Trusted Computing Base: The TCB of CELESTIAL includes the CELESTIAL compiler consisting of the two syntax translations, the F* model of the blockchain (Section III), the

F* toolchain itself, and the Solidity compiler (these components are colored blue in Figure 1). With these components in our TCB, formal verification of smart contracts in CELESTIAL guarantees that when the compiled Solidity contracts are run on the blockchain, they behave as per their specifications. We leave it as future work to minimize trust on our F* blockchain semantics (say, by testing it against a Solidity test suite).

c) *Solidity Language Restrictions*: CELESTIAL does not support `delegatecall` which is used to call functions from other contracts in a way that the callee may directly change the state of the calling address, thereby breaking the function call abstraction. Since this is insecure (for example, the ParityWallet [18] attack exploited it), the secure development recommendations suggest against its use [3]. CELESTIAL also does not support embedding EVM assembly. To check the prevalence of these features in real-world contracts, we performed an empirical study. In summary, we found that not more than 45% of highly used and highly valued contracts use these features, and even then in controlled manner where their usage is restricted to a small set of libraries.

d) *Modeling Limitations*: Our F* semantics does not model gas consumption. As a result, CELESTIAL contracts may revert due to out-of-gas exceptions. The model also does not cover low-level failures such as callstack depth overflow. However, these failures can only cause the transaction to revert and therefore do not compromise the verification guarantees. Since we do not model all runtime exceptions, this is one of the reasons that the `tx_reverts` condition for a function is an under-specification for when the function may revert. We also do not precisely model block-level parameters such as timestamp.

III. VERIFYING CELESTIAL CONTRACTS IN F*

CELESTIAL compiles the contracts and their specifications to F*, which are then verified against a trusted F* library modeling the blockchain semantics. The library consists of the definition of the blockchain state datatype and a custom F* *effect* that encapsulates this state behind the abstraction of an effect layer. We have carefully designed this abstraction to ensure that the verification is scalable and fully automated. The contracts call the stateful API exported by the library and specify precise changes to the blockchain state in their pre- and postconditions, that are verified by F*.

A. Blockchain state

We model the blockchain state as consisting of 3 main elements: (a) state of all the contracts (i.e. values of the contract fields), (b) contract balances, and (c) an event log. Since in CELESTIAL all contract fields are private, a contract can directly read or write only its own fields, while interacting with the other contracts through method calls. The event log models the per-transaction event log of the Ethereum blockchain; contracts can use the Solidity `emit` API to output events to this log.

a) *Contracts state*: We model the state of all the contracts in the blockchain as a heterogeneous map from addresses to records, where the record corresponding to a contract instance contains the values of all its fields. For the `Item` contract from Listing 2, the record type would be:

```
type item_t = { market : address; seller : address; price : uint }
```

Below is the API provided by the contract map (# parameters are implicit parameters inferred by F* at the call sites):

```
type address = uint (* 256 bit unsigned integers *)
val contract (a:Type) : Type (* a is the record of contract fields *)
val cmap : Type (* the heterogeneous contracts map *)

val live (#a:Type) (c:contract a) (m:cmap) : prop
val sel (#a:Type) (c:contract a) (m:cmap{live c m}) : a
val create (#a:Type) (m:cmap) (x:a) : contract a & cmap
val upd (#a:Type) (c:contract a) (m:cmap{live c m}) (x:a) : cmap
val addr_of (#a:Type) (c:contract a) : address
```

The API defines the type `address` as 256 bit unsigned integers. The contract type is parametric over the record type `a` that contains all the contract fields; for the `Item` contract, type `a` will be instantiated with `item_t`. Type `cmap` is the heterogeneous contracts map type.

The `sel` function returns the `a`-typed record value mapped to a contract instance in the map. The API requires that the contract be `live` in the map (type `m:cmap{live c m}` is a refinement type that requires that the `m` argument at the call sites satisfies `live c m`). The liveness requirement basically says that the contract must be present in the contracts map, preventing `sel` to be called with arbitrary addresses. The `create` function returns the freshly created contract and the new `cmap` that includes a mapping for the new contract, internally assigning a fresh address to the new contract. The API is fully implemented in F*, we elide the implementation details for space reasons; all of our development is available online at <https://github.com/microsoft/verisol/tree/celestial/Celestial>.

b) *Contracts balance*: We model the contracts balance using a map from addresses to `uint` (the type of 256-bits unsigned integers). An alternative would have been to add balance as another one of the contract fields (thus maintaining them as part of the contracts map), but a separate map allows us to specify the balances for external accounts, that do not have an entry in the contracts map.

c) *Event log*: The event log is a list of events, where each event records the destination address, a string for event type, and a payload (`a:Type` & `a` is a dependent tuple that packages a `Type` and a value of that type):

```
type event = { to : address; ev_type : string; payload : (a:Type & a) }
type log = list event
```

With these components, the blockchain state is the following record type:

```
type bstate = { cmap : cmap; balances : Map.t address uint; log : log }
```

B. Libraries for arrays and maps

We have implemented F* libraries for modeling Solidity arrays and maps—the uses of arrays and maps in CELESTIAL contracts are translated to uses of these F* libraries.

Our current implementation only supports dynamically-sized arrays for now, support for compile time fixed-sized arrays is future work. The libraries export operations that match the corresponding Solidity API, and several lemmas that enable the contracts to reason about their properties. For example, following is a snippet of our array library:

```
val array (a:Type) : Type (* an array with element type a *)
val push (#a:Type) (s:array a{length s < uint_max}) (x:a) : array a
val push_length (#a:Type) (s:array a{length s < uint_max}) (x:a)
  : Lemma (requires T) (ensures (length (push s x) == length s + 1))
```

C. An F^* effect for contracts

Having set up the model for the blockchain state, we now add a layer on top so that the contracts may manipulate the state and precisely specify the modifications in pre- and postconditions, while making sure that the verification complexity does not get out-of-hands. We leverage the type-and-effect system of F^* for this purpose.

F^* distinguishes value types such as `uint` from *computation types*. Computation types specify the effect of a computation, its result type, and optionally some specifications (e.g. pre- and postconditions) for the computation. For example, `Tot uint` classifies pure, terminating computations that return a `uint` value. Similarly `uint → Tot uint` is the type of pure, terminating functions that take a `uint` argument and return a `uint` result. `uint → uint` is a shorthand for `uint → Tot uint`; all the blockchain state functions that we have seen so far have an implicit `Tot` effect.

Following Ahman et al. [21], a state and exception effect for computations that operate on mutable state and may throw exceptions is as follows (`st` is the type of mutable state):

```
type result (a:Type) = (* the return type of the computations *)
  Success : x:a → result a
  Error : e:string → result a
```

```
effect STEXN a st (pre:st → prop) (post:st → result a → st → prop) = ...
```

The semantics of the computations in the `STEXN` effect may be understood as follows: a computation `e` of type `STEXN a st pre post` when run in an initial state ($s_0:st$) satisfying `pre s0`, terminates either by throwing an exception (modeled as returning an `Error`-valued result) or by returning a value of type `a` (modeled as returning `Success`-valued result). In either case, the final state ($s_1:st$) is such that `post s0 r s1` holds, where `r` is the return value of the computation. F^* also supports divergent effects, in which case the computations are also allowed to diverge. The `STEXN` effect in F^* comes with a program logic for verifying such computations.

a) *Customizing STEXN for contracts:* Contract computations naturally fall into the state and exception effect; they read from and write to the mutable blockchain state, and they may throw an exception by calling `revert`.

However, the `revert` operation in Ethereum is slightly different from exceptions in, say, OCaml in that it also reverts the underlying state to what it was at the beginning of the transaction, while in OCaml, the state changes are retained. To accommodate this, we instantiate the state `st` in `STEXN` with

```
type st = { tx_begin : bstate; current : bstate }
```

where the field `tx_begin` snapshots the state at the beginning of a transaction. Contracts modify the current state, unless they `revert`, in which case the current state is reset to `tx_begin`. Thus, we define the `ETH` effect for smart contracts as follows:

```
(* state + exception with st as the state *)
effect ETH (a:Type) (pre:st → prop) (post:st → result a → prop) =
  STEXN a st pre post
```

Using `ETH` effect, we implement the APIs for `begin_transaction`, `revert`, and `commit_transaction` as follows:

```
let begin_transaction () : ETH unit (requires λ_ → T)
  (ensures λs0 r s1 → is_success r ∧ s0 == s1) = () (* no op *)

let revert () : ETH unit (requires λ_ → T)
  (ensures λs0 r s1 → is_err r ∧ s1 == {s0 with current=s0.tx_begin}) = ...

let commit_transaction () : ETH unit (requires λ_ → T)
  (ensures λs0 r s1 → is_succ r ∧ s1 == {s0 with tx_begin=s0.current}) = ...
```

The function `begin_transaction` is a no-op, its precondition is trivial (T), while its postcondition states that it does not `revert` (`is_success r`) and it leaves the state unchanged (`s0 == s1`). `revert`, on the other hand, returns an error value, and its output state `s1` is same as its input state `s0` with `current` component replaced with the snapshot `s0.tx_begin`, i.e. the state at the beginning of the transaction. `commit_transaction` is opposite, it replaces the `tx_begin` component with `s0.current` to commit the current state.

The function to get the current state for a contract is as follows, note that the contract is selected from the current component of the state:

```
let get_contract (#a:Type) (c:contract a) : ETH a
  (requires λs → live c s.current.cmap)
  (ensures λs0 x s1 → x == Success (sel c s.current.cmap) ∧
    s0 == s1) = ...
```

Similarly, the library provides functions `send` to transfer balance to a contract and `emit` to emit an event to the event log.

To make our specifications easier to read and write, we define the following effect abbreviation:

```
effect Eth (a:Type) (pre:bstate → prop) (revert:bstate → prop)
  (post:bstate → a → bstate → prop)
  = ETH a (requires λs → pre s.current)
  (ensures λs0 r s1 →
    (revert s0.current ==> Error? r) ∧
    (Success? r ==> post s0.current (Success?.x r) s1.current))
```

The pre- and postconditions in the `Eth` effect are written over the current blockchain state (`bstate`), as opposed to over the `st` record. Further, the postcondition is a predicate on a value of type `a`—it only specifies what happens when the contract function terminates successfully. The `revert` predicate is a predicate on the input state, which if valid means that the function reverts. We find this abbreviation well-suited for our examples, providing the full-flexibility of the `ETH` effect to the programmers is of course possible.

CELESTIAL translates each contract to an F^* module, where the contract methods are translated to F^* functions in the `Eth` effect. Every function gets explicit parameters for self, sender, value in the case of payable functions, and (underspecified)

block-level parameters such as timestamp; after these the function specific parameters follow.

The F^* precondition of each function gets to assume the liveness of the contract and the contract invariant. Since these functions can be called by arbitrary, non-verified code, we cannot expect the callers to satisfy more sophisticated preconditions. The postcondition of each function includes the liveness, the contract invariant, and other function-specific postconditions.

The translation of a function body uses the private, per-field getters and setters, also emitted by the translation. Calls to public functions of other contracts are translated to calls to corresponding functions in other F^* modules (contracts). Library calls to arrays, maps, etc. translate to corresponding libraries calls in F^* .

We make a final comment regarding the correctness of the various translations. Since the CELESTIAL source language is just Solidity with specifications, the CELESTIAL to Solidity translation is only spec erasure. The translation to F^* is again quite systematic, and therefore, amenable to auditing. Formally proving that the CELESTIAL to F^* translation is semantics preserving is an interesting and challenging future work.

IV. IMPLEMENTING CELESTIAL

The translators to F^* , for specifications as well as implementation, are combined 2300 lines of Python code. The spec-erasing translator to Solidity is about 750 lines of Python code. The blockchain model is around 1200 lines of F^* code. We target the 0.6.8 version of the Solidity compiler for generating EVM bytecode. To aid developer experience, we have written a plugin for Visual Studio Code [16] that supports full syntax highlighting for CELESTIAL. If developers require access to the CELESTIAL specifications in the generated Solidity, we can easily tweak the CELESTIAL to Solidity translation to preserve the specifications as comments.

Limitations: We focused our implementation efforts on Solidity constructs used in our case studies. We currently do not support syntactic features such as inheritance, abstract contracts and tuple types. These mostly only provide syntactic sugar that should be easy to support in future versions of CELESTIAL. Our implementation currently also does not support passing arrays and structs as arguments to functions. While our implementation allows loops in contract functions, we currently do not support writing loop invariants. We also only provide weak specifications for block level constructs (such as timestamp, number and gaslimit), transaction level constructs (such as origin and gasprice), and functions for obtaining hashes (such as keccak256 and sha256).

Contract Local Reasoning: Calling external contracts can lead to *reentrant* behavior where the external contract calls back into the caller, which is often difficult to reason about. CELESTIAL disallows such behaviors by checking for *external callback freedom* (ECF) [28], [42] which states that every contract execution that contains a reentrant callback is *equivalent* to some behavior with no reentrancy. When this property holds, it is sufficient to reason about non-reentrant

```
1 contract A {
2     bool lock;
3     function foo () public
4         tx_reverts lock
5     { if(lock) { revert; } ... }
6
7     function bar (address x) {
8         lock = true;
9         // external call
10        x.call(...);
11        lock = false;
12        ...
13    }
```

Listing 4: Ensuring External Callback Freedom

behaviors only: any specification over those set of behaviors will hold for all behaviors as well. Thus, ECF allows for contract-local reasoning.

CELESTIAL has two ways of checking for ECF; one of these must hold for each external call. The first is a lightweight syntactic check from VERX [42]. An external call is deemed ECF compliant if it is guaranteed to only be called at the end of a transaction. In other words, for any public method that may transitively invoke an external call, it must ensure that it does not read or write to the blockchain state after the call. External calls that do not fall in this category must satisfy CELESTIAL’s second check that asserts that any callbacks made by an external call are guaranteed to revert. We explain this check using the CELESTIAL contract shown in Listing 4. There is an external call in method bar on line 10. To prevent reentrancy, the programmer uses a contract field called lock and follows the protocol that the lock will be assigned true when making an external call. Furthermore, each public method of the contract (such as foo) will revert if lock is set to *true*. It is easy to see that if the external contracts tries to call back a method of A, the transaction will abort.

CELESTIAL’s translation to F^* adds a sequence of assertions preceding each external call (that does not satisfy CELESTIAL’s first check). For each public method of the contract, it takes the tx_reverts condition on the method, say ϕ , and inserts `assert ϕ` before the external call. This will ensure that a call back to a public method is guaranteed to revert.

V. EVALUATION

We evaluate the development experience with CELESTIAL by writing verified versions of 8 Solidity smart contracts, including real-world contracts spanning crypto-currency tokens, wallets, marketplace, auctions and governance. Some of these contracts are “high-valued”, holding millions of dollars of financial assets or having processed millions of transactions.

For each contract, we added detailed functional specifications. If the verification failed, we minimally modified the code in order to discharge the verification conditions. For contracts which required such modifications, we additionally measured the gas consumption overhead, using Truffle [13]. We performed our experiments using an Intel Core i7-7600U dual-core CPU, with 16GB RAM, and running Windows 10. Table I summarizes the various case studies that we performed.



Fig. 3: The AssetTransfer state machine. The dashed arrow indicates a buggy state transition.

Due to lack of space, we discuss details of 3 of the case studies here. We refer interested readers to our Technical Report [25] for a detailed discussion of all the case studies. The sources for all the case studies are available at <https://github.com/microsoft/verisol/tree/celestial/Celestial>.

Benchmark	#C	#Sol	CELESTIAL		V-Time (sec)
			#Spec	#Impl	
AssetTransfer*	1	130	0	18	4.26
OpenZeppelin ERC20	4	1 1	9	200	8.82
BinanceCoin*	2	133	25	136	29.98
WrappedEther*	1	62	62	114	20.00
EtherDelta*	1	281	5	351	63.9
Consensus MultiSig*	2	3 8	163	289	.80
SimpleAuction*	1	66	61	101	22.45
Governance Contract	1	41	121	149	86.86

TABLE I: CELESTIAL case studies. We report the number of contracts in the application (#C), LOC of the original Solidity implementation (#Sol), LOC of the CELESTIAL version, divided between specification (#Spec) and implementation (#Impl), and finally the F* verification time (averaged over 3 runs). Benchmarks marked with * used a safe arithmetic library, which is added towards #Impl.

A. AssetTransfer

Application: AssetTransfer [10] is a microbenchmark that provides a smart contract based solution for transferring assets between a buyer and a seller. The contract encodes asset transfer as a finite state machine (FSM) (Figure 3), a common design pattern [11], [39], with the different states denoting the varying stages of approval for the transfer. The contract has notions of *roles*, such as Buyer and Seller, and state transitions are guarded by appropriate roles (for example, the contract can transition from Active to OfferPlaced when the Seller invokes the MakeOffer method).

Specifications. Figure 3 is also the specification for this contract, that is, we must ensure that each of the contract methods respect the transitions mentioned in the FSM diagram. For example, the following is the spec for MakeOffer:

```
function MakeOffer (uint _price)
  modifies [sellingPrice, state, log]
  tx_revert (old(state) != Active && msg.sender != Seller)
  post (state == OfferPlaced && sellingPrice == _price)
  { // implementation }
```

The spec ensures that the method makes the correct state transition (Active \rightarrow OfferPlaced), and this transition can only be caused by the Seller. Interestingly, this spec failed to verify, which led us to discover two bugs in the implementation. These bugs could potentially leave the whole

transfer in a frozen state. For instance, one of the bugs led to the erroneous state transition shown in Figure 3. It caused the contract to mistakenly transition to the SellerAccept state, even after both the Seller and Buyer had accepted the transfer, which makes the final state (Accept) to become unreachable. Fixing these bugs allowed verification to go through. Previous work [47] has noted similar bugs in a different version of the contract. The original contract also had overflow/underflow vulnerabilities, which we eliminated using runtime checks.

Performance. We ran both contracts (CELESTIAL-generated Solidity and original Solidity) through a typical asset-transfer workflow. On an average, the CELESTIAL version consumed $1.12\times$ more gas compared to the original. We account for both the contract as well as any associated library, for instance for safe arithmetic, when measuring the deployment cost.

B. ERC20 Tokens

Application. ERC20 is a standard [4] for Ethereum cryptocurrencies (or *tokens*). Till date, over 400K ERC20 tokens have been deployed on Ethereum, handling financial assets worth *billions of dollars*. We formally verified the OpenZeppelin ERC20 contract [8], which is a popular reference implementation of some of the key ERC20 functions, such as transferring tokens from one account to another and approving third parties to spend tokens on a user’s behalf. We also verified the ERC20-based BinanceCoin (BNB) [2] token.

Specifications. We based some of our specifications on earlier efforts to formally verify the OpenZeppelin ERC20 token [6], [47]. The following shows an excerpt. The implementation maintains the balance (number of issued tokens) for each contract address using a `_balances` map. CELESTIAL allows us to easily express the important invariant (line 4) that the sum over the balances for each user equals the total number of tokens issued.

```
1 contract ERC20 {
2   mapping (address => uint) _balances;
3   uint _totalSupply; // total issued tokens
4   invariant _balanceAndSellerCredits {
5     _totalSupply = sum_mapping(_balances)
6   }
7 }
```

The remaining specifications capture the business logic of key ERC20 functions. The example below shows the postcondition for the `_transfer` method that is used for atomically debiting a source account, and crediting the amount in a destination account. The postcondition ensures that the correct debit and credit operations occur in the source and destination accounts, and all other accounts remain unchanged.

```
1 function _transfer (address from, address to, uint amt)
2   private tx_reverts ..., modifies [...]
3   pre _balances[from] >= amt &&
4     _balances[to] + amt <= uint_max
5   post ite(from == to, _balances == old(_balances),
6     _balances == old(_balances)[
7       from => old(_balances)[from] - amt,
8       to => old(_balances)[to] + amt])
9   { // implementation }
```

The ERC20 token makes copious use of arithmetic operations. OpenZeppelin designed a SafeMath Library [9] to perform runtime checks for overflows and underflows, which

the original ERC20 token leverages to ensure runtime safety for arithmetic operations. In contrast, we used the CELESTIAL safe arithmetic operations in public functions, and eliminated runtime checks altogether in private functions when the arithmetic was provably safe.

C. Governance Contract

Application. We study a contract from Microsoft that manages a consortium of mutually-trusted members interacting on a *private* Ethereum blockchain. The contract comprises a set of rules governing operations such as inviting fresh members to join the consortium and adding or removing existing members. The contract is complex, since it maintains many correlated data structures, loops and access control policies, with each logical operation involving intricate changes to multiple data structures. Due to the proprietary nature of the contract, we abstain from showing code or specifications for it explicitly. We did not include several functions in the original contract, whose operations were orthogonal to the governance logic.

Specifications. We briefly describe some of the important properties that we proved.

- 1) Among members in the consortium, some are designated as being “administrators”. An important invariant is that the number of administrators cannot be zero (otherwise the consortium freezes with no further transaction processing).
- 2) In the contract, logical units of information are maintained in aggregate by several data structures. For example, the contract maintains an array of existing members. However, members can either be referenced by a string identifier, or an address. Thus, the contract maintains a couple of additional mappings that maintain, respectively, associations between string identifiers and addresses, to the correct indices in the array. We specify several invariants to ensure that these data structures are always consistent. For example, we specify that there are no duplicates in the array, no two string identifiers map to the same array index, and the value of each string identifier must not exceed the length of the array of members.
- 3) We precisely captured the postconditions for operations such as member additions, where we ensure that the operation only updates the necessary keys and indices, while leaving the remaining entries untouched.

We note that some of these properties are similar to those proved by Lahiri et al [35] for a variation of an open-source governance contract [14].

VI. RELATED WORK

The literature on ensuring correctness of smart contracts can be classified into the following broad categories.

Surveys and Best Practices. There is a wealth of available material that highlights known vulnerabilities and exploits in smart contracts [22], [24], [41], [46]. These efforts have resulted in literature suggesting best coding practices for Solidity [5], [12]. CELESTIAL is inspired by these practices, for instance, by ruling out low-level instructions as well as uncontrolled reentrancy, however, the restrictions are not just

for avoiding programming pitfalls, but rather to aid semantic verification.

Testing. Frameworks like Truffle [13] allow users to write unit and integration tests for smart contracts in JavaScript. The transactions are typically executed in an in-memory mock of the EVM, such as Ganache [7]. In addition to testing functional behaviors and finding bugs, such tests reveal useful diagnostic information such as gas consumption.

Contract Analysis. A large number of tools have been developed that statically analyze smart contracts (Solidity source code or EVM bytecode) to reveal various vulnerabilities. Examples include MadMax [27] (targeting vulnerabilities due to gas exceptions) and Slither [26] (for identifying security vulnerabilities). Oyente [38] leverages symbolic execution to rule out several classes of vulnerabilities. ContractFuzzer [33] offers a fuzzing based solution for identifying security bugs.

Solythesis [37] is a source-to-source Solidity compiler that instruments the Solidity code with runtime checks to enforce invariants, but specifications particular to each function can’t be specified in this framework and it has a significantly high gas overhead because of the runtime checks. VeriSmart [44] offers a highly precise verifier for ensuring arithmetic safety of Ethereum smart contracts, which discovers transaction invariants, but is unable to capture quantified transaction invariants. Tools like teEther [34] leverage symbolic execution to find vulnerable executions and automatically generate exploits.

Each of these tools target a known set of vulnerabilities and offer specialized solutions for them. In contrast, CELESTIAL verifies custom specifications of contracts, relying on verification to rule out all vulnerabilities against that specification.

Formal Verification. VeriSol [35], [47] checks conformance between a state-machine-based workflow and the smart contract implementation, for contracts of Azure Blockchain Workbench [1]. VeriSol does not check for reentrancy; it simply assumes its absence, as opposed to CELESTIAL that enforces it as part of the contract verification. Further, VeriSol does not model arithmetic over/underflow, or check for unsafe type casts, which were an important aspect of our case studies.

VerX [15], [42] is another formal verification tool. VerX uses a syntactic check to ensure ECF (which we use in CELESTIAL as well), however it cannot verify that the program in Listing 4 satisfies ECF. VerX aims for automation of verification by inferring predicates in an abstraction-refinement loop. Such techniques tend to be limited in their ability to reason with quantifiers; VerX uses special built-in predicates like *sum* for quantified reasoning over maps. CELESTIAL, on the other hand, allows for the full power of first-order reasoning with quantifiers. VerX implements its own custom symbolic execution, whereas CELESTIAL uses a simple syntax translation to F* and delegates all analysis to the mature F* verifier. Unfortunately, the VerX tool is not openly available for further comparisons.

Some verification tools work at the level of EVM bytecode [30], [31], [40], [43], instead of Solidity source level. This is more precise and removes the Solidity compiler from the TCB, however, it is also more time consuming and hard to

scale to the larger, complex contracts that we have evaluated in Section V. Bhargavan et al. [23] provide an approach to translate a subset of Solidity to F^* for verification, as well as a method to decompile EVM bytecode to F^* to check low-level properties such as establishing worst-case gas bounds for a transaction. Their work is presented as a proof-of-concept only, with limited evaluation and restricted to a small subset of the language.

VII. CONCLUSION


We presented CELESTIAL, a framework for developing formally verified smart contracts. CELESTIAL provides fully automated verification, using F^* , of Solidity contracts annotated with functional correctness specifications. With the help of several real-world case studies, we conclude that formal verification can be made accessible to smart contract developers for programming high-assurance contracts. Our next steps include enriching our F^* model of blockchain with more features and validating it using the Solidity testsuite as well as exploring proofs of cross-transaction properties.

REFERENCES

- [1] Azure blockchain workbench. <https://azure.microsoft.com/en-us/solutions/blockchain/>.
- [2] Binance coin. <https://www.binance.com/en>.
- [3] Consensus secure development recommendations. <https://consensus.github.io/smart-contract-best-practices/recommendations/>.
- [4] Eip 20: Erc-20 token standard. <https://eips.ethereum.org/EIPS/eip-20>.
- [5] Ethereum smart contract security best practices. <https://consensus.github.io/smart-contract-best-practices/>.
- [6] Formal verification of erc20 implementations with verisol. <https://forum.openzeppelin.com/t/formal-verification-of-erc20-implementations-with-verisol/1824>.
- [7] Ganache. <https://github.com/trufflesuite/ganache>.
- [8] Openzeppelin erc20. <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol>.
- [9] Openzeppelin safemath. <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/SafeMath.sol>.
- [10] Remix ethereum ide. <https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/asset-transfer>.
- [11] Solidity docs: State machines. <https://solidity.readthedocs.io/en/v0.6.8/common-patterns.html#state-machine>.
- [12] Solidity security considerations. <https://solidity.readthedocs.io/en/v0.6.8/security-considerations.html>.
- [13] Truffle suite. <https://www.trufflesuite.com/>.
- [14] Validator set contracts. <https://github.com/Azure-Samples/blockchain/tree/master/ledger/template/ethereum-on-azure/permissioning-contracts/validation-set>.
- [15] Verx. <https://verx.ch/>.
- [16] Visual studio code. <https://code.visualstudio.com/>.
- [17] Understanding the dao attack. <https://www.coindesk.com/understanding-dao-hack-journalists>, 2016.
- [18] The parity wallet hack explained. <https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/>, 2017.
- [19] Etherscan: Contract accounts. <https://etherscan.io/accounts/c>, 2020.
- [20] Solidity v0.7.2. <https://solidity.readthedocs.io/en/v0.7.2/>, 2020.
- [21] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 515–529. ACM, 2017.
- [22] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.
- [23] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kula-tova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. Formal verification of smart contracts: Short paper. In Toby C. Murray and Deian Stefan, editors, *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, pages 91–96. ACM, 2016.
- [24] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks and defenses. *CoRR*, abs/1908.04507, 2019.
- [25] Samvid Dharanikota, Suvam Mukherjee, Chandrika Bhardwaj, Aseem Rastogi, and Akash Lal. Celestial: A smart contracts verification framework. Technical Report MSR-TR-2020-43, Microsoft, December 2020.
- [26] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 8–15. IEEE / ACM, 2019.
- [27] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, 2018.
- [28] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzy, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.
- [29] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 531–548. ACM, 2019.
- [30] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 204–217. IEEE Computer Society, 2018.
- [31] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, volume 10323 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2017.
- [32] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [33] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 259–269. ACM, 2018.
- [34] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1317–1333. USENIX Association, 2018.
- [35] Shuvendu K. Lahiri, Shuo Chen, Yuepeng Wang, and Isil Dillig. Formal specification and verification of smart contracts for azure blockchain. *CoRR*, abs/1812.08829, 2018.
- [36] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
- [37] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on*

- Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 438–453. ACM, 2020.
- [38] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016.
 - [39] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In Sarah Meiklejohn and Kazuo Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 523–540. Springer, 2018.
 - [40] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 175–188. ACM, 2014.
 - [41] Daniel Pérez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? *CoRR*, abs/1902.06710, 2019.
 - [42] Anton Permenov, Dimitar Dimitrov, Petar Tsankov, Dana Drachler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*, pages 18–20, 2020.
 - [43] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010.
 - [44] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VERISMART: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1678–1694. IEEE, 2020.
 - [45] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.
 - [46] Antonio Lopez Vivar, Alberto Turégano Castedo, Ana Lucila Sandoval Orozco, and Luis Javier García-Villalba. An analysis of smart contracts security threats alongside existing solutions. *Entropy*, 22(2):203, 2020.
 - [47] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, volume 12031 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2019.

The Civil Verifier

Bernhard Kragl 
Amazon Web Services and IST Austria

Shaz Qadeer
Facebook

Abstract—Civil is a static verifier for concurrent programs designed around the conceptual framework of layered refinement, which views the task of verifying a program as a sequence of program simplification steps each justified by its own invariant. Civil verifies a layered concurrent program that compactly expresses all the programs in this sequence and the supporting invariants. This paper presents the design and implementation of the Civil verifier.

I. INTRODUCTION

Correctness of critical specifications of concurrent systems rests upon invariants about the global system state. The classical approach to static verification is to represent the entire organizational structure—processes, threads, procedures, looping, branching, sequencing—of a concurrent system as a flat transition relation that encodes its operational semantics. Further reasoning is performed on this transition relation. This approach leads to massively complex invariants that are hard to specify for the programmer and difficult to verify via automated tools.

$$\begin{array}{l}
 a: x := n \\
 b: \text{acquire}(l) \\
 c: t_1 := x \\
 d: x := t_1 + 1 \\
 e: \text{release}(l) \\
 f: \text{assert } x = n + 2
 \end{array}
 \parallel
 \begin{array}{l}
 \text{acquire}(l) \\
 t_2 := x \\
 x := t_2 + 1 \\
 \text{release}(l)
 \end{array}$$

Fig. 1. Parallel increment (version 0).

We motivate our work using the program in Figure 1. This program starts with a single thread that initializes a global variable x to a constant n , creates two threads that run in parallel each incrementing x by 1 while holding the lock l , waits for the two threads to finish, and then asserts that $x = n + 2$. The goal of verification is to prove this assertion for all values of n and all executions of the program.

The classical approach to verification of concurrent programs models the verification problem in Figure 1 as a transition system shown in Figure 2, comprising an initial predicate *Init*, a transition predicate *Next*, and a safety predicate *Safe*. To prove that all reachable states of the transition system satisfy the predicate *Safe*, an inductive invariant *Inv* must be invented such that $\text{Init} \Rightarrow \text{Inv}$, $\text{Inv} \wedge \text{Next} \Rightarrow \text{Inv}'$, and $\text{Inv} \Rightarrow \text{Safe}$.

This research was performed while Bernhard Kragl was at IST Austria, supported in part by the Austrian Science Fund (FWF) under grant Z211-N23 (Wittgenstein Award).

Init: $pc = pc_1 = pc_2 = a$

Next:

$$\begin{aligned}
 pc &= a \wedge pc' = pc_1 = pc_2 = b \wedge x' = n \wedge eq(l, t_1, t_2) \\
 pc_1 &= b \wedge pc'_1 = c \wedge l = \bigcirc \wedge l' = \textcircled{1} \wedge eq(pc, pc_2, x, t_1, t_2) \\
 pc_1 &= c \wedge pc'_1 = d \wedge t'_1 = x \wedge eq(pc, pc_2, l, x, t_2) \\
 pc_1 &= d \wedge pc'_1 = e \wedge x' = t_1 + 1 \wedge eq(pc, pc_2, l, t_1, t_2) \\
 pc_1 &= e \wedge pc'_1 = f \wedge l' = \bigcirc \wedge eq(pc, pc_2, x, t_1, t_2) \\
 pc_2 &= b \wedge pc'_2 = c \wedge l = \bigcirc \wedge l' = \textcircled{2} \wedge eq(pc, pc_1, x, t_1, t_2) \\
 pc_2 &= c \wedge pc'_2 = d \wedge t'_2 = x \wedge eq(pc, pc_1, l, x, t_1) \\
 pc_2 &= d \wedge pc'_2 = e \wedge x' = t_2 + 1 \wedge eq(pc, pc_1, l, t_1, t_2) \\
 pc_2 &= e \wedge pc'_2 = f \wedge l' = \bigcirc \wedge eq(pc, pc_1, x, t_1, t_2) \\
 pc_1 &= pc_2 = f \wedge pc' = f \wedge eq(pc_1, pc_2, l, x, t_1, t_2)
 \end{aligned}$$

Safe: $(pc = f \Rightarrow x = n + 2) \wedge$
 $(pc_1 \in \{c, d, e\} \Rightarrow l = \textcircled{1}) \wedge (pc_2 \in \{c, d, e\} \Rightarrow l = \textcircled{2})$

Fig. 2. Transition relation of the program in Figure 1. The lock l can be either available (value \bigcirc), or held by the first or second thread (values $\textcircled{1}$ and $\textcircled{2}$). The predicate eq denotes unmodified variables, e.g., $eq(l)$ means $l' = l$.

This approach is clearly problematic for several reasons. First, the encoding as a transition system flattens and eliminates the syntactic structure of the program. Forcing the programmer to think about the inductive invariant at the level of this encoding significantly reduces productivity. Second, the inductive invariant is likely to have as much case analysis as the encoded transition relation, making it even more tedious and unproductive for the programmer to specify it. For example, the inductive invariant for our example program is larger than its transition relation. This trivial parallel increment program is just the tip of the iceberg; the task of specification and verification explodes in complexity if we turn our attention to realistic implementations of large concurrent systems.

There are two broad approaches to the problem of inductive invariants for concurrent systems. One approach is automatic generation of inductive invariants [1], [2], [3] eliminating the need to specify them manually. Another approach is to specify them via annotations on the structured program itself [4], [5] reducing the cognitive burden on the programmer. Civil falls into this latter class of techniques; its contribution is to allow more proofs to be expressed on the structured program.

Civil proposes an alternative proof strategy which encourages the programmer to think in terms of a sequence of program versions that increasingly simplify the original program. Denoting the program in Figure 1 as version 0, we show three progressively simpler versions in Figure 3.

The simplification from version 0 to version 1 is based on mover types [6], [7]. Acquiring of lock l is a right mover, release of lock l is a left mover, and accesses to the shared variable x protected by the lock l are left and right movers.

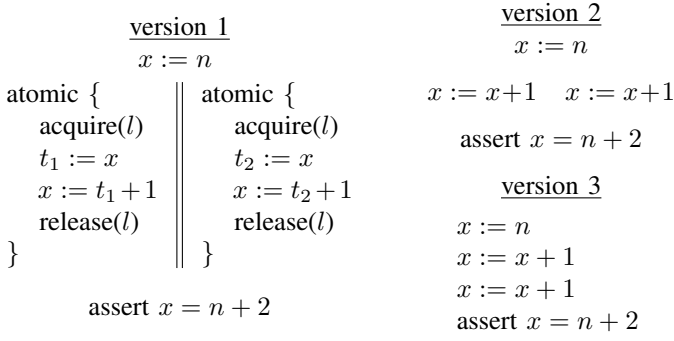


Fig. 3. Simplifying parallel increment.

Consequently, the code fragment executed by each child thread can be treated as an atomic block which executes in one step.

The simplification from version 1 to version 2 summarizes each atomic block with an atomic increment of x , while hiding global variable l and local variables t_1 and t_2 . This summarization is possible because each atomic block leaves the value of l unchanged.

Finally, the simplification from version 2 to version 3 applies mover types again. Since each atomic increment is both a left and right mover, the two parallel increments can be converted into a sequence of two increments. Version 3 can be verified trivially by constructing a sequential verification condition and using an SMT solver to discharge it.

There are several advantages of the Civl approach. First, the transition relation of the program is never exposed to the programmer who specifies program versions using the familiar syntax of structured concurrent programs. Second, although an invariant may be needed to justify a program transformation in general, each invariant is simpler because it justifies only one transformation. Finally, invariants, even when they are needed, are supplied by annotating the structured program itself.

Section II presents a high-level overview of layered refinement, the collection of techniques underlying the Civl approach. Taken together, these techniques increase proof productivity by allowing the correctness argument to be expressed as a single layered concurrent program [8]. This section is targeted to an expert in the theory of concurrency verification and may be skipped on a first reading of the paper. Section III presents the modeling and specification features available to a Civl user through concrete examples.

Since the first published description of Civl [9], we have reimplemented the verifier completely. Section IV describes the current architecture of the Civl implementation as a conservative extension of the Boogie verifier.

The main contribution of Civl is a methodology supported by automated reasoning for implementing verified concurrent systems. We present two arguments that Civl improves the state of the art in constructing verified programs. First, Civl clearly allows new proofs of concurrent systems to be expressed. Second, these proofs have been accomplished on many programs by many researchers including several who were not involved in the design and implementation of Civl. Section V presents this accumulated experience.

II. LAYERED REFINEMENT

Civil advocates layered refinement over structured concurrent programs. Instead of proving the safety of a program in one shot, the new approach allows the programmer to specify a chain of increasingly simpler programs starting from the original program. Each link of the chain, from program P to program Q, represents a single simplification that may be viewed as an abstraction from P to Q or a refinement from Q to P. The correctness of the program is established piecemeal by focusing on the simpler invariant required for each refinement step separately. Most importantly, all the layers and the supporting invariants are specified as a structured and layered concurrent program [8], thus hiding the low-level transition relation from the programmer.

Layered concurrent programs introduce a succinct presentation for multi-layer refinement proofs, which offer two major advantages for interactive proof construction. First, through a syntax for expressing “data layering” (i.e., which variables live on which layers) and “control layering” (i.e., which operations live on which layers), it is easy for the user to write, refine, and maintain a proof outline. Second, a layered concurrent program expresses only the changes in the program from one layer to the next. Thus, layered concurrent programs can result in much smaller proofs, especially for large programs.

While traditional approaches view refinement as a mechanism to specify behavior of concurrent programs, Civl views refinement as a tactic to simplify verification of safety properties. Consequently, the simulation relation justifying the refinement step in Civl is computed but never revealed to the programmer who focuses only on the program layers and the connecting invariants. The viability of the layered refinement approach depends on the existence of program simplification tactics that are easy to use by the programmer and whose justification can be checked automatically. Civl incorporates a number of such tactics described below.

Creating atomic blocks. The Civl programming model comprises concurrently-executing and dynamically-created tasks operating over global memory, each access to which must be encapsulated inside an indivisible atomic action. Global variables model either shared memory or communication channels. Civl uses a theory of commutative atomic actions [6], [7] to create sequential code blocks that appear to execute atomically, despite accesses to global state by multiple atomic actions in the code block.

Creating atomic actions. An atomic code block might be internally complex, due to sequencing, branching, looping, and recursion. Civl summarizes such a code block with an atomic action that hides all the internal details in favor of a declarative specification. Thus, atomic actions in Civl are used to model both low-level execution primitives and high-level summary specifications. To support such diverse usage, an atomic action in Civl generalizes a guarded command [10] to include a specification of failure [11] (in addition to blocking or successful execution) and the creation of asynchronous activity in the form of pending asyncs [12].

Synchronizing asynchrony. Civl supports elimination of pending asyncs from the atomic actions in a program via a tactic known as inductive sequentialization [13]. Introduction and elimination of pending asyncs in atomic actions together enable a program simplification that provides the appearance of executing in one step a collection of atomic computations executing asynchronously. This tactic amplifies the use of commutative atomic actions to allow summarization of both synchronous and asynchronous computation.

Civl allows introduction and hiding of global and local variables to change the state representation of the program. This change often results in a program whose atomic actions become commutative and thus the other tactics mentioned above become applicable. Variable introduction is performed as part of the tactic that creates atomic blocks; calls to special atomic actions assign meaning to the introduced variables. Variable hiding is performed as part of the tactic that creates atomic actions from atomic blocks; the created atomic action does not refer to the hidden variables.

Variable introduction and hiding in Civl has two other benefits. First, variable introduction naturally allows the user to introduce an arbitrary safety specification for the program. Second, it becomes unnecessary to support the notion of ghost state present in most provers for concurrent programs. Changing the state representation of the program often addresses the need for ghost state. Also, a variable may be introduced and hidden at the same layer for those special cases when ghost state is needed purely for invariant specification.

The tactic that creates atomic actions often needs constraints on the reachable states of the program. These constraints are supplied via yield invariants [14] which are named and parameterized invariants that can be reused and suitably instantiated across multiple program locations where interference may happen. Yield invariants combine the precision and flexibility of location invariants [4] with the compactness and modularity of rely-guarantee specifications [5]. Civl supports local reasoning with permissions that are redistributed by atomic actions and otherwise passed around the program without duplication [14]. Permissions are useful in proving locally both that yield invariants are interference-free and that atomic actions satisfy desired commutativity properties.

Civl supports the verification of arbitrary safety properties. Civl’s notion of correctness is that the lowest-layer program is free of assertion failures. Arbitrary safety properties are expressible as assertions because auxiliary state (e.g., history variables) can be introduced into the program in addition to program state.

The client of a system constructed with layered refinement only needs to check that the established high-level specification captures the desired property. The details of a layered proof are not trusted since they are checked by Civl. However, the introduction of auxiliary state into the system at the lowest layer, sometimes needed to express a specification, is trusted.

III. PROGRAMMING AND PROVING IN CIVL

In this section we illustrate the input language and the verification features of Civl. The presentation is necessarily brief and selective. Detailed documentation is available at our website civl-verifier.github.io.

Syntax. Civl is built on top of Boogie [15], a language and verifier for sequential programs. Boogie provides standard features for imperative programming such as assignments, sequencing, branching, looping, and procedures. Additionally, it provides specification features such as assert and assume statements, loop invariants, preconditions, postconditions, and axioms. The expression language of Boogie is first-order logic with built-in theories such as uninterpreted functions, integers, bitvectors, datatypes, and arrays. Civl adds the keywords `async` (asynchronous procedure call), `par` (parallel procedure call), and `yield` (yield point) to express concurrent behaviors. All other syntactic extensions are implemented using generic *attributes* which attach to abstract syntax tree nodes of a Boogie program. Attributes are of the form `{:attr e1, e2, ...}`, where `attr` is the attribute name and `e1, e2, ...` are parameter expressions of the attribute.

Atomic actions. Every access to a global variable has to be encapsulated into an atomic action. An atomic action consists of a *gate*, a one-state predicate that specifies the condition under which the action can execute or otherwise fail, and a *transition relation*, a two-state predicate that specifies the possible state updates of the action. Atomic actions are capable of specifying uniformly both low-level operations (like writing to a memory location or sending a message on a channel) and high-level operations (like acquiring a lock or reaching consensus in a distributed system). For example, the left column in Figure 4 shows atomic actions which acquire and release a lock, modeled by the global variable `l`. The Boogie procedures are identified as atomic actions by the `:right/:left` annotations which also declare their mover types; actions that are non-movers are annotated with `:atomic`. The action `AcquireSpec` blocks until `l` equals `None()` (denoting the availability of the lock) and then updates `l` to `Some(tid)` (denoting that the lock is held by the current thread with thread id `tid`). Conversely, `ReleaseSpec` asserts that the current thread holds the lock (the `assert` statement specifies the gate) and updates `l` to `None()`.

Program layers. In a Civl proof, the user explicitly organizes the program into layers using *layer annotations*. Variables and atomic actions have a *layer range*. In Figure 4, variable `l` is introduced at layer 1 and hidden at layer 2, and action `AcquireSpec` only exists at layer 2.

Concurrent computations are expressed by *yielding procedures*. The yielding procedure `Acquire` in Figure 4 acquires a lock by repeatedly invoking the compare-and-swap operation `CAS_b` to atomically set the global Boolean variable `b` from `false` to `true`. A yielding procedure is subject to interference from other concurrent threads at any point during its execution. However, `Acquire` is declared to *refine* the atomic action `AcquireSpec` at layer 1. This means that Civl checks that

```

var {:layer 1,2} l: Option Tid;
procedure {:right} {:layer 2,2}
AcquireSpec{:linear "tid"} tid: Tid)
modifies l;
{
  assume l == None();
  l := Some(tid);
}
procedure {:left} {:layer 2,2}
ReleaseSpec{:linear "tid"} tid: Tid)
modifies l;
{
  assert l == Some(tid);
  l := None();
}

var {:layer 0,1} b: bool;
procedure {:yields} {:layer 1}
{:refines "AcquireSpec"}
{:yield_preserves "LockInv"}
Acquire{:layer 1}{:linear "tid"} tid: Tid)
{
  var t: bool;
  while (true)
    invariant {:layer 1}{:yields}
      {:yield_loop "LockInv"} true;
  {
    call t := CAS_b(false, true);
    if (t) {
      call set_l(Some(tid));
      break;
    }
  }
}

procedure {:intro} {:layer 1}
set_l(v: Option Tid)
modifies l;
{ l := v; }
procedure {:yields} {:layer 2}
{:refines "ClientSpec"}
{:yield_preserves "LockInv"}
Client{:layer 1,2} {:hide}
{:linear "tid"} tid: Tid)
{
  call Acquire(tid);
  ...
  call Release(tid);
}
procedure {:atomic} {:layer 3,3}
ClientSpec()
{ ... }

```

Fig. 4. A layered program, showing a lock implementation and its client. Left: Atomic actions for acquiring and releasing a lock. Middle: A spinlock implementation that refines the atomic action specification. Right: Introduction action for proving the lock refinement and a client of the lock.

Acquire “behaves like” `AcquireSpec`, and thus clients of the former can ignore the details of its implementation and instead reason with the atomic behavior of the latter. `Acquire` uses the global Boolean variable `b`, while `AcquireSpec` uses the global lock variable `l`. The connection between these two different representations is established by the *introduction action* `set_l`, which sets `l` from `None()` to `Some(tid)` when `b` is set from `false` to `true`. Finally, the yielding procedure `Client` protects a critical section with calls to `Acquire` and `Release` and declares that it refines the action `ClientSpec` at layer 2.

The layer annotation of a yielding procedure denotes its *disappearing layer*. The procedure exists (with changing bodies) on all layers below and up to its disappearing layer. For example, `Acquire` exists on layer 0 and 1, and `Client` exists on layer 0, 1, and 2. Intuitively, a procedure is replaced with its refined atomic action above its disappearing layer.

Figure 4 encodes four program layers. Layer 0 is the most concrete program. It contains procedure `Client` which calls procedure `Acquire`, and `Acquire` implements a spinlock using calls to `CAS_b`; `b` is the only global variable, and `Client` and `Acquire` have no input parameters. Layer 1 introduces the global variable `l` and the local input parameters `tid`, along with the introduction action `set_l` (the call to `set_l` does not exist at layer 0). At layer 2, `Acquire` is gone and the body of `Client` is rewritten to make calls to the actions `AcquireSpec` and `ReleaseSpec`; `b` is hidden and `l` is the only global variable. At layer 3, `Client` is also gone, and any potential calls to `Client` are replaced by its atomic summary `ClientSpec`; global variable `l` and the parameter `tid` do not exist anymore.

Layering provides a form of modularity. At layer 2 we do not care about how the lock is implemented, and at layer 3 we do not care that a lock was used at all. The applied proof tactics (variable introduction, variable hiding, and atomic blocks) simplify the necessary invariants on every layer.

Yield sufficiency. *Civil* partitions the bodies of yielding procedures into *yield-to-yield fragments*. The following code locations are *yield points*: procedure entry and exit, loop head-

ers annotated with `{:yields}`, and explicit `yield` statements. Context switches are only considered at yield points, and the code between two yield points is a *yield-to-yield fragment*. At layer 1, in `Acquire` every loop iteration (i.e., call to `CAS_b`) is a *yield-to-yield fragment*, and in `Client` there is a *yield* before and after every call. At layer 2, something interesting happens. The body of `Client` does not call any procedures anymore (the calls are to atomic actions now), and thus `Client` has only a single *yield-to-yield fragment*. *Civil* justifies this simplification using *reduction* [6], [7]. Concretely, using the fact that `AcquireSpec` is a *right mover* and `ReleaseSpec` is a *left mover*. In general, every *yield-to-yield fragment* is checked to be a sequence of right movers, followed by at most one non-mover, followed by a sequence of left movers.

Refinement. To justify the summarization of a yielding procedure at layer n by an atomic action, *Civil* checks that in every execution of the procedure, the effect of the refined action happens in exactly one *yield-to-yield fragment* and that other *yield-to-yield fragments* leave the layer- $(n + 1)$ state unchanged. In `Acquire`, every loop iteration where `CAS_b` fails leaves `l` unchanged, while the (final) iteration where `CAS_b` succeeds also updates `l` to `Some(tid)` and thus produces the effect of `AcquireSpec`.

Invariants. *Civil* performs refinement checking modularly, by considering every *yield-to-yield fragment* in isolation. This usually requires certain properties to hold at yield points, notwithstanding any interference from other concurrent threads. *Civil* supports location invariants [4] and yield invariants [14], which are checked to be interference-free across all *yield-to-yield fragments* in the program. Yield invariants are named and parameterized invariants that can be reused and suitably instantiated across multiple yield points. The following code shows the yield invariant `LockInv`.

```

procedure {:yield_invariant} {:layer 1} LockInv();
requires b <=> (l != None());

```

In `Acquire` (Figure 4), `LockInv` is attached to the procedure entry and exit using the `:yield_preserves` annotation, and to the loop header using the `:yield_loop` annotation. We give examples of parameterized yield invariants below.

Permissions. Certain invariants, like those connecting local variables from different scopes, can be tedious to express and propagate. Civl addresses this problem using *linear permissions*. Program variables can be declared as *linear*, from which Civl calculates the *available* variables at every control location, assigns every available variable a set of *permissions*, and ensures that there is no duplication across these permission sets. Civl allows the user to customize the type of permissions and the assignment of permissions to variables.

The lock specification in Figure 4 uses linearity to express unique thread identifiers. The type declaration

```
type {:linear "tid"} Tid;
```

specifies the permissions for the *linear domain* `tid` to be of type `Tid`, the type of thread identifiers. This means that every variable that is linear under domain `tid` gets assigned a set of `Tid` values. The assignment is specified using *collector* functions. Civl uses the following default collector in the absence of a user-specified collector.

```
function {:linear "tid"} TidCol(x: Tid) : [Tid]bool
{ MapConst(false)[x := true] }
```

We use a map from `Tid` to `bool` to model a set. The polymorphic map constructor `MapConst` applied to `false` returns a map set to `false` everywhere representing an empty set. `TidCol` assigns linear variables of type `Tid` (like the input parameter `tid` of `AcquireSpec` and `ReleaseSpec`) the single value the variable contains as its permission. Consider an instance of `AcquireSpec` and an instance of `ReleaseSpec` with parameters `tid1` and `tid2`, respectively. By linearity, Civl gets to assume that the multiset $\text{TidCol}(\text{tid1}) \uplus \text{TidCol}(\text{tid2}) = \{\text{tid1}, \text{tid2}\}$ does not contain any duplicates, which implies $\text{tid1} \neq \text{tid2}$. This assumption is used to show that the `AcquireSpec` instance commutes to the right of the `ReleaseSpec` instance, an important part of the proof that `AcquireSpec` and `ReleaseSpec` satisfy their mover types.

Figure 5 presents an example inspired by barrier synchronization to demonstrate how permissions are useful in proving invariants. The program has two global variables, `barrier` and `count`, to represent the set of identifiers inside the barrier and the number of threads outside the barrier, respectively. The atomic actions `EnterBarrier` and `ExitBarrier` encode entering and exiting the barrier by a thread, respectively. The yield invariant `ThreadInv` is parameterized by a thread identifier `j` and indicates that `j` is in the barrier. Typically, a thread with identifier `i` would enter the barrier by calling `EnterBarrier(i)`, yield to other threads by calling `ThreadInv(i)`, and then exit the barrier by calling `ExitBarrier(i)`. The linearity of parameter `j` of `ThreadInv` and parameter `i` of `ExitBarrier` allows us to assume that `j` and `i` are distinct, and therefore `ThreadInv` is preserved by `ExitBarrier`. Preservation by `EnterBarrier` is trivial since this action only adds elements to `barrier`.

Permission redistribution. Now consider the following yield invariant `BarrierInv` that indicates that the sum of the size of `barrier` and `count` is equal to `N`, the total number of threads.

```
var {:layer 0,1} barrier: [Tid]bool;
var {:layer 0,1} count: int;

procedure {:atomic} {:layer 1} EnterBarrier(
  {:linear "tid"} i: Tid)
modifies barrier;
{
  barrier[i] := true;
  count := count - 1;
}

procedure {:atomic} {:layer 1} ExitBarrier(
  {:linear "tid"} i: Tid)
modifies barrier;
{
  assert barrier[i];
  barrier[i] := false;
  count := count + 1;
}

procedure {:yield_invariant} {:layer 1} ThreadInv(
  {:linear "tid"} j: Tid);
requires barrier[j];
```

Fig. 5. Using permissions to prove invariants.

```
procedure {:yield_invariant} {:layer 1} BarrierInv();
requires Size(barrier) + count == N;
```

This invariant cannot be proved on the code in Figure 5. The action `EnterBarrier` does not preserve `BarrierInv` whenever `barrier[i]` already holds upon entry. This condition, of course, cannot happen in the program, since a thread only calls `EnterBarrier` when it is outside the barrier. But this constraint is not encoded in the current specification. An attempt to encode this constraint would be to make the global variable `barrier` linear. However, this strategy would force us to drop the linear annotation on parameter `i` of `ExitBarrier` which would then make `ThreadInv` unprovable.

To solve this programming problem, we present a more sophisticated use of permissions that depends on custom collectors and new linearity annotations on local variables. The datatype declaration

```
type {:linear "perm"} {:datatype} Perm;
function {:constructor} Left(i: Tid): Perm;
function {:constructor} Right(i: Tid): Perm;
```

specifies the permissions for a new linear domain `perm`. The datatype `Perm` has two constructors `Left` and `Right`; each constructor wraps a thread identifier to create a `Perm` value. The collectors for `perm` are shown below.

```
function {:linear "perm"} TidCol(x: Tid) : [Perm]bool
{ MapConst(false)[Left(x) := true][Right(x) := true] }

function {:linear "perm"} TidSetCol(xs: [Tid]bool)
: [Perm]bool
{ (lambda p: Perm :: is#Left(p) && xs[i#Left(p)]) }
```

The collector `TidCol` defines the permissions stored in a single thread identifier `x` as the set comprising `Left(x)` and `Right(x)`. The collector `TidSetCol` collects the permissions in a set of thread identifiers `xs` by collecting `Left(x)` for each element `x` in `xs`. Additionally, there is the following default collector for type `Perm`.

```
function {:linear "perm"} PermCol(x: Perm) : [Perm]bool
{ MapConst(false)[x := true] }
```

Figure 6 shows the revised code for our example which now uses the linear domain `perm` throughout. The global

```

var {:layer 0,1} {:linear "perm"} barrier: [Tid]bool;
var {:layer 0,1} count: int;

procedure {:atomic} {:layer 1} EnterBarrier(
  {:linear_in "perm"} i: Tid)
returns ({:linear "perm"} p: Perm)
modifies barrier;
{
  barrier[i] := true;
  count := count - 1;
  p := Right(i);
}

procedure {:atomic} {:layer 1} ExitBarrier(
  {:linear_in "perm"} p: Perm, {:linear_out "perm"} i: Tid)
modifies barrier;
{
  assert p == Right(i) && barrier[i];
  barrier[i] := false;
  count := count + 1;
}

procedure {:yield_invariant} {:layer 1} ThreadInv(
  {:linear "perm"} p: Perm, j: Tid);
requires p == Right(j) && barrier[j];

```

Fig. 6. Permission redistribution in atomic actions.

variable `barrier` is linear and consequently a store of permissions. The signatures and implementation of `EnterBarrier`, `ExitBarrier`, and `ThreadInv` have also changed.

We now present the intuition behind the revised implementation. `EnterBarrier` splits the permissions $\{\text{Left}(i), \text{Right}(i)\}$ contained in its input parameter i into $\text{Left}(i)$ which is put into `barrier` and $\text{Right}(i)$ which is returned via the output parameter p . The `linear_in` annotation on i indicates that the permissions in i are consumed by the call and are therefore unavailable afterwards. The permission p and the unavailable thread identifier i are used to call `ThreadInv`. Finally, when `ExitBarrier` is called with p and i and i is removed from `barrier`, the permission $\text{Left}(i)$ is also removed from `barrier`. This permission becomes available to be joined with $\text{Right}(i)$ contained in p so that the full permission set $\{\text{Left}(i), \text{Right}(i)\}$ is put into i which becomes available after the call. This protocol is indicated by the `linear_in` annotation on p and the `linear_out` annotation on i .

This example shows that permissions can be redistributed without duplication by an atomic action among global variables and its parameters. This ability to soundly redistribute permissions allows us to compactly express and prove coordination protocols.

Asynchrony. Asynchronous invocations—calls that create a new concurrent thread of computation without the caller waiting for the operation to complete—are challenging to specify and verify. `Civl` provides the inductive sequentialization [13] proof rule to sidestep the arduous task of inventing complex inductive invariants that capture all possible interleavings of an asynchronous program.

Consider the action `ASYNC_SUM` in Figure 7. It uses an output variable `PAs` that represents *pending asyncs*, asynchronous operations that are spawned by `ASYNC_SUM` but executed asynchronously at some later time. Concretely, `ASYNC_SUM` creates the multiset of pending asyncs $\text{set_of_ADD}(1, n) = \{\text{ADD}(1), \text{ADD}(2), \dots, \text{ADD}(n)\}$, which could be refined to a

```

procedure {:atomic}{:layer 1}{:IS "SUM","INV"}{:elim "ADD"}
ASYNC_SUM (n: int)
returns ({:pending_async "ADD"} PAs:[PA]int)
modifies x;
{
  assert n >= 0;
  PAs := set_of_ADD(1, n);
}

procedure {:atomic}{:layer 2} SUM (n: int)
modifies x;
{
  assert n >= 0;
  x := x + (n * (n+1)) div 2;
}

procedure {:left}{:layer 1} ADD (i: int)
modifies x;
{ x := x + i; }

procedure {:IS_invariant}{:layer 1} INV (n: int)
returns ({:pending_async "ADD"} PAs:[PA]int,
  {:choice} choice:PA)
modifies x;
{
  var i: int;
  assert n >= 0;
  assume 0 <= i && i <= n;
  x := x + (i * (i+1)) div 2;
  PAs := set_of_ADD(i+1, n);
  choice := ADD(i+1);
}

```

Fig. 7. Sequentialization of the asynchronously computed sum from 1 to n . We are omitting annotations that support automated reasoning with quantifiers.

procedure that asynchronously invokes `ADD` in a `while` loop.

The annotations on `ASYNC_SUM` tell `Civl` instead to convert it into `SUM`, by *eliminating* from it the pending asyncs to `ADD` using the *invariant action* `INV`. `SUM` adds to x the value $\frac{n(n+1)}{2}$, which is the cumulative effect of the asynchronous `ADD` operations. The key is that `INV` only talks about a single interleaving of the `ADD` operations: `ADD(1); ADD(2); ...; ADD(n)`. It represents any prefix of this single interleaving as follows. It (1) nondeterministically picks i between 0 and n denoting the number of finished `ADD`'s, (2) increases x by $\frac{i(i+1)}{2}$ to capture the effect of executing `ADD(1)` to `ADD(i)`, (3) creates pending asyncs for `ADD(i+1)` to `ADD(n)`, and (4) specifies that the next pending async we wish to execute in our sequential order is `ADD(i+1)`. `INV` represents `ASYNC_SUM` with $i = 0$, `SUM` with $i = n$, and the induction order from i to $i+1$ is specified by the user through the output variable `choice`. The justification for this sequential reduction is that `ADD` is a left mover, and thus can always be commuted to the desired location in the sequentialization.

IV. IMPLEMENTATION

`Civl` is implemented as a conservative extension of the Boogie verifier. The extensions to the syntax (Section III) and the verification engine do not affect ordinary Boogie programs. The Boogie verifier itself is implemented as a pipeline with a sequence of phases—parsing, type checking, verification condition generation, solver invocation, and error reporting. For every procedure, a verification condition in SMT-LIB format is passed to an SMT solver running in a separate process. If an error is discovered, a diagnostic error trace is calculated by examining the model returned by the solver.

The implementation of *Civil* adds two more phases into the pipeline of the Boogie verifier. Initially, the *Civil* attributes are parsed together with the rest of the Boogie program and the standard Boogie type checker is run. Then, the *Civil type checker* validates the *Civil* attributes and converts them into internal data structures. Next, the *Civil processor* compiles all proof obligations related to concurrency down to sequential Boogie procedures. Finally, the existing Boogie pipeline for converting procedures into verification conditions takes over.

Civil type checker. The type checker has three main parts.

First, a *layer analysis* [8] checks that the layer annotations are consistent. This analysis ensures that all program layers encoded by the input layered program are well-formed, e.g., that variables accessed and procedures/actions called on some layer actually exist on that layer. It also ensures the soundness of our refinement check. For example, in Figure 4 we could not refine *Client* at layer 1, because its callee *Acquire* first needs to be converted to the action *AcquireSpec*, which happens from layer 1 to layer 2. For sound variable introduction, only introduction actions and invariants are allowed to access global variables at their introduction layer. For example, at layer 1 only *set_l* and *LockInv* refer to *l*, whereas *AcquireSpec* only refers to it at layer 2.

Second, a *yield sufficiency analysis* [7] checks, for each layer separately, that it is safe to consider context switches only at yield points. This check is implemented by computing a simulation relation [16] between a labeled control-flow graph and a specification automaton that encodes all sequences of mover types allowed by Lipton’s reduction theorem [6]. The specification automaton is shown in panel ① of Figure 8. Panel ② shows the labeled graph for procedure *Acquire* at layer 1. Node n_0 represents the loop head. Since the loop is yielding, the edge to the loop condition n_1 is labeled Y. At n_1 we either exit the loop and thus the entire procedure on the private edge to n_3 , or we execute the non-mover *CAS_b* on the edge to n_2 labeled N. At n_2 , corresponding to the if condition, we either execute the introduction action *set_l* and break from the loop, or we loop back to the loop head n_0 , both of which are private edges. Panels ③ and ④ show that the calls to the yielding procedures *Acquire* and *Release* are labeled with Y at layer 1 but with the mover type of their respective refined atomic action at layer 2. For simplicity, *Civil* does not allow a yield-to-yield fragment that starts within a loop to wrap around the loop head, and thus checks that every loop that contains a Y edge is a yielding loop.

Third, a *linear flow analysis* [14] computes the available linear variables at each control location of a procedure, and ensures that calls to procedures, atomic actions, and yield invariants satisfy their linear interfaces. The following code snippet refers to Figure 6.

```
// i available, p unavailable
call p := EnterBarrier(i);
// i unavailable, p available
call ThreadInv(p, i);
// i unavailable, p available
call ExitBarrier(p, i);
// i available, p unavailable
```

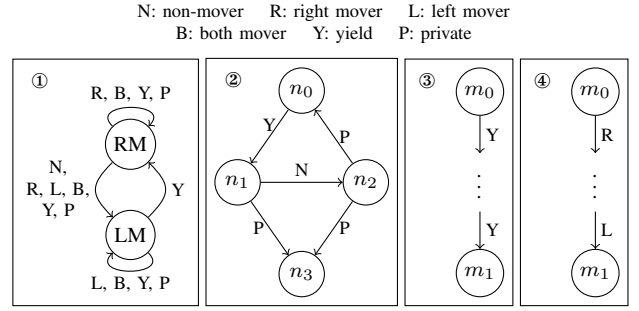


Fig. 8. Labeled control-flow graphs for yield sufficiency analysis of Figure 4. ① Specification automaton. ② Acquire at layer 1. ③ Client at layer 1. ④ Client at layer 2.

EnterBarrier requires *i* to be available and consumes it, making *p* available in return. The unavailable *i* can be used in places where it is not required to be linear, in particular the calls to *ThreadInv* and *ExitBarrier*. After *ExitBarrier* which consumes *p*, variable *i* is available again.

Civil processor. To target Boogie’s verification-condition generator, *Civil* eliminates layers, concurrency, and linearity from the input layered concurrent program by creating a collection of sequential *checker procedures*. There are two advantages to this approach. First, modular decomposition into checker procedures improves scalability by creating small verification problems. Second, verification failures in checker procedures are processed to create targeted error messages. In the following we explain the categories of checker procedures *Civil* generates. We do not have the space to present detailed encodings; we suggest that interested readers use the command-line flag `-civilDesugaredFile` to inspect the plain Boogie program generated by the *Civil* processor.

A common functionality required by multiple checker procedures is the computation of a logical transition relation from the code representation of an atomic action. For each code path, *Civil* computes a path constraint from its static single assignment form, and then iteratively eliminates intermediate copies of variables by finding and inlining definitions. Variables that cannot be eliminated are existentially quantified. The transition relation is the disjunction over all path formulas.

Permission redistribution among linear variables occurs through assignment, parameter passing, and mutation in atomic actions. The first two sources of redistribution are tracked by the syntactic flow analysis in the *Civil* type checker. For the third source, a checker procedure for each atomic action ensures that no permission duplication occurs due to its execution. This semantic check involves user-supplied collector functions. For example, the checker procedure for *ExitBarrier* from Figure 6 validates the postcondition

$$\text{TidSetCol}(\text{barrier}) \uplus \text{TidCol}(i) \subseteq \text{TidSetCol}(\text{old}(\text{barrier})) \uplus \text{PermCol}(\text{old}(p)),$$

stating that the permissions flowing into the action through *barrier* and *p* must be a subset of the permissions flowing out through *barrier* and *i*. The resulting non-duplication guarantee among linear variables is injected into all the following checks as a free assumption.

```

procedure CommutativityChecker(tid_1: Tid, tid_2: Tid)
requires tid_1 != tid_2; // derived from linearity
requires l == Some(tid_2); // gate of ReleaseSpec
modifies b, l;
{
  call AcquireSpec(tid_1); // inlined
  call ReleaseSpec(tid_2); // inlined
  // trans. rel. of ReleaseSpec(tid_2); AcquireSpec(tid_1)
  assert l == Some(tid_1);
}

```

Fig. 9. Commutativity checker for AcquireSpec and ReleaseSpec.

The mover type of each atomic action is verified by pairwise checks against every atomic action with an overlapping layer range. Each such check is encoded by multiple checker procedures to account for commutativity of both failing and successful behaviors. For example, the commutativity check between AcquireSpec and ReleaseSpec is shown in Figure 9. Recall that this check succeeds because the first call blocks due to the constraint we get from linearity. In addition, each left mover and introduction action is separately checked to have a failing or successful behavior from each initial state.

Invariants are verified separately for each layer n , resulting in a checker procedure for each yielding procedure with disappearing layer at least n . Civl constructs the checker procedure from the code of the yielding procedure as follows. First, calls to invariants and introduction actions at layers other than n are dropped and calls to yielding procedures with disappearing layers lower than n are rewritten to calls of their respective refined actions. Next, asynchronous and parallel calls (of which ordinary calls are a special case) are translated. An asynchronous call to a yielding procedure is translated into an assertion of the precondition of the procedure. An asynchronous call to an action is either synchronized or converted into a pending async [12]. A parallel call may contain arms that are actions, yield invariants, or yielding procedures. Each such call is rewritten into a sequence comprising calls to actions and parallel calls whose arms are either yield invariants or yielding procedures. For example, $\text{par } A \mid P \mid I \mid B \mid C \mid Q \mid D$ with actions A, B, C and D , procedures P and Q , and invariant I , is rewritten to $\text{call } A; \text{par } P \mid I; \text{call } B; \text{call } C; \text{par } Q; \text{call } D$. All calls to atomic actions are inlined. Any parallel call remaining at this point is a yield where interference is possible. Next, each yield is instrumented to record a snapshot of the global variables immediately after the yield. This snapshot is used to assert the preservation of all invariants in the program at the end of a yield-to-yield fragment. Finally, each parallel call (with arms that are yielding procedures or yield invariants) comprising a yield is itself desugared as follows: (1) assert preconditions of yielding procedures and yield invariants, (2) havoc all global variables, (3) assume postconditions of yielding procedures and yield invariants. The soundness of this translation of concurrent code to sequential code is ensured by the yield sufficiency analysis of the Civl type checker. A side condition for asynchronous calls forbids global state updates between an asynchronous call to a yielding procedure and the next yield

point. Additionally, there are restrictions on the sequence of arms in a parallel call. For example, any left mover must occur before any right mover, and there cannot be both a yielding procedure and a non-mover in the sequence.

At the disappearing layer n of every yielding procedure, a checker procedure verifies refinement of the specified atomic action by tracking two local Boolean variables, pc and ok , each initialized to false. The variable pc is set to true as soon as a yield-to-yield fragment modifies any layer- $(n+1)$ state; before any such modification it is asserted that pc is false. The variable ok is set to true as soon as a yield-to-yield fragment modifies the layer- $(n+1)$ state according to a transition admitted by the refined action; ok is asserted to be true when the procedure returns. Overall, we check that layer- $(n+1)$ state is modified at most once, and that a behavior of the refined action occurs at least once.

Each invocation of the inductive sequentialization [13] rule results in a collection of checker procedures, one each for the base and conclusion case and one for the inductive step corresponding to each eliminated pending async.

V. EXPERIENCE

Civl has been used in many efforts to develop verified concurrent systems, both by the authors of Civl and by other researchers. These efforts include a concurrent garbage collector [9], a Paxos implementation [13], and implementations of concurrent data structures: the FastTrack data-race detector [17], Chase-Lev deque [18], and Java weakly-consistent objects [19]. Civl has also been used to prototype techniques for verification under TSO semantics [20]. Civl is fast enough to be used for interactive development. Even on our large benchmarks, verification time is a few seconds.

Our experience suggests that Civl’s specification mechanisms—layering, commutativity, yield invariants—are natural for users. These features aid discovery of provable implementations by encouraging the user to think about different layers of abstraction, the primitives for each layer, and suitable organization of the reasoning technique at each layer. In addition, layers enable partitioning of work among multiple developers each working on the proof of a particular layer with agreed-upon interfaces between layers.

We present more details about two major case studies to provide anecdotal evidence for the improvements in developing verified concurrent systems enabled by Civl.

Concurrent Garbage Collector. An author of this paper together with other researchers used Civl to develop a verified concurrent garbage collector and object allocator that improves upon the mark-and-sweep garbage collector by Dijkstra et al. [21] in two ways. First, the new collector supports more than one mutator running in parallel with the collector. Second, it requires a write-barrier only on updates of heap pointers but not on root modifications. The Civl implementation is realistic, given in terms of individual CPU operations. The refined specification comprises high-level atomic actions for object allocation and access, that provide the illusion of unbounded memory in which individual objects are not reused.

The proof is done via a sequence of 6 program transformations connecting 7 program layers. Layer 0 is described in terms of individual atomic CPU operations. Layer 0 \rightarrow 1 introduces locks and atomic actions for read/write accesses. Layer 1 \rightarrow 2 uses the locks and protected accesses to construct higher-level atomic operations that are used in the barrier synchronization algorithm for root scanning and in the mark-sweep algorithm. The collector operates in three phases—idle, mark, and sweep. Layer 2 \rightarrow 3 reasons about the coordination between the collector and the mutators to make phase changes safely. The mark algorithm performs a depth-first search of the heap starting from the roots. The stack in this search comprises “gray” objects. Layer 3 \rightarrow 4 changes the representation of the gray objects to a set. Layer 4 \rightarrow 5 reasons about the root scanning algorithm that internally uses barrier synchronization to create an atomic action that scans all roots in one step. Reasoning about the write barrier also happens during this transformation. Layer 5 \rightarrow 6 reasons about the mark-sweep algorithm using the atomic actions for scanning roots, maintaining the set of gray objects, and changing object colors. The garbage collector is hidden entirely, leaving the client with atomic actions for allocating objects, reading and writing object fields, and checking object equality.

This proof was constructed and reported in 2015 [9]. Since then, Civl has been rewritten but the proof has been maintained and improved. The current artifact is 2031 LOC and verifies in 25s on a standard Mac. The biggest improvement happened with the introduction of yield invariants [14] which reduced the verification time by a factor of 10.

Paxos. The Paxos protocol [22] establishes consensus among a set of unreliable nodes in an asynchronous network without a central coordinator. This protocol lies at the core of any system with replicated state. It is difficult to both understand and implement. The authors of this paper together with other researchers constructed a verified implementation [13] of single-decree Paxos, which establishes consensus on a single value. The verified implementation only uses primitive atomic actions, like reading or writing a single memory address, and sending or receiving a single message.

The proof is constructed via a sequence of 2 program transformations done over 3 layers. Layer 0 implements event handlers using primitive atomic actions for sending and receiving network messages, and for updates to the local state and decision variable at each Paxos node. The transformation from layer 0 to layer 1 converts event handlers to atomic actions at the granularity typically used to describe protocols in papers. At the same time, this transformation changes the state representation to make it easier to apply the next transformation. The invariant justifying this transformation simply connects the two state representations. The transformation from layer 1 to layer 2 uses inductive sequentialization [13] to create a single atomic action where consensus is reached in one step by nondeterministically setting decisions at each node consistently. The invariant justifying this transformation captures the intuition of the protocol. It has 4 conjuncts and

is considerably simpler than the invariants in other published proofs of the Paxos protocol. For example, the proof [23] using Ivy has 5 other supporting invariants in addition to the 4 used in the Civl proof. The current artifact for the Civl proof is 1116 LOC and verifies in 7s on a standard Mac.

VI. RELATED WORK

In this section we compare Civl to other *reusable tools* that have *support for concurrency*.

TLA+ [24] and Event-B [25] are two classic tools for refinement reasoning over transition systems. Ivy [26] verifies transition systems using a restricted modeling and specification language (notably without functions and arbitrary quantification) that makes verification conditions decidable. While Ivy requires manual effort to encode distributed systems concepts in this restricted language, Civl requires manual effort to automate quantifier reasoning. Ivy also has a synchronous, reactive programming language from which it can extract asynchronous, distributed implementations [27]. This programming model, which cannot express fine-grained concurrency, can be encoded in Civl by threading a linear parameter through atomic actions and procedures. Ivy provides liveness reasoning and information hiding via modules.

Iris [28] is a Coq-based formalization of a program logic suitable for reasoning about fine-grained concurrent programs with higher-order ghost state. The focus in Iris is to clarify and simplify concurrent separation logics around a few primitive concepts in order to provide a suitable foundation for developing reasoning mechanisms for concurrent programs. Compared to Iris, Civl is less flexible but provides more automation on a programming notation that supports standard models of concurrent programming. ReLoC [29] is a logic built on top of Iris for interactively proving contextual refinement judgments.

Chalice [30] verifies monitor invariants, in addition to absence of data races and deadlocks, on a small Java-like concurrent programming language. VeriFast [31] supports separation logic specifications, resource invariants, and higher-order ghost state on concurrent C and Java programs. Prusti [32] uses the guarantees of the Rust type system to simplify the manual annotation effort. VerCors [33] builds on separation logic specifications and provides verification features for several concurrent programming idioms, e.g., based on histories and process algebra. VCC [34] is a verifier for concurrent C programs. VCC allows the programmer to construct a custom verification methodology via extensive support for the introduction of ghost types and values. Noninterference is accomplished via a network of type-level global invariants which together must satisfy certain stability and admissibility conditions. Similar to Civl, these tools use SMT solvers as the reasoning engine, exploit programmer interaction, and support modular reasoning. Civl provides features not present in these tools such as layered refinement and yield invariants.

Anchor [35], a successor to Calvin-R [36], is a lightweight verifier for a small Java-like programming language. Anchor allows the programmer to compactly specify conditional mover types for read and write accesses of shared object fields.

It is less modular than Civl and other tools discussed here; inlining is used extensively to deal with procedure calls.

Armada [37] is a language and verifier that implements layers, mover types, and explicit noninterference reasoning. Armada is inspired by Civl but also supports weak memory and extensibility via new simplification tactics. While Civl represents all program layers in a single layered concurrent program, Armada connects explicitly written programs using proof scripts that invoke mechanized theorems.

VII. CONCLUSION

The Civl static verifier aids the development of verified concurrent systems through language-integrated proof structuring mechanisms, an array of program-simplifying proof tactics, and modular and automatable verification conditions. The modeling features provided in Civl are general; they can be specialized to many different domains by building custom linguistic support and automation. For example, it is possible to use Civl as the verification backend for domain-specific languages suitable for developing implementations of distributed protocols, concurrent data structures, or even system-level hardware implementations. Overall, Civl opens many new opportunities in development of programming tools for concurrent systems.






Civl's capabilities to generate verification conditions for checking commutativity, refinement, and noninterference can be leveraged individually by a verifier. It is also conceivable to design a programming language that supports layering and atomic actions natively, and uses Civl as a backend for verification. This language would generate executable code from the lowest-layer program which invokes atomic actions whose implementation is provided by the language runtime.

Our experience suggests that progress on the following important challenges should increase the applicability and usability of Civl. First, Civl's verification conditions have quantifiers which can result in unpredictable verification times. Domain-specific techniques for automatic quantifier instantiation or language mechanisms for conveniently specifying instances would help. Second, Civl supports linear maps [38] for reasoning about disjoint but flat memory. Extension to support reasoning about nested linear maps would make it easier to encode standard heap programming models. Third, layered programs in Civl are challenging to comprehend, edit, and refactor; tools to help with these tasks would be helpful. A module system for factoring out libraries and their layered proofs would aid the development of large verified systems.

REFERENCES

- [1] A. Gupta, C. Popeea, and A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs," in *POPL*, 2011.
- [2] A. Farzan, Z. Kincaid, and A. Podelski, "Proof spaces for unbounded parallelism," in *POPL*, 2015.
- [3] A. Farzan and A. Vandikas, "Reductions for safety proofs," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 2020.
- [4] S. S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, no. 5, 1976.
- [5] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, 1983.
- [6] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, 1975.
- [7] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *PLDI*, 2003.
- [8] B. Kragl and S. Qadeer, "Layered concurrent programs," in *CAV*, 2018.
- [9] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran, "Automated and modular refinement reasoning for concurrent programs," in *CAV*, 2015.
- [10] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, 1975.
- [11] T. Elmas, S. Qadeer, and S. Tasiran, "A calculus of atomic actions," in *POPL*, 2009.
- [12] B. Kragl, S. Qadeer, and T. A. Henzinger, "Synchronizing the asynchronous," in *CONCUR*, 2018.
- [13] B. Kragl, C. Enea, T. A. Henzinger, S. O. Mutluergil, and S. Qadeer, "Inductive sequentialization of asynchronous programs," in *PLDI*, 2020.
- [14] B. Kragl, S. Qadeer, and T. A. Henzinger, "Refinement for structured concurrent programs," in *CAV*, 2020.
- [15] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, 2005.
- [16] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *FOCS*, 1995.
- [17] J. R. Wilcox, C. Flanagan, and S. N. Freund, "VerifiedFT: a verified, high-performance precise dynamic race detector," in *PPoPP*, 2018.
- [18] S. O. Mutluergil and S. Tasiran, "A mechanized refinement proof of the Chase-Lev deque using a proof system," *Computing*, vol. 101, no. 1, 2019.
- [19] S. Krishna, M. Emmi, C. Enea, and D. Jovanovic, "Verifying visibility-based weak consistency," in *ESOP*, 2020.
- [20] A. Bouajjani, C. Enea, S. O. Mutluergil, and S. Tasiran, "Reasoning about TSO programs using reduction and abstraction," in *CAV*, 2018.
- [21] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: An exercise in cooperation," *Commun. ACM*, vol. 21, no. 11, 1978.
- [22] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, 1998.
- [23] 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, 2017.
- [24] L. Lamport, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*, 2002.
- [25] J. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *Int. J. Softw. Tools Technol. Transf.*, vol. 12, no. 6, 2010.
- [26] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," in *PLDI*, 2016.
- [27] K. L. McMillan and O. Padon, "Ivy: A multi-modal verification tool for distributed algorithms," in *CAV*, 2020.
- [28] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *J. Funct. Program.*, vol. 28, 2018.
- [29] D. Frumin, R. Krebbers, and L. Birkedal, "ReLoC: A mechanised relational logic for fine-grained concurrency," in *LICS*, 2018.
- [30] K. R. M. Leino and P. Müller, "A basis for verifying multi-threaded programs," in *ESOP*, 2009.
- [31] B. Jacobs and F. Piessens, "Expressive modular fine-grained concurrency specification," in *POPL*, 2011.
- [32] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 2019.
- [33] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn, "The VerCors tool set: Verification of parallel and concurrent software," in *IFM*, 2017.
- [34] E. Cohen, M. Moskal, W. Schulte, and S. Tobies, "Local verification of global invariants in concurrent programs," in *CAV*, 2010.
- [35] C. Flanagan and S. N. Freund, "The Anchor verifier for blocking and non-blocking concurrent software," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020.
- [36] S. N. Freund and S. Qadeer, "Checking concise specifications for multithreaded software," *J. Object Technol.*, vol. 3, no. 6, 2004.
- [37] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao, "Armada: low-effort verification of high-performance concurrent programs," in *PLDI*, 2020.
- [38] S. K. Lahiri, S. Qadeer, and D. Walker, "Linear maps," in *PLPV*, 2011.

Synthesizing Pareto-Optimal Interpretations for Black-Box Models

Hazem Torfah¹ , Shetal Shah² , Supratik Chakraborty² , S. Akshay² , Sanjit A. Seshia¹ 

¹University of California at Berkeley

{torfah, ssesia}@berkeley.edu

²Indian Institute of Technology Bombay

{shetals, supratik, akshayss}@cse.iitb.ac.in

Abstract—We present a new multi-objective optimization approach for synthesizing interpretations that “explain” the behavior of black-box machine learning models. Constructing *human-understandable* interpretations for black-box models often requires balancing conflicting objectives. A simple interpretation may be easier to understand for humans while being less precise in its predictions vis-a-vis a complex interpretation. Existing methods for synthesizing interpretations use a single objective function and are often optimized for a single class of interpretations. In contrast, we provide a more general and multi-objective synthesis framework that allows users to choose (1) the class of syntactic templates from which an interpretation should be synthesized, and (2) quantitative measures on both the correctness and explainability of an interpretation. For a given black-box, our approach yields a set of Pareto-optimal interpretations with respect to the correctness and explainability measures. We show that the underlying multi-objective optimization problem can be solved via a reduction to quantitative constraint solving, such as weighted maximum satisfiability. To demonstrate the benefits of our approach, we have applied it to synthesize interpretations for black-box neural-network classifiers. Our experiments show that there often exists a rich and varied set of choices for interpretations that are missed by existing approaches.

I. INTRODUCTION

Machine learning (ML) components, especially deep neural networks (DNNs), are increasingly being deployed in domains where trustworthiness and accountability are major concerns. Such domains include health care [5], automotive systems [28], finance [21], loans and mortgages [25], [33], and cyber-security [10] among others. For a system to be considered accountable and trustworthy, it is necessary to provide understandable explanations to (possibly expert) humans of why the system took specific actions/decisions in response to inputs of concern. This requires the availability of models that are human-understandable, and that also predict the outcome of different components of the system with reasonable accuracy. Laws and regulations, such as the General Data Protection Regulation (GDPR) in Europe [1], are already emerging with requirements on explainability of ML components in such systems. Unfortunately, the working of ML components like DNNs can be extremely complex to comprehend, and more so when the components are used as black boxes. Therefore, there is an urgent need for automated techniques that generate “easy-to-understand” and “targeted” interpretations of black-box ML components, with formal guarantees about tradeoffs between correctness and explainability.

Synthesizing a “good” interpretation of a black-box ML component often requires striking the right balance between correctness or accuracy of the interpretation (measured in terms of fidelity, misclassification rate of predictions etc.) and explainability or understandability (approximated by the size of the ML model – e.g., depth of decision tree/list/diagram, number and nature of predicates used, etc.). In most cases, the correctness and explainability measures are in direct conflict with each other. Thus, a simple interpretation that is easily understood by humans may disagree in its predictions with the output of a black-box ML component for many input instances, whereas an interpretation that correctly predicts the output for most input instances may be too large and unwieldy for human comprehension. This is not surprising since components like DNNs are often used to learn highly non-trivial functions for which simple models are not available. Therefore, *synthesis of interpretations for black-box ML components is inherently a multi-objective optimization problem with conflicting objectives, and Pareto optimality is the best we can hope for when synthesizing such interpretations.*

The literature contains a rich collection of techniques for synthesis of interpretations for black-box ML components (see, for example, recent surveys by [2] and [13]). Most of these approaches optimize a single correctness measure (e.g. misclassification rate on a set of samples) while systematically constraining some explainability measure (e.g. number of nodes or depth of a decision tree). Examples of such techniques include [19] wherein sparse logical formulae are synthesized, and also recent approaches to learning optimal decision trees using constraint programming [35]–[37], item-set/rulelist mining [3] and SAT-based techniques [6], [18], [27], among others. These approaches often allow efficient generation of a *single* interpretation with high correctness measure and satisfying user-provided explainability constraints. However, no formal guarantees of Pareto-optimality (w.r.t. correctness and explainability) are provided. Furthermore, these techniques do not compute the set of *all* Pareto-optimal interpretations, thereby constraining the choice of which interpretation to use for a given application.

In this paper, we present a novel multi-objective optimization approach for synthesizing Pareto-optimal interpretations of black-box ML components, using an off-the-shelf quantitative constraint solver (weighted MaxSAT solver in

our case). For each problem instance, our approach yields a set of interpretations that correspond to *all* Pareto-optimal combinations of correctness and explainability measures. This contrasts sharply with earlier approaches such as [3], [6], [18], [19], [27], [35]–[37] that always yield a single interpretation, leaving the user with no choice of exploring the trade-off between correctness and explainability of alternative interpretations. Similar to existing work, we use syntactic constraints to restrict the class of interpretations over which to search. Unlike earlier approaches, however, we do not combine quantitative correctness and explainability measures into a single optimization objective. Any such mapping of an inherently multi-dimensional optimization problem to the uni-dimensional case results in exclusion of some Pareto-optimal solutions in general. Given that quantitative explainability measures are often just approximations of subjective preferences of the end-user, we believe it is important to present the entire set of Pareto-optimal interpretations, and leave the choice of the “best” interpretation to the user. As our experiments show, there is significant diversity among Pareto-optimal interpretations, and a user aware of this diversity can make an informed choice for a specific application.

The syntactic constraints considered in this paper restrict the space of interpretations to decision diagrams (a generalization of decision trees) with specified bounds on the number of nodes, predicates and branching factors. For simplicity, we let the set of predicates be pre-determined but potentially large, and with possibly different relative preferences for different predicates. We focus on the setting where the black-box ML model can only be treated as an input-output oracle, i.e., given an input, we can observe its output and nothing else. Additionally, we do not have access to training or test data used to create the black-box component. Our correctness measure is therefore based on querying the black-box component with random samples chosen from its input space, where the sample set size is carefully chosen to provide statistical guarantees of near-optimality. Our explainability measure takes into account user preferences of predicates and also size of the interpretation, preferring smaller interpretations over larger ones. The overall framework is, however, general enough to admit other syntactic classes (beyond decision diagrams), and also other correctness and explainability measures.

We have implemented our approach in a prototype tool and applied it to synthesize Pareto-optimal interpretations for some black-box neural network classifiers. Our results exhibit the richness of choices available to the end-user in each case, none of which would be exposed by existing methods that generate only a single optimal interpretation. Indeed, we find that significant improvements in explainability can sometimes be achieved by only a marginal reduction of accuracy.

Our primary contributions can be summarized as follows:

- 1) We formulate the Pareto-optimal interpretation synthesis problem for black-box ML components.
- 2) We show that finding a single Pareto-optimal interpretation can be formulated as a weighted MaxSAT

problem, for some meaningful choices of correctness and explainability scores.

- 3) We present a divide-and-conquer algorithm for synthesizing interpretations for *all* Pareto-optimal combinations of correctness and explainability scores.
- 4) We provide formal guarantees of soundness, completeness and universality of our algorithm, and also statistical guarantees of near-optimality when only a subset of behaviors of a black-box component is sampled.
- 5) We build a prototype tool and apply it to a collection of black-box neural network classifiers: our results show that significant diversity exists among Pareto-optimal interpretations which earlier tools fail to discover.

II. MOTIVATING EXAMPLE

We start with an example, adapted from [11], that illustrates the diversity that exists among Pareto-optimal interpretations of black-box ML models. Consider a scenario where an airplane uses a neural network to autonomously taxi along a runway, relying on a camera sensor. Suppose the plane is expected to follow the runway centerline within a tolerance of 2.5 meters. The airplane is equipped with monitoring modules that decide under what circumstances certain learning-enabled components can be trusted to behave correctly. One of these monitoring modules decides under what conditions the camera-based perception module, that determines the distance to the centerline, can be trusted to deliver the right values. For example, the monitoring module may use the weather condition, time of day, and initial positioning of the airplane to decide whether the perception module’s output is reliable. We wish to reason about this black-box monitoring module, and hence need an understandable interpretation for it.

Given a set of user-defined predicates (viz. clouds, time of day, and initial position of the plane), the user may favor certain predicates over others, and also favor concise interpretations. By giving favorability weights to each predicate, we can define an explainability score that is related to the number of nodes in the interpretation and also to the predicates used (this is detailed later). The prediction accuracy of an interpretation is measured w.r.t a set of examples sampled from the black box, and is represented by a correctness score. Our approach explores the space of interpretations, searching for concise interpretations that use more favored predicates and also have high accuracy. Clearly, to find a “good” interpretation that meets these conflicting goals, one must explore *all* Pareto-optimal interpretations w.r.t. the criteria above.

Figure 1 shows three of the many Pareto-optimal interpretations our approach synthesized for the monitoring black-box. Each of these has its own pros and cons, and is incomparable with the others. The user can now choose the interpretation that best suits the user’s purpose. For example, if interpretation size is not of concern but accuracy is, then Figure 1(b) is the best choice. However, if the user wants concise models with favored predicates (related to time of day and initial position), then Figure 1(a) is the best choice. The user may also choose the interpretation in Figure 1(c), which is only

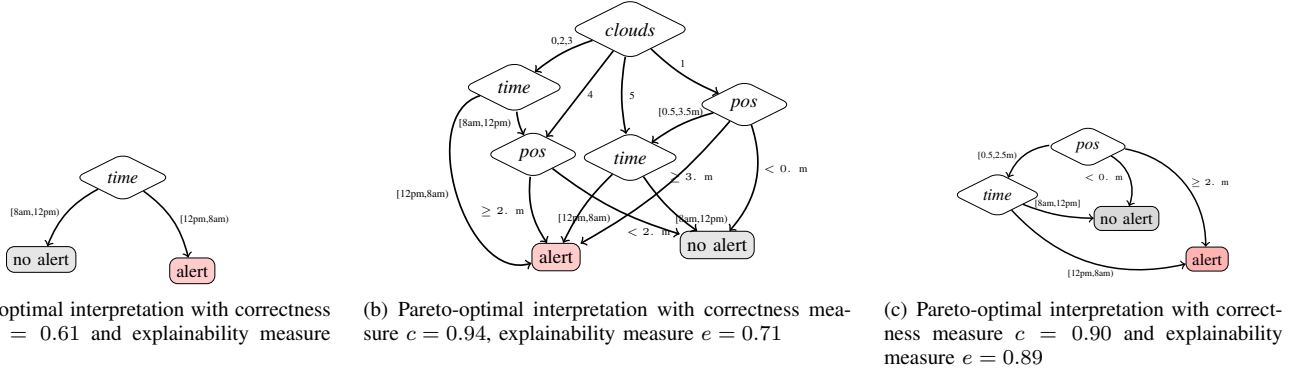


Fig. 1. Pareto-optimal decision diagram interpretations for the black-box monitoring component that decides based on time of day, cloud types, and initial position of an airplane whether to trust a perception module to help the plane track the centerline of a runway. The correctness score is given by the prediction accuracy w.r.t. to the used sample set. The explainability score is the normalized sum of weights of used predicates and unused nodes.

slightly less accurate than that in Figure 1(b), but has a higher explainability score. In fact, Figure 1(c) represents a healthy balance between accuracy and explainability. According to it, the perception module can be trusted only during morning hours if the plane starts no more than 2.5m from the centerline, or at any time if the plane starts within 0.5m of the centerline.

Tools that use a single-objective function to synthesize interpretations can only find one of these Pareto-optimal interpretations, depending on the relative weights given to accuracy and explainability. The rich diversity among Pareto-optimal interpretations is completely missed by such tools, effectively restricting the user’s choice of a “good” interpretation.

III. PARETO-OPTIMAL INTERPRETATION SYNTHESIS

In this section, we formalize the Pareto-optimal interpretation synthesis problem and present a solution (for specific choices of correctness and explainability scores) using a quantitative constraint satisfaction engine. In our case, this engine is an off-the-shelf weighted maximum satisfiability solver. The key idea is that the user sets syntactic restrictions on the class of considered interpretations as well as quantitative objectives for evaluating the interpretations. The quantitative objectives are defined using two inherently incomparable measures – the explainability measure and the correctness measure. The explainability measure relates to “ease” of understanding of the interpretation by an end-user, while the correctness measure relates to how precisely the interpretation explains the behavior of the black-box model on a given set of samples. Examples of quantitative correctness measures include accuracy, recall, precision, F1-score [34], while examples of explainability measures include those that reward usage of concise interpretations and less complex predicates.

Since our access to the black-box model is only via input/output samples, the correctness measure referred to above is defined with respect to a set of samples, and not with respect to the black-box model in its entirety. While this may appear ad-hoc at first sight, we show in Section IV that rigorous statistical guarantees can indeed be provided with sufficiently many samples.

A. Formal problem definition

We now give a formal definition of the Pareto-optimal interpretation synthesis problem. An interpretation is simply a syntactic structure, viz. decision tree, decision diagram, linear model, etc. We will fix a class of interpretations \mathcal{E} over an input domain \mathcal{I} and output domain \mathcal{O} . For an interpretation $E \in \mathcal{E}$, we define $f_E \in (\mathcal{I} \rightarrow \mathcal{O})$ to be the semantic function that is computed by E . Note that different interpretations may compute the same semantic function.

Every interpretation $E \in \mathcal{E}$ is associated with a pair of real-valued measures (c, e) , where c is the correctness measure and e is the explainability measure of E . We define a partial order \preceq on such pairs as: $(c, e) \preceq (c', e')$ iff $c \leq c'$ and $e \leq e'$. Given a set X of (c, e) pairs, we define $\max^{\preceq} X$ to be the set of \preceq -maximal pairs in X . An interpretation E with the pair of measures (c, e) is said to be *Pareto-optimal* if (c, e) is maximal over pairs of measures of all interpretations.

Definition 1 (Pareto-optimal interpretation synthesis): Let \mathcal{E} be a syntactic class of interpretations over inputs \mathcal{I} and outputs \mathcal{O} . Further, let $\mathcal{S} \subseteq \mathcal{I} \times \mathcal{O}$ be a set of samples, $\Delta_{\mathcal{C}}: (\mathcal{I} \rightarrow \mathcal{O}) \times 2^{(\mathcal{I} \times \mathcal{O})} \rightarrow \mathbb{R}^{\geq 0}$ be a correctness measure, and $\Delta_{\mathcal{E}}: \mathcal{E} \rightarrow \mathbb{R}^{\geq 0}$ an explainability measure. The Pareto-optimal interpretation synthesis problem $\langle \mathcal{E}, \mathcal{S}, \Delta_{\mathcal{C}}, \Delta_{\mathcal{E}} \rangle$ is the multi-objective problem of finding a Pareto-optimal interpretation $E \in \arg \max_{E' \in \mathcal{E}}^{\preceq} (\Delta_{\mathcal{C}}(f_{E'}, \mathcal{S}), \Delta_{\mathcal{E}}(E'))$.

We interpret $\Delta_{\mathcal{C}}(f_E, \mathcal{S})$ as a measure of closeness between the semantic function f_E of interpretation E and the semantic constraints defined by a set \mathcal{S} of samples. An optimally correct interpretation is one with maximal closeness. An example of such a measure is the *prediction accuracy* $\frac{|\{(i, o) \in \mathcal{S} \mid f_E(i) = o\}|}{|\mathcal{S}|}$. The problem can also be defined in terms of the “distance” between an interpretation and the semantic constraints defined by \mathcal{S} , in which case, the optimization problem is one of minimization. An example of such a measure is the *misclassification rate*, which is one minus the prediction accuracy. Similarly, for $\Delta_{\mathcal{E}}(\cdot)$, we choose to define it as a reward function that we want to maximize, but it can also be dually defined as a cost function we want to minimize.

For each \preceq -maximal pair of measures, there can be multiple corresponding interpretations realizing the measures. We don't distinguish between them for purposes of this paper. The following definition is therefore relevant.

Definition 2 (Minimal representative set): A set Γ of Pareto-optimal interpretations is a minimal representative set for $\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle$ if for every $(c, e) \in \max_{E \in \mathcal{E}} (\Delta_C(f_E, \mathcal{S}), \Delta_E(E))$, there is exactly one interpretation $E' \in \Gamma$ such that $(\Delta_C(f_{E'}, \mathcal{S}), \Delta_E(E')) = (c, e)$.

Our goal can therefore be stated as one of finding a minimal representative set of interpretations for a black-box model.

B. Synthesis via weighted maximum satisfiability

We now discuss how to synthesize one (of possibly many) Pareto-optimal interpretation for specific choices of \mathcal{E} , Δ_C and Δ_E , by encoding the synthesis problem as a *weighted maximum satisfiability* problem (weighted MAXSAT). For purposes of our discussion, we choose \mathcal{E} to be the class of *bounded multi-valued decision diagrams*, i.e., decision diagrams with multiple branching at each node, where the branching is governed by decision predicates, and with a bound on the number of decision nodes (see, e.g., diamond nodes in Figure 1). We use prediction accuracy as the correctness measure, and define the explainability measure with weights (denoting preferences) on the predicates and on the number of used nodes. The encoding for several other classes of interpretations, such as decision trees, decision rules, etc. and for other explainability and correctness measures can be done similarly.

We start by recalling the weighted MAXSAT problem. A Boolean formula φ over variables in a set X is said to be in conjunctive normal form (CNF) if φ is of the form $C_1 \wedge C_2 \wedge \dots \wedge C_m$, where each C_i is a disjunction of literals (i.e. variables or negations of variables). An assignment $\sigma: X \rightarrow \{0, 1\}$ is an assignment of truth values to variables. If a clause C_i evaluates to 1 under σ , we say σ satisfies C_i , denoted by $\sigma \models C_i$.

Definition 3 (Weighted Maximum Satisfiability): Given a Boolean formula $\varphi = \bigwedge_{i=1}^m C_i$ in CNF and a weight function $w: \{C_1, \dots, C_m\} \rightarrow \mathbb{R}^{\geq 0}$ that assigns a non-negative real weight to each clause, the weighted MAXSAT problem is to find an assignment σ which maximizes $\sum_{\{C_i \mid \sigma \models C_i\}} w(C_i)$. In a variant of the above definition, the clauses in φ are partitioned into *hard* and *soft* clauses. The problem now is to find an assignment σ that satisfies *all hard clauses* and maximizes the sum of weights of satisfied soft clauses. We use this variant for encoding our problem.

At a high level, for an instance $\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle$ of the Pareto-optimal interpretation synthesis problem, we define its encoding as a conjunction of four formulae. Specifically, $\phi_{\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle} = \phi_{\mathcal{E}} \wedge \phi_{\mathcal{S}} \wedge \phi_{\Delta_C} \wedge \phi_{\Delta_E}$ where, (i) $\phi_{\mathcal{E}}$ encodes the syntactic restrictions, i.e., bounded multi-valued decision diagrams with the permitted predicates (features and branchings) and labels; (ii) $\phi_{\mathcal{S}}$ encodes the semantic constraints, i.e., the relation between the samples in \mathcal{S} and an interpretation satisfying $\phi_{\mathcal{E}}$; (iii) ϕ_{Δ_C} encodes the correctness measure, e.g., in case of prediction accuracy it encodes whether an interpretation agrees on a sample; and finally (iv) ϕ_{Δ_E} defines constraints

that encode certain structural aspects of an interpretation, e.g., what predicates were chosen and whether a node was used. We discuss some details of these formulas below, leaving the full encoding to the long version of this paper at [31].

a) *Encoding of the interpretation class ($\phi_{\mathcal{E}}$):* We start by discussing the encoding for our interpretation class of bounded multi-valued decision diagrams over inputs \mathcal{I} and outputs \mathcal{O} . These diagrams are restricted by a finite set of decision predicates, denoted by P . For example, in Figure 1(a), the initial node uses the “time of day” predicate with branchings: $\{[8\text{am}-12\text{pm}], [12\text{pm}-8\text{am}]\}$. Let L be a set of output labels, e.g., in Figure 1, we have two labels, “alert” and “no alert”. An interpretation $E \in \mathcal{E}$ is a multi-valued decision diagram over a finite set of nodes \mathcal{N} , where each internal node corresponds to a decision predicate $p \in P$ and each leaf to an output label $\ell \in L$. Outgoing transitions of a node are labelled according to the branchings of the predicate corresponding to the node. We remark that features are distinct from inputs to the black-box. For example, in the decision diagrams in Figure 1 the feature “pos” uses the latitude and longitude inputs to compute the initial position of the plane. Furthermore, the same predicate may appear on different nodes in the decision diagram, but not more than once along a path. For a given P , L , and a bound n on the number of nodes \mathcal{N} in the decision diagram, the formula $\phi_{\mathcal{E}}$ encodes an acyclic decision diagram of at most n -nodes over a set P of predicates, with leaves labeled by elements of L .

b) *Encoding of the samples ($\phi_{\mathcal{S}}$):* The formula $\phi_{\mathcal{S}}$ encodes the relation between the samples and the interpretation $\phi_{\mathcal{E}}$. It uses an auxiliary variable $m_{(i,o)}$ for each sample (i, o) in the set \mathcal{S} . Logically, $m_{(i,o)}$ is set to true iff the interpretation given by a satisfying assignment of $\phi_{\mathcal{E}}$ produces the output label o when fed the input i . For decision diagrams, this is encoded by symbolically matching the input i to a decision path in the diagram, and by comparing the value of o with that of the label reached at the end of the decision path. Note that the number of these auxiliary variables grows linearly with the size of the sample set.

c) *Encoding the correctness measure (ϕ_{Δ_C}):* To encode Δ_C , we add a unit soft clause (i.e., a clause with only one literal) $m_{(i,o)}$ for each sample (i, o) . By assigning appropriate weights to these unit clauses and by maximizing the sum of weights of satisfied clauses (see Definition 3), we obtain an interpretation that maximizes Δ_C with respect to the sample set \mathcal{S} . E.g., if Δ_C represents the prediction accuracy, then assigning a weight of 1 to each unit clause $m_{(i,o)}$ gives us an interpretation that agrees on a maximal number of samples in \mathcal{S} . If the user is interested in interpretations that agree on certain types of samples, then higher weights should be given to these samples. More precisely, to define such measures Δ_C , the user can provide a function $w: \mathcal{I} \times \mathcal{O} \rightarrow \mathbb{R}$, that defines these weights. For example, in the case of prediction accuracy, w is the constant function 1.

d) *Encoding the explainability measure (ϕ_{Δ_E}):* To encode Δ_E , we add a unit clause u_{γ} for each syntactic structure γ of an interpretation in \mathcal{E} and give it a weight according to how

important γ is. For example, in the case of decision diagrams, using some predicates may be more favorable than others. To encode this, we add unit clauses $u_{(i,p)}$ that are set to true iff predicate p is used in node i , and assign higher weights for clauses representing favorable predicates. Moreover, predicates with fewer branches can be favored by using soft clauses with appropriate weights. To further reward the synthesis of decision diagrams with fewer nodes, we can also add unit soft clauses u_i for each node i that is set to true iff node i is not reachable from the root node in an interpretation satisfying $\phi_{\mathcal{E}}$, and give them positive weights. In this case, by maximizing the satisfaction of these clauses, we reward the synthesis of small decision diagrams.

In our weighted MAXSAT formulation, we require that all clauses resulting from a Tseitin encoding (i.e., a transformation into CNF) of the formula $\phi_{\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle}$, except the unit soft clauses mentioned above, be hard clauses. On feeding the above formula to a MAXSAT solver, it returns a satisfying assignment giving a concrete instantiation of the decision diagram template that maximizes the sum of weights of $m_{(i,o)}$ and u_γ clauses.

The encoding described above is specific to a particular choice of \mathcal{E} , Δ_C and Δ_E . However, similar encoding can be done for a much wider class of interpretations, and explainability and correctness measures. In fact, most types of interpretation classes used in the literature, viz. decision trees, decision diagrams, decision lists and sets of bounded depth/size admit encoding as Boolean formulas. In addition, if the computation of explainability and correctness measures can be encoded using arithmetic circuits of bounded bit-width, the Pareto-optimal interpretation synthesis problem can be reduced to weighted MAXSAT by assigning appropriate weights to bits in the bit-vector representing the measures. The following theorem applies to our encoding, and to all other similar encodings referred to above.

Theorem 1 (Pareto-optimality): Every solution of the weighted MAXSAT problem $\phi_{\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle}$ gives a solution for the Pareto-optimal interpretation synthesis problem $\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle$.

C. Exploring the set of Pareto-optimal interpretations

We now present an algorithm for computing a minimal representative set of Pareto-optimal interpretations. The algorithm is based on the key observation that every Pareto-optimal measure (c, e) splits the space of measures into four regions, depicted in Figure 2(a), (1) a region $R_1^{c,e}$ of measures for which there exists no solution, namely, all measures $(c', e') \neq (c, e)$ with $c' \geq c$ and $e' \geq e$, otherwise (c, e) would not be Pareto-optimal, (2) a region $R_2^{c,e}$ of measures that are not Pareto-optimal, namely, all points $(c', e') \neq (c, e)$ with $c' \leq c$ and $e' \leq e$, (3) a region $R_3^{c,e}$ with measures of potential Pareto-optimal interpretations with better correctness measures, i.e., those with measures (c', e') with $c' > c$ and $e' < e$, and lastly (4) a region $R_4^{c,e}$ with measures of potential Pareto-optimal interpretations with better explainability measures, i.e., points (c', e') with $c' < c$ and $e' > e$. By synthesizing a first Pareto-

optimal interpretation using the procedure from last section, and then dividing the search space into corresponding regions (1)-(4), our algorithm proceeds by searching for further Pareto-optimal interpretations with better correctness in region (3) and better explainability in region (4). This process is repeated for every Pareto-optimal interpretation found by our algorithm, thus, directing the search into smaller and smaller regions until no new Pareto-optimal interpretation can be found.

This is detailed in Algorithm 1 and the exploration process it implements is illustrated in Figure 2. For $\mathcal{E}, \mathcal{S}, \Delta_C$, and Δ_E , Algorithm 1 returns a minimal representative set Γ of interpretations for all Pareto-optimal measures. To synthesize a Pareto-optimal interpretation within a given region of measures, Algorithm 1 relies on the procedure QUINTSYNT which given $\mathcal{E}, \mathcal{S}, \Delta_C$, and Δ_E , in addition to a lower-bound $\delta_{\mathcal{E}}^l$ and upper-bound $\delta_{\mathcal{E}}^u$ on the explainability measure, returns a Pareto-optimal interpretation E with explainability measure e such that $\delta_{\mathcal{E}}^l \leq e \leq \delta_{\mathcal{E}}^u$. QUINTSYNT effectively solves an extension of the weighted MaxSAT instance defined in the last section, in which we additionally require the explainability measure to satisfy the constraints given by the lower-bound $\delta_{\mathcal{E}}^l$ and upper-bound $\delta_{\mathcal{E}}^u$. This can be done by extending the formula ϕ in the last section with a fifth conjunct $\phi_{\delta_{\mathcal{E}}^l, \delta_{\mathcal{E}}^u}$. This conjunct is satisfied if the sum of weights of the used syntactic structures (e.g. in the case of decision diagrams, this will be sum of weights of the satisfied clauses $u_{(i,p)}$ and u_i) lies within the given bounds. We leave details of this encoding to [31], but intuitively, we encode a binary adder that sums up the weights of satisfied $u_{(i,p)}$ and u_i clauses and compare the results to binary encodings of the bounds. To fix the number of bits to encode both the adder and bounds, we normalize the weights to values between 0 and 1 up to a certain floating-point precision k . Now let us go further into Algorithm 1 while elaborating on why it suffices to only bound the explainability measure when exploring regions (3) and (4) depicted in Figure 2(a).

Initially, Algorithm 1 explores the entire set of Pareto-optimal solution space. To this end, the exploration set W is initialized with the point $(0, 1, 0)$ (line 2) defining a lower bound on the explainability measure, an upper-bound on the explainability measure, and a lower-bound on the correctness measure, respectively. For every point $(\delta_{\mathcal{E}}^l, \delta_{\mathcal{E}}^u, \delta_C)$ in W , QUINTSYNT synthesizes a Pareto-optimal region within the explainability measure bounds defined by $\delta_{\mathcal{E}}^l$ and $\delta_{\mathcal{E}}^u$ (line 5). If an interpretation E is found with measures c and e , i.e., $E \neq \perp$ (line 6), the algorithm further divides the search space based on the following case distinction:

- if $c > \delta_C$, then a new Pareto-optimal interpretation with measures (c, e) is found and the regions $R_3^{c,e}$ and $R_4^{c,e}$ defined by the points $(\delta_{\mathcal{E}}^l, \downarrow e, c)$ and $(\uparrow e, \delta_{\mathcal{E}}^u, \delta_C)$, respectively, are added to W (lines 9 and 10). The operators \downarrow and \uparrow define the predecessor and successor value of the value e (we assume that the values are discrete and hence the predecessor and successor exist). For example, if the interpretation synthesized by QUINTSYNT is one with measures c', e' as depicted in Figure 2(b), then the region

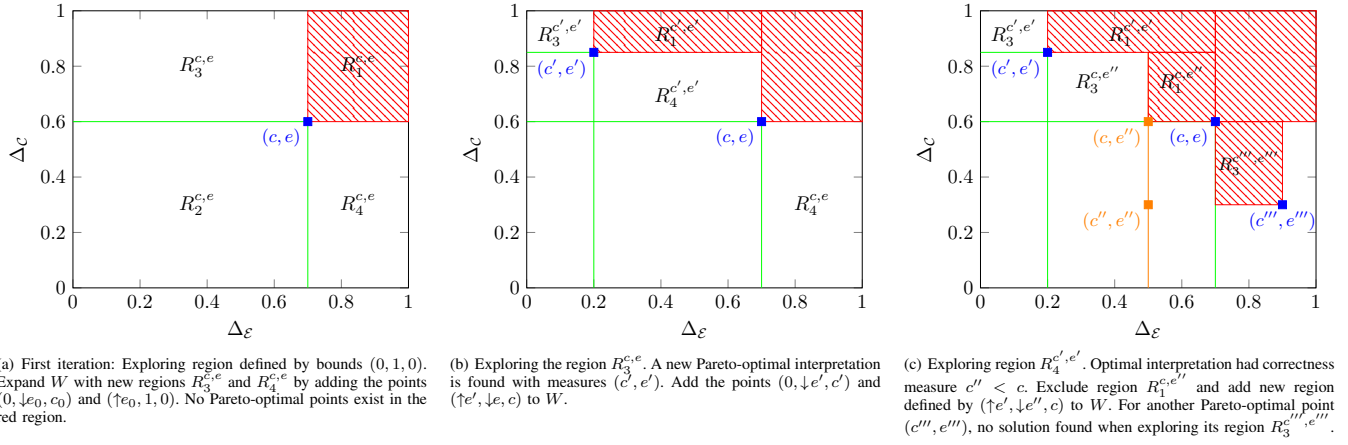


Fig. 2. An illustration of Algorithm 1.

Algorithm 1 EXPLOREPOI

Input: $\mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E$

Output: Minimal representative set Γ for $\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle$

```

1:  $\Gamma := \emptyset$ 
2:  $W := \{(0, 1, 0)\}$ 
3: while  $W \neq \emptyset$  do
4:    $(\delta_E^l, \delta_E^u, \delta_C) := \text{pop}(W)$ 
5:    $(E, (c, e)) = \text{QUINTSYNT}(\mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E, \delta_E^l, \delta_E^u)$ 
6:   if  $E \neq \perp$  then
7:     if  $c > \delta_C$  then
8:        $\Gamma := \Gamma \cup \{(E, (c, e))\}$ 
9:        $\text{push}(W, (\delta_E^l, \downarrow e, c))$ 
10:       $\text{push}(W, (\uparrow e, \delta_E^u, \delta_C))$ 
11:     else
12:        $\text{push}(W, (\delta_E^l, \downarrow e, \delta_C))$ 
13:     end if
14:   end if
15: end while
16: return  $\Gamma$ 

```

$R_4^{c',e'}$ is captured by the point $(\uparrow(e'), \downarrow(e), c)$. The region $R_3^{c',e'}$ is captured by $(0, \downarrow(e'), c')$. Notice that we do not need to include an upper bound on the correctness measure as it is already implicitly defined by the $R_1^{c,e}$ region of any Pareto-optimal point (c, e) . For example, in Figure 2(b) the upper bound on the correctness for region $R_4^{c',e'}$ is already captured through the fact that no Pareto-optimal solutions exist in $R_1^{c',e'}$.

- if $c \leq \delta_C$, then (c, e) cannot be Pareto-optimal, because we already know that there is a Pareto-optimal interpretation with measures $(\delta_C, \uparrow \delta_E^u)$. In this case, we can exclude the search in the region $R_1^{\delta_C, e}$, because if there was any Pareto-optimal interpretation with measures (\hat{c}, \hat{e}) in $R_1^{\delta_C, e}$, then QUINTSYNT would have found this interpretation. Thus, Algorithm 1 further prunes the search region to a smaller region defined by $(\delta_E^l, \downarrow e, \delta_C)$ (line 12). For example, if Algorithm 1 used QUINTSYNT

to synthesize an interpretation from $R_4^{c',e'}$, and returned a solution with measures (c'', e'') as depicted in Figure 2(c), then we can exclude the search in region $R_1^{c,e''}$ and add the region $R_3^{c,e''}$ to W .

Lastly, if QUINTSYNT returns no interpretation, then we can immediately exclude the searched region from further exploration and thus no new points are added to W in this case. For example, as shown in Figure 2(c), if QUINTSYNT found no Pareto-optimal interpretations in $R_3^{c'',e''}$, then this region is excluded from the search and Algorithm 1 continues with the next available point in W .

Next we show some important properties of Algorithm 1.

Lemma 1 (Soundness): For an instance $\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle$ of the Pareto-optimal interpretation synthesis problem, if $(E, (c, e)) \in \text{EXPLOREPOI}(\mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E)$, then $(c, e) \in \max_{E' \in \mathcal{E}}^{\preceq} (\Delta_C(f_{E'}, \mathcal{S}), \Delta_E(E'))$.

In the rest of this section, we assume that each of the explainability measures has finitely many discrete values, as they are defined as floating points up to a certain precision. Thus, we obtain that the range of Δ_E is finite, which allows us to obtain the following results.

Lemma 2 (Completeness): For an instance $\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle$ of the Pareto-optimal interpretation synthesis problem, if $(c, e) \in \max_{E' \in \mathcal{E}}^{\preceq} (\Delta_C(f_{E'}, \mathcal{S}), \Delta_E(E'))$, then there is an interpretation E with measures (c, e) such that $(E, (c, e)) \in \text{EXPLOREPOI}(\mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E)$.

We summarize the correctness result next which follows immediately from Lemmas 1 and 2.

Theorem 2 (Correctness of Algorithm 1): For a class of interpretations \mathcal{E} , a finite set of samples \mathcal{S} , and measures Δ_C and Δ_E , the algorithm EXPLOREPOI terminates and returns a minimal representative set for $(\mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E)$.

Algorithm EXPLOREPOI solves the interpretation synthesis problem as a multi-objective optimization problem. If we were to solve the same problem using single-objective optimization, it would be necessary to combine the accuracy and explainability measures for every interpretation to yield a single hybrid measure. Let $\lambda : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ be a function that yields such a

measure. Since higher values of c and e always increase the desirability of an interpretation, we require λ to be *strictly increasing*, i.e., $(c, e) \prec (c', e') \implies \lambda(c, e) < \lambda(c', e')$. For example, $\lambda(c, e) = w_1 \cdot c + w_2 \cdot e$ is a strictly increasing function for every $w_1, w_2 > 0$. Then, for any (c, e) pair that is maximal wrt such a function λ , our algorithm can find an interpretation with this measure pair. Formally,

Theorem 3 (Universality): For every strictly increasing function $\lambda : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ and every $\langle \mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E \rangle$ if $E \in \arg \max_{E' \in \mathcal{E}} (\lambda(\Delta_C(f_{E'}, \mathcal{S}), \Delta_E(E')))$, then there exists an interpretation $E^* \in \mathcal{E}$ such that (i) $\Delta_C(f_E, \mathcal{S}) = \Delta_C(f_{E^*}, \mathcal{S})$, (ii) $\Delta_E(E) = \Delta_E(E^*)$, and (iii) $(E^*, (\Delta_C(f_{E^*}, \mathcal{S}), \Delta_E(E^*))) \in \text{EXPLOREPOI}(\mathcal{E}, \mathcal{S}, \Delta_C, \Delta_E)$.

We conclude the section with some remarks on Algorithm 1.

Remark 1: Algorithm 1 can also be applied interactively as a conversation between synthesizer and user. Given a Pareto-optimal interpretation, the user may guide the search to interpretations that are more explainable or to those with more accuracy, until the user has found an optimal interpretation.

Remark 2: Note that there might be multiple interpretations with the same pair (c, e) . In this case, Algorithm 1 will add only one of them as a representative interpretation, since the others are indistinguishable wrt correctness and explainability.

Finally, we can also search for Pareto-optimal solutions based on regions solely bounded on the correctness measure. We choose to use bounds on the explainability measure, because the sample sets tend to be large and will result in much larger encodings.

IV. STATISTICAL GUARANTEES FOR BLACK-BOX MODELS

In Section III, the correctness of an interpretation E , defined using a measure Δ_C , was determined with respect to a set of samples \mathcal{S} obtained from the black-box model \mathcal{B} . Our approach guarantees that E is optimal for \mathcal{S} and the measure Δ_C . Our ultimate goal, however, is to synthesize an interpretation E that is optimal with respect to the entire black-box model \mathcal{B} , i.e., w.r.t. the set $\mathcal{S}_B = \{(i, o) \mid f_B(i) = o, i \in \mathcal{I}\}$. Obtaining an exhaustive set of samples from a black-box model is often not practical. The question that we, therefore, raise in this section is: *how large must \mathcal{S} be such that it is not misleading, i.e., optimal interpretations synthesized by our approach for \mathcal{S} do not overfit the set, and thus the guarantees obtained over \mathcal{S} can be adopted for \mathcal{S}_B ?*

The answer to the above question lies in the theory of *Probably Approximately Correct (PAC) Learnability* [32]. The notion of a *loss function*, ℓ , that must be minimized to obtain an optimal interpretation, is central to this discussion. For our purposes, the loss function may be viewed as $1 - \Delta_C$, where the range of the (normalized) correctness measure Δ_C is assumed to be $[0, 1]$. Thus for every $(i, o) \in \mathcal{I} \times \mathcal{O}$, and $f \in \mathcal{I} \rightarrow \mathcal{O}$, we define $\ell(f, (i, o)) = 1 - \Delta_C(f, \{(i, o)\})$. For technical reasons, we also assume that for every set \mathcal{S} of (i, o) samples, we have $\Delta_C(f, \mathcal{S}) = \frac{\sum_{(i, o) \in \mathcal{S}} \Delta_C(f, \{(i, o)\})}{|\mathcal{S}|}$. This is true, for example, if Δ_C is the prediction accuracy (the loss function being the misprediction rate in this case). Note

that in this case, the loss function for the sample set \mathcal{S} is given by $\frac{\sum_{(i, o) \in \mathcal{S}} \ell(f, (i, o))}{|\mathcal{S}|} = 1 - \Delta_C(f, \mathcal{S})$.

A class of interpretations (or hypotheses) \mathcal{E} over inputs \mathcal{I} and outputs \mathcal{O} is said to be PAC-learnable with respect to the set $Z = \mathcal{I} \times \mathcal{O}$ and a loss function $\ell : (\mathcal{I} \rightarrow \mathcal{O}) \times Z \rightarrow [0, 1]$, if there exists a function $m_{\mathcal{E}} : (0, 1)^2 \rightarrow \mathbb{N}$ and a learning algorithm with the following property: For every $\epsilon, \delta \in (0, 1)$ and for every distribution D over Z , when running the learning algorithm on $m \geq m_{\mathcal{E}}(\epsilon, \delta)$ i.i.d. samples generated by D , the algorithm returns a hypothesis E such that, with probability (confidence) of at least $1 - \delta$, $L_D(f_E) - \min_{E' \in \mathcal{E}} L_D(f_{E'}) \leq \epsilon$, where $L_D(f_E) = \mathbb{E}_{z \sim D}[\ell(f_E, z)]$. Furthermore, choosing an interpretation $E \in \mathcal{E}$ that minimizes $\frac{\sum_{z \in \mathcal{S}} \ell(f_E, z)}{|\mathcal{S}|}$ suffices for the learning algorithm in the above definition [32].

It is known that every finite class of interpretations is PAC-learnable due to the uniform convergence property [32]. In fact, the sample complexity, i.e., the function $m_{\mathcal{E}}$, can be determined in such cases in terms of $|\mathcal{E}|$, δ and ϵ . Under the standard *realizability assumption*, i.e. assuming \mathcal{E} includes an interpretation E such that f_E implements the semantic function f_B of the black-box, $m_{\mathcal{E}}$ is bounded above by $\lceil \frac{\log(|\mathcal{E}|/\delta)}{\epsilon} \rceil$. This bound increases to $\lceil \frac{2 \log(2|\mathcal{E}|/\delta)}{\epsilon^2} \rceil$ if we do not make the realizability assumption [32].

From the results above, if we use the $m_{\mathcal{E}}$ bound for the sample size, we get interpretations that are very close to the optimal interpretation within the class \mathcal{E} with high probability. Of course, sans the realizability assumption, this does not necessarily mean the obtained interpretation is very close to the black-box model. The latter depends highly on the class of interpretations. Note also that the price for the PAC guarantee is that we may have to work with an increased size of the sample set \mathcal{S} , as given by $m_{\mathcal{E}}$. In general, this affects the scalability of our synthesis procedure, since size of the weighted MAXSAT formula increases linearly with $|\mathcal{S}|$. This can limit how small δ and ϵ can be in practice. Nevertheless, as we show in Section V, we are able to use fairly small values of δ and ϵ in our experiments.

V. EVALUATION

a) Benchmarks: We apply our approach to three black-box models: a *decision module* for predicting the performance of a perception module in an airplane (AP), a *bank loan predictor* (BL), and a *solvability predictor* (TP).

The decision module predicts, based on the time of day, the cloud types, and initial positioning of an airplane on a runway, whether a perception module used by the plane can be trusted to behave correctly. The decision module is an implementation of a decision tree that was trained on data collected from 200 simulations, using the XPlane (x-plane.org) simulator.

The bank loan predictor is a deep neural network that was trained on synthetic data that we created. The training set included 100000 entries chosen such that majority of people with age between 18 to 29 years, and those with age between 30 and 49 years but with income less than \$6000, were denied the loan. The network has five dense fully connected hidden

layers with 200 ReLU’s each, in addition to a softmax layer and the output layer comprised of two nodes.

The solvability predictor is a neural network built to predict the solvability of first-order formulas by a theorem prover with respect to percentage of unit clauses and average clause length in a formula. The network had three hidden dense fully connected layers each with 200 ReLU’s. The data used to train the neural network can be found on the UCI machine learning repository [8]. We used the data for heuristic H1 from [8], thus predicting solvability for H1.

b) Experiments and setup: We conducted two types of experiments: (1) application of our exploration algorithm on the three benchmarks (2) performance evaluation of QUINTSYNT. The MaxSAT engine used an implementation of RC2 in PySAT [16], [17]. All experiments were conducted on a 2.4GHz Quad-core machine with 8GB of RAM. For additional details of the experiments and results, please see [31].

c) Exploring the Pareto-optimal space: We ran our approach on the three benchmarks mentioned above. We used confidence measure $\delta = 0.05$ and error margin $\epsilon = 0.05$ to determine the size of the sample set (as given in Table I) under the realizability assumption referred to in Section IV. Figures 3(a) to 3(c) show the measures of the Pareto-optimal interpretations found by our exploration algorithm. We used prediction accuracy for correctness (recall this satisfies the technical assumption mentioned in Section IV), and an explainability measure that favored decision diagrams of smaller size with predicates having a fewer number of branchings.

For all three benchmarks we found a variety of interpretations with interesting tradeoffs between the correctness and explainability measures, reflected by the blue squares in each plot. The exploration algorithm shows that searching for interpretations that are optimal only in size or in accuracy may result in unfavorable solutions. For example, in Figure 3(a) we see that the interpretation with highest accuracy has very low explainability. However, a very small tradeoff in accuracy resulted in significantly more explainable interpretations.

d) Performance: Table I presents our results on each benchmark and gives the confidence value δ , error rate ϵ and the number of samples $|S|$ used for each run. The number of Pareto-optimal points (PO), total number of points explored (TNP) and minimum, maximum and median times to find a Pareto-optimal interpretation are also shown. The number shown in parenthesis next to each benchmark is the number of predicates used. From Table I we can see that the number of Pareto-optimal (PO) points is considerably smaller than the total number of points explored (TNP). The minimum time taken to find an interpretation was less than 3 seconds for all benchmarks, but there were a few points in the Pareto-optimal space where finding an interpretation took considerably more time (see the maximum times). For most Pareto-optimal points though, the time taken to find an interpretation was less than 20 seconds, as demonstrated by the median values. If an interpretation did not exist for a combination of correctness and explainability measures, the MaxSAT solver returned UNSAT in less than a second in all performance runs.

TABLE I
PERFORMANCE OF QUINTSYNT: EXPLORATION OF THE ENTIRE
PARETO-OPTIMAL SPACE

Bench mark	δ, ϵ	$ S $	Explored (PO, TNP)	min time (s)	max time (s)	median time (s)	unsat time (s)
Theorem Prover (6)	0.05, 0.05	338	4, 20	0.767	3.392	1.138	< 1
	0.05, 0.03	703	3, 28	2.051	18.148	3.643	< 1
Air plane (3)	0.05, 0.05	333	7, 25	1.709	388.527	5.696	< 1
	0.05, 0.03	555	5, 26	2.513	616.520	11.222	< 1
Bank Loan (4)	0.05, 0.05	365	7, 27	1.927	387.599	8.975	< 1
	0.05, 0.03	608	4, 27	2.855	1299.196	17.998	< 1

As none of the other interpretation synthesis tools in the literature compute the set of all Pareto optimal interpretations, we omit comparison with other tools (any such comparison wouldn’t be fair, especially when using different notions for explainability). However, to understand if the variation in running times is inherent to the problem, we performed a similar experiment with MinDS, a tool for learning decision sets [38]. In MinDS, correctness and explainability are combined in a single objective and the contribution of the explainability measure is governed by a parameter λ . We ran MinDS for 15 values of λ and found interpretations for all these values. We observed again (Table II) that the time taken to find interpretations for some λ was much more than others.

Note that unlike in our approach, running MinDS in this manner does not guarantee that the entire Pareto-optimal space of interpretations has been obtained. Finding all Pareto optimal points by varying the weights of explainability and correctness measures is also not feasible, since this requires trying out all (infinitely many) weight combinations. While some decision sets learned by MinDS were indeed semantically equivalent to some of the Pareto-optimal interpretations synthesized by our approach, some interpretations that our methods found did not have a decision set counterpart within the range of weights we experimented on. We emphasize that running approaches like MinDS that combine explainability and correctness measures into single objective function may result in the same interpretation being returned for different combinations of weights. This can be avoided using our exploration method.

TABLE II
ILLUSTRATING VARIATION IN RUNNING TIMES EVEN ON
NON-EXHAUSTIVE PARETO SEARCH WITH MINDS

Bench mark	δ, ϵ	$ S $	min time (s)	max time (s)	median time (s)
Theorem Prover (6)	0.05, 0.05	338	0.707	0.813	0.719
	0.05, 0.03	703	0.687	0.798	0.725
Air plane (3)	0.05, 0.05	333	0.771	364.456	7.603
	0.05, 0.03	555	0.748	757.639	9.687
Bank Loan (4)	0.05, 0.05	365	0.744	25.819	1.165
	0.05, 0.03	608	0.738	52.388	0.841

VI. RELATED WORK

There is a large body of work on interpreting black-box models, where a dominant paradigm is to generate labeled data samples and obtain an interpretable model representation in terms of input features, some of which were discussed in the introduction. In some applications, the aim is to explain the

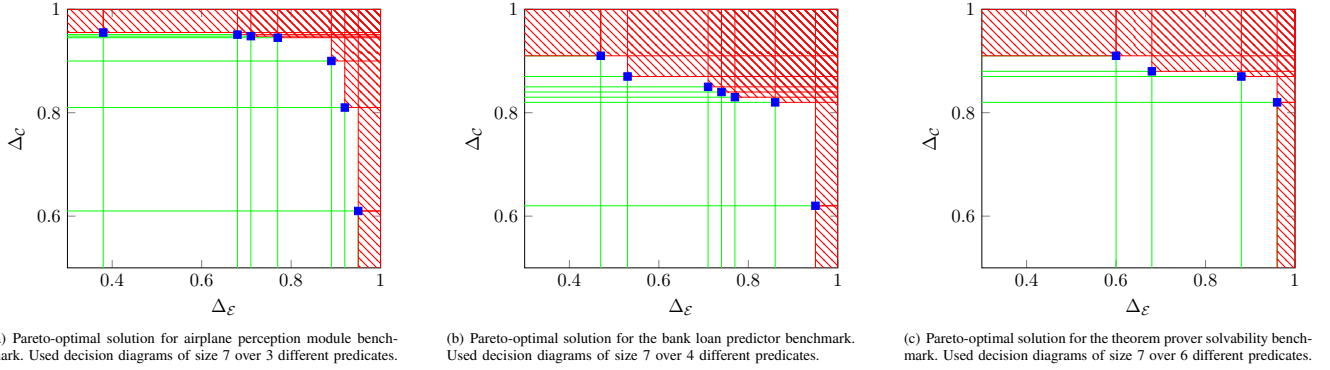


Fig. 3. Exploring Pareto-optimal solutions for three benchmarks. The size of the sample sets used for constructing interpretations was computed based on confidence values $\delta = 0.05$ and error margin $\epsilon = 0.05$, as well as the size of the class of interpretation in each benchmark.

output of a black-box model in the neighbourhood of a specific input, and specialized techniques [12], [24], [29], [30], [39] give such local and robust explanations. Other applications use techniques like model distillation (in the form of decision trees [7], [9], [20], [22], [23]), counterfactual explanations [26] etc. For further information on these techniques, we refer to reader to the excellent surveys in [2], [13].

The work in [15], [38] comes closest to ours. In [38], the authors encode the problem of finding an interpretation as optimal decision sets (to a weighted MAXSAT formulation). They present two variants: (i) optimize on accuracy (100%) while constraining the explainability (number of literals), and (ii) directly minimize the size of decision sets at the cost of accuracy. In [15], sparse optimal decision trees are built using an objective function that combines misclassification rate and number of leaves. Solution approaches like these give a single point of the optimized function in the Pareto-optimal space and hence a single value for the correctness and explainability measures.

Our Pareto-optimal interpretation synthesis problem formulation can also be related to Structural Risk Minimization (SRM), which is well-studied in the literature. Like in SRM, we have two orthogonal measures – one that depends only on the structure/complexity of the hypothesis/interpretation, and the other that depends on how well the hypothesis/interpretation “explains” the given sample set. The SRM formulation (e.g., see [32], Section 7.2) effectively combines these two measures into one and treats the problem as a single-objective optimization problem. In contrast, our Pareto-optimal synthesis problem is inherently a multi-objective optimization problem. As mentioned in the introduction, such a multi-objective optimization problem cannot be reduced to a single-objective optimization problem in general, without potentially excluding some (possibly important) solutions.

Finally, we note that the idea of using SAT (and related) solvers for systematically searching for all Pareto-optimal points has been used in other settings earlier (see, for example, systems biology applications in [4], [14]). However, their use in finding Pareto-optimal interpretations for black-box ML components appears not to have been explored earlier.

VII. CONCLUSION AND FUTURE WORK

We have presented a new approach to automatically generate a complete set of Pareto-optimal interpretations for black-box ML models, which works in the absence of training or test data sets. Our interpretations are obtained by instantiating user-provided decision diagram templates, and satisfy optimality conditions, while also providing formal guarantees on the tradeoff between accuracy and explainability. We have presented an empirical evaluation demonstrating that our approach produces compact, accurate explanatory interpretations for neural networks used for applications such as autonomous plane taxiing, predicting bank loans and classifying theorem-provers. The discovery of multiple Pareto-optimal interpretations, as opposed to a single one, demonstrates the value of the multi-objective approach.


The current work focuses on finite classes of possible interpretations, although we allow a class to be combinatorially large. The weighted MAXSAT encoding allows us to solve this problem symbolically by leveraging significant recent advances in MaxSAT solving that scale to very large solution spaces. Using a finite, yet large hypothesis class permits us to strike a balance between generality and practical efficiency of our approach. An interesting avenue for futurework would be to see if our approach can be extended to interpretation classes of infinite cardinality but finite Vapnik-Chervonenkis (VC) dimension. While the overall problem formulation, the notions of Pareto-optimality of explanations, and our algorithm for finding representative sets of explanations easily adapt to this setting, we would need to go beyond the current weighted MAXSAT formulation to find individual Pareto-optimal interpretations. Using an optimization modulo theories (OMT) encoding is a promising direction for such a generalization.


Acknowledgments. This work is partially supported by NSF grants 1545126 (VeHICaL), 1646208 and 1837132, by the DARPA contracts FA8750-18-C-0101 (AA) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, and by Toyota under the iCyPhy center. We would also like to express our gratitude to the anonymous reviewers for their in-depth reviews, constructive suggestions and various pointers.


REFERENCES

- [1] General Data Protection Regulation (GDPR). <https://gdpr.eu/>, 2018.
- [2] Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on Explainable Artificial Intelligence (XAI). *IEEE Access*, 6:52138–52160, 2018.
- [3] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning Optimal Decision Trees Using Caching Branch-and-Bound Search. In *AAAI 2020*, pages 3146–3153. AAAI Press, 2020.
- [4] S. Akshay, Sukanya Basu, Supratik Chakraborty, Rangapriya Sundarajan, and Prasanna Venkatraman. Functional Significance Checking in Noisy Gene Regulatory Networks. In *Principles and Practice of Constraint Programming*, pages 767–785, 2019.
- [5] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning. *Nature biotechnology*, 2015.
- [6] Florent Avellaneda. Efficient Inference of Optimal Decision Trees. In *AAAI 2020*, pages 3195–3202. AAAI Press, 2020.
- [7] Olcay Boz. Extracting Decision Trees from Trained Neural Networks. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, New York, NY, USA, 2002. Association for Computing Machinery.
- [8] James P. Bridge, Sean B. Holden, and Lawrence C. Paulson. Machine Learning for First-Order Theorem Proving - Learning to Select a Good Heuristic. *J. Autom. Reasoning*, 53(2):141–172, 2014. <https://archive.ics.uci.edu/ml/datasets/First-order+theorem+proving>.
- [9] Mark W. Craven and Jude W. Shavlik. Extracting Tree-Structured Representations of Trained Networks. In *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS '95, page 24–30. Cambridge, MA, USA, 1995. MIT Press.
- [10] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3422–3426. IEEE, 2013.
- [11] Daniel J. Fremont, Johnathan Chiu, Dragos D. Margineantu, Denis Osipychyev, and Sanjit A. Seshia. Formal analysis and redesign of a neural network-based aircraft taxiing system with VeriFAL. In *32nd International Conference on Computer Aided Verification (CAV)*, July 2020.
- [12] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Dino Pedreschi, Franco Turini, and Fosca Giannotti. Local Rule-Based Explanations of Black Box Decision Systems. *CoRR*, abs/1805.10820, 2018.
- [13] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A Survey of Methods for Explaining Black Box Models. *ACM Comput. Surv.*, 51(5), August 2018.
- [14] Friedman A. M. He L. and Bailey-Kellogg C. A divide-and-conquer approach to determine the Pareto frontier for optimization of protein engineering. *Proteins*, 80(3):790–806, 2012.
- [15] Xiyang Hu, Cynthia Rudin, and Margo Seltzer. Optimal Sparse Decision Trees. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.
- [16] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.
- [17] Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient MaxSAT solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019.
- [18] Mikolás Janota and António Morgado. SAT-Based Encodings for Optimal Decision Trees with Explicit Paths. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020*, volume 12178 of *Lecture Notes in Computer Science*, pages 501–518. Springer, 2020.
- [19] Susmit Jha, Tuhin Sahai, Vasumathi Raman, Alessandro Pinto, and Michael Francis. Explaining AI Decisions Using Efficient Methods for Learning Sparse Boolean Formulae. *J. Autom. Reasoning*, 63(4):1055–1075, 2019.
- [20] U. Johansson and L. Niklasson. Evolving decision trees using oracle guides. In *2009 IEEE Symposium on Computational Intelligence and Data Mining*, pages 238–244, 2009.
- [21] Eric Knorr. How PayPal beats the bad guys with machine learning. <http://www.infoworld.com/article/2907877/machine-learning/how-paypal-reduces-fraud-with-machine-learning.html>, 2015.
- [22] R. Krishnan, G. Sivakumar, and P. Bhattacharya. Extracting decision trees from trained neural networks. *Pattern Recognition*, 32(12):1999 – 2009, 1999.
- [23] Sanjay Krishnan and Eugene Wu. PALM: Machine learning explanations for iterative debugging. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, HILDA'17, New York, NY, USA, 2017. Association for Computing Machinery.
- [24] Scott M Lundberg and Su-In Lee. A Unified Approach to Interpreting Model Predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.
- [25] Douglas Merrill. AI is coming to take your mortgage woes away. <https://www.forbes.com/sites/douglasmerrill/2019/04/04/ai-is-coming-to-take-your-mortgage-woes-away/>, April 2019.
- [26] Christoph Molnar. *Interpretable Machine Learning*. 2019. <https://christophm.github.io/interpretable-ml-book/>.
- [27] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and João Marques-Silva. Learning Optimal Decision Trees with SAT. In Jérôme Lang, editor, *International Joint Conference on Artificial Intelligence, IJCAI 2018*. ijcai.org, 2018.
- [28] NVIDIA. Nvidia tegra drive px: Self-driving car computer, 2015.
- [29] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “Why Should I Trust You?”: Explaining the Predictions of Any Classifier. In *Knowledge Discovery and Data Mining*, KDD '16. Association for Computing Machinery, 2016.
- [30] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-Precision Model-Agnostic Explanations. In *AAAI Conference on Artificial Intelligence*, 2018.
- [31] Hazem Torfah Shetal Shah, Supratik Chakraborty, S. Akshay, and Sanjit A. Seshia. Synthesizing pareto-optimal interpretations for black-box models. *CoRR arXiv*, abs/2108.07307, 2021.
- [32] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, USA, 2014.
- [33] Justin Sirignano, Apaar Sadhwani, and Kay Giesecke. Deep learning for mortgage risk, 2016.
- [34] Pang-Ning Tan, Michael S. Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.
- [35] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning Optimal Decision Trees using Constraint Programming (extended abstract). In Christian Bessière, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4765–4769. ijcai.org, 2020.
- [36] Sicco Verwer and Yingqian Zhang. Learning Decision Trees with Flexible Constraints and Objectives Using Integer Optimization. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming*, pages 94–103. Cham, 2017. Springer International Publishing.
- [37] Sicco Verwer and Yingqian Zhang. Learning Optimal Classification Trees Using a Binary Linear Program Formulation. In *AAAI 2019*, pages 1625–1632. AAAI Press, 2019.
- [38] Jinqiang Yu, Alexey Ignatiev, Peter J. Stuckey, and Pierre Le Bodic. Computing Optimal Decision Sets with SAT. In *Principles and Practice of Constraint Programming*, pages 952–970. Cham, 2020. Springer International Publishing.
- [39] Xin Zhang, Armando Solar-Lezama, and Rishabh Singh. Interpreting Neural Network Judgments via Minimal, Stable, and Symbolic Corrections. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 4874–4885. Curran Associates, Inc., 2018.

Dynamic Partial Order Reductions for Spinloops

Michalis Kokologiannakis
MPI-SWS
Kaiserslautern, Germany
michalis@mpi-sws.org 

Xiaowei Ren
The University of British Columbia
Vancouver, Canada
xiaowei@ece.ubc.ca 

Viktor Vafeiadis
MPI-SWS
Kaiserslautern, Germany
viktor@mpi-sws.org 

Abstract—Stateless model checking (SMC) coupled with dynamic partial order reduction (DPOR) is an effective way for automatically verifying safety properties of loop-free concurrent programs. SMC, however, does not work well for programs with loops because it cannot distinguish loop iterations that make progress from ones that revisit the same state. This results in redundant exploration that dominates the verification time.

We present SAVER (Spinloop-Aware Verifier), a memory-model-agnostic SMC/DPOR extension that detects *zero-net-effect* spinloops and avoids redundant explorations that lead to the same local state. As confirmed by our experiments, SAVER achieves an exponential reduction in verification time and outperforms state-of-the-art tools in a variety of real-world benchmarks.

Index Terms—stateless model checking, spinloops

I. INTRODUCTION

Stateless model checking (SMC) [1] is a prominent technique for verifying safety properties of concurrent programs, especially under weak memory consistency [2]–[6]. The key design choice that makes SMC scale is that it does not record the set of states explored, but rather uses alternative techniques, namely *dynamic partial order reduction* (DPOR) [7], [8], to avoid exploring the same state multiple times. The downside of this choice, however, is that SMC struggles with spinloops, i.e., loops that continuously read a shared variable until some condition holds: as SMC does not record the set of visited program states, it cannot distinguish loop iterations that make progress from those that return to the same state. To make matters even worse, such loops are ubiquitous in real-world concurrent programs, whether lock-based or lock-free.

Consequently, spinloops typically have to be *bounded*. Since bounding generally sacrifices the soundness of the verification, one would like to use fairly large loop bounds to be confident enough that the program verified is correct. Doing so, however, is practically infeasible. A loop bound of $N \geq 2$ typically leads to an exponential blowup in the state space, since the model checker explores the possibility of each spinloop failing 0, 1, ..., $N - 1$ times and, for each failure, all possible stores from which the spinloop loads(s) can read.

To avoid the blowup, the solution is to use a bound of $N = 1$. So far, this is typically done manually by rewriting the program to use **assume** statements (a.k.a. **await**), special verifier commands that block the execution of the relevant thread when the condition of the **assume** is violated.

The goal of this paper is to determine *conditions* under which it is sound to do such conversions automatically. As we shall see, this turns out to be quite challenging.

First, spinloops cannot be adequately detected by a simple syntactic criterion. Since programming languages have many ways of creating spinloops (e.g., while loops, repeat-until loops, for-loops, goto statements), their detection is best done after converting each program thread into a *control-flow graph* (CFG). However, even there, simply removing the CFG backedges for side-effect-free loops (i.e., loops with no stores to global variables or to local variables that are live at the loop header) is insufficient, as illustrated by the program below. As a convention, in our examples, we use x, y, z for global (shared) variables and a, b, c, \dots for registers.

```
do  a := x  ||  b := x
while (a ≠ 0) || while (b ≠ 0) b := x  (LOOP-PEEL)
```

While the loop in thread I can be easily bounded by converting it into $a := x; \mathbf{assume}(a = 0)$, the one in thread II cannot because b is “live” at the header of the loop (its value is used in the loop).

Second, some spinloops may have side-effects, but these either do not occur on all their iterations or are never observed by the other threads (e.g., writing to a global variable that is not concurrently read) or cancel each other out (e.g., incrementing and then decrementing a variable, acquiring and releasing a lock). As an example of the latter kind, consider the following *zero-net-effect* (ZNE) spinloops extracted from a lock implementation.

```
while (true)           while (true)
  a := fetch_add(x, 1)  b := fetch_add(x, 1)
  if (a = 0) break      if (b = 0) break
  fetch_add(x, -1)      fetch_add(x, -1)
// critical section    // critical section
fetch_add(x, -1)        fetch_add(x, -1)
                        (INC-DEC-SPIN)
```

Each thread tries to acquire the lock by incrementing x . If the lock was already taken, it decrements x and tries again. The lock is finally released by decrementing x . Since each decrement cancels out the previous increment, we would like to avoid considering loop iterations with a decrement, i.e., unsuccessful lock acquisition attempts. The soundness of doing so depends on the context. If, for instance, there is another thread repeatedly reading x , it may observe the value of x flickering, which cannot happen if we bound the ZNE loops to a single iteration. Similarly, if another thread writes to x concurrently, the loop may no longer have a zero net effect, rendering the transformation unsound.

To address these challenges, we develop SAVER (Spinloop-Aware Verifier), a model checker that reduces spinloops to a single iteration. SAVER works at the level of reduced control flow graphs, obtained by merging bisimilar nodes. Whenever a spinloop cannot be shown to be side-effect-free statically, SAVER dynamically checks that the reduced spinloop iterations have a zero net effect (in particular, that the context does not observe any of their effects), and if the check fails, it rolls back the transformation.

We remark that our results are independent of the *memory consistency model*: they hold not only for sequential consistency (SC), but also for weak memory models, which admit executions that cannot be expressed as program interleavings.

II. PRELIMINARIES

In this section, we review how programs can be represented as control flow graphs (§ II-A), how their executions can be modeled as execution graphs (§ II-B), and how DPOR enumerates these executions (§ II-C).

A. Control Flow Graphs

To avoid cluttering the presentation, we omit all features irrelevant to loops and concurrency. We represent a concurrent program P as a top-level parallel composition of threads, each of which is modeled as a control-flow graph (CFG). A CFG is a directed graph whose nodes are program labels and whose edges are labeled with instructions of the following form:

$$\text{Inst} \ni i ::= r := e \mid \mathbf{error} \mid \mathbf{assume}(e) \mid r := x \mid x := e \mid r := \text{fetch_add}(x, e) \mid r := \text{CAS}(x, e_1, e_2)$$

where r ranges over registers (i.e., local variables), x over global (shared) variables, and e over simple expressions built from integer constants n , registers, and arithmetic operators:

$$\text{Exp} \ni e ::= n \mid r \mid e_1 + e_2 \mid e_1 - e_2 \mid \dots$$

Instructions comprise plain assignments; **error**, that halts the program (e.g., due to a safety violation); **assume**(e), that blocks the calling thread if e has the value zero; and memory accesses. Memory accesses include $r := x$, that reads the value of x and stores it in r ; $x := e$, that stores the value contained in e in the global variable x ; $r := \text{fetch_add}(x, e)$ (fetch-and-increment) that atomically increments the value of x by the value of e and returns the old value to r , and $r := \text{CAS}(x, e_1, e_2)$ (compare-and-swap), that atomically compares the value stored in location x with the value of e_1 , and if they are equal, replaces the value of x with the value of e_2 . The $r := \text{CAS}(x, e_1, e_2)$ instruction always returns the result of the comparison in r . We also use the term *load instruction* to refer to $r := x$, $r := \text{CAS}(x, e_1, e_2)$, and $r := \text{fetch_add}(x, e)$ instructions, while we use *store instruction* to refer to $x := e$, $r := \text{CAS}(x, e_1, e_2)$, and $r := \text{fetch_add}(x, e)$ instructions.

We assume that input programs are deterministic in that each node n either has at most one successor (for standard program statements), or it has two successors labeled with **assume**(e) and **assume**($\neg e$) respectively (for conditionals and loops). As an example, Fig. 1 shows the CFGs for the

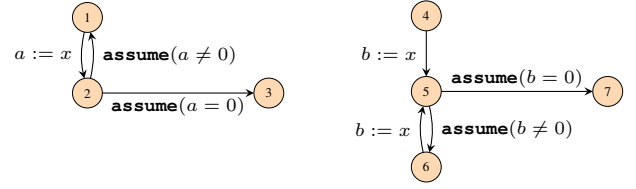


Fig. 1. CFGs for the two threads of LOOP-PEEL.

two threads of the LOOP-PEEL program from §I. The loops generate cycles in the CFGs, and the conditional tests (whether to execute another loop iteration or to exit the loop) generate the edges labeled with **assume** statements.

A *path* π in a CFG is an alternating sequence of nodes and instructions corresponding to edges in the CFG, starting and ending with a node. That is, π is of the form $n_1 i_1 n_2 i_2 n_3 \dots n_{k-1} i_{k-1} n_k$ where (n_j, i_j, n_{j+1}) is an edge in the CFG for all $1 \leq j < k$. As it is common in the literature, we are primarily interested in *simple paths*, which do not visit the same node twice, except possibly by their last node. A (simple) path is *cyclic* if it starts and ends with the same node, while a *lasso* path is one whose end node is one of its intermediate nodes. We write $|\pi|$ to denote the length of the path (i.e., the number of edges it contains), and $\pi(k)$ to project the k^{th} node and/or instruction of the path.

We say that node a *dominates* b if all paths from the entry node of the CFG to b contain a . Given a path π in a CFG, we say that a node h of π is its *header* if it dominates all nodes in π . By definition, paths can have at most one header; in the case of reducible graphs, every cyclic path has a header. For example, in Fig. 1, nodes 1 and 5 are the headers of the two cyclic paths, respectively.

A *loopy path* is a simple path that starts and ends at its header. Formally, a simple path π is called a *loopy path* of an edge $n \rightarrow h$ if $\pi(1) = \pi(|\pi|) = h$ and $\pi(|\pi| - 1) = n$ and h dominates all nodes in π (i.e., h is a header of π).

B. Execution Graphs

In order to keep our approach as general as possible, we follow the standard axiomatic approach of Alglave *et al.* [9] and represent the executions of a concurrent program as *execution graphs*. Using execution graphs allows us to keep our formalism memory-model-agnostic, as our contributions do not depend on a particular memory consistency model.

Execution graphs have two basic components:

- (i) a set of events (nodes), that represent the memory accesses performed by the program, and
- (ii) some relations on these events (edges), such as the *program order*, which relates events in the same thread, and the *reads-from* relation, which relates reads to writes they are reading from.

The semantics of a program P is given by the set of execution graphs that correspond to the instructions of the program and satisfy the consistency predicate of the underlying memory model. The purpose of the consistency predicate is to rule

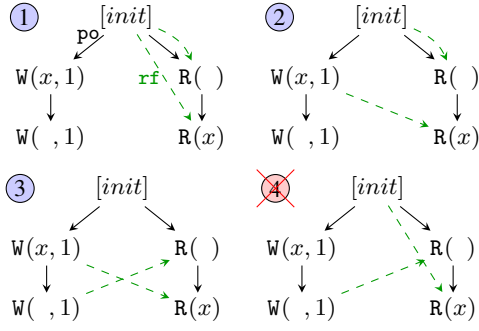


Fig. 2. MP: three consistent execution graphs under SC.

out executions with nonsensical edges, such as a load reading from a store later in program order or a store that has been overwritten by another store before the load.

To see how execution graphs model the executions of a program, consider the following example:

$$\begin{array}{l} x := 1 \parallel a := y \\ y := 1 \parallel b := x \end{array} \quad (\text{MP})$$

Under SC, the MP program has three consistent executions, shown in Fig. 2, where the solid edges represent the program order and the green dashed edges the reads-from relation. As can be seen, execution ④ is inconsistent under SC—the consistency predicate of SC forbids the load of x to read from the initial state as the load is already aware of the $x := 1$ store. This execution, however, is allowed under certain weak memory models, such as the ‘relaxed’ fragment of RC11 [10].

Let us now formally describe events and execution graphs. For a more extensive discussion regarding execution graphs, we refer interested readers to Kokologiannakis *et al.* [5].

Definition 1. An event, $e \in \text{Event}$, is either an initialization event $\langle \text{init } l \rangle$ for a location $l \in \text{Loc}$ or a thread event $\langle t, i, \text{lab} \rangle$ where $t \in \text{Tid}$ is a thread identifier, $i \in \text{Idx} \triangleq \mathbb{N}$ is a serial number inside each thread, and $\text{lab} \in \text{Lab}$ is a label that takes one of the following forms:

- Read label: $R(l)$ where $l \in \text{Loc}$ is the location accessed.
- Write label: $W(l, v)$ where $l \in \text{Loc}$ is the location accessed, and $v \in \text{Val} \triangleq \mathbb{Z}$ is the value written.
- Error label: **error**.
- Blocked label: **blocked**, generated by **assume**(e) statements when e is false.
- ZNE label: **ne**(x), which is used to mark ZNE loops.

Definition 2. An execution graph G consists of:

- 1) a set $G.E$ of events that includes initialization events for all locations accessed by the program, and
- 2) a function $G.\text{rf}$, called the reads-from map, that maps each read event of G to a same-location write event of G from where it gets its value.

Our formal definition of execution graphs does not record the program order (po) as an explicit component because it

Algorithm 1 Dynamic Partial Order Reduction

```

1: procedure VERIFY( $P$ )
2:    $\langle G, \Gamma \rangle \leftarrow \langle G_\emptyset, \Gamma_\emptyset \rangle$ 
3:   do
4:     VISITONE( $P, G, \Gamma$ )
5:   while  $\langle G, \Gamma \rangle \leftarrow \text{pop}(\Gamma)$ 

6: procedure VISITONE( $P, G, \Gamma$ )
7:   while  $\text{consistent}(G) \wedge a \leftarrow \text{next}_P(G)$  do
8:      $G.E \leftarrow G.E \uplus \{a\}$ 
9:     if  $a \in \text{error}$  then exit(“error”)
10:    else if  $a \in R$  then
11:      let  $\{w_0\} \uplus ws = G.E \cap W_{\text{loc}(a)}$ 
12:       $G \leftarrow \text{SetRF}(G, w_0, a)$ 
13:       $\Gamma \leftarrow \text{push}(\Gamma, \{\text{SetRF}(G, w, a) \mid w \in ws\})$ 
14:    else if  $a \in W$  then
15:      CALCREVISITS( $G, \Gamma, a$ )
16:    CHECKZNEVALIDITY( $G, a$ )

```

can be defined directly from our representation of events:

$$\text{po} \triangleq \{ \langle \langle \text{init } l \rangle, \langle t, i, \text{lab} \rangle \rangle \mid \forall l, t, i, \text{lab} \} \cup \{ \langle \langle t_1, i_1, \text{lab}_1 \rangle, \langle t_2, i_2, \text{lab}_2 \rangle \rangle \mid t_1 = t_2 \wedge i_1 < i_2 \}$$

Initialization events precede all non-initialization events in po, while events in the same thread are ordered according to their serial numbers. Events from different threads are unordered.

C. Dynamic Partial Order Reduction

DPOR verifies a program by generating all of its consistent execution graphs and checking that none of them contains an error. To do so, DPOR typically assumes some basic properties of the consistency predicate, such as prefix-closedness and extensibility [5], which are satisfied by all known memory models that follow the graph representation of § II-B.

This graph representation is also very helpful for DPOR because it encodes the independence relation that is traditionally used by DPOR algorithms to decide which interleavings should be explored. Indeed, under sequential consistency, each graph corresponds to the set of thread interleavings that are equivalent under the *reads-from equivalence* [11], [12] (or under Mazurkiewicz equivalence if we extend the graphs to also record the coherence order).

Algorithm 1 shows the general structure of a DPOR algorithm. The procedure VERIFY verifies a concurrent program P by starting from the graph G_\emptyset containing only the initialization events and an empty environment Γ_\emptyset (Line 2), and exploring the executions of P one by one by calling VISITONE (Line 4). VISITONE does most of the exploration work: it explores one full execution of P and populates Γ with alternative exploration options. These exploration options recorded in Γ are later explored by VERIFY (Line 5).

At each step, VISITONE extends the current execution G by one event a (obtained via $\text{next}_P(G)$), as long as G remains consistent according to the memory model (Line 7). If there

are no more events to add, then G is complete, and VISITONE returns. If a denotes an error (e.g., an assertion violation), it is reported to the user and verification terminates (Line 9).

If a is a read, then it must read from some write in G . To this end, VISITONE calculates the set of all writes in G on the same location as a (Line 11), and chooses one write w_0 as the reads-from option for a (Line 12). For all other same-location writes, an alternative execution is added to Γ so that it can be explored later by VERIFY (Line 13).

If a is a write, it needs to revisit existing reads of the same location in G , because a was not present in the graph when VISITONE was considering possible reads-from options for these reads. To that end, VISITONE calls CALCREVISITS (Line 15), which extends Γ with such alternative explorations. Since the discussion on how these explorations are calculated is not relevant for this paper, we do not present it here; we refer interested readers to Kokologiannakis *et al.* [5], where CALCREVISITS is explained in detail.

Note that Algorithm 1 does not have any special treatment for **assume** statements. Whenever $\text{next}_p(G)$ encounters an **assume** statement whose condition is not satisfied, it returns a blocked event and stops scheduling that thread thereafter. When VERIFY later pops some graph that does not contain the blocked label (e.g., because the graph represents an alternative exploration choice before the blocked event), the thread will be again schedulable, and other options that might not block the **assume** will be considered.

III. BOUNDING EFFECT-FREE SPINLOOPS

Effect-free loop iterations that do not exit the loop are almost unobservable: they do not affect the set of reachable program states, and so can be ignored when verifying safety properties of a program. (We note that for liveness properties, effect-free loop iterations cannot be discarded that simply. An infinite sequence of such effect-free iterations, unless prevented by some fairness assumption about the program's semantics, yields a non-terminating run of the program.)

What remains to be clarified is what exactly constitutes an effect-free loop iteration. Clearly, the iteration should not be writing to a global variable, as otherwise other threads may be able to observe whether the iteration took place or not. Similarly, it should also not be assigning to any local registers that could affect the subsequent execution of the thread itself, i.e., to any variables that are *live* at the header of the loop. Assigning to a dead variable is harmless because, by definition, it does not affect the subsequent execution of the thread, even if technically it might reach a slightly different local state (differing only in the values of dead variables).

We note that spinloops need to be effect-free only along looping paths—they may well have side-effects on paths exiting the loop. This is frequently the case for CAS-loops, such as the following implementation of an atomic increment:

```
do
  a := x
  success := CAS(x, a, a + 1)      (CAS-LOOP)
while (¬success)
```

```
while (true)
  h := head
  t := tail
  n := next[h]
  h' := head
  if (h ≠ h') continue
  if (h = t)
    if (n) break
    CAS(tail, t, n)
  else
    b := CAS(head, h, n)
    if (b) break
```

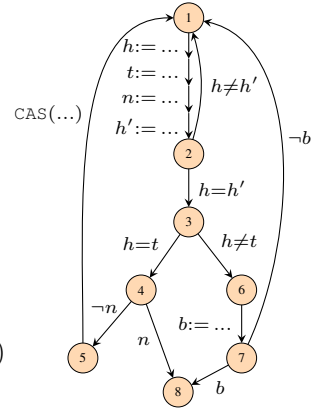


Fig. 3. Simplified dequeue operation from the *ms-queue* benchmark and its CFG, whose instructions are abbreviated. In the code, *hea*, *next*, and *tail* are global variables, while *b*, *h*, *h'*, *n*, and *t* are local registers.

Here, even though the loop contains a CAS, which is generally an effectful instruction, along the looping path, the CAS fails, and so the path is effect-free.

We also note that loops often have multiple looping paths, only some of which are effect-free. Consider, for instance, the **while** loop in Fig. 3, which is extracted from the *ms-queue* benchmark of §VIII. It contains three loopy paths. The first (through the **continue** statement) is trivially effect-free because it contains only loads and assignments to dead variables. (All local variables are dead at the loop header.) The second path (when $h = t$) can have side-effects—the CAS to *tail*. The third path (when $h \neq t$) is again effect-free because whenever its CAS succeeds, the function returns.

Let us now make these intuitions more formal. A path π is *pure* if it either contains no store instructions or, if it contains any, all of them are failed CASes. That is, whenever $\pi(i)$ is a store instruction, then it is of the form $r := \text{CAS}(x, e_1, e_2)$ and there is $i < j < |\pi|$ such that $\pi(j) = \text{assume}(\neg r)$ and for all $i < k < j$, $\pi(k)$ does not assign to r .

Pure paths do not affect the global state, but can affect the local state. A loopy path does not affect the local state if it always reaches the same local state it started from. A simple approximation to reaching the same state is for the path to not assign to any variable that is live at its header. Putting these conditions together, an *effect-free spinloop* is a pure loopy path that does not assign to any variable live at its header. Formally:

Definition 3. A CFG edge $n \rightarrow h$ is an effect-free spinloop backedge if every loopy path of $n \rightarrow h$ is pure and assigns only to registers dead at h .

The *spin-assume* transformation removes all effect-free spinloop backedges from the CFG. Returning to the example in Fig. 1, the edge $2 \rightarrow 1$ is an effect-free spinloop backedge; removing it transforms thread I of LOOP-PEEL into $a := x; \text{assume}(a = 0)$. In contrast, the backedge of thread II ($6 \rightarrow 5$) is not effect-free and so the spin-assume transformation does not affect thread II.

IV. DETECTING MORE KINDS OF SPINLOOPS

While the spin-assume transformation defined in the previous section can detect typical cases of **do-while** spinloops, it does not apply to **while** loops that have a non-trivial condition.

The main problem is that the registers used to evaluate the condition are live at the loop header, and so any loop iterations that update these registers are deemed effectful. As a simple example, consider the spinloop of thread II of LOOP-PEEL from §I: register b is live at the beginning of the loop, and so the body of the loop ($b := x$) is effectful. (Formally, in the CFG of Fig. 1, register b is live at node 5—the loop header.)

One simple way to resolve this problem is to apply a compiler transformation called *loop rotation*, which moves the loop exit checks to the end of the loop. Applying loop rotation transforms the second thread of LOOP-PEEL as follows:

```

 $b := x$ 
while ( $b \neq 0$ )
   $b := x$ 
  ~~~~~
   $b := x$ 
  if ( $b \neq 0$ )
    do  $b := x$  while ( $b \neq 0$ )

```

The transformed loop can be bounded with the spin-assume transformation yielding executions with at most two loads of x . We note that this bounding outcome is suboptimal, since thread I of LOOP-PEEL is bounded with a single load of x .

A better approach for this example is to exploit *bisimilarity* among CFG nodes. Two nodes are bisimilar if they produce the exact same computations, i.e., if their outgoing edges can be matched 1-to-1 in a way that every two matched edges are labeled with the same instruction and lead to bisimilar nodes. Bisimilarity can be computed as a greatest fixed point, starting with the identity relation (i.e., each node being bisimilar to itself) and adding pairs of nodes whenever they have matching outgoing edges to nodes already calculated to be bisimilar. For example, in Fig. 1, nodes 4 and 6 are bisimilar because they both have only one outgoing edge labeled with the same instruction ($b := x$) and leading to the same node (5).

Having detected that two (distinct) nodes a and b are bisimilar, we can then merge them into one node by redirecting b 's incoming edges to a and deleting node b . For example, merging nodes 4 and 6 of Fig. 1 would add an edge from 5 to 4 with label **assume**($b \neq 0$), and remove node 6. Effectively, this transformation converts the second thread of LOOP-PEEL to a **do-while** loop analogous to that in its first thread, which makes the spin-assume transformation applicable.

We note that merging bisimilar nodes is not always strictly better than loop rotation. There are cases where loop rotation (or a similar transformation called *jump threading*) can transform a loop into the **do-while** form, but no two distinct bisimilar nodes exist. Such cases frequently arise with **CAS** loops like the following.

```

success := false
while ( $\neg$ success)
   $a := x$ 
  success := CAS( $x, a, a + 1$ )

```

(CAS-LOOP2)

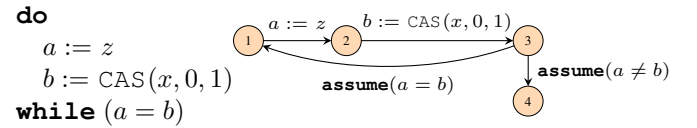
Here, the spin-assume transformation is not directly applicable to CAS-LOOP2 because *success* is live at the loop header

and is updated by the loop body. Loop rotation and/or jump threading, followed by dead assignment elimination, convert this program to CAS-LOOP, which can be handled by the spin-assume transformation. By contrast, merging bisimilar nodes does not change the program, since the program does not contain the same instruction twice.

V. DYNAMICALLY CHECKING PURITY

The spin-assume transformation as described in §III uses a completely static definition of purity. If a CAS along a CFG path cannot be determined to always fail, the path is deemed effectful. This is, however, suboptimal for two reasons.

First, using a static purity definition prevents us from transforming paths that are pure only under certain contexts. For instance, consider the thread below, and assume that it is running as part of a program that only writes the value 0 to z (this might not be inferable statically):



In this case, the (only) loop path of this thread will not be deemed pure (as the CAS is not followed by an **assume**($\neg b$) statement), even though it will never produce observable effects in its running context as a will always be 0.

Second, in cases where a loop path contains a CAS that *does* have observable effects, it is wasteful to explore executions where such a CAS fails. To see this, consider again the dequeue operation of the `ms-queue` example in Fig. 3. As explained in §III, the second loop path of this operation is not pure, as it potentially has side-effects. Still, it does not make sense to consider iterations where the CAS of this path fails, as they both do not contribute to the loop exiting, and they produce no observable side-effects.

Leveraging the insights above, we say that a CFG backedge $n \rightarrow h$ is a *potentially effect-free spinloop backedge* if every loop path of $n \rightarrow h$ assigns only to registers dead at h . The *dynamic-spin-assume transformation* marks all potentially effect-free spinloop backedges with a dynamic purity check. Whenever the $\text{next}_P(G)$ function of Algorithm 1 encounters such a check, it validates whether G contains any write event originating from the respective loop iteration and, if not, it returns a blocked event, thereby blocking the execution of the respective thread. Otherwise, if the loop iteration did generate a write event, $\text{next}_P(G)$ proceeds with the next event.

In fact, the dynamic purity check described above can be relaxed even further: SAVER allows loop iterations to contain write events, as long as these only affect memory locations that are not reachable by other threads. In turn, this proves very useful in cases where some initialization writes need to take place as part of a loop.

To see an example of this, consider the push operation of the `treiber-stack` benchmark (cf. Fig. 4). First, a node to be inserted to the stack is created, but this node cannot be initialized fully: its *next* field needs to point to the existing

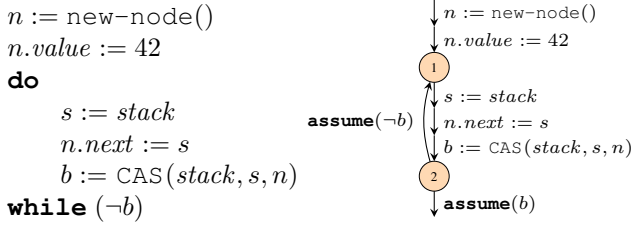


Fig. 4. Simplified push operation from the `treiber-stack` benchmark with its CFG: `stack` is a global variable, while `b`, `n`, and `s` are registers.

top of the stack, but the stack top might change between the time it is read, and the time the node is created. Thus, the push operation first reads the stack, sets it as the node's `next`, and then tries to atomically replace the stack with the newly created node. If the replacement succeeds, the operation exits; otherwise, it tries again. Notice, however, that, as long as the replacement CAS does not succeed, the store to the node's `next` remains unobserved by the other threads. Thus, it is safe to consider failed CAS loop iterations as effect-free, and block their exploration.

As a final remark, we observe that validating effect-free loops dynamically makes SAVER resilient to more aggressive loop rotation passes that convert loops to a canonical form containing a single backedge (see §VII).

VI. HANDLING ZERO-NET-EFFECT SPINLOOPS

Let us now consider the more challenging case of *zero-net-effect* (ZNE) loops. Recall that these are spinloop iterations that do have side-effects but (1) whose side-effects cancel each other out, and (2) whose intermediate effects are not observed by other threads. While condition (1) can be checked pretty well statically, condition (2) has to be checked dynamically. In the discussion below, we focus on ZNE loops that arise because of an atomic increment being followed by an atomic decrement of the same location and value.

A decrement instruction at node k is a *canceling decrement* in a loop h if all of h 's loopy paths that contain node k also contain a prior opposite increment instruction, and the paths are effect-free modulo two instructions. More formally:

Definition 4. A node k in a (minimal) CFG cycle with header h is a canceling decrement if it has a (unique) outgoing edge of the form $r_1 := \text{fetch_add}(x, n)$, and for every loopy path π of h such that $\pi(i) = k$ for some $1 < i < |\pi|$, there exists $j < i$ such that $\pi(j) = r_2 := \text{fetch_add}(x, -n)$ for some r_2 , and replacing the instructions at $\pi(i)$ and $\pi(j)$ with plain assignments to r_1 and r_2 yields an effect-free path.

SAVER's *spin-zne* transformation annotates all canceling decrements so that when $\text{next}_P(G)$ encounters them for the first time (cf. Algorithm 1, Line 7), it generates a $\text{ne}(x)$ event and blocks the thread instead of generating a read event and afterwards a write event. The $\text{ne}(x)$ event serves as a marker for SAVER to validate that the transformation is sound.

Validation of ZNE loops happens every time a new event e is added to the graph by calling the `CHECKZNEVALIDITY`

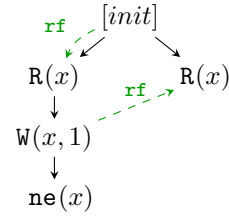


Fig. 5. Execution graph encountered during the exploration of ZNE-OBS.

routine (Algorithm 1, Line 16). If we use the pair $\langle w, z \rangle$ to represent a blocked ZNE loop iteration with w being the event corresponding to the increment of the ZNE loop and z being the ne event, the addition of e can render the reduction of the $\langle w, z \rangle$ loop unsound in one of the following two ways.

First, if e writes to the same location as w , it can be ordered (in coherence) between w and the blocked decrement (after z), and so, unless e is also an atomic increment, w and its corresponding decrement will no longer cancel each other out.

Second, if e reads from w and there is already some other read event reading from w , then, in an alternate execution, it is possible for e to read from the canceling decrement instead of w , thereby observing the value of the shared variable flickering. To see this, consider the example below.

<pre> while (true) a := fetch_add(x, 1) if (a = 42) break fetch_add(x, -1) </pre>	<pre> b := x if (b) c := x assert(c) </pre>	(ZNE-OBS)
---	---	-----------

Note that the loop of the first thread fulfills the conditions of a ZNE loop, and so the second `fetch_add()` will be annotated by the *spin-zne* transformation.

Figure 5 shows the execution graph arising from adding the events of thread I and then adding the read event corresponding to the $b := x$ instruction of thread II in the case it reads the incremented value of x . Next, we have to add the event corresponding to $c := x$. In this graph, the only consistent option for this event is to also read the incremented value of x , which satisfies the subsequent assertion. Yet, if we had the decrement of x instead of the ne event in the graph, c could also have read the value 0 from the decrement, and the `assert` would have failed. Thus, it is clear that concurrent reads can render the transformation of ZNE spinloops unsound.

Therefore, `CHECKZNEVALIDITY`(G, e) (cf. Algorithm 2) checks whether either of these two conditions holds for any existing $\text{ne}(x)$ event in the graph (where x is the location accessed by e), and if so, it removes the ne event(s) and unblocks the corresponding thread(s), which will eventually add the missing decrement event(s) and restore soundness.

Other cases of ZNE loops can be handled in a similar manner. For example, consider spinloops containing matching lock acquisitions and releases. In such a case, acquiring the lock acts as the increment operation and releasing the lock as the matching decrement. Statically, it therefore suffices to check that each lock release in the spinloop has its corresponding lock acquisition earlier in the same spinloop iteration.

Algorithm 2 ZNE Spinloop Validity Check

```

1: procedure CHECKZNEVALIDITY( $G, e$ )
2:   if  $e$  is a write other than from a fetch_add() then
3:      $G.E \leftarrow G.E \setminus \text{ne}(\text{location-of}(e))$ 
4:   else if  $e \in R_{\text{loc}(x)} \wedge \exists e' \neq e. G.\text{rf}(e') = G.\text{rf}(e)$  then
5:      $G.E \leftarrow G.E \setminus \text{matching-zne}(G.\text{rf}(e))$ 

```

Dynamically, we simply check that no other thread accesses the lock besides by calling the acquire and release methods.

VII. IMPLEMENTATION

We implemented SAVER as an extension to the open-source GENMC tool [5], [13]. GENMC is a state-of-the-art stateless model checker for C/C++ programs that works at the level of LLVM Intermediate Representation (LLVM-IR), and can verify programs under weak memory models such as RC11 [10] and IMM [14]. SAVER is implemented as (a) a collection of transformation passes that modify GENMC’s input before the latter starts the verification procedure, and (b) slight modifications to GENMC’s DPOR algorithm that handle the dynamic checks for pure and ZNE loops.

As expected, SAVER imposes negligible overhead over GENMC, as its transformations take place statically, before the verification procedure starts, and the dynamic conditions for purity and ZNE loops can be checked in $\mathcal{O}(n)$ time (where n is the size of the graph), which is dominated by GENMC’s existing consistency checks.

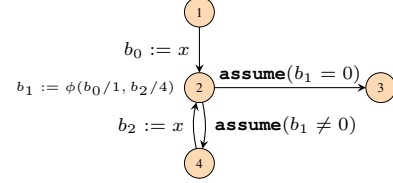
We conclude this section with some remarks regarding the implementation of loop rotation and the merging of bisimilar nodes over GENMC/LLVM.

In the case of loop rotation, we have implemented our own custom loop rotation pass that applies to loops whose rotation is deemed worthwhile. Although LLVM already contains an implementation of loop rotation, that implementation performs a more aggressive transformation by converting loops to a canonical form containing a single backedge. That is, if the loop contains multiple backedges, it constructs a new node with a backedge to the loop header and redirects all the existing backedges to the new node. This latter transformation is detrimental to the static detection of effect-free paths because it would, for example, conflate the three loopy paths of `ms-queue`’s `dequeue` operation (Fig. 3), thereby disabling the spin-assume transformation for the two that are effect-free. To avoid this unintended consequence, one would then have to undo this transformation (e.g., by invoking a form of jump threading) or rely on dynamic purity checks (§V). Instead, and to be able to statically transform as many loops as possible, we opted for implementing our own loop rotation pass, that transforms simple loops like CAS-LOOP2; loops that are not captured by our loop rotation pass are handled dynamically.

In the case of merging of bisimilar nodes, there are also a couple of points worth mentioning. First, detecting bisimilar nodes on LLVM is more complicated than what was discussed in §IV because LLVM represents programs in *static single assignment* (SSA) form. The effect of this design choice is that

there are never two nodes with identical assignments on their outgoing edges, since by the SSA definition each assignment is to a different register. Therefore, the standard bisimilarity algorithm outlined earlier in this section will not detect any nodes as being bisimilar!

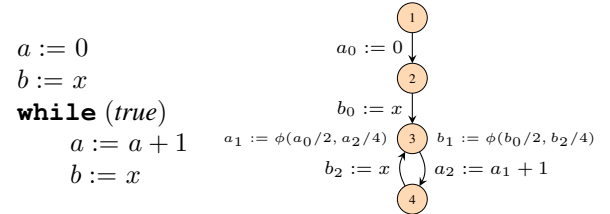
As an example, consider the “SSA-CFG” of thread II of the LOOP-PEEL program from §I, which is shown below.



The SSA-CFG is an enriched kind of CFG whose nodes may have ϕ -guards that define a variable differently depending on the incoming control flow path. For instance, in the SSA-CFG above, at node 2, b_1 is defined to be equal to b_0 if node 2 is reached from node 1, or to b_2 if it is reached from node 4.

In order to match nodes 1 and 4, our bisimilarity implementation has not only to account for ϕ -nodes, but also unify the variables b_0 and b_2 . It does so by collecting equality constraints and solving them by unification. For each node with more than one incoming edge, the algorithm starts iterating backwards for each pair of predecessors, and collects the constraints under which these predecessors are equal, simplifying them along the way. The iteration stops when some nodes cannot be equal under any constraints, or the entry node has been reached. At that point, any pair of nodes whose constraints can be trivially solved (namely, nodes 1 and 4 above) are deemed bisimilar.

Besides making bisimilarity detection more complex, SSA also affects the merging of bisimilar nodes. Consider the program below along with its SSA-CFG.



As can be seen, each of the assignments is to a different register, and node 3 contains two ϕ -guards (one for a and one for b) selecting the appropriate register to use depending on the incoming branch. With the algorithm outlined above one can detect that nodes 2 and 4 are bisimilar. However, one cannot simply add an edge $a_2 := a_1 + 1$ from node 3 to node 2 because that would violate the SSA form. To ensure that the resulting CFG is well-formed we also have to introduce a ϕ -guard at node 2 to say which version of a should be used for node 2. Our implementation achieves this by *moving* ϕ -guards the incoming values of which have not been deemed bisimilar (e.g., the ϕ -guard for a here) to the new loop header, along with any other incoming edges these ϕ -guards have.

VIII. EVALUATION

In this section, we evaluate the effectiveness of SAVER’s optimizations on a variety of benchmarks. Our evaluation comprises two distinct parts, with the first part concerning the overall performance of SAVER in a real-world setting, and the second part evaluating the effectiveness of employing individual transformations.

In general, we observe that applying the transformations introduced in this paper typically leads to *exponential gains* in real-world benchmarks with spinloops. Key to these gains are SAVER’s dynamic checks for spinloop purity and/or validity of ZNE spinloops, as well as the bisimilarity-based reduction of CFGs, which enables more spinloops to be bounded.

We conducted all experiments on a system with an Intel(R) Core(TM) i5-6600 CPU (4 cores @ 3.30GHz) and 16GB of RAM, running a custom Debian-based distribution. We used LLVM 7 for GENMC (v0.5.3). All reported times are in seconds. We set the timeout limit to 30 minutes.

A. Overall Performance

We start by evaluating SAVER on some challenging data structures utilizing weak-memory atomics that we harvested from the literature, including all data-structure benchmarks from GENMC’s original paper [5]. Since we want to measure the effectiveness of SAVER’s optimizations over the existing GENMC implementation, we do not compare against other tools and use GENMC as a baseline for our comparison. Since GENMC already contains a simple heuristic that converts some very simple **do-while** spinloops into **assume** statements, we use two versions of GENMC: one with its heuristic disabled and one with it enabled.

As can be seen in Table I, these benchmarks demonstrate that SAVER is extremely effective in a real-world setting, and that SAVER’s extensions combined lead to exponential gains. For all these benchmarks apart from **mutex-musl**, we have used an unroll value of $N + 1$ (where N is the number of threads, shown in parentheses) for both GENMC and SAVER to avoid manually unrolling any loops that spawn threads or initialize thread-local variables. For **mutex-musl** an unroll value of 2 and some manual unrolling was used, to keep the state space manageable. The transformations that SAVER applies are shown on the rightmost column, where S, D, Z, L, and B stand for spin-assume, dynamic-spin-assume, zne-assume, loop-rotation, and bisimilarity, respectively.

As can also be seen, GENMC’s simple heuristic is of rather limited value. It works very well only for the first two benchmarks (**mcslock** and **qspinlock**), where it matches the performance of SAVER. For the next three benchmarks (**seqlock**, **mpmc-queue**, and **linuxrwlocks**), it reduces the number of executions explored, but is still much slower than SAVER. Specifically, for **mpmc-queue(4)** and **linuxrwlocks(4)** GENMC does not manage to terminate within the time limit, while for **seqlock(4)** it needs 30.71 seconds. For the remaining eight benchmarks, GENMC’s heuristic does not apply at all.

SAVER, on the other hand, is able to employ its transformations (even if only partially) on all the benchmarks and, with

TABLE I
REAL-WORLD BENCHMARKS

	GENMC _S	GENMC	SAVER		
	<i>Execs</i>	<i>Execs</i>	<i>Execs</i>	<i>Time</i>	<i>Trans</i>
mcslock(3)	5964	336	336	0.09	S
mcslock(4)	⊙	26 232	26 232	6.20	S
qspinlock(2)	12	6	6	0.02	S
qspinlock(3)	13 764	564	564	0.09	S
seqlock(3)	430	147	9	0.03	S
seqlock(4)	3 670 360	87 980	88	0.21	S
mpmc-queue(3)	1 232 884	15 808	166	0.12	S, D
mpmc-queue(4)	⊙	⊙	39 706	193.41	S, D
linuxrwlocks(3)	14 059 037	38 033	24	0.04	B, S, Z
linuxrwlocks(4)	⊙	⊙	1060	0.36	B, S, Z
chase-lev(5)	17 367	17 367	3835	0.20	S
chase-lev(6)	778 581	778 581	41 055	2.39	S
treiber-stack(3)	426	426	18	0.10	S, D
treiber-stack(4)	1 546 168	1 546 168	484	0.61	S, D
mutex(2)	18	18	12	0.09	S, D
mutex(3)	59 760	59 760	7086	0.54	S, D
mutex-musl(2)	34	34	26	0.09	S, D
mutex-musl(3)	652 104	652 104	361 296	28.20	S, D
ttaslock(3)	11 031	11 031	162	0.10	S, D
ttaslock(4)	⊙	⊙	20 760	2.46	S, D
twalock(3)	1338	1338	96	0.10	S
twalock(4)	1 018 872	1 018 872	6144	0.72	S
ms-queue(3)	1389	1389	75	0.09	L, S, D
ms-queue(4)	⊙	⊙	10 662	28.13	L, S, D
scgather(3)	7560	7560	90	0.04	Z
scgather(4)	1 247 400	1 247 400	2520	1.07	Z

the exception of **mutex-musl**, this leads to a huge reduction in verification time over GENMC. That is, even if in some cases, SAVER only applies spin-assume/zne-assume in some of the data-structure’s methods, or even in some paths of a particular method, SAVER is still orders of magnitude faster than GENMC. Concretely, for all benchmarks, SAVER is able to transform at least one of the spinloops completely into an **assume** statement. For **seqlock**, SAVER reduces the read paths; for **mpmc-queue**, it reduces both the enqueue and dequeue methods; for **linuxrwlocks**, the read_lock and write_lock methods, for **chase-lev**, the steal method; for **treiber-stack**, the pop method; for **mutex**, **mutex-musl**, **ttaslock**, and **twalock**, various spinloops in the lock and unlock paths; for **ms-queue**, the enqueue and dequeue methods; and for **scgather** the check method. Finally, the smaller gains in verification time for **mutex-musl** are due to the small unroll value used and the fact that SAVER’s transformations do not apply to all the benchmark loops.

B. Employing Dynamic Purity/Unobservability Checks

As it can be seen from Table I, in more than half of the benchmarks, SAVER checked the purity of a spinloop or the non-observability of its intermediate effects dynamically. Dynamic checking proves useful for three cases.

First, in cases like **ms-queue**, plain spin-assume is not enough to fully transform some spinloop iterations into

TABLE II
BENEFITS OF BISIMILARITY

	SAVER _{\B\L}		SAVER _{\B}		SAVER	
	Execs	Time	Execs	Time	Execs	Time
ws+r-peeled(3)	9 264 697	81.01	5418	0.04	1	0.01
ws+r-peeled(4)	83 357 632	1353.39	13 419	0.18	1	0.01
w+rs-peeled(3)	⊖	⊖	893 025	4.75	1	0.01
w+rs-peeled(4)	⊖	⊖	⊖	⊖	1	0.03

assume statements because they contain possibly succeeding CAS operations. Recall from Fig. 3 that the second loop path of the simplified dequeue implementation is not effect-free. By adding a dynamic check to the relevant backedge, SAVER only considers iterations where the CAS actually succeeds, thus greatly reducing the state space of the program.

Second, in other cases (e.g., [mutex](#) and [ttaslock](#)), dynamic-spin-assume is necessary as spinloops contain function calls possibly containing side-effects. As it is difficult to determine statically whether these side-effects will actually take place in the particular calling context, the check is deferred to runtime.

Third, the unobservability checks both for initialization writes in failed CAS loops (e.g., [treiber-stack](#)) and for ZNE loops ([linuxrwlocks](#) and [scgather](#)) are very hard to perform statically with sufficient precision. As such, performing them dynamically is the only viable option.

C. Employing Loop Rotation and Bisimilarity Reduction

Loop rotation and bisimilarity reduction are similarly important in some real-world test cases. Even though they do not yield any performance improvements on their own, they are instrumental in making the spin-assume and zne-assume transformations applicable to more complex cases. Specifically, in benchmarks like [ms-queue](#) and [linuxrwlocks](#), spin-assume and zne-assume are not applicable without loop rotation and bisimilarity respectively. And, in fact, these are not the only cases that we have encountered; there are many ways to rewrite the same benchmarks so that they also require bisimilarity and/or loop rotation, thus rendering these transformations a necessity, as opposed to an enhancement.

As a further demonstration of their usefulness, we consider two synthetic test cases inspired by the LOOP-PEEL example. In these tests, some threads repeatedly write to a shared variable, which is read by readers that employ schemes similar to LOOP-PEEL’s second thread. As explained in §III, spin-assume is not directly applicable in such cases because the live variables of the header are redefined within the loop. Thus, we used an unroll value of 3, and manually unrolled any loops utilized by the writer threads. For these benchmarks, we used three SAVER versions: the default version that employs both bisimilarity and loop rotation (SAVER), a version where bisimilarity is disabled (SAVER_{\B}) and a version where both bisimilarity and loop rotation are disabled (SAVER_{\B\L}). The results can be seen in Table II.

With bisimilarity reduction, SAVER transforms the spinloops into **assume** statements and only explores one execution,

since only one combination of values satisfies the **assumes**. Applying only loop rotation is equivalent to transforming the syntactic spinloops in these programs into **assume** statement but keeping the peeled iteration. Thus, SAVER_{\B} explores a much larger number of executions, which affects the verification time. Applying neither transformation (SAVER_{\B\L}) explores a huge number of executions and often timeouts. These results highlight the necessity of being resilient against small syntactic variations as, even if a single read is not taken into account when transforming a spinloop into an **assume**, the state space might grow exponentially.

IX. RELATED WORK AND CONCLUSIONS

We have presented a set of automated techniques for soundly bounding various kinds of spinloops to a single iteration, which empowers SMC to reason effectively about programs containing such spinloops. Although our contribution was presented in terms of SMC, it should be equally applicable to SAT/SMT-based bounded model checking (BMC) implemented by different tools (e.g., [15]–[17]).

Although there is a large body of work on model checking concurrent programs (e.g., [12], [18]–[22]), we are not aware of any other automated technique for bounding such a wide range of spinloops including potentially effect-free and ZNE loops. NIDHUGG [3], [23], RCMC [4] and GENMC [5], [13] are the only other tools we are aware of that automatically transform some spinloops to **assume** statements but they limit themselves to very simple busy-wait loops with no side-effects and no CAS instructions and they are not resilient to simple syntactic variations of such loops. POET [24] does recognize spinloop iterations that do not make progress, but saves the program state in order to do so.

Since both SMC and BMC cannot handle programs with executions of unbounded length, most tools bound the number of allowed loop iterations by a user-specified bound. Other tools like CDSHECKER [2] use a memory-liveness bound to ensure termination for spinloops. As shown in §VIII, bounding techniques in general are inferior to converting spinloops to **assume** statements in terms of scalability.

Bounding of spinloops to a single iteration is, however, not a totally new idea. In a rather different context, Flanagan *et al.* [25] have used purity for proving atomicity of concurrent libraries treating effect-free spinloops as though they had been reduced to **assume** statements. Elmas *et al.* [26] have also performed similar transformations in their tool QED, which allows a programmer to initiate a sequence of reductions and abstractions to statically establish correctness of a program.

ACKNOWLEDGMENTS

This work was supported by a European Research Council (ERC) Consolidator Grant for the project “PERSIST” under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101003349).

REFERENCES

- [1] P. Godefroid, “Model checking for programming languages using VeriSoft,” in *POPL 1997*, Paris, France: ACM, 1997, pp. 174–186. DOI: 10.1145/263699.263717.
- [2] B. Norris and B. Demsky, “CDSChecker: Checking concurrent data structures written with C/C++ atomics,” in *OOPSLA 2013*, ACM, 2013, pp. 131–150. DOI: 10.1145/2509136.2509514.
- [3] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, “Stateless model checking for TSO and PSO,” in *TACAS 2015*, ser. LNCS, vol. 9035, Berlin, Heidelberg: Springer, 2015, pp. 353–367. DOI: 10.1007/978-3-662-46681-0_28.
- [4] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, “Effective stateless model checking for C/C++ concurrency,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 17:1–17:32, Dec. 2017, ISSN: 2475-1421. DOI: 10.1145/3158105.
- [5] M. Kokologiannakis, A. Raad, and V. Vafeiadis, “Model checking for weakly consistent libraries,” in *PLDI 2019*, New York, NY, USA: ACM, 2019, DOI: 10.1145/3314221.3314609.
- [6] M. Kokologiannakis and V. Vafeiadis, “HMC: Model checking for hardware memory models,” in *ASPLOS 2020*, ser. ASPLOS ’20, Lausanne, Switzerland: ACM, 2020, pp. 1157–1171, ISBN: 9781450371025. DOI: 10.1145/3373376.3378480.
- [7] C. Flanagan and P. 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.
- [8] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, “Optimal dynamic partial order reduction,” in *POPL 2014*, New York, NY, USA: ACM, 2014, pp. 373–384. DOI: 10.1145/2535838.2535845.
- [9] J. Alglave, L. Maranget, and M. Tautschnig, “Herding cats: Modelling, simulation, testing, and data mining for weak memory,” *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, 7:1–7:74, Jul. 2014, ISSN: 0164-0925. DOI: 10.1145/2627752.
- [10] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, “Repairing sequential consistency in C/C++11,” in *PLDI 2017*, Barcelona, Spain: ACM, 2017, pp. 618–632, ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062352.
- [11] P. A. Abdulla, M. F. Atig, B. Jonsson, M. Lång, T. P. Ngo, and K. Sagonas, “Optimal stateless model checking for reads-from equivalence under sequential consistency,” *Proc. ACM Program. Lang.*, vol. 3, 150:1–150:29, OOPSLA Oct. 10, 2019. DOI: 10.1145/3360576.
- [12] M. Chalupa, K. Chatterjee, A. Pavlogiannis, N. Sinha, and K. Vaidya, “Data-centric dynamic partial order reduction,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 31:1–31:30, Dec. 2017, ISSN: 2475-1421. DOI: 10.1145/3158119.
- [13] M. Kokologiannakis and V. Vafeiadis, “GenMC: A model checker for weak memory models,” in *CAV 2021*, A. Silva and K. R. M. Leino, Eds., ser. LNCS, vol. 12759, Springer, 2021, pp. 427–440. DOI: 10.1007/978-3-030-81685-8_20.
- [14] A. Podkopaev, O. Lahav, and V. Vafeiadis, “Bridging the gap between programming languages and hardware weak memory models,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 69:1–69:31, Jan. 2019, ISSN: 2475-1421. DOI: 10.1145/3290382.
- [15] E. M. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *TACAS 2004*, ser. LNCS, vol. 2988, Berlin, Heidelberg: Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.
- [16] N. Gavrilenco, H. Ponce-de-León, F. Furbach, K. Heljanko, and R. Meyer, “BMC for weak memory models: Relation analysis for compact SMT encodings,” in *CAV 2019*, I. Dillig and S. Tasiran, Eds., Cham: Springer International Publishing, 2019, pp. 355–365, ISBN: 978-3-030-25540-4. DOI: 10.1007/978-3-030-25540-4_19.
- [17] S. Burckhardt, R. Alur, and M. M. K. Martin, “CheckFence: Checking consistency of concurrent data types on relaxed memory models,” in *PLDI 2007*, New York, NY, USA: ACM, 2007, pp. 12–21. DOI: 10.1145/1250734.1250737.
- [18] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, “Finding and reproducing heisenbugs in concurrent programs,” in *OSDI 2008*, USENIX Association, 2008, pp. 267–280. [Online]. Available: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf (visited on 11/16/2020).
- [19] E. Albert, P. Arenas, M. G. de la Banda, M. Gómez-Zamalloa, and P. J. Stuckey, “Context-sensitive dynamic partial order reduction,” in *CAV 2017*, R. Majumdar and V. Kunčák, Eds., Cham: Springer International Publishing, 2017, pp. 526–543, ISBN: 978-3-319-63387-9. DOI: 10.1007/978-3-319-63387-9_26.
- [20] E. Albert, M. Gómez-Zamalloa, M. Isabel, and A. Rubio, “Constrained dynamic partial order reduction,” in *CAV 2018*, H. Chockler and G. Weissenbacher, Eds., Cham: Springer International Publishing, 2018, pp. 392–410, ISBN: 978-3-319-96142-2. DOI: 10.1007/978-3-319-96142-2_24.
- [21] N. Zhang, M. Kusano, and C. Wang, “Dynamic partial order reduction for relaxed memory models,” in *PLDI 2015*, New York, NY, USA: ACM, 2015, pp. 250–259. DOI: 10.1145/2737924.2737956.
- [22] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo, “Optimal stateless model checking under the release-acquire semantics,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 135:1–135:29, Oct. 2018, ISSN: 2475-1421. DOI: 10.1145/3276505.
- [23] P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson, “Stateless model checking for POWER,” in *CAV 2016*, ser. LNCS, vol. 9780, Berlin, Heidelberg: Springer, 2016, pp. 134–156. DOI: 10.1007/978-3-319-41540-6_8.
- [24] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening, “Unfolding-based partial order reduction,” in *CONCUR 2015*, ser. LIPIcs, vol. 42, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 456–469. DOI: 10.4230/LIPIcs.CONCUR.2015.456.
- [25] C. Flanagan, S. N. Freund, and S. Qadeer, “Exploiting purity for atomicity,” *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 275–291, 2005. DOI: 10.1109/TSE.2005.47.
- [26] T. Elmas, S. Qadeer, and S. Tasiran, “A calculus of atomic actions,” in *POPL 2009*, Z. Shao and B. C. Pierce, Eds., ACM, 2009, pp. 2–15. DOI: 10.1145/1480881.1480885.

Robustness between Weak Memory Models

Soham Chakraborty

EEMCS, TU Delft

Email: s.s.chakraborty@tudelft.nl

Abstract—

Robustness of a concurrent program ensures that its behaviors on a weak concurrency model are indistinguishable from those on a stronger model. Enforcing robustness is particularly useful when porting or migrating applications between architectures. Existing tools mostly focus on ensuring sequential consistency (SC) robustness which is a stronger condition and may result in unnecessary fences.

To address this gap, we analyze and enforce robustness between weak memory models, more specifically for two mainstream architectures: x86 and ARM (versions 7 and 8). We identify robustness conditions and develop analysis techniques that facilitate porting an application between these architectures. To the best of our knowledge, this is the first approach that addresses robustness between the hardware weak memory models.

We implement our robustness checking and enforcement procedure as a compiler pass in LLVM and experiment on a number of standard concurrent benchmarks. In almost all cases, our procedure terminates instantaneously and insert significantly less fences than the naive schemes that enforce SC-robustness.

I. INTRODUCTION

Robustness analysis checks whether a program running on a weak memory consistency model demonstrates only the behaviors that are allowed by a stronger model. Robust programs can therefore be seamlessly migrated from one model to another as far as their concurrent behaviors are concerned. If a program is not robust, we can insert fences to enforce robustness.

Robustness analysis is especially beneficial in porting applications [1, 2] where it is crucial to preserve the observable behaviors of a running application. For instance, consider the porting of an application written for x86 to ARM. Since the x86 model is stronger than the ARM models (x86 exhibits less behavior), x86-robustness abstracts the underlying ARM machine specification to an outside observer. Consider the following programs where initially $X = Y = 0$.

$$\begin{array}{l} X = 1; \parallel Y = 1; \\ a = Y; \parallel b = X; \end{array} \quad (\text{SB}) \quad \left| \quad \begin{array}{l} a = X; \parallel b = Y; \\ Y = 1; \parallel X = 1; \end{array} \quad (\text{LB})$$

Both x86 and ARM allow same set of concurrent executions in the SB program and hence indistinguishable on x86 and ARM. Therefore SB can be ported seamlessly between these architectures. Now consider the porting of the LB program from x86 to ARM. x86 disallows $a = b = 1$ but ARM allows the outcome. Hence the LB program in ARM is not x86-robust. To enforce x86-robustness we insert fences in both threads and restrict the $a = b = 1$ outcome.

Checking and enforcing robustness to a stronger but non-SC model from a weaker model can play a key role in migrating programs between architectures having weak concurrency

models. Existing SC-robustness approaches may not provide an optimal solution as they check a stronger constraint and hence may introduce additional fences. For example, if we use an SC-robustness checker for SB, it identifies that the $a = b = 0$ outcome is allowed on ARM but disallowed in SC. Hence the analyzer inserts two full fences (DMB in ARMv7 and DMBFULL in ARMv8) between the memory accesses in both threads which are unnecessary in this case.

To address this scenario we propose robustness analysis and enforcement between weak memory models of two mainstream architectures: x86 and ARM (version 8 and 7). As ARMv8 is a stronger model than ARMv7, we also study ARMv8-robustness for ARMv7 to enable application porting between these ARM models. We also check SC-robustness in x86, ARMv8, ARMv7 and restrict relaxed memory behaviors.

In this paper we propose M - K robustness where M is a stronger model than K and M can also be a non-SC model unlike existing approaches in [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]. We propose the M - K robustness conditions in §III and prove their correctness [15]. Our proposed M - K robustness conditions ensure that if a K -consistent execution satisfies the M - K condition then the execution is also M -consistent. We check if certain memory access pairs are appropriately ordered in a K -consistent execution so that the execution shows no weaker behavior. Otherwise we insert fences to enforce order and restrict the weaker behaviors. However, as fences are costly, we investigate if it is possible to weaken the robustness constraints for the memory access pairs which are on same-location or are ordered by dependencies. We observe that these relations suffice in x86 and ARMv8, but the results in ARMv7 are counter-intuitive.

- We note that dependency based ordering *preserved-program-order* (ppo) is not strong enough to ensure robustness in ARMv7. Consider the following ARMv7 program.

$$\begin{array}{l} a = T; \parallel X = 2; \parallel b = X; \parallel c = Y; \parallel Z = 1; \parallel d = Z; \\ X = a; \parallel Y = b; \parallel Z = c; \parallel T = d; \end{array} \quad (\text{WP})$$

The execution in Fig. 4 exhibits non-SC behavior though all the memory access pairs result in ppo relations due to data dependencies. Even an intermediate full fence in one of these threads cannot restrict the relaxed behavior.

- We evaluate the role of same-location program-order relation in defining robustness conditions. On ARMv7, same-location read-write access pair is unordered (see ARM-Weak [16] example in Fig. 2). Yet if all *external-program-orders* (see §III) are on same-location or have intermediate fences then the program exhibits only SC behavior.

In §IV we propose static analyses to check if a program is M - K robust based on the respective conditions. Otherwise we insert fences to enforce robustness. These analyses are computed in polynomial time as shown in §IV-C unlike the robustness checkers which explore program executions and are of significantly higher computational complexity.

The robustness checking procedures analyze the programs with thread functions. In these programs each thread function may result in any number of concurrent threads in an execution. Thus our analysis is parameterized by the thread functions and the analyses are applicable to all the programs having same thread functions.

We have implemented the analyses procedures in a tool called *Fency* based on LLVM [17] and have evaluated on several well known concurrent programs [8, 14]. We compare the SC-x86 robustness analysis of *Fency* to existing SC-TSO robustness results of Trencher [8] that explore program executions by model checkers. Yet, *Fency* is quite precise and matches Trencher in most of the programs. Moreover, *Fency* does not use external model checkers or SAT/SMT solvers and therefore is significantly fast in most of the cases.

We also compare *Fency* to a *naive* fence insertion scheme that do not use robustness analysis. *Fency* inserts significantly fewer fences than the naive scheme in several benchmarks. Moreover, empirical evaluations show that if a model W is weaker than M then ensuring W - K robustness often requires fewer fences than ensuring M - K robustness. Thus precise robustness analysis is indeed beneficial for many cases instead of using SC-robustness checkers.

Outline and Contributions. §II reviews the concurrency models. §III proposes the M - K robustness conditions. §IV explains our approach to check and enforce robustness. §V examine the experimental results. §VI discusses the related work and we conclude in §VII. The proofs and additional details are in the supplementary material [15].

II. CONCURRENCY MODELS

In this section we review SC, x86, ARMv8, and ARMv7 concurrency. For all models we follow a common syntax.

$$\begin{aligned} E &::= r | v | E + E | E * E | E \leq E | \dots \\ C &::= \text{skip} | C; C | t = E | t = X | X = E | \text{RMW}(X, E, E) \\ &\quad | \text{Fence} | \text{RMW}(X, E) | \mathbf{br} \text{ label} | \mathbf{br} \text{ label label} | \dots \\ P &::= X = v; \dots X = v; \{C \parallel \dots \parallel C\} \end{aligned}$$

An expression E results from thread-local temporary (t), value (v), and arithmetic operations (E). Command $t = X$ returns the value of a shared memory location X to a thread-local register r and $X = E$ writes the evaluation of expression E to X . The $\text{RMW}(X, E_r, E_w)$ atomically compares the values of X and E_r ; if equal then X is written to the value of E_w and set r . If the value of X is not equal to the value of E_r then the RMW fails. Command $\text{RMW}(X, E_r)$ atomically updates the value of X with the value of E_r and returns the value of X to r . A failed RMW performs only read access. A fence orders certain memory accesses. We use conditional and

unconditional branches for program's control flow. Finally, a program consists of a set of initialization writes followed by a parallel composition of thread commands. Unless otherwise mentioned, the initializations set all memory locations to zero.

A. Program Semantics and Execution Graphs

We follow the axiomatic models for all architectures [18, 19, 20, 21, 22, 23, 24, 25, 26]. In these axiomatic models a program's semantics is defined by a set of consistent executions. An execution consists of a set of events and relations.

Event. An event $\langle \text{id}, \text{tid}, \text{lab} \rangle$ consists of unique identifier id , thread identifier $\text{tid} \in \mathbb{N}$, and a label lab based on the respective executed memory or fence access. A label is of the form $\langle \text{op}, \text{loc}, \text{val} \rangle$ where op , loc , and val are operation type, location, and read or written value.

Preliminaries. Given a binary relation P on events, $\text{dom}(P)$ and $\text{codom}(P)$ are its domain and its range. P^{-1} , $P^?$, P^+ , and P^* are inverse, reflexive, transitive, and reflexive-transitive closures of P respectively. P_ℓ denotes P related event pairs on same locations i.e. $P_\ell \triangleq \{(e, e') \in P \mid e.\text{loc} = e'.\text{loc}\}$ and $P_{\neq \ell} \triangleq P \setminus P_\ell$ denote the P related event pairs on different locations. $\text{imm}(P)$ defines the immediate P relation, i.e. $\text{imm}(P) \triangleq \exists a, b. P(a, b) \wedge \nexists c. P(a, c) \wedge R(c, b)$. $P; S$ is the relational composition of the binary relations P and S . Finally, $[A]$ is an identity relation on a set A .

R , W , and F are the set of read, write, and fence events. The events are related by primitive relations: strict partial order program-order (po) captures the syntactic order among the events, reads-from (rf) relates a write event to a read event that justifies its read value, and strict total order coherence-order (co) relates same-location writes.

Execution. An execution is of the form $X = \langle E, \text{po}, \text{rf}, \text{co} \rangle$ where $X.E$ is the set of events in X . The set of po , rf , and co relations between the events in $X.E$ are $X.\text{po}$, $X.\text{rf}$, and $X.\text{co}$. Execution X is *well-formed* if $X.\text{po}$ is total in each thread and every read reads-from some write, i.e. $X.R \subseteq \text{codom}(X.\text{rf})$.

We derive a number of relations from these primitive relations. Relation $\text{rmw} \subseteq \text{imm}(\text{po}) \cap ([R] \times [W])_\ell$ denotes atomic update where a read has an immediate po -successor write on the same location. The non- rmw read and write events are load (Ld) and store (St) events.

$$\text{Ld} \triangleq R \setminus \text{dom}(\text{rmw}) \quad \text{St} \triangleq W \setminus \text{codom}(\text{rmw})$$

A successful RMW generates an rmw and a failed RMW generates a Ld event. We use $a \cdot b \triangleq [\{a\}; \text{imm}(\text{po}); \{b\}]$ to denote that a and b are immediate po related events.

Relation WR denotes a write-read event pair on different locations that does not have any intermediate rmw .

$$\text{WR} \triangleq ([W]; \text{po}_{\neq \ell}; [R]) \setminus (\text{po}; \text{rmw}; \text{po})$$

The from-read (fr) relation relates a pair of same-location read and write events r and w where r reads-from a write w' which is co -before w , that is, $\text{fr} \triangleq \text{rf}^{-1}; \text{co}$. For example, in Fig. 1a the $R(X, 0)$ and $W(X, 1)$ events are in fr relation.

We categorize the relations as external and internal based on whether the events are also in po relation. Considering rf ,

co, and fr relations rfi, coi, fri and rfe, coe, fre denote the internal and external relations respectively.

$$\begin{aligned} \text{rfe} &\triangleq \text{rf} \setminus \text{po} & \text{coe} &\triangleq \text{co} \setminus \text{po} & \text{fre} &\triangleq \text{fr} \setminus \text{po} \\ \text{rfi} &\triangleq \text{rf} \cap \text{po} & \text{coi} &\triangleq \text{co} \cap \text{po} & \text{fri} &\triangleq \text{fr} \cap \text{po} \end{aligned}$$

For example, the rf and fr edges in Fig. 1a edges are rfe and fre edges respectively. Based on the rfe, coe, and fre we define *extended-coherence-order* (eco) on same location events: $\text{eco} \triangleq (\text{rfe} \cup \text{coe} \cup \text{fre})^+$.

Consistency Axioms. An axiomatic model is defined by a set of axioms. An execution is consistent in a model if it satisfies all its axioms. An axiom violation can be captured by a cycle on the respective execution graph.

B. Formal Models

Now we move to the axiomatic definitions based on various relations. We elide some definitions here due to space constraint which we discuss in the technical appendix [15].

In these models a store access writes value v on location x and generates an event with label $W(x, v)$. A load access reads value v from x and generates an event with label $R(x, v)$. A successful RMW on x reads value v' and writes value v to generate a pair of $R(x, v')$ and $W(x, v)$ events that are in *rmw* relation. A failed RMW generates an $R(x, v')$ event. The full fences in x86, ARMv8, and ARMv7 are MFENCE, DMBFULL, and DMB respectively. A full fence generate an event with label F. ARM architectures also provides ISB fence to order a pair of reads. In ARMv7 an ISB access along with control (cmp) and jump (bc) instructions generate cmp; bc; ISB that result in ctrl_{ISB} between a pair of read events in an execution [19]. In ARMv8 an ISB generates an ISB event.

ARMv8 Specific Accesses. In addition, ARMv8 has synchronizing memory accesses such as release write, acquire read, and acquirePC load which are denoted by events with label $L(x, v)$, $A(x, v)$, and $Q(x, v)$. ARMv8 also provide DMBLD and DMBST fences that generate F_{LD} , and F_{ST} events. Finally, $L \subseteq W$, $A \subseteq R$, $Q \subseteq L$, and $F, F_{\text{LD}}, F_{\text{ST}}$ are the set of release, acquire, acquirePC, and full, load, store fence events.

All these models satisfy coherence and atomicity properties. **Coherence.** The property enforces SC per location i.e. in an execution all accesses on same memory locations are totally ordered. A complete execution graph X satisfies coherence if $X.\text{po}_\ell \cup X.\text{rf} \cup X.\text{co} \cup X.\text{fr}$ is acyclic.

Atomicity. An execution X violates atomicity if there is an intermediate write on same location between *rmw* related read and write events. In that case $X.\text{fre}(r, \cdot)$ and $X.\text{coe}(\cdot, \cdot)$ hold where r and \cdot are $X.\text{rmw}$ -related events and \cdot is another write on the same location as r and \cdot .

SC. An well-formed execution X is SC when:

- $(X.\text{po}_\ell \cup X.\text{rf} \cup X.\text{fr} \cup X.\text{co})$ is acyclic (SC)
- $X.\text{rmw} \cap (X.\text{fre}; X.\text{coe}) = \emptyset$ (atomicity)

The executions in Fig. 1 are inconsistent in SC. For example, the SB execution has $\text{po} \cup \text{fr}$ cycle. Note that coherence constraint is included in (SC) axiom as $\text{po}_\ell \subseteq \text{po}$ holds and therefore if $(X.\text{po}_\ell \cup X.\text{rf} \cup X.\text{fr} \cup X.\text{co})$ is acyclic then $(X.\text{po}_\ell \cup X.\text{rf} \cup X.\text{fr} \cup X.\text{co})$ is also acyclic.

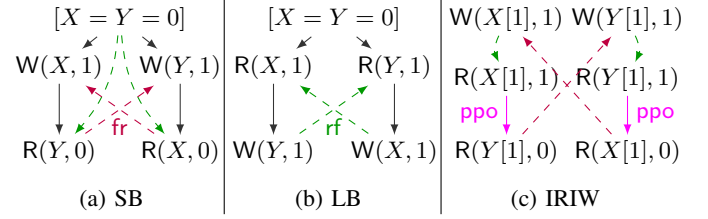


Fig. 1: Distinguishing executions: SB execution is disallowed in SC but allowed in x86 and ARM. SC and x86 disallow LB execution but ARM models allow it. IRIW execution is disallowed in SC, x86, ARMv8, but allowed in ARMv7.

x86. Relation x86-preserved-program-order (*xppo*) orders read-read, read-write, write-write access pairs. Relation *implied* signifies that an intermediate *rmw* or F acts as a full fence. Based on these relations x86 defines x86-happens-before (*xhb*). Finally, x86 defines its consistency constraints for a well-formed execution.

- $X.\text{po}_\ell \cup X.\text{rf} \cup X.\text{fr} \cup X.\text{co}$ is acyclic (sc-per-loc)
- $X.\text{rmw} \cap (X.\text{fre}; X.\text{coe}) = \emptyset$ (atomicity)
- $X.\text{xhb}$ is acyclic where (GHB)
 - $\text{xhb} \triangleq \text{xppo} \cup \text{implied} \cup \text{rfe} \cup \text{fr} \cup \text{co}$ where
 - $\text{xppo} \triangleq ((W \times W) \cup (R \times W) \cup (R \times R)) \cap \text{po}$
 - $\text{implied} \triangleq \text{po}; [\text{dom}(\text{rmw}) \cup F] \cup [\text{codom}(\text{rmw}) \cup F]; \text{po}$

x86 satisfies coherence and atomicity by (sc-per-loc) and (atomicity) axioms respectively. Axiom (GHB) ensures a global order based on *xhb* relation. The model allows Fig. 1a but disallows the executions in Figs. 1b and 1c.

ARMv8. In ARMv8 relation observed-by ($\text{obs} \subseteq \text{eco}$) relates same-location external events. Relation atomic-ordered-by ($\text{aob} \subseteq \text{po}_\ell$) orders events based on *rmw* and acquire or acquirePC events. The dependency-ordered-before (*dob*) captures dependency based ordering between events e.g. $\text{data} \cup \text{addr} \subseteq \text{dob}$. Relation barrier-ordered-by (*bob*) orders events by fences and stronger memory accesses as follows.

$$\begin{aligned} \text{bob} &\triangleq \text{po}; [F]; \text{po} \cup [R]; \text{po}; [F_{\text{LD}}]; \text{po} \cup [W]; \text{po}; [F_{\text{ST}}]; \text{po}; [W] \\ &\quad \cup [L]; \text{po}; [A] \cup \text{po}; [L] \cup [A \cup Q]; \text{po} \cup \text{po}; [L]; \text{coi} \end{aligned}$$

A full fence orders all accesses, a load fence orders a read with its successors, and a store fence orders a pair of writes. A release access is ordered with its predecessors and an acquire or acquirePC is ordered with its successors. Release and acquire accesses are ordered. Finally, (a, b) is ordered if b is a write and there is an intermediate release store on the same-location as b . Based on these relations ARMv8 defines *Ordered-before* (*ob*) order: $\text{ob} \triangleq (\text{obs} \cup \text{dob} \cup \text{aob} \cup \text{bob})^+$. A well-formed ARMv8 execution X is consistent when:

- $X.\text{po}_\ell \cup X.\text{rf} \cup X.\text{co} \cup X.\text{fr}$ is acyclic (internal)
- $X.\text{rmw} \cap (X.\text{fre}; X.\text{coe}) = \emptyset$ (atomicity)
- $X.\text{ob}$ is irreflexive (external)

These axioms allow the executions in Figs. 1a and 1b but disallows the execution in Fig. 1c by the (external) axiom.

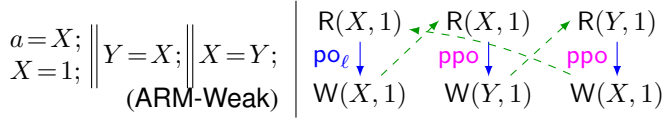


Fig. 2: Outcome $a = 1$ is allowed in ARMv7.

ARMv7. ARMv7 orders memory accesses in a thread by *preserved-program-order* (ppo) based on dependencies or $\text{fence} \subseteq \text{po}; [F]; \text{po}$ relation. ARMv7 also defines happens-before (ahb) and propagation ($\text{prop} \subseteq R_1; \text{fence}; R_2$) relations that can order events across threads. Finally a well-formed ARMv7 execution X is consistent when:

- $(X.\text{po}_\ell \cup X.\text{rf} \cup X.\text{fr} \cup X.\text{co})$ is acyclic. (sc-per-loc)
- $X.\text{rmw} \cap (X.\text{fre}; X.\text{coe}) = \emptyset$ (atomicity)
- $X.\text{fre}; X.\text{prop}; X.\text{ahb}^*$ is irreflexive. (observation)
- $(X.\text{co} \cup X.\text{prop})$ is acyclic. (propagation)
- $X.\text{ahb}$ is acyclic. (no-thin-air)

Axiom (observation) constrains the set of writes from which reads may read-from; if a write is in $\text{prop}; \text{ahb}^*$ relation with a same-location read r then r does not read from $'$ which is co-before . (propagation) ensures that prop does not contradict co and (no-thin-air) constrain causality cycle.

ARMv7 allows the executions in Fig. 1 including IRIW with $a = c = 1, b = d = 0$ outcome in the following program.

$$X[1] = 1; \left\| \begin{array}{l} a = X[1]; \\ b = Y[a]; \end{array} \right\| \left\| \begin{array}{l} c = Y[1]; \\ d = X[c]; \end{array} \right\| Y[1] = 1; \text{ (IRIW)}$$

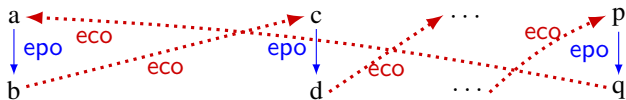
In addition read-write accesses on same-location can be unordered in ARMv7. As a result, the ARM-Weak program in Fig. 2 has an execution with $a = 1$ outcome.

III. ROBUSTNESS ANALYSIS AND ENFORCEMENT

In this section we first define M - K robustness and then propose the M - K robustness conditions.

Definition 1. A program is M - K robust if all its K -consistent executions are also M -consistent.

Suppose a K -consistent execution X violates an axiom from M -consistency. The violation results in a cycle in X . If the cycle contains no po edge then it is formed by rfe , fre , and coe edges on same location events. The cycle also violates coherence. This is not possible as execution X is K -consistent and all K models we are considering satisfy coherence. So the cycle consists of a set of po -edges along with the eco edges between them. We define these po edges as *external-program-order* (epo) i.e. $\text{epo} \triangleq \text{po} \cap (\text{codom}(\text{eco}) \times \text{dom}(\text{eco}))$.



Thus we represent an axiom violation as a $(\text{epo}; \text{eco})^+$ cycle where all the epo edges on the cycle are not sufficiently ordered. To enforce order we insert fences to strengthen these epo edges and restrict a cycle to enforce M - K robustness.

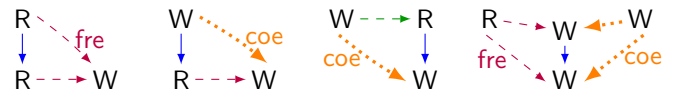


Fig. 3: Coherence ensures $\text{eco}; \text{epo}_\ell \cup \text{epo}_\ell; \text{eco} \subseteq \text{eco}$.

Theorem 1. A program P is M - K robust if in all its K -consistent execution X , $X.\text{epo} \subseteq X.R$ holds where R is defined as M - K condition as follows.

- (SC-x86) $\text{xppo} \cup \text{po}_\ell \cup \text{implied}; \text{po}^?$
- (SC-ARMv8) $\text{po}_\ell \cup (\text{aob} \cup \text{dob} \cup \text{bob})^+$
- (x86-ARMv8) $\text{po}_\ell \cup (\text{aob} \cup \text{bob} \cup \text{dob})^+ \cup \text{WR}$
- (SC-ARMv7) $\text{po}_\ell \cup \text{fence}$
- (x86-ARMv7) $\text{po}_\ell \cup \text{fence} \cup \text{WR}$
- (ARMv8-ARMv7) $\text{po}_\ell \cup [W]; \text{po} \cup \text{fence}$

Next, we explain the M - K conditions for the concurrency models. The correctness proofs for these robustness conditions are in the technical appendix [15].

A. Robustness of x86 Programs

From the SC-x86 condition in Theorem 1, relation xppo orders read-read, read-write, and write-write pairs. So if an x86 execution violates SC-x86 robustness then it contains a $(\text{epo}; \text{eco})^+$ cycle with one or multiple epo edges that are in WR relation. If it is on same location then there is an alternative $(\text{eco}; \text{epo})^+$ cycle as shown in Fig. 3 that also denote the violation. The $\text{implied}; \text{po}^?$ relation can order a write-read pair by intermediate rmw or F .

Consider the SB execution from Fig. 1a in x86. The epo edges do not satisfy SC-x86 condition and the execution is non-SC. If we insert fences between the store-load pairs in each thread then the program exhibits only SC behaviors.

B. Robustness of ARMv8 Programs

SC-ARMv8 Robustness. Suppose an ARMv8 execution contains a $(\text{epo}; \text{eco})^+$ cycle that violates SC-ARMv8 robustness. If an epo_ℓ edge is on the cycle then as shown in Fig. 3 there is an alternative $(\text{epo}; \text{eco})^+$ cycle without the edge.

Now consider an $(\text{epo}; \text{eco})^+$ cycle where each epo on the cycle is in $(\text{aob} \cup \text{bob} \cup \text{dob})^+$ relation. In that case $((\text{aob} \cup \text{bob} \cup \text{dob})^+; \text{eco})^+$ cycle implies an ob cycle which is not possible as an ARMv8 consistent execution satisfies (external). The epo edges in SB and LB executions in Fig. 1 do not satisfy the SC-ARMv8 condition. The executions are allowed in ARMv8 but not in SC.

x86-ARMv8 Robustness. The x86-ARMv8 robustness condition orders all epo relations except WR pairs as WR is also unordered in x86. Hence an ARMv8 execution exhibits only x86 behavior if the x86-ARMv8 condition holds. Consider the SB execution from Fig. 1a in ARMv8; both the epo edges are also in WR and the execution is x86 consistent.

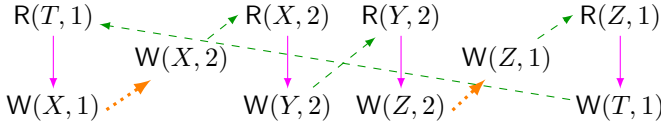


Fig. 4: ARMv7 allows the execution of the WP program.

C. Robustness of ARMv7 Programs

SC-ARMv7 Robustness. The ARMv7 model uses po_ℓ and $fence$ relations to order epo edges for SC-ARMv7 robustness.

The ppo and po_ℓ do not guarantee SC-ARMv7 robustness as shown in the execution in Fig. 2. If we insert fences in the second and third threads the execution is disallowed in ARMv7 and the resulting program is SC-ARMv7 robust.

Moreover, ppo relations in all epo edges do not ensure SC behavior in an execution. For instance, the WP program execution in Fig. 4 is non-SC even though the epo edges are ppo -ordered. Note that, even if we insert an intermediate DMB in one of the threads the cycle is still possible in ARMv7.

x86-ARMv7 Robustness. To ensure x86-robustness, ARMv7 orders all epo relations except write-read pairs. Consider the SB program execution in Fig. 1a where the epo edges are WR pairs and the execution is consistent in both ARMv7 and x86.

ARMv8-ARMv7 Robustness. ARMv8-ARMv7 robustness requires to order all $epo_{\neq \ell}$ relations except write-read and write-write pairs. In this case also ppo relation cannot order $epo_{\neq \ell}$ edges. Hence the cycle in the ARMv7 execution in Fig. 4 is disallowed in ARMv8 as it is an ob cycle.

IV. CHECKING AND ENFORCING ROBUSTNESS

In this section we lift the semantic notion of M - K robustness to the program syntax and propose static analyses to check and enforce robustness in the following steps.

- 1) *Identify program components which may run concurrently.*
We consider fork-join parallelism and identify the thread functions where each function may create multiple threads.
- 2) *Memory-access pair graph construction.* We identify the memory accesses in thread functions and construct a memory-access pair graph (MPG) that captures the potential epo and eco edges in the executions.
- 3) *Checking robustness.* If an MPG contains a cycle then we check whether each access pair on the cycle is ordered. If so then all K -consistent execution of the program preserve M - K robustness condition and as a result all K consistent executions of these programs are also M consistent.
- 4) *Enforcing robustness.* If the memory access pairs on the cycle are not ordered we insert appropriate fences between the memory access pairs. These fences disallow these cycle in the executions in the K consistency model and in turn enforce M - K robustness.

A. MPG Construction

Let $\{f_1, f_2, \dots, f_n\}$ be the set of thread functions in a program that may run in parallel. Let $\mathcal{C} = \langle \mathcal{V}, \mathcal{E} \rangle$ be a control

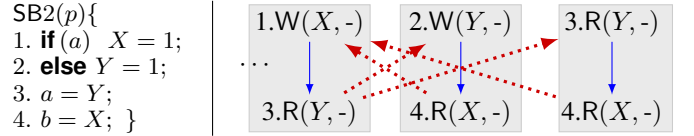


Fig. 5: Subgraph of SB2 MPG with potential epo and eco edges. $SB2(\text{true}) \parallel SB2(\text{false})$ violates SC-x86 robustness.

flow graph (CFG) of a thread function where $\mathcal{C.V}$ are the instruction nodes and $\mathcal{C.E}$ are the set of control flow edges. We analyze the thread functions' CFGs to construct an MPG.

Helper Definitions. We define following helper conditions.

- $CFG(f)$ returns the control-flow-graph of a function f .
- $ma\ AA(i, j)$ checks if i and j may access same location.
- $ac(\mathcal{C}, A)$ returns the primitives in \mathcal{C} which create A events or rmw relations i.e. $ac(\mathcal{C}, A) \triangleq \{i \mid \llbracket i \rrbracket \in A\}$. In this case $ac(\mathcal{C}, rmw)$ returns the accesses that create RMW primitives.
- $\mathcal{P}(\mathcal{C}, i, j)$ checks if there is a path from i to j on the control flow graph \mathcal{C} i.e. $\mathcal{P}(\mathcal{C}, i, j) \triangleq (i, j) \in [\mathcal{C.V}]^+; \mathcal{C.E}^+; [\mathcal{C.V}]$.
- $MM(\mathcal{C})$ returns the set of memory access pairs in a control flow graph \mathcal{C} where the second access is reachable from the first access. These pairs depict the potential epo edges i.e. $MM(\mathcal{C}) \triangleq \{(i, j) \mid i, j \in ac(\mathcal{C}, W \cup R) \wedge \mathcal{P}(\mathcal{C}, i, j)\}$.

Definition 2. An MPG is of the form $\mathbb{G} = \langle \mathbb{V}, \mathbb{E} \rangle$ where $\mathbb{G.V}$ is the set of shared memory access pairs and $\mathbb{G.E}$ denote the set of edges between the nodes. An edge from $(a, b) \in \mathbb{G.V}$ to $(c, d) \in \mathbb{G.V}$ implies that b and c may access same location.

Procedure BuildG in Fig. 6 constructs an MPG. In BuildG line 2-4 appends the memory access pairs from $CFG(f_1), CFG(f_1), \dots, CFG(f_n)$ to \mathbb{V} . Line 5-8 compute the $\mathbb{G.E}$ edges. An edge between (a, b) and (c, d) denotes that $ma\ AA(b, c)$ holds. Note that we also create $\mathbb{G.E}$ edges between access pairs from the same thread function. It is because multiple concurrent threads may execute same thread function and access pairs from a function may result in events which are concurrent in an execution. In this case we effectively analyze all programs of the form $f_1 \parallel \dots \parallel f_1 \parallel \dots \parallel f_n \parallel \dots \parallel f_n$.

B. Checking robustness on MPG

A cycle in MPG \mathbb{G} implies a potential $(epo; eco)^+$ cycle in an execution. $\mathcal{C}(\mathbb{G})$ returns the set of access pairs that may create cycle(s) in the MPG \mathbb{G} i.e.

$$\mathcal{C}(\mathbb{G}) \triangleq \{n \mid n \in \mathbb{G.V} \wedge \exists m, o \in \mathbb{G.V}. \\ m \neq n \wedge o \neq n \wedge \mathbb{G.E}(m, n) \wedge \mathbb{G.E}(n, o)\}$$

We do create any self loop in \mathbb{G} on n . A self loop on n implies that n may create concurrent event pair (p, q) and (r, s) in an execution where $eco(q, r)$ or $eco(p, s)$ holds which implies $(p, q), (r, s) \in po_\ell$. However, po_ℓ is included in all M - K robustness condition and therefore multiple event pairs from n does not create any new robustness violation.

If $\mathcal{C}(\mathbb{G})$ has any unordered access pair following respective Ord condition then we report M - K robustness violation.

example. Consider the SB2 function in Fig. 5. The program SB2(true) || SB2(false) violates SC-x86 robustness due to an execution where R(Y,0) and R(X,0) is possible in the first and second threads respectively. We construct the MPG from {1, 2, 3, 4} accesses. The subgraph in Fig. 5 contains a cycle of (1, 3) and (2, 4) that depicts SC-x86 robustness violation.

1) Defining Ord Conditions

To define an Ord condition we use the following definitions.

- $\text{mustAA}(i, j)$ checks if i and j always access same location.
- Procedure $\text{getG}(i)$ returns the CFG \mathcal{C} of instruction i .
- \mathcal{P}_{nf} checks if there exist any path from i to j on the CFG \mathcal{C} without passing through a fence in F . Else in all executions the events from i and j are ordered by a set of fences.

$$\mathcal{P}_{\text{nf}}(\mathcal{C}, i, j, F) \triangleq \mathcal{P}(\langle \mathcal{C}.\mathcal{V} \setminus F, \mathcal{C}.\mathcal{E} \setminus B \rangle, i, j) \\ \text{where } B = (G.\mathcal{V} \times F) \cup (F \times G.\mathcal{V})$$

- $\text{isW}(i)$ and $\text{isR}(i)$ check if the access i is write and read respectively.
- $\text{isWR}(\mathcal{C}, i, j)$ checks if i and j are write-read pair which may access different locations without any intermediate RMW. In an execution i and j may create a WR relation.

$$\text{isWR}(\mathcal{C}, i, j) \triangleq \text{isW}(i) \wedge \text{isR}(j) \wedge \neg \text{mustAA}(i, j) \\ \wedge \exists u (u \in \text{ac}(\mathcal{C}, \text{rmw}) \\ \wedge \mathcal{P}(\mathcal{C}, i, u) \wedge \mathcal{P}(\mathcal{C}, u, j))$$

x86. The Ord condition for SC-x86 robustness is as follows.

$$\text{Ord}(\text{SC}, \text{x86}, \mathcal{C}, i, j) \triangleq \text{isR}(i) \vee \text{isW}(j) \vee \text{mustAA}(i, j) \\ \vee \neg \mathcal{P}_{\text{nf}}(\mathcal{C}, i, j, \text{ac}(\mathcal{C}, F))$$

The $\text{isR}(i)$ and $\text{isW}(j)$ conditions ensure **xppo** relations between the events generated from i and j . $\text{mustAA}(i, j)$ checks if i and j generated events pairs are in **epo_ℓ** relation. The \mathcal{P}_{nf} condition checks if there are intermediate fences between i and j generated events in all executions. The Ord condition is satisfied in LB and IRIW but violated in the SB program.

In x86 a successful RMW results in **rmw** which acts as an intermediate fence. But a failed RMW generates a read event only and it does not act as a fence. Therefore an RMW operation between a pair of memory access does not ensure that the access pair is ordered in all execution. However, if an RMW is used in a *wait-loop* where the loop terminates only when the RMW is successful then the RMW in the *wait-loop* acts as a fence in all x86 terminating executions. For these programs we strengthen SC-x86 robustness checking condition as follows.

$$\text{SOrd}(\text{SC}, \text{x86}, i, j) \triangleq \text{isR}(i) \vee \text{isW}(j) \vee \text{mustAA}(i, j) \\ \vee \neg \mathcal{P}_{\text{nf}}(\mathcal{C}, i, j, \text{ac}(\mathcal{C}, F \cup \text{rmw}))$$

ARMv8(A8). $\text{isL}(i)$, $\text{isA}(i)$, $\text{isAQ}(i)$ check if an access i is a release, acquire, acquire/acquirePC respectively. $\text{isLA}(i, j)$ holds for a release, acquire access pair (i, j) . $\text{Lcoi}(i)$ returns the set of release-writes that access same-location as

i . $\text{RA}(\mathcal{C}, i)$ returns the set of acquire-reads that is reachable from i through some release-writes.

$$\text{RA}(\mathcal{C}, i) \triangleq \{a \mid \text{isA}(a) \wedge \neg \mathcal{P}_{\text{nf}}(\mathcal{C}, i, a, \text{ac}(\mathcal{C}, L))\} \\ \text{Lcoi}(\mathcal{C}, i) \triangleq \{ \mid \text{isL}(\) \wedge \text{mustAA}(\ , i) \}$$

We now define the Ord condition for SC-ARMv8 robustness where $B \triangleq \text{ac}(\mathcal{C}, F) \cup \text{RA}(i)$. It results in $B_F = \text{po}; [F]; \text{po} \cup \text{po}; [L]; \text{po}[A]; \text{po} \subseteq \text{bob}$ that acts as a fence on an **epo**. Moreover we define $\text{isRR}(i, j) \triangleq \text{isR}(i) \wedge \text{isR}(j)$, $\text{isRW}(i, j) \triangleq \text{isR}(i) \wedge \text{isW}(j)$, $\text{isWW}(i, j) \triangleq \text{isW}(i) \wedge \text{isW}(j)$.

$$\text{Ord}(\text{SC}, \text{A8}, \mathcal{C}, i, j) \triangleq \text{mustAA}(i, j) \quad (1)$$

$$\vee (\neg \mathcal{P}_{\text{nf}}(\mathcal{C}, i, j, B)) \vee \text{isLA}(i, j) \vee \text{isAQ}(i) \vee \text{isL}(j) \quad (2)$$

$$\vee (\text{isRR}(i, j) \wedge \neg \mathcal{P}_{\text{nf}}(\mathcal{C}, i, j, B \cup \text{ac}(\mathcal{C}, F_{\text{LD}}))) \quad (3)$$

$$\vee (\text{isRW}(i, j) \wedge \neg \mathcal{P}_{\text{nf}}(\mathcal{C}, i, j, B \cup \text{ac}(\mathcal{C}, F_{\text{LD}}) \cup \text{Lcoi}(\mathcal{C}, j))) \quad (4)$$

$$\vee (\text{isWW}(i, j) \wedge \neg \mathcal{P}_{\text{nf}}(\mathcal{C}, i, j, B \cup \text{ac}(\mathcal{C}, F_{\text{ST}}) \cup \text{Lcoi}(\mathcal{C}, j))) \quad (5)$$

The definition ensures that the generated events from i and j are in (1) po_ℓ or in one of the following bob relations: (2) $B_F \cup [L]; \text{po}; [A] \cup [A \cup Q]; \text{po} \cup \text{po}; [L]$, (3) $B_F \cup [R]; \text{po}; [F_{\text{LD}}]; \text{po}$, (4) $B_F \cup [R]; \text{po}; [F_{\text{LD}}]; \text{po} \cup \text{po}; [L]; \text{coi}$, (5) $B_F \cup [W]; \text{po}; [F_{\text{ST}}]; \text{po}; [W] \cup \text{po}; [L]; \text{coi}$. The overall condition ensures SC-ARMv8 robustness. The condition is satisfied in IRIW but violated in SB and LB.

The dob and aob relations also order memory accesses. From the definition $\text{aob} \subseteq \text{po}_\ell$ which is already captured by (1). We do not include dob in the Ord condition as a dependency can be optimized away after the robustness analysis which may result in a non-robust program even when we report the original program to be robust.

Next, we define x86-ARMv8 robustness condition where an (i, j) access pair is ordered or may generate a WR pair.

$$\text{Ord}(\text{x86}, \text{A8}, \mathcal{C}, i, j) \triangleq \text{Ord}(\text{SC}, \text{A8}, \mathcal{C}, i, j) \vee \text{isWR}(\mathcal{C}, i, j)$$

SB and IRIW satisfy the condition but LB violates it.

ARMv7(A7). We define the Ord condition to ensure the SC-ARMv7 robustness condition in all ARMv7 executions. Then we extend the Ord for SC-ARMv7 to define the Ord conditions for x86-ARMv7 and ARMv8-ARMv7 robustness.

$$\text{Ord}(\text{SC}, \text{A7}, \mathcal{C}, i, j) \triangleq \text{mustAA}(i, j) \vee (\neg \mathcal{P}_{\text{nf}}(\mathcal{C}, i, j, \text{ac}(\mathcal{C}, F)))$$

$$\text{Ord}(\text{x86}, \text{A7}, \mathcal{C}, i, j) \triangleq \text{Ord}(\text{SC}, \text{A7}, \mathcal{C}, i, j) \vee \text{isWR}(\mathcal{C}, i, j)$$

$$\text{Ord}(\text{A8}, \text{A7}, \mathcal{C}, i, j) \triangleq \text{Ord}(\text{SC}, \text{A7}, \mathcal{C}, i, j) \vee \text{isW}(i)$$

The memory access pairs in the LB program satisfies the ARMv8-ARMv7, and the SB program satisfies the x86-ARMv7, ARMv8-ARMv7 conditions.

2) Robustness Analysis and Enforcement Procedure

The MKRobust procedure in Fig. 6 checks M - K robustness on an MPG \mathbb{G} : (line 3) we first compute \mathcal{C} (\mathbb{G}). (line 4-7) if an access pair (a, b) in \mathcal{C} (\mathbb{G}) is on a cycle then we check if (a, b) is ordered by the Ord condition. (line 8) returns the unordered memory access pairs O .

If O is empty then the program is M - K robust. Else Enforce procedure insert appropriate fences to enforce robustness. Procedure getF returns a fence based on the access type a and


```

1: procedure BuildG( $\{f_1, \dots, f_n\}$ )
2:   for  $f \in \{f_1, \dots, f_n\}$  do
3:      $\mathcal{C} \leftarrow \text{CFG}(f)$ ;
4:      $\mathbb{V} \leftarrow \mathbb{V} \cup \text{MM}(\mathcal{C})$ ;
5:   for  $(a, b) \in \mathbb{V}$  do
6:     for  $(c, d) \in \mathbb{V}$  do
7:       if  $\text{ma AA}(b, c)$  then
8:          $\mathbb{E} \leftarrow \mathbb{E} \cup \{(a, b), (c, d)\}$ ;
9:   return  $\langle \mathbb{V}, \mathbb{E} \rangle$ ;
10: end procedure

1: procedure MKRobust( $M, K, \mathbb{G}$ )
2:    $O \leftarrow \emptyset$ ;
3:    $AB \leftarrow \mathcal{C}(\mathbb{G})$ ;
4:   for  $(a, b) \in AB$  do
5:      $\mathcal{C} \leftarrow \text{getG}(b)$ ;
6:     if  $\neg \text{Ord}(M, K, \mathcal{C}, a, b)$  then
7:        $O \leftarrow O \cup \{(a, b)\}$ ;
8:   return  $O$ ;
9: end procedure

1: procedure Enforce( $K, O$ )
2:    $H \leftarrow \emptyset$ ;
3:   for  $(a, b) \in O$  do
4:     if  $b \notin H$  then
5:        $f \leftarrow \text{getF}(K, a, b)$ ;
6:        $\text{insertF}(\text{getG}(b), a, b, f)$ ;
7:        $H \leftarrow H \cup \{b\}$ ;
8: end procedure

 $\mathbb{G} \leftarrow \text{BuildG}(\{f_1, \dots, f_n\})$ ;  $O \leftarrow \text{MKRobust}(M, K, \mathbb{G})$ ;  $\text{Enforce}(K, O)$ ;

```

Fig. 6: Static M - K robustness analysis and enforcement.

```

1: procedure getF( $K, a, b$ )
2:   if  $K == \text{x86}$  then return  $\text{new}(\text{MFENCE})$ ;
3:   if  $K == \text{A7}$  then return  $\text{new}(\text{DMB})$ ;
4:   if  $K == \text{A8}$  then
5:     if  $\text{isW}(a) \wedge \text{isR}(b)$  then return  $\text{new}(\text{DMBFULL})$ ;
6:     if  $\text{isW}(a) \wedge \text{isW}(b)$  then return  $\text{new}(\text{DMBST})$ ;
7:     if  $\text{isR}(a)$  then return  $\text{new}(\text{DMBLD})$ ;
8:   end procedure

1: procedure insertF( $\mathcal{C}, a, b, f$ )
2:    $\mathcal{V}' \leftarrow \mathcal{C}.\mathcal{V} \cup \{f\}$ ;
3:    $\mathcal{E}_1 \leftarrow \mathcal{C}.\mathcal{E} \cup \{(f, b)\}$ 
4:    $\mathcal{E}' \leftarrow \mathcal{E}_1 \cup \{(e, f) \mid \mathcal{C}.\mathcal{E}^+(e, b)\} \cup \{(f, e) \mid \mathcal{C}.\mathcal{E}^+(b, e)\}$ 
5:   return  $\langle \mathcal{V}', \mathcal{E}' \rangle$ ;
6: end procedure

```

Fig. 7: Procedure getF and insertF.

b in the memory model K . Procedure insertF inserts the fence f between a and b . Note that one inserted fence may order multiple access pairs. These methods are defined in Fig. 7. In case of x86 and ARM programs we insert MFENCE and DMB respectively. In ARMv8 we first insert DMBFULL followed by DMBLD and then DMBST fences.

C. Complexity of Robustness

To analyze the complexity of the robustness algorithm we analyze the main procedures: BuildG, MKRobust, and Enforce which perform MM, \mathcal{P}_{nf} , and \mathcal{C} computations. Given a program with n statements, the number of shared memory accesses and control flow edges are bound by n and n^2 respectively. Hence MM contain maximum n^2 elements and \mathcal{P}_{nf} computation is bound by traversing n^2 edges. So procedure BuildG constructs an MPG graph with maximum $|\text{MM}| = n^2$ nodes and $|\text{MM}|^2 = n^4$ edges. Hence \mathcal{C} computation traverses maximum n^4 edges. In procedure MKRobust, for each node in MPG, we check (i) if it is on the cycle by computing \mathcal{C} (ii) if yes then it performs \mathcal{P}_{nf} computation for the memory access pair. Hence MKRobust overall incurs $n^2 * (n^4 + n^2) = n^6 + n^4$ computation. Next, procedure Enforce takes maximum n^2 computation for each access pair in MM and for overall incurs

maximum $n^2 * |\text{MM}| = n^4$ computation. Hence, the robustness checking and enforcement computation is bounded by $O(n^6)$ which is polynomial in terms of the program size.

V. EXPERIMENTAL EVALUATION

Implementation. We implement the robustness analysis and enforcement techniques in *Fency* (for FENCE analysis) as LLVM compiler passes for x86, ARMv8, and ARMv7 programs. We leverage the existing analyses in LLVM. The CFG analyses are used to define MM, Path, \mathcal{P} , and \mathcal{P}_{nf} conditions. We define the ma AA and mustAA conditions using memory operand type and alias analyses provided in LLVM.

We run the analyses on a MacOS machine having a 2.4GHz 8-Core Intel i9 processor with 64 GB RAM.

Benchmarks. We analyze a number of well-known concurrent algorithms and data structures [14, 27] including global barrier (Barrier) construct, mutual exclusion algorithms (by Dekker, Peterson, and Lamport), different lock algorithms (e.g. Spinlock, Seqlock, Ticketlock), non-blocking write protocol (NBW), read-copy-update (RCU) programs, work-stealing queue in Cilk, and ChaseLev dequeue. These programs use C11 [28, 29] atomic accesses extensively. The release-acquire(RA)/TSO/SC versions indicate the memory model for which the respective version is developed. The number of lines in the LLVM IR (.ll) files vary between 100-400 which indicate the approximate size of an analyzed CFG.

Naive fence insertion scheme. We compare *Fency* to a naive scheme which does not use robustness information in fence insertion. The naive scheme works as follows.

- Eliminate existing fences in concurrent threads.
- Enforce robustness by fence insertion in concurrent threads.
 - (x86) Insert MFENCE after load, store, and RMW accesses.
 - (ARMv8) Insert DMBLD after non-acquire loads and DMBFULL for other memory accesses.
 - (ARMv7) Insert DMB after all memory accesses.

A. Experimental Results

In Figs. 8 and 9 we report the results of some benchmarks. The full results are in the supplementary material [15]. For comparison we also provide the number of fences required by

Prog.	SC-x86		Trencher	
	result	⟨sec	result	⟨sec
Barrier	6 0 X 2	⟨0.005	X 2	⟨0.004
Dekker-TSO	20 4 ✓ 0	⟨0.002	✓ 0	⟨0.007
Peterson-SC	14 0 X 2	⟨0.004	X 2	⟨0.013
Lamport-SC	17 0 X 4	⟨0.019	X 4	⟨0.107
Spinlock	14 0 ✓ 0	⟨0.004	✓ 0	⟨0.007
Ticketlock	12 0 ✓ 0	⟨0.004	✓ 0	⟨0.006
Seqlock	7 0 ✓ 0	⟨0.004	✓ 0	⟨0.582
RCU-offline	33 4 X 3	⟨0.038	X -	⟨0.246
Cilk-TSO	22 2 ✓ 0	⟨0.011	X 0	⟨2.039
Cilk-SC	22 0 ✓ 0	⟨0.010	✓ 2	⟨6.322

Prog.	ARMv7					
	SC		x86		ARMv8	
	result	⟨sec	result	⟨sec	result	⟨sec
Barrier	6 2 X 2	⟨0.012	6 2 ✓ 0	⟨0.002	6 2 ✓ 0	⟨0.002
Dekker-TSO	20 8 X 6	⟨0.003	20 8 X 6	⟨0.007	20 8 X 6	⟨0.009
Peterson-SC	14 0 X 12	⟨0.002	14 0 X 10	⟨0.002	14 0 X 8	⟨0.003
Lamport-SC	17 7 X 10	⟨1.699	17 7 X 8	⟨1.659	17 7 X 5	⟨1.698
Spinlock	18 12 ✓ 0	⟨0.141	18 12 ✓ 0	⟨0.133	18 12 ✓ 0	⟨0.133
Ticketlock	14 8 ✓ 0	⟨0.025	14 8 ✓ 0	⟨0.022	14 8 ✓ 0	⟨0.023
Seqlock	9 6 X 2	⟨0.006	9 6 X 2	⟨0.002	9 6 X 2	⟨0.002
RCU-offline	36 19 X 17	⟨0.335	36 19 X 15	⟨0.334	36 19 X 10	⟨0.339
Cilk-TSO	33 10 X 6	⟨2.455	33 10 X 6	⟨2.411	33 10 X 6	⟨2.427
Cilk-SC	33 8 X 7	⟨2.445	33 8 X 7	⟨2.410	33 8 X 7	⟨2.411

Fig. 8: Robustness analyses and enforcement for x86 and ARMv7 programs.

the naive schemes as well as the results from state-of-the-art x86-robustness checker Trencher [8].

Interpreting the Results. The (SC-K) entries in the tables are of the form (a|b(✓/X) c ⟨ d) where

- ‘a’: number of fences required by naive scheme.
- ‘b’: number of existing fences in the program.
- ‘c’: number of fences inserted by proposed scheme.
- ‘✓/X’ symbol denotes if a program is M - K robust or not.
- ‘d’: time taken by the robustness pass in seconds.

In ARMv8 we show total number of DMB(FULL/LD/ST) fences. We use $\#(a-(b+c))$ less fences than the naive schemes e.g. from Fig. 8 the Barrier program requires $6-(0+2)=4$ less fences than the naive scheme to enforce SC-x86 robustness.

For Trencher we analyze the encoded programs taken from [14]. We report if the program is SC-x86 robust (✓/X), number of inserted fences (i.e. ‘c’) and the execution time (i.e. ‘d’). Trencher fence insertion does not terminate for RCU-offline.

1) Checking Robustness

x86 programs. We report the SC-x86 robustness analysis results of *Fency* in Fig. 8 (and in [15]) and compare the results from Trencher. on the corresponding programs.

The SC-x86 robustness analysis in *Fency* is quite precise and agrees to Trencher in all cases except Lamport-RA, Lamport-TSO, and Cilk-SC programs. Lamport-(RA/TSO) have unordered write-read pairs that generate WR relations and hence *Fency* report SC-robustness violation though these access pairs never execute concurrently in any x86 execution. Moreover, in most cases *Fency* insert same number of fences as Trencher.

We note a subtle case in Cilk-SC. It has an access sequence $a = R_{RLX}(T); W_{RLX}(T, a-1); R_{RLX}(H)$. Trencher reports SC-violation due to the WR pair. However, LLVM combines the load and store of T and create an atomic fetch-and-sub: $a = R_{RLX}(T); W_{RLX}(T, a-1) \rightsquigarrow a = fsub(T, 1)$. Hence the resulting x86 program ensures SC-robustness which *Fency* reports correctly.

We also note the execution time of *Fency* and of Trencher. Trencher incurs significantly more time for the Seqlock, Cilk-

Prog.	ARMv8			
	SC		x86	
	result	⟨sec	result	⟨sec
Barrier	6 2 X 2	⟨0.009	6 2 X 0	⟨0.007
Dekker-TSO	20 8 X 4	⟨0.007	20 8 X 4	⟨0.011
Peterson-SC	14 0 X 11	⟨0.001	14 0 X 10	⟨0.001
Lamport-SC	17 7 X 9	⟨0.007	17 7 X 9	⟨0.008
Spinlock	18 12 X 4	⟨0.017	18 12 X 4	⟨0.009
Ticketlock	14 8 X 2	⟨0.006	14 8 X 2	⟨0.007
Seqlock	9 6 X 2	⟨0.002	9 6 X 2	⟨0.005
RCU-offline	35 16 X 17	⟨0.157	35 16 X 19	⟨0.160
Cilk-TSO	33 10 X 7	⟨0.025	33 10 X 7	⟨0.024
Cilk-SC	33 8 X 8	⟨0.011	33 8 X 8	⟨0.012

Fig. 9: Robustness analyses & enforcement in ARMv8.

TSO, Cilk-SC programs and does not terminate for RCU-offline fence insertion. Trencher exhibits comparable efficiency in certain programs e.g. Spinlock, Ticketlock. However, in these programs also if we increase the number of threads by replicating the thread functions then Trencher incurs orders of seconds to check and enforce robustness. At the same time Trencher inserts more fences. On the other hand, the analyses in *Fency* are parameterized by thread functions and therefore are unaffected by the number of executing threads.

ARMv8 programs. In Fig. 9 (and in [15]) we report the robustness results of the ARMv8 programs. The ARMv8 programs violate SC and x86 robustness as the programs contain independent memory accesses on different locations which are unordered in ARMv8.

As ARMv8 is weaker than x86, the programs (e.g. Barrier) which violate SC-x86 robustness also violate SC-ARMv8 robustness. Moreover, there are programs which are SC-x86 robust but violates SC-ARMv8 robustness such as dekker-TSO. These programs violate both SC-ARMv8 and x86-ARMv8 robustness due to unordered accesses that result in $[R]; po_{\neq \ell}; [R]$ or $[W]; po_{\neq \ell}; [W]$ relation in an execution. These access pairs are ordered in x86 but not in ARMv8 and hence violate x86-ARMv8 robustness.

Robustness of ARMv7 programs. In general the ARMv7 programs violate robustness when x86 or ARMv8 are not robust as shown in Fig. 8 (and in [15]). However, C11 release/acquire/SC accesses which generate full fences in ARMv7 and synchronizing accesses in ARMv8 which act as half fences. As a result, in some programs the ARMv7 version enforce stronger ordering than the ARMv8 version. Hence the ARMv7 programs are robust unlike the ARMv8 programs. For example, Consider the C11 event (without read/written values) sequences from Spinlock and Ticketlock programs and their C11 to ARMv8 and ARMv7 mappings [30].

$$R(X) \cdot W_{sc}(Y) \cdot R(Z) \mapsto R(X) \cdot L(Y) \cdot R(Z) \quad (\text{C-v8})$$

$$R(X) \cdot W_{sc}(Y) \cdot R(Z) \mapsto R(X) \cdot F \cdot W(Y) \cdot F \cdot R(Z) \quad (\text{C-v7})$$

The reads are unordered in ARMv8 and may violate SC-ARMv8. The ARMv7 event sequence is ordered by fences that leads to SC-ARMv7 robustness.

The Barrier (and Peterson-RA-b) program violates SC-ARMv7 due to unordered store-load pairs, but satisfies x86 and ARMv8 robustness. Some ARMv7 programs violate SC, x86, ARMv8 robustness due to unordered read-read pairs.

2) Enforcing robustness.

In most of the programs enforcing weaker model requires less number of inserted fences. However, certain ARMv8 programs (e.g. lamport-SC) incur less fences to enforce SC-ARMv8 than x86-ARMv8. Consider the ARMv8 sequence $W(X) \cdot R(X) \cdot R(Y) \cdot W(Y)$ that may violate SC-ARMv8 and x86-ARMv8. To ensure SC-ARMv8 we insert a DMBFULL that results in $W(X) \cdot R(X) \cdot F \cdot R(Y) \cdot W(Y)$ sequence. To ensure x86-ARMv8 we insert a DMBLD and a DMBST to generate a $W(X) \cdot R(X) \cdot F_{LD} \cdot R(Y) \cdot F_{ST} \cdot W(Y)$ sequence.

3) Performance of Robustness Analyses

We have already compared the execution times of SC-x86 robustness analysis in *Fency* and *Trencher*. In case of ARM program versions *Fency* incurs less than a second except for ARMv7 Cilk-(TSO/SC) programs. The timings of *Fency* analyses vary among different program versions. It is because LLVM may optimize a program differently for different architectures. So the number of memory accesses (parameter ‘a’ in Figs. 8 and 9) and the number of memory access pairs vary. Moreover, the CFGs in different architectures also differ which affect the \mathcal{P}_{nf} and \mathcal{C} computations.

VI. RELATED WORK

SC-robustness is studied against TSO [3, 4, 5, 6, 7, 8, 9, 10], PSO [11, 12], POWER [13], and Release-Acquire [14] models by exploring possible executions using model checking tools. On the contrary, we analyze and transform programs as LLVM passes without exploring program executions.

[8] check and enforce SC-robustness for parameterized programs for any number of threads. It reduces the robustness checking problem to parameterized reachability analysis on possible executions. Instead, our approach is static and parameterized over the thread functions for any number of threads.

PORTHOS [31] checks portability of a program from one model to another, particularly from POWER to TSO by encoding models in SAT/SMT solvers. On the contrary, we check robustness or portability of ARM models which are different from POWER. In addition, our analysis enable fence insertion to enforce robustness unlike PORTHOS.

A number of approaches [32, 8, 33, 34, 35, 18, 6, 11] propose fence insertion to ensure SC. Among these fence insertion schemes our approach is closer to static approaches [34, 18, 35]. [18] use delay-set analysis to ensure SC for weak memory programs. [35] proved that identifying minimal set of fences is NP-hard and proposed minimal fence insertion based on control flow analysis. Similar to [35], we analyze control flow graph without exploring the executions.

[32] checks SC-robustness against x86 and POWER, and restore SC by inserting lock-unlock or RMW constructs. [34] proposed fence insertion in POWER to strengthen a program to release/acquire semantics which has same ordering constraints between memory accesses as TSO. On the contrary, we propose M - K robustness; we define robustness conditions for ARMv7 and ARMv8 programs and show that *ppo* is not sufficient to enforce SC in ARMv7. Moreover, we analyze parameterized programs unlike these approaches.

We extend abstract event graph (AEG) from [34] and propose memory pair graph in our analyses. An AEG captures the possible execution graphs statically for a given set of threads and statically detect possible robustness-violating cycles which may occur in an execution. The proposed memory-access pair graph (MPG) also considers that the program is parameterized where each thread function may create multiple threads and hence construct the event graph on all memory access pairs from all threads. Then similar to AEG we statically detect possible robustness-violating cycles on MPG. However, our fence insertion may not be optimal; identifying optimal fence insertion is an well studied problem [35, 18, 34] which we will pursue in the context of M - K robustness.

VII. CONCLUSION AND FUTURE WORK

In this paper we identify robustness conditions for x86, ARMv8, and ARMv7 relaxed memory models. Based on these identified conditions we check M - K robustness. If robustness is violated we insert appropriate fences to enforce robustness. We implement our approach as LLVM compiler passes and evaluate the efficiency on a number of well-known concurrent algorithms and data structures.



Going forward we want to extend the analyses to other concurrency features in x86 and ARM models [36]. We would also like to extend these analyses to other architectures such as RISC-V [37] and Power [38].

REFERENCES

- [1] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran, “Edge computing: the case for heterogeneous-isa container migration,” in *VEE’20*, 2020, pp. 73–87.

- [2] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H. Chuang, V. Legout, and B. Ravindran, “Breaking the boundaries in heterogeneous-isa datacenters,” in *ASPLOS 2017*, 2017, pp. 645–659.
- [3] S. Burckhardt and M. Musuvathi, “Effective program verification for relaxed memory models,” in *CAV’08*, 2008, pp. 107–120.
- [4] A. Bouajjani, R. Meyer, and E. Möhlmann, “Deciding robustness against total store ordering,” in *ICALP’11*, 2011, pp. 428–440.
- [5] J. Burnim, K. Sen, and C. Stergiou, “Sound and complete monitoring of sequential consistency for relaxed memory models,” in *TACAS’11*, 2011, pp. 11–25.
- [6] A. Linden and P. Wolper, “A verification-based approach to memory fence insertion in relaxed memory systems,” in *SPIN’11*, 2011, pp. 144–160.
- [7] A. Gotsman, M. Musuvathi, and H. Yang, “Show no weakness: Sequentially consistent specifications of tso libraries,” 2012.
- [8] A. Bouajjani, E. Derevenetc, and R. Meyer, “Checking and enforcing robustness against TSO,” in *ESOP 2013*, 2013, pp. 533–553.
- [9] P. A. Abdulla, M. F. Atig, and T.-P. Ngo, “The best of both worlds: Trading efficiency and optimality in fence insertion for tso,” in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2015, pp. 308–332.
- [10] A. Bouajjani, C. Enea, S. O. Mutluergil, and S. Tasiran, “Reasoning about tso programs using reduction and abstraction,” in *CAV’18*, 2018, pp. 336–353.
- [11] A. Linden and P. Wolper, “A verification-based approach to memory fence insertion in pso memory systems,” in *TACAS’13*, 2013, pp. 339–353.
- [12] P. A. Abdulla, M. F. Atig, M. Lång, and T. P. Ngo, “Precise and sound automatic fence insertion procedure under pso,” in *Networked Systems*, 2015, pp. 32–47.
- [13] E. Derevenetc and R. Meyer, “Robustness against power is pspace-complete,” in *ICALP’14*, ser. LNCS, vol. 8573, 2014, pp. 158–170.
- [14] O. Lahav and R. Margalit, “Robustness against release/acquire semantics,” in *PLDI 2019*, 2019, pp. 126–141.
- [15] S. Chakraborty, “Technical appendix.” 2021, available at <https://www.st.ewi.tudelft.nl/sschakraborty/mkrobustness.html>.
- [16] A. Podkopaev, O. Lahav, and V. Vafeiadis, “Promising compilation to armv8,” in *ECOOP’17*, 2017.
- [17] “The LLVM compiler infrastructure,” <http://llvm.org/>.
- [18] D. E. Shasha and M. Snir, “Efficient and correct execution of parallel programs that share memory,” *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988.
- [19] J. Alglave, L. Maranget, and M. Tautschnig, “Herd cats: modelling, simulation, testing, and data-mining for weak memory,” *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, 2014.
- [20] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, “Repairing sequential consistency in C/C++11,” in *PLDI 2017*, 2017, pp. 618–632, technical Appendix Available at <https://plv.mpi-sws.org/scfix/full.pdf>.
- [21] J. Alglave and L. Maranget, “herd7 consistency model simulator,” <http://diy.inria.fr/www/>.
- [22] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, “Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8,” *PACMPL*, vol. 2, no. POPL, pp. 19:1–19:29, 2018.
- [23] “SC cat file,” 2021, available at <https://github.com/herd/herdtools7/blob/master/herd/libdir/sc.cat>.
- [24] “x86 cat file,” 2021, available at <https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso.cat>.
- [25] “Armv8 cat file,” 2021, available at <https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat>.
- [26] “Armv7 cat file,” 2021, available at <https://github.com/herd/herdtools7/blob/master/herd/libdir/arm.cat>.
- [27] B. Norris and B. Demsky, “CDSChecker: Checking concurrent data structures written with C/C++ atomics,” in *OOPSLA’13*, 2013.
- [28] ISO/IEC 9899, “Programming language C,” 2011.
- [29] ISO/IEC 14882, “Programming language C++,” 2011.
- [30] “C/C++11 mappings to processors,” <https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html>.
- [31] H. Ponce de León, F. Furbach, K. Heljanko, and R. Meyer, “Portability analysis for weak memory models. porthos: One tool for all models,” 08 2017.
- [32] J. Alglave and L. Maranget, “Stability in weak memory models,” in *CAV 2011*, 2011, p. 50–66.
- [33] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav, “Dynamic synthesis for relaxed memory models,” in *PLDI ’12*, 2012, pp. 429–440.
- [34] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, “Don’t sit on the fence: A static analysis approach to automatic fence insertion,” *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 6:1–6:38, 2017.
- [35] J. Lee and D. A. Padua, “Hiding relaxed memory consistency with a compiler,” *IEEE Transactions on Computers*, vol. 50, no. 8, pp. 824–833, 2001.
- [36] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard, and L. Maranget, “Armed cats: Formal concurrency modelling at arm,” vol. 43, no. 2, 2021.
- [37] “RISC-V specification.” <https://riscv.org/technical/specifications/>.
- [38] “Powerisa public.v3.1.” https://wiki.raptorcs.com/wiki/File:PowerISA_public.v3.1.pdf.

Pruning and Slicing Neural Networks using Formal Verification

Ori Lahav  and Guy Katz 

The Hebrew University of Jerusalem, Jerusalem, Israel

{ori.lahav, guykatz}@cs.huji.ac.il

Abstract—Deep neural networks (DNNs) play an increasingly important role in various computer systems. In order to create these networks, engineers typically specify a desired topology, and then use an automated training algorithm to select the network’s weights. While training algorithms have been studied extensively and are well understood, the selection of topology remains a form of art, and can often result in networks that are unnecessarily large — and consequently are incompatible with end devices that have limited memory, battery or computational power. Here, we propose to address this challenge by harnessing recent advances in DNN verification. We present a framework and a methodology for discovering redundancies in DNNs — i.e., for finding neurons that are not needed, and can be removed in order to reduce the size of the DNN. By using sound verification techniques, we can formally guarantee that our simplified network is equivalent to the original, either completely, or up to a prescribed tolerance. Further, we show how to combine our technique with *slicing*, which results in a *family* of very small DNNs, which are together equivalent to the original. Our approach can produce DNNs that are significantly smaller than the original, rendering them suitable for deployment on additional kinds of systems, and even more amenable to subsequent formal verification. We provide a proof-of-concept implementation of our approach, and use it to evaluate our techniques on several real-world DNNs.

I. INTRODUCTION

The wide-spread adoption of *deep learning* [17] has caused a significant leap forward in many domains within computer science. *Deep neural networks (DNNs)* have now become the state of the art solution for a myriad of real-world problems, such as game playing [40], image recognition [41], and autonomous vehicles [5], [25]. This trend is likely to continue and intensify, thus creating an urgent need for tools and techniques to analyze and manipulate DNNs.

A part of the appeal of DNNs is that they are produced in a mostly automated way. In order to create a DNN for a particular task at hand, engineers first specify the network architecture — specifically, the number of layers in the network, the size and type of each layer, and the inter-layer connections. Then, they invoke an automated training algorithm for assigning weights to the network’s edges [17]. While the automated training process has been extensively studied and is generally well understood [17], the choice of network architecture is still performed according to various rules of thumb, and is considered a form of art. This can often lead to a choice of architecture that is wasteful — i.e., which results in a large DNN, whereas a smaller DNN could have achieved similar accuracy [15], [19], [23]. For DNNs intended to run on devices

with limited resources (e.g., mobile phones, or embedded circuits), excessive DNN size can be a limiting factor [25].

One successful approach for mitigating this difficulty is to first train a large network, and then shrink it by removing *redundant neurons*. Informally, we say that a neuron is redundant if removing it does not change the DNN’s output; and thus, removing it from a network N results in a smaller network, N' , that is *equivalent* to N . In order to identify redundant neurons within a DNN, prior work has focused primarily on *heuristic pruning*: heuristically identifying neurons and edges that contribute little to the network’s output, removing these neurons, and then performing additional training of the network [19], [23]. These methods have been highly successful in reducing DNN sizes, but they provide no formal guarantees; i.e., the removed neurons are not guaranteed to have been redundant, and the simplified network can thus be dramatically different from the original, producing different results for various inputs [35].

Recently, there has been a surge of interest in the formal verification of neural networks (e.g., [2], [14], [20], [26], [28], [32], [46], and many others). These new capabilities have made it possible to identify and remove redundancies in a network, in a way that *guarantees* that the smaller network is completely equivalent to the original [15]. Specifically, Gokulanathan et al. showed how verification could be used to identify and remove “dead” neurons, i.e. neurons whose output is 0 regardless of the network’s inputs. This approach was shown to reduce network sizes by up to 10%, which is quite significant, while preserving complete equivalence to the original network.

Here, we propose a new technique, which also attempts to apply formal verification in order to remove neurons from a DNN, but which is significantly stronger. Specifically, our technique: (i) can identify additional kinds of redundant neurons (beyond “dead” neurons), whose removal does not affect the network’s outputs at all; and (ii) can identify additional redundant neurons, whose removal *does* affect the network’s outputs, but only up to a small, provable bound.

Finally, we propose a method that takes our approach to the extreme, by integrating it with *network slicing*. This method, in which a network is simplified into a family of much smaller sub-networks, is appropriate for cases where fast inference is crucial: an input is checked to identify the appropriate sub-network for handling it, and then only that network needs to be evaluated for that specific input. Slicing is achieved by

partitioning the DNN’s input domain into small sub-domains, maintaining a separate DNN for each input sub-domain, and then applying the aforementioned simplification techniques on each of these DNNs. We demonstrate that the use of small input sub-domains causes many neurons to become redundant, and consequently removable.

For evaluation purposes, we implemented our approach in an open-source, publicly available tool [33]. As a backend, our tool uses the Marabou DNN verification tool [29]. We note, however, that our approach is agnostic of the underlying verification engine — indeed, it could be integrated with any other tool, and will consequently benefit from any development in DNN verification technology. We evaluated our approach on a set of airborne collision avoidance networks [25], obtaining highly favorable results. Specifically, we were able to achieve a reduction of up to 71% in overall network sizes, while keeping the outputs identical (up to a prescribed tolerance) to those produced by the original DNN. This reduction in network sizes is a significant improvement over the previous state of the art [15]. Further, while prior techniques were specifically tailored to networks with only a specific activation function (i.e., rectified linear units [15]), our technique is applicable to multiple kinds of DNNs.

The rest of this paper is organized as follows. In Section II, we provide the necessary background on DNNs and their verification. Next, in Section III we present the basic building block of our approach, namely the removal of a single neuron. We then specify multiple kinds of neurons that can be removed in Section IV, and discuss the simultaneous removal of neurons in Section V. Subsequently, in Section VI we present how *input slicing* and simplification can be used to improve network evaluation time. An evaluation appears in Section VII, followed by a discussion of related work in Section VIII. We then conclude in Section IX.

II. BACKGROUND: DNNs AND THEIR VERIFICATION

A deep neural network [17] is a directed, acyclic graph, whose nodes (also referred to as *neurons*) are grouped into layers. The first layer is the *input layer*; the final layer is the *output layer*; and the intermediate layers are the *hidden layers*. When the network is evaluated, the input neurons are assigned some values (e.g., sensor readings), and these values are then propagated through the network, layer by layer, until the output values are computed. In *regression* networks, the numeric value of the output is of interest, while in the case of *classification* networks, the output neurons correspond to possible *labels* that the network can classify the input into; and the label whose neuron obtained the highest score is the one returned by the network.

Each layer in the DNN has a type, which determines how its neuron values are computed. Here, we will focus on two types: *weighted sum* layers, and *piecewise-linear activation* layers. In a weighted-sum layer, the value of a neuron y is computed as $y = b + \sum c_i v_i$ for neurons v_i from preceding layers, where the *weights* c_i are determined when the network

is first trained. In a piecewise-linear activation layer, the value of neuron y is computed as

$$y = \begin{cases} a_1 x + b_1 & \text{if } s_1 \leq x < s_2, \\ a_2 x + b_2 & \text{if } s_2 \leq x < s_3, \\ \dots & \dots \\ a_k x + b_k & \text{if } s_k \leq x \leq s_{k+1} \end{cases}$$

where x is a neuron from some preceding layer, and the a_i , b_i and s_i parameters determine the piecewise linear function being computed. A common example of a piecewise-linear activation function is the ReLU function, given by

$$y = \max(x, 0) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

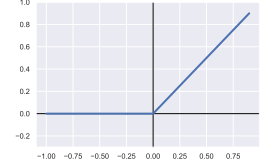


Fig. 1: The ReLU function.

(see Fig. 1). Together, weighted-sum layers and piecewise-linear activation functions make up many common DNN architectures [17]. Typically, they are used in alternation (see Fig. 2). Extending our approach to activation functions that are not piecewise-linear remains a work in progress.

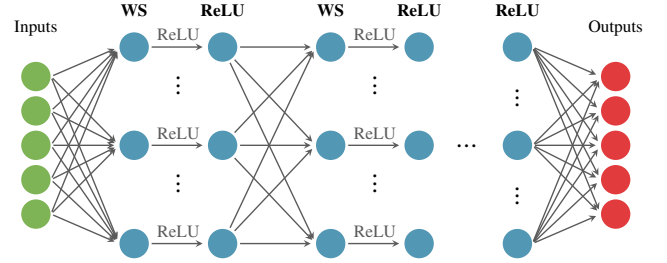


Fig. 2: An illustration of a DNN with alternating weighted-sum (WS) and ReLU layers.

More formally, we regard a DNN N with k inputs and m outputs as a mapping $\mathbb{R}^k \rightarrow \mathbb{R}^m$. The DNN is given as a sequence of layers L_1, \dots, L_n , where L_1 is the input layer and L_n is the output layer. We use s_i to denote the size of layer L_i , and use $v_i^1, \dots, v_i^{s_i}$ to refer to the individual neurons of L_i . We use V_i to refer to the column vector $[v_i^1, \dots, v_i^{s_i}]^T$. When the network is being evaluated, we assume that the input values V_1 are given, and that V_2, \dots, V_n are computed iteratively. The type of each hidden layer is given via the mapping $T_N : \mathbb{N} \rightarrow \mathcal{T}$. For simplicity we set $\mathcal{T} = \{\text{weighted-sum, ReLU}\}$, although our technique applies to all types of piecewise-linear activation functions.

In a weighted-sum layer L_i , each neuron v_i^j is associated with a linear function $v_i^j = b_i^j + \sum c_{l,t} \cdot v_l^t$; i.e., v_i^j is computed as a weighted-sum of neurons v_l^t from preceding layers $l < i$, plus a bias value b_i^j . In a ReLU layer L_i , each neuron v_i^j is associated with a specific neuron v_l^t from a preceding layer $l < i$, and its value is given by $v_i^j = \text{ReLU}(v_l^t) = \max(v_l^t, 0)$. Note that each neuron’s value depends only on neurons from preceding layers.

In recent years, various security and safety issues have been discovered in DNNs [26], [43]. This has led the verification community to study the *DNN verification problem* [36]. Generally, this problem is defined by a set of constraints P on the DNN's inputs, and a set of constraints Q on the DNN's outputs; and solving it entails finding (or proving the non-existence of) an input x such that $P(x) \wedge Q(N(x))$; i.e., an input x that satisfies the input condition, and is mapped by the DNN to a point that satisfies the output condition. When P and Q characterize an unsafe behavior of the DNN, an UNSAT answer to the aforementioned query indicates that the DNN is safe; whereas a SAT answer, accompanied by a satisfying assignment, demonstrates an unsafe behavior. This formalization is sufficiently expressive for capturing many properties of interest [26]. Many approaches for solving the DNN verification problem have been proposed recently (e.g., [14], [20], [26], [46], and many others). The techniques we discuss in this work use a DNN verification engine as a backend, and do not depend on the precise method used — and so we do not elaborate on this topic. We refer the interested reader to [36] for a survey.

III. REMOVING A SINGLE NEURON

The core of our DNN simplification approach is the identification, and then the removal, of *redundant neurons*. Given a DNN N , we seek to identify a redundant neuron v_i^j , and then produce another network, N' , which is identical to N except for the redundant neuron that has been removed. Ideally, we would like to ensure that N and N' are equivalent; i.e., that $\forall x. N(x) = N'(x)$. Because N' is obtained from N by removing a neuron, it is smaller; and this process can be repeated iteratively, to eventually obtain a significantly smaller network that is equivalent to N . Of course, the key points that need addressing are: (i) how to technically remove a redundant neuron from the network; and (ii) how to identify redundant neurons. In this section we focus on the first challenge, and describe the mechanics of removing a neuron.

In order to maintain compatibility with the original network, we will refrain from removing neurons from the network's input or output layers; all other neurons are considered candidates for removal. We distinguish between neurons in weighted-sum layers, and neurons in activation function layers. In fact, our proposed approach only supports the removal of weighted-sum neurons that feed only into other weighted-sum neurons; and the removal of activation function neurons will be performed by first transforming them into weighted-sum neurons, as described in later sections.

Consider a neuron v computed as a weighted-sum

$$v = b_v + \sum c_i \cdot x_i,$$

where x_i are neurons from preceding layers. Suppose that v only feeds into other weighted-sum neurons, and let u be such a neuron:

$$u = b_u + c \cdot v + \sum d_i \cdot y_i,$$

where y_i are again neurons from preceding layers. In this case, u 's equation can be updated into:

$$u = (b_u + c \cdot b_v) + \sum c \cdot c_i \cdot x_i + \sum d_i \cdot y_i.$$

If this process is repeated for every (weighted-sum) neuron that v feeds into, then afterwards v will have no outgoing edges. Consequently, v could then be eliminated from the network altogether. It is straightforward to show that such an operation will never affect the value of u , and that the modified network will thus be completely equivalent to the original. Also, identifying neurons that can be eliminated is simple, and amounts to searching for weighted-sum neurons that are only connected to other weighted-sum neurons.

In practice, DNN topology usually alternates between weighted-sum and activation function layers, and so consecutive weighted-sum neurons are likely to be scarce. Our strategy will thus be to replace activation function neurons with weighted-sum neurons, in a way that will enable neuron removal while preserving network accuracy. As an example, let us consider a ReLU neuron, $y = \text{ReLU}(x)$. Because of layer-type alternation, it is reasonable to assume that x is a weighted-sum neuron. In this case, if we can express y as a linear function of x , i.e. $y = ax + b$ for some a and b , then the previous case of two consecutive weighted-sum neurons applies: we can remove x entirely, change y 's type to weighted-sum, and connect y to x 's inputs. Further, if y also feeds into weighted-sum neurons, then we can apply simplification once again, and remove y as well. An illustration appears in Fig. 3.

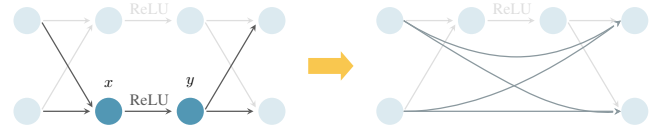


Fig. 3: Illustration: removing a neuron. x is a weighted-sum neuron which feeds into y , a ReLU neuron. After converting y into a weighted-sum neuron, both x and y can be removed.

The aforementioned steps constitute the framework of our approach — to repeat, until saturation, the two steps: (i) identify any weighted-sum neurons that only feed into weighted sum neurons, and remove them; and (ii) identify any activation function neurons that can be changed into weighted-sum neurons, without harming the network's accuracy. The key remaining issue is how to identify those neurons to which step 2 can be applied. We elaborate on this issue in the following sections.

IV. LINEARIZING ACTIVATION FUNCTIONS

We next propose various criteria for determining which activation function neuron can be changed into weighted-sum neurons. Applying these criteria in practice is discussed later, in Section V.

Phase Redundancy. In order to transform an activation function neuron into a weighted-sum neuron without changing the network's outputs, we leverage the properties of

piecewise-linear functions. Let x be a weighted-sum neuron and let $y = f(x)$ be an activation function neuron; then, by definition, the value range of x is divided into segments $[s_1, s_2], [s_2, s_3], \dots, [s_k, s_{k+1}]$, and in each segment y is a linear function (a weighted-sum) of x . If we are able to discover that x is in fact restricted to one of these segments, i.e. $s_i \leq x < s_{i+1}$ for some i , then we can safely discard the constraint $y = f(x)$ and replace it with a linear constraint $y = a_i x + b_i$, thus changing y to be a weighted-sum neuron. We stress that this change does not alter the value of y , and consequently does not alter the network’s outputs. When this phenomenon occurs, we say that y is *phase-redundant*. For the ReLU function, this happens if we discover that $x < 0$ (y is *inactive-redundant*), or $x \geq 0$ (y is *active-redundant*). As previously stated, transforming the piecewise-linear constraint into a linear one will often allow us to eliminate two neurons from the network, without changing its outputs.

Forward Redundancy. Phase-redundancy captures the case where an activation function neuron is fixed to a single linear phase, for all possible inputs. However, there actually exist *unstable* activation-function neurons, i.e. neurons not fixed to a particular linear phase, which can still be soundly transformed into weighted-sum neurons computing one of these linear phases. Intuitively, this happens when neuron y ’s assignment affects its k succeeding layers, for some $k > 0$, but gets “canceled out” in layer $k + 1$. A small, illustrative example appears in Fig. 4. When replacing y with a weighted-sum neuron only affects neurons that are at most k layers away from y , we say that y is *k-forward-redundant*. Much like phase-redundant neurons, *k-forward-redundant* neurons can be removed from the network without harming its accuracy.

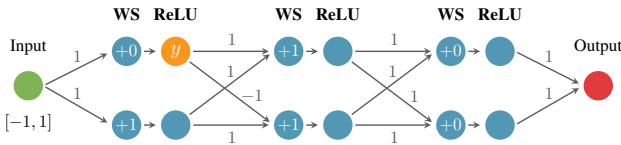


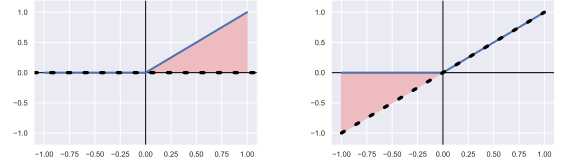
Fig. 4: The orange ReLU neuron, marked y , is 2-forward-redundant. Replacing y with a constant zero affects the following WS and ReLU layers, but it does not affect the last WS layer (and thus the network output). For example, observe that if we input 1 into the network, y evaluates to 1, and the network’s output evaluates to 12. This output value is unchanged even if we replace y ’s value with 0. A careful examination of the network reveals that this will always be the case, regardless of the network’s input value.

More formally, let v_i^j be an activation function neuron, and let N' be a network obtained from N replacing v with a weighted-sum neuron $v_i^j = b_i^j + \sum c_k x_k$. Let V_1 denote an input vector, on which both N and N' are evaluated; and let V_2, \dots, V_n and V'_2, \dots, V'_n denote the layer evaluations of N and N' (respectively) on V_1 . If, for every V_1 , it holds that $V_{i+k} = V'_{i+k}$, then we say that neuron v_i^j is *k-forward-redundant* (note that this implies $V_{i+k'} = V'_{i+k'}$ for every $k' > k$). We note that a neuron that is phase-redundant is also *k-forward-redundant*, for any $k \geq 0$.

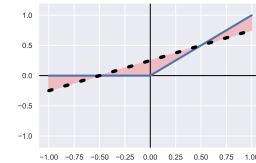
Relaxed Redundancy. So far, we discussed replacing a piecewise-linear activation neuron with a weighted-sum neuron that corresponds to one of the activation function’s linear segments; e.g., in the case of $y = \text{ReLU}(x)$, neuron y would be changed into a weighted-sum neuron computing either $y = 0$ or $y = x$. We observe that, although these linear functions are natural candidates for replacing the original constraint, in fact any linear function $y = \ell(x)$ could be used. Specifically, given an activation function $y = f(x)$ and some known lower and upper bounds lb and ub for x (computed, e.g., using interval arithmetic [26] or abstract interpretation [14], [46]), we propose to find a linear function $\ell(x)$ that has *minimal error* compared to $f(x)$. We define this error to be

$$\max_{lb \leq x \leq ub} |f(x) - \ell(x)|$$

See Fig. 5 for an illustration of replacing a ReLU constraint, whose phase is not fixed, with three linear constraints. In each illustration, the blue line is the ReLU, the dashed line is the linear replacement, and the red area is the introduced error. In case (c), the maximal introduced error (the height of the red region) is the smallest among the three options.



(a) Replacing a ReLU with the zero function (b) Replacing a ReLU with identity function



(c) Replacing a ReLU with an arbitrary linear function

Fig. 5: Replacing a ReLU with linear functions.

Unlike in the phase-redundancy and *k-forward-redundancy* cases, setting $y = \ell(x)$ will introduce some imprecision to the network’s output. The motivation is that by replacing $y = f(x)$ with $y = \ell(x)$ that has minimal error, we would be introducing only a small imprecision, while enabling the removal of y . Let e_t be some user-defined error threshold; when replacing $y = f(x)$ with $\ell(x)$ introduces an error e such that $e \leq e_t$, we say that neuron y is *relaxed-redundant*.

Let us focus on the $y = \text{ReLU}(x)$ function as an example, and suppose we know that $x \in [lb, ub]$. If $lb < 0$ and $ub > 0$, the neuron is not phase-redundant. In this case, a linear function $y = \ell_m(x)$ with minimal error can be easily

computed, and is given by:

$$l_m(x) = \frac{ub}{ub - lb} \cdot x + \frac{-lb \cdot ub}{2(ub - lb)}.$$

It is straightforward to check that the maximum error is obtained when $x = 0$, and it is given by $\frac{-lb \cdot ub}{2(ub - lb)}$ (a proof appears in Appendix 1 in the extended version of this paper [34]). Unsurprisingly, when lb or ub are close to 0, the error becomes very small — indicating that such ReLUs, which are “almost phase-redundant”, could be removed at a small cost to precision. It should be noted, however, that minimizing the maximum error introduced by the removal of a single neuron does not necessarily minimize the overall imprecision introduced to the network’s outputs.

Result-Preserving Redundancy. In classification networks, it may be acceptable to give up some precision, as long as the output label for each input is unchanged; i.e., if the original network classified input x as label l with 80% confidence, it may be acceptable to remove neurons in a way that reduces this confidence to 60%, as long as x is still classified as l .

More formally, let $y = f(x)$ be an activation neuron in a network N , and let N' denote the same network with y replaced by a weighted sum neuron, $y = \ell(x)$. If, for every input vector V_1 , it holds that $\text{argmax}(V_n) = \text{argmax}(V'_n)$, i.e. if both networks classify each input vector in the same way (regardless of the actual output neuron values computed), then we say that neuron y is *result-preserving redundant*. See Fig. 6 for an example.

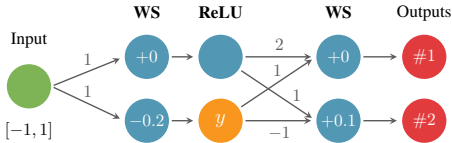


Fig. 6: The **orange** ReLU, marked y , is result-preserving redundant and can be replaced with a constant zero. Observe that any input in range $(0.1, 1]$ is classified as label $\#1$, while any input in range $[-1, 0.1)$ is classified as label $\#2$. The ReLU in **orange** is active only for inputs in $(0.2, 1]$, and it only increases the confidence in label $\#1$. For example, the network output for input 0.5 is $[1.3, 0.3]^T$, and after replacing y with 0 the output becomes $[1.0, 0.6]^T$. Label $\#1$ still wins, but with a lower confidence. Thus, y is result-preserving redundant — replacing it with a constant zero does not change the winning class, for the entire input domain.

Note that result-preserving redundancy is, in a way, more permissive than the previous categories: we do not directly try to bound the imprecision introduced, but rather only try to maintain the same output *label* for every input. Clearly, any neuron that is phase-redundant or k -forward-redundant is also result-preserving; and it is reasonable to assume that relaxed-redundant neurons with a small error would also be result-preserving redundant. The motivation for considering this kind of redundancy is that, due to its more permissive nature, it can identify additional redundant neurons.

Our definition of result-preserving redundancy can also be slightly relaxed, to exclude inputs whose classification was

borderline; i.e., inputs whose highest-scored label and the second-highest label received very similar scores. Intuitively, with this alteration, a neuron is considered result-preserving redundant if it does not change the classification of any inputs which were previously classified with a high degree of confidence, but may flip the classification of inputs about which the DNN was not sure to begin with. The motivation for this change is to allow the removal of additional neurons.

V. NEURON REMOVAL STRATEGIES

In Section III we laid the theoretical foundations of our DNN simplification approach, by defining four kinds of redundant neurons that could be removed to reduce network size. There exist many strategies for applying these definitions in practice, in order to reduce network sizes. Intuitively, a good strategy is one that identifies large sets of neurons that can be removed simultaneously, in a way that is computationally efficient. In this Section, we propose one such strategy, which we have empirically observed to perform well.

Step 1: Bound Estimation using MILP. Let v be an activation function neuron which we are considering for removal. In this context, it is useful to deduce lower and upper bounds for v that are as tight as possible. Such bounds could lead, for example, to the classification of v as phase-redundant, or enable us to compute $l_m(v)$ and declare v to be relaxed-redundant.

Mixed-Integer Linear Programming (MILP) [9] is a well-studied method for solving a system of linear constraints with real and integer variables. In the context of DNN verification, MILP can be used to derive lower and upper bounds on the values that the various neurons in the DNN can obtain [10], [44]. This is done by encoding a linear over-approximation of the neural network into the MILP solver, and then using the solver’s objective function to maximize/minimize each of the individual neurons. For example, after encoding a network N , we could set the solver’s objective function to $1 \cdot v$, where v is some neuron in N ; and the optimal solution discovered would then constitute v ’s upper bound.

As a first step in the simplification process, we propose to run such MILP queries for every neuron that is candidate for removal. The number of resulting queries can be large — two queries per neuron, one for each bound — but the gains are significant, as the discovered bounds can often be quite tight [44]. At the end of this step, we immediately remove all phase-redundant neurons.

In practice, it is useful to run the MILP solver with a short timeout (e.g., 10 second) for each neuron. In case a timeout occurs, modern solvers are able to provide a sound approximation of the optimal solution [38]. In our experiments, we observed that this initial step already detects a large number of phase-redundant neurons.

Step 2: Simulations. After the MILP phase is concluded, we are left with multiple activation-function neurons whose phases are not yet fixed. It is possible that some of these neurons are also phase-redundant, but that the bounds discovered

in the MILP pass were too loose to indicate this. It is also possible that they are k -forward-redundant or result-preserving redundant. At this point we wish to quickly *rule out* as many of these candidates as possible, before applying computationally expensive steps to dispatch the remaining candidates.

To do this, we follow in the footsteps of Gokulanathan et al. [15], and apply *simulations*; i.e., we evaluate the network on a large number of random inputs, and for each input record the values assigned to the network’s neurons. Simulations can easily show that a neuron is not phase-redundant, by demonstrating two different inputs for which the neuron is in two different linear phases. Similarly, they can show that a neuron is not k -forward-redundant or result-preserving redundant.

Step 3: Formal Verification. After the MILP and simulation phases, we are left with activation-function neurons that are candidates for removal, if we can prove them redundant. We now apply formal verification to classify these remaining neurons. Specifically, for each candidate neuron v , we: (i) apply verification to check whether v is fixed to one if its linear phases, and is hence phase-redundant; and if not, (ii) if N is a classification network, apply verification to check whether v is result-preserving redundant; else, if N is a regression network, apply verification to check whether v is k -forward-redundant, for a value of k that corresponds to the output layer. Each of these conditions can be posed as a DNN verification query, as described next. As soon as a neuron is marked redundant, it is removed, and the process continues.

In order to determine whether $v = f(x)$ is phase-redundant, we must check whether x is restricted to a certain linear segment. Let $[s_1, s_2], [s_2, s_3], \dots, [s_k, s_{k+1}]$ be the set of possible segments. For each such segment $[s_i, s_{i+1}]$, we can encode the DNN into the verifier, and pose the query: $\exists V_1. (x < s_i) \vee (x > s_{i+1})$. If the answer is UNSAT, we know that x is indeed fixed into segment $[s_i, s_{i+1}]$. An illustration appears in Fig. 7.

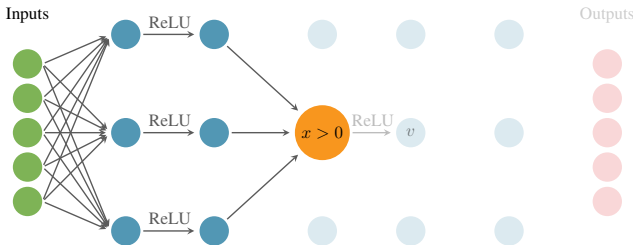


Fig. 7: A query for determining whether ReLU node $v = \text{ReLU}(x)$ is *phase-redundant*: we check whether it is possible that $x > 0$, and if not, we conclude that v is inactive-redundant. To facilitate the verification process, the neurons in subsequent layers, as well as all other neurons in layer 2 (grayed out), are not encoded.

Determining whether $v = f(x)$ is k -forward-redundant is done by creating a query where the part of the network starting from the neuron in question is duplicated. One copy of the network is the unmodified one, and in the other copy $v = f(x)$ is replaced with a weighted-sum neuron, $v' = \ell(x)$. We query the verifier whether it is possible that a neuron k layers away from v is assigned different values in the original and

modified copies. If the answer is UNSAT, the neuron is k -forward-redundant. See Fig. 8 for an illustration.

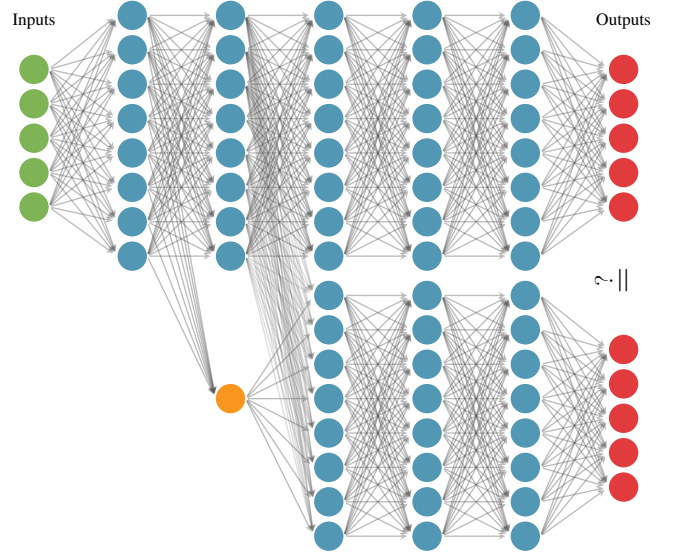


Fig. 8: **4-Forward-Redundancy** query illustration. The neuron in **orange** is the neuron being checked for forward-redundancy. In this case the layer being checked is at distance 4, which happens to be the output layer.

Determining whether $v = f(x)$ is result-preserving redundant is done by creating a query similar to the k -forward-redundant case, only this time we ask the verifier whether there exists an input that the two networks classify differently. If the answer is UNSAT, we know that the neuron is indeed result-preserving redundant.

Step 4: Relaxed Redundancy and Accumulative Error. The aforementioned steps were aimed at identifying and removing redundant neurons, without introducing any imprecision into the simplified network. Last but not least, we discuss the removal of relaxed-redundant neurons. Recall that relaxed-redundant neurons are determined by a user-specified error threshold e_t . Identifying these neurons is thus a local operation, that does not require verification; for every neuron we can compute the maximum error introduced by replacing it with l_m , and see whether it exceeds the threshold.

While each relaxed-redundant neuron can be identified locally, removing multiple neurons simultaneously runs the risk of compounding the overall error, beyond the permitted threshold. To circumvent this issue and allow the efficient removal of multiple relaxed-redundant neurons, we introduce the following lemma:

Lemma 1. *Let N be a neural network, and let N' be a simplified network, obtained from N by removing relaxed-redundant neurons u_1, \dots, u_n . Consider another neuron v in N' that is relaxed-redundant, and let e_{in} denote the error to v 's input, previously introduced by the removal of u_1, \dots, u_n . Let e_v denote the error introduced by the removal of v . Then, if we remove v , the overall error introduced to its output is*

upper bounded by:

$$e_{in} + e_v$$

This lemma tells us that the iterative removal of relaxed-redundant neurons does not compound the introduced error; instead, the error introduced by the removal of each neuron is only added to the error already introduced by the removal of other neurons. This enables us, through a straightforward computation, to upper bound the overall imprecision (on the output layer) that the removal of a set of relaxed-redundant neurons might cause. Consequently, our proposed strategy is to begin removing relaxed-redundant neurons with small error rates, each time recomputing the overall network inaccuracy, until hitting the prescribed overall error threshold. A full, formal description of these claims appears in Appendix 2 in the extended version of this paper [34].

VI. INTRODUCING REDUNDANCIES VIA INPUT SLICING

So far, our simplification efforts have hinged on the existence of redundant neurons. Next, we introduce a technique that can cause neurons to become redundant, even if they are initially not so.

The core idea is to: (i) *slice the input domain* \mathcal{D} of the DNN N into smaller sub-domains $\mathcal{D}_1, \dots, \mathcal{D}_n$; (ii) duplicate the original network n times, resulting in networks N_1, \dots, N_n , such that network N_i is associated with domain \mathcal{D}_i ; and (iii) apply the simplification process described in Section V for each N_i , separately. Intuitively, splitting the input domain into sub-domains can serve to separate “simpler” inputs regions, in which many neurons are phase-redundant, from more “complex” input domains where neurons fluctuate between phases. Various heuristics can be used for splitting the input domain, depending on the network in question. A simple splitting method, which we used in our evaluation, is to split the range of each input coordinate into n even sub-ranges.

After the slicing and simplification is done, we are left with a family of DNNs N_1, \dots, N_n , which are together equivalent to the original N . Evaluation is then performed in two steps: given an input vector V_1 , we first identify the domain \mathcal{D}_i to which V_1 belongs; and then compute $N_i(V_1)$ and return the result. As our evaluation shows, the resulting N_i networks can be quite small, resulting in a significant improvement to the expected number of operations required for evaluating the network. This improvement might come at the expense of increased space requirements for storing the resulting family of networks, making this approach suitable for cases where space is abundant but fast inference is crucial. We note that, as a side effect, the resulting networks may be easier to verify [46], [48].

Discussion: Dependency on Input Dimensions. Our proposed slicing method relies on splitting the input domain, by restricting input neurons to various values. This approach works quite well on DNNs with relatively few input neurons (e.g., the ACAS Xu family of networks [25]; see Section VII for details). For networks with a larger number of input neurons (e.g., image recognition networks), the number of input sub-domains might be prohibitively large. Indeed, a similar

phenomenon has been observed for verification techniques that rely on input slicing [46], [48].

One approach for mitigating this difficulty is through performing slicing not on the input layer, but on some smaller intermediate layer L_k in the network. Then, the network would be evaluated by evaluating the original network’s layers $L_1 \dots L_{k-1}$, and then using the values computed for layer L_k in choosing from a set of networks for continuing the evaluation. We speculate that for an intermediate layer of a moderate size, this approach could lead to improved performance over input slicing. We leave this for future work.

Extreme Slicing: Complete Linearization. We observe that input slicing can be used to completely linearize every sub-domain of the input space; that is, if the resulting sub-domains are sufficiently small, then in each network N_i all activation functions will become phase-redundant, effectively collapsing the DNN into a linear transformation. Additionally, even if the slicing does not fix the phase of all activation function neurons, extreme slicing tends to decrease the error introduced by removing relaxed-redundant neurons; and thus, complete linearization could be achieved by removing these neurons, even if they have not become phase-redundant. This linearization approach can thus be regarded as providing us with a simple, piecewise-linear approximation of the network as a whole — with an upper bound on the error in each sub-domain. Our experimental results in Section VII demonstrate very low error rates on most sub-domains.

Complete linearization incorporates a trade-off: in order to obtain very small, nearly-linear networks, the input domain would have to be sliced many times. Users can fine-tune the number of slices used, and consequently the sizes of the resulting DNNs, to their specific needs.

VII. EVALUATION

We created a proof-of-concept implementation of our approach as a Python framework, available online [33] (together with all benchmarks reported in this section). The framework provides all the functionality discussed so far: after importing a network, it can run MILP queries to compute neuron bounds; perform simulations; and identify phase-redundant, k -forward-redundant and result-preserving redundant neurons, by running verification queries. The framework uses the Gurobi [38] MILP solver and the Marabou [29] DNN verification engine as backends, although other backends could also be used.

For evaluation purposes, we conducted extensive experiments on the ACAS Xu system: an airborne collision avoidance system, implemented as a family of 45 neural networks [25]. Each of these neural networks has 5 input neurons, 5 output neurons, and 6 hidden layers with 50 neurons each and ReLU activation functions (310 neurons in total). Keeping the network sizes small was a key consideration in developing the ACAS Xu system [25], making it a prime candidate on which to apply simplification techniques.

We began by comparing our approach to that of Gokulanathan et al. [15], which is the current state-of-the-art in

verification-based simplification of DNNs. Their technique can be regarded as a private-case of ours, in which only specific phase-redundant neurons (specifically, inactive-redundant ReLUs) are removed. We compared that approach to our framework, configured to identify and remove both active-redundant and inactive-redundant ReLUs, and also to remove relaxed-redundant neurons. We ran both tools on all 45 ACAS Xu networks; the results appear in Table I.

TABLE I: Phase-Redundancy and Relaxed-Redundancy on ACAS Xu networks.

	Inactive Redundant	Active Redundant	Relaxed-Redundant		
			$\epsilon = 10^{-4}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-2}$
% of all neurons	4%	4.2%	4.2%	4.6%	4.9%
% of redundant neurons	baseline	3.5%	5.3%	13.6%	21.5%
output error bound	0	0	0.02	2.64	525.1

The table depicts the accumulated numbers of redundant neurons, when read from left to right (which is the order in which the techniques were applied). First, inactive-redundant neurons are removed (this is the technique of [15]), accounting for 4% of all neurons in the network. Active-redundant neurons are next, removing another 0.2% of all neurons, which is a 3.5% increase in the number of removed neurons. Finally, relaxed-redundant neurons are removed, with three possible alternative ϵ values. The most permissive one, $\epsilon = 10^{-2}$, leads to the removal of 4.9% of the neurons in total, which is a 21.5% increase over the baseline — but the resulting network error bound in this case, 525.1, is quite high. $\epsilon = 10^{-3}$ appears a better choice, with a total removal rate of 4.6% and a significantly smaller error bound of 2.64. We note that our evaluation indicates that the output error bounds currently computed are far from tight; devising tighter bounding schemes is a work in progress.

In our second experiment, we evaluated our complete simplification pipeline. First, we applied input-slicing, dividing the input domain into 32,768 equal sub-domains (3 rounds of bisecting the range of each of the 5 input neurons in 2). Next, for each sub-domain we: (i) ran MILP and removed any discovered phase-redundant neurons; (ii) ran simulations, and then formal verification to discover and remove any remaining phase-redundant neurons; and (iii) identified all result-preserving neurons, and greedily attempted to simultaneously remove large sets thereof, using verification. We note that identifying the largest set possible of result-preserving neurons that can be removed simultaneously is a difficult problem, and our current heuristic was a simple, greedy approach. Devising more sophisticated heuristics is left for future work.

We ran the MILP step on all 32,768 sub-domains, which resulted in the discovery of 67.3% phase-redundant neurons on average in each sub-domain. We continued to run the pipeline on a sample of 50 sub-domains selected at random. Most notably, we observed an average removal of 82.5%

redundant neurons (out of all neurons in the network), with 7.2% additional neurons still candidates for removal, but for which the underlying verification engine timed-out. Of the 82.5% removed neurons, 70.2% were phase-redundant, which is a very significant increase from the 4.2% neurons removed when the pipeline was run over the entire input domain. This demonstrates the high effectiveness of input slicing. In addition, about 21% of phase-redundant neurons were active-redundant, which signifies the importance of the generalization from “dead neurons” [15] to phase-redundancy. The remaining 12.3% neurons removed were result-preserving redundant. Fig. 9 shows the breakdown.

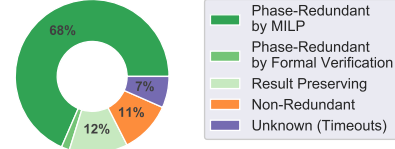


Fig. 9: Redundant neuron removal, averaged over 10 ACAS Xu input sub-domains.

Slicing is highly beneficial for neuron removal, but results in a large number of sub-domains that need to be checked. Within our pipeline, verification steps are the most expensive, whereas MILP queries and simulations are relatively cheap. We observe, however, that MILP queries already account for most of the removed neurons. Specifically, 68.5% of all phase-redundant neurons removed were discovered through MILP (about 83% of all redundant neurons), with a 10 second timeout for each individual MILP query.

The next step, namely simulations, is also computationally cheap and highly effective. For each sub-domain, we ran 100,000 simulations; and out of the of 31.5% neurons which were still candidates for removal after the MILP phase, an average of 26.4% of the neurons were ruled not phase-redundant through simulations. This left only a small number of candidates to be dispatched through verification (5.1% of the neurons), which in turn discovered the remaining 1.7% redundant neurons, on average. In our experiment, each Marabou verification query was run with a 4-hour timeout.

As discussed above, we used a fairly naïve strategy for discovering result-preserving redundant neurons. Specifically, we ran formal verification on each candidate neuron to check whether it was individually result-preserving redundant; this resulted in a set of candidates for removal. Then, we ran result-preserving simulations, iteratively removing additional candidate neurons from the network, as long as the simulations could not find a counter-example to the redundancy of the currently removed set. Finally, we ran a single verification query to verify that removing our selected neurons was indeed a result-preserving operation. On 75% of the sub-domains checked, this strategy worked. In sub-domains where we were successful, we found an additional 24.6% forward-redundant and result-preserving redundant neurons; whereas in sub-

domains where we were not successful, we had a similar amount of candidates for removal on average.

In the final step of our experiment, we tested our hypothesis that slicing can lead to the complete linearization of some of the sub-domains. Indeed, for some of the sub-domains explored, the simplification pipeline was able to remove *all* neurons, resulting in a DNN that is effectively a linear transformation. We noticed, however, a high variability — for example, in another sub-domain we were only able to remove 58% of the neurons. See Fig. 10 for additional details. We conclude that there is an inherent difference between the sub-domains: apparently, some of them compute simpler transformations than others.

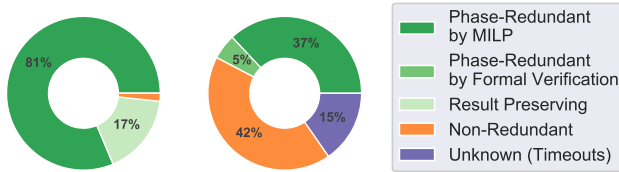


Fig. 10: An “almost” linear sub-domain (left) vs. a complex sub-domain (right).

VIII. RELATED WORK

The pruning of DNNs in order to reduce their sizes has received significant attention from the machine learning community in recent years. The most common approaches are based on heuristically identifying neurons and edges that seem to contribute little to the network’s output, removing these neurons and edges, and performing additional training of the network [19], [23]. Other approaches apply quantization: by using fewer bits to store the network’s weights or activation functions, the DNN’s footprint is decreased [21], [22], [39]. A common trait of these approaches is that, while they achieve a significant reduction in memory, they provide no guarantees about the resemblance of the smaller network to the original.

The most closely related work to our own is that of Gokulanathan et al. [15]. There, the authors use formal verification to remove dead neurons from a network, ensuring that the resulting network is equivalent to the original. Additionally, simulations are used to reduce the number of verification queries that need to be dispatched. Our work uses similar principles, but significantly extends them: we consider additional kinds of redundancy (phase-redundancy, k -forward-redundant, and result-preserving redundancy) that produce equivalent networks, and also relaxed-redundancy which removes additional neurons by introducing a bounded amount of imprecision.

Our work uses the Marabou DNN verification engine as a backend [1], [7], [13], [18], [27], [29], [30], [42]; but any of the many approaches and tools that have been proposed in recent years could be used as well. These approaches leverage SMT solvers (e.g., [20]), based on LP and MILP solvers (e.g., [6], [11], [37], [44]), the propagation of symbolic intervals and abstract interpretation (e.g., [14], [45]–[47]), abstraction-refinement techniques (e.g., [3], [12]), and many

others. Recent work has extended beyond answering yes/no questions about DNNs, targeting tasks such as automated DNN repair [16], [31] and quantitative verification [4]. Verification approaches have also been proposed for recurrent networks [24], [49], which could potentially also be simplified. As DNN verification technology improves, the scalability of our approach will also increase.

IX. CONCLUSION AND FUTURE WORK

Neural networks often suffer from a high degree of redundancy, which affects evaluation time, memory footprint and verification costs. In this paper we presented a novel technique to identify and remove such redundancy. Our framework is customizable, allowing users to safely trade network precision for size reduction, while maintaining the introduced imprecision within a prescribed bound.

In the future, we plan to extend our work along multiple axes. Specifically, we plan to research more intelligent techniques for input domain slicing than coordinate-splitting; and also compositional techniques that would allow us to split the network into several sub-networks, identify redundancies in each of them, and then re-combine the pruned network into a single network that is smaller than the original. In addition, we plan to explore ways of combining our pruning techniques with techniques from the related field of Boolean circuit simplification [8].

Acknowledgements. We thank Ittai Rubinstein and Haoze Wu for their contributions to this project. The project was partially supported by the Israel Science Foundation (grant number 683/18) and the Binational Science Foundation (grant number 2017662).

REFERENCES

- [1] 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)*, 2021.
- [2] 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.
- [3] P. Ashok, V. Hashemi, J. Kretinsky, and S. Mühlberger. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2020.
- [4] 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.
- [5] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. <http://arxiv.org/abs/1604.07316>.
- [6] 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.
- [7] N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. <https://arxiv.org/abs/1709.10207>.
- [8] S.-C. Chang, M. Marek-Sadowska, and K.-T. Cheng. Perturb and Simplify: Multilevel Boolean Network Optimizer. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1494–1504, 1996.

- [9] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
- [10] 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.
- [11] 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.
- [12] Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 43–65, 2020.
- [13] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. Annual Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2021.
- [14] 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.
- [15] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.
- [16] 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.
- [17] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
- [18] D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Assessing Robustness of Neural Networks. In *Proc. 16th. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
- [19] S. Han, H. Mao, and W. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, 2015. Technical Report. <http://arxiv.org/abs/1510.00149>.
- [20] 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.
- [21] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, pages 4107–4115, 2016.
- [22] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research*, 18(1):6869–6898, 2017.
- [23] F. Iandola, S. Han, M. Moskewicz, K. Ashraf, W. Dally, and K. Keutzer. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and < 0.5MB Model Size, 2016. Technical Report. <http://arxiv.org/abs/1602.07360>.
- [24] 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.
- [25] K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
- [26] 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.
- [27] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, pages 19–26, 2017.
- [28] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021. To appear.
- [29] 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.
- [30] Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 83–89, 2019.
- [31] 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.
- [32] L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, 2018. Technical Report. <https://arxiv.org/abs/1801.05950>.
- [33] O. Lahav and G. Katz. Code: Pruning and Slicing Neural Networks using Formal Verification, 2021. <https://github.com/vbcr1f/redo>.
- [34] O. Lahav and G. Katz. Pruning and slicing neural networks using formal verification, 2021.
- [35] L. Liebenwein, C. Baykal, B. Carter, D. Gifford, and D. Rus. Lost in Pruning: The Effects of Pruning Neural Networks beyond Test Accuracy, 2021. Technical Report. <https://arxiv.org/abs/2103.03014>.
- [36] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. Algorithms for Verifying Deep Neural Networks, 2019. Technical Report. <http://arxiv.org/abs/1903.06758>.
- [37] 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>.
- [38] G. Optimization. The Gurobi MILP Solver, 2021. <https://www.gurobi.com/>.
- [39] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: Imagenet Classification using Binary Convolutional Neural Networks. In *Proc. 14th European Conf. on Computer Vision (ECCV)*, pages 525–542, 2016.
- [40] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
- [41] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014. Technical Report. <http://arxiv.org/abs/1409.1556>.
- [42] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU networks, 2020. Technical Report. <http://arxiv.org/abs/2010.03258>.
- [43] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. <http://arxiv.org/abs/1312.6199>.
- [44] 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>.
- [45] H. Tran, S. Bak, 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.
- [46] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, 2018.
- [47] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. Dhillon, and L. Daniel. Towards Fast Computation of Certified Robustness for ReLU Networks, 2018. Technical Report. <http://arxiv.org/abs/1804.09699>.
- [48] 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.
- [49] 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 Conf. of European Conference on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.

Towards Scalable Verification of Deep Reinforcement Learning

Guy Amir, Michael Schapira and Guy Katz
The Hebrew University of Jerusalem, Jerusalem, Israel
{guyam, schapiram, guykatz}@cs.huji.ac.il

Abstract—Deep neural networks (DNNs) have gained significant popularity in recent years, becoming the state of the art in a variety of domains. In particular, deep reinforcement learning (DRL) has recently been employed to train DNNs that realize control policies for various types of real-world systems. In this work, we present the *whiRL 2.0* tool, which implements a new approach for verifying complex properties of interest for DRL systems. To demonstrate the benefits of *whiRL 2.0*, we apply it to case studies from the communication networks domain that have recently been used to motivate formal verification of DRL systems, and which exhibit characteristics that are conducive for scalable verification. We propose techniques for performing k-induction and semi-automated invariant inference on such systems, and leverage these techniques for proving safety and liveness properties that were previously impossible to verify due to the scalability barriers of prior approaches. Furthermore, we show how our proposed techniques provide insights into the inner workings and the generalizability of DRL systems. *whiRL 2.0* is publicly available online.

I. INTRODUCTION

In recent years, *deep neural networks* (DNNs) [23] have become highly popular due to their ability to produce state-of-the-art results in multiple fields, e.g., image recognition [34], text classification [37], game playing [45], and many others [7]. DNNs used in such contexts have been shown to successfully learn, by training on data, a model that *generalizes* to previously unseen inputs. In particular, *deep reinforcement learning* (DRL) [40] has been recently used to train DNNs to learn control policies for complex computer and networked systems, surpassing the state-of-the-art in a variety of application domains, including database management [60], compiler optimization [41], congestion control [27], [39] on the Internet, routing [53], compute-resource scheduling [9], [42], adaptive video streaming [38], [43], and many more.

Despite the overwhelming success of DNNs, many safety issues pertaining to them have been identified [22], [51], demonstrating that although DNN models potentially yield excellent performance, they also suffer from many weaknesses. For instance, it has been shown that DNNs can be manipulated into performing severe errors through only slight distortions to their inputs [17]. This phenomenon, called *adversarial perturbations*, plagues effectively all modern DNNs.

Adversarial perturbations, alongside other safety and security vulnerabilities, have brought about a surge of interest in formally verifying the correctness of DNNs. A plethora of approaches for DNN verification have been proposed in recent years (e.g., [19], [25], [30], [55]). Unfortunately, in general,

all proposed tools face significant scalability barriers, which render them unable to verify state-of-the-art, industrial DNNs with millions of parameters. Furthermore, even when applied to small DNNs, these tools are often restricted to verifying simplistic properties. The scalability challenge is further aggravated in the DRL context, which involves *sequential* DNN-informed decision making, and so reasoning about repeated invocations of the DNN, where the outcome of one invocation can influence the input to the DNN in subsequent invocations. Consequently, the applicability of recently introduced DNN verification tools to complex properties and systems of practical interest remains extremely limited.

To begin bridging this gap, we previously introduced a tool called *whiRL 1.0* [16], which enables verifying certain safety and liveness properties, or identifying violations, for practical DRL systems. We demonstrated *whiRL 1.0*'s usefulness by verifying properties of interest for three systems from the *communication networking* domain. We identified such systems to be prime candidates for verification for two main reasons: first, state-of-the-art DNNs in this domain tend to be of moderate sizes, which are within reach of existing verification technology; and second, meaningful and complex specifications can be formulated and verified because the inputs for these systems are carefully handcrafted and reflect important semantic meaning (as opposed to raw pixel data in computer vision applications, for example). *whiRL 1.0*, which combines DNN verification techniques with bounded model checking, uses a black-box DNN verification engine as a backend, and can thus benefit from any future improvements to DNN verification technology. As exemplified by our promising initial results in [16], *whiRL 1.0* constituted a first step towards enhancing the reliability of DRL systems.

Still, *whiRL 1.0* had severe limitations: most notably, although it successfully generated violations of desired properties, it was incapable of proving that properties of practical significance held without making very strong assumptions, e.g., that runs of the considered system terminate within a very small number of steps. However, the executions of real-world systems are often infinite, or finite but consisting of many steps. In such scenarios, *whiRL 1.0* and other DRL verification tools are unable to prove that most relevant properties hold.

In this work, we present *whiRL 2.0* [1] — a verification engine for DRL systems. *whiRL 2.0* significantly extends the capabilities of the original *whiRL 1.0* tool to accommodate verifying complex properties. In particular, while *whiRL 1.0*

was limited to verifying basic safety properties, *whiRL 2.0* utilizes *k-induction* techniques for proving both safety and liveness properties of DRL systems. In addition, *whiRL 2.0* uses *invariant inference* techniques to quickly prove properties that could otherwise be quite difficult to verify. *whiRL 2.0* also incorporates *abstraction* methods for providing some visibility into the DRL system’s operation. We demonstrate the effectiveness of these techniques by revisiting the three case studies involving state-of-the-art DRL systems to which *whiRL 1.0* has been applied in [16]: the *Aurora* [27] Internet congestion controller, the *Pensieve* [43] adaptive video streamer, and the *DeepRM* [42] compute resource scheduler. We are able to prove various properties of these systems that, to the best of our knowledge, were beyond the reach of prior state-of-the-art tools, including the original *whiRL 1.0* tool.

The rest of this paper is organized as follows. Section II covers basic background on DNNs, DRL systems, and DNN verification. Next, in Section III we present our *whiRL 2.0* verification tool, and describe its novelties and main components. We present *whiRL 2.0*’s semi-automated invariant inference in Section IV, and discuss the tool’s implementation in Section V. Our case studies are described in Section VI, followed by related work in Section VII. We conclude in Section VIII.

II. BACKGROUND

A. Deep Neural Networks and Deep Reinforcement Learning

A deep neural network (DNN) [23] is a directed graph, where the nodes (also called neurons) are organized in layers. In feed-forward DNNs, data flows from the first (*input*) layer, onto a sequence of intermediate (*hidden*) layers, and finally into a final (*output*) layer. The network is evaluated by assigning values to the input layer’s neurons, and then iteratively computing the assignment of each of the hidden layers, until reaching the output layer and returning its evaluation to the user.

More specifically, the value of each neuron in the hidden and output layers is computed using the values of neurons in the preceding layer. Each such layer has a *type*, which determines the exact way in which its neuron values are computed. One common layer type is the *weighted sum* layer, in which each neuron is computed as an affine combination of the values of neurons in the preceding layer, based on edge weights and bias values determined as part of the DNN’s training process. Another popular layer type is the *rectified linear unit* (*ReLU*) layer, where each node y is connected to a single node x from the preceding layer, and its value is computed by $y = \text{ReLU}(x) = \max(0, x)$. In this paper we will focus on weighted sum and ReLU layers, although there exist many additional layer types, such as *max-pooling* and *hyperbolic tangent*, to which our technique may be extended.

Fig. 1 depicts a toy DNN comprising an input layer with two neurons, followed by a weighted sum layer and a ReLU layer. For input $V_1 = [1, 3]^T$, the second layer’s computed values are $V_2 = [18, -3]^T$. In the third layer, the ReLU functions are applied, resulting in $V_3 = [18, 0]^T$. Finally, the network’s single output is $V_4 = [54]$.

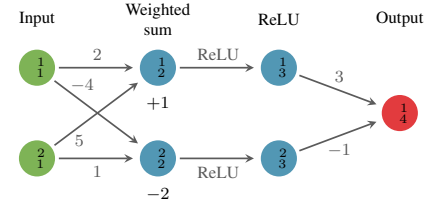


Fig. 1: A toy DNN. The values above the edges are weights, and the values below the vertices are biases.

Formally, a DNN N that receives k inputs and returns n outputs is a mapping $\mathbb{R}^k \rightarrow \mathbb{R}^n$. The DNN consists of a sequence of m layers L_1, \dots, L_m , where L_1 is the input layer and L_m is the output layer. We use s_i to denote layer L_i ’s size, and $v_i^1, \dots, v_i^{s_i}$ to denote L_i ’s individual neurons. We refer to the column vector $[v_i^1, \dots, v_i^{s_i}]^T$ as V_i . During evaluation, the input values V_1 are fed to the network’s input layer, and V_2, \dots, V_n are computed iteratively.

Each weighted sum layer L_i has a weight matrix W_i of dimensions $s_i \times s_{i-1}$ and a bias vector B_i of size s_i . These W_i and B_i are set at training time, and determine how V_i is computed: $V_i = W_i \cdot V_{i-1} + B_i$. For a ReLU layer L_i , the values of V_i are computed by applying the ReLU to each individual neuron in its preceding layer: $v_i^j = \text{ReLU}(v_{i-1}^j)$.

In *deep reinforcement learning* (DRL) [40], a DNN, called the *agent*, learns a *policy* π , which maps each possible observed *environment state* s to an *action* a . During training, at each discrete time-step $t \in 0, 1, 2, \dots$, a *reward* r_t is displayed to the agent, based on the action a_t it chose to perform after observing the environment’s state at that time s_t . This reward is used for tuning the agent DNN’s weights. The DNNs produced using DRL fall within the same general architecture described above; the difference lies in the training process, which is aimed at generating a DNN that computes a mapping π that maximizes the *expected cumulative discounted return* $R_t = \mathbb{E}[\sum_t \gamma^t \cdot r_t]$. The *discount factor*, $\gamma \in [0, 1)$, controls the effect that past decisions have on the total expected reward.

B. Verification of Deep Neural Networks

A DNN verification query typically includes a DNN N , a pre-condition P on N ’s input, and a post-condition Q on N ’s output [28]. The verification algorithm’s goal is to find a concrete input x_0 such that $P(x_0) \wedge Q(N(x_0))$ (the SAT case), or prove that no such x_0 exists (the UNSAT case). Typically, we use the pre-condition P to express some states of the environment that the network might encounter, and use the post-condition Q to encode the *negation* of the behavior we would like N to exhibit in these states. Thus, when the verification algorithm returns UNSAT, this implies that the desired property always holds. Conversely, a SAT result indicates that the desired property does not always hold, and this is demonstrated by the discovered counter-example x_0 .

For example, observe the toy DNN in Fig. 1, and suppose we wish to verify that the DNN’s output is strictly larger than 5, for any input, i.e., for any $x = \langle v_1^1, v_1^2 \rangle$, it holds that $N(x) =$

$v_4^1 > 5$. This is encoded as a verification query by choosing a pre-condition which does not restrict the input, i.e., $P = (\text{true})$, and by setting $Q = (v_4^1 \leq 5)$, which is the *negation* of our desired property. For this verification query, a sound verifier will return SAT, and a feasible counter-example such as $x = \langle 0, -1 \rangle$, which produces $v_4^1 = 0 \leq 5$. Hence, the property does not hold for this DNN.

Verifying DRL Systems. Beyond the general challenges of verifying DNNs (most notably, scalability), verifying DRL systems involves additional challenges. These challenges stem from the fact that DRL agents typically run within reactive systems, and are invoked multiple times, with the inputs to each invocation usually affected by the outputs of previous invocations. This means that (i) the specifications for DRL systems need to account for multiple invocations; and (ii) the scalability issue is aggravated, because the verifier needs to consider multiple consecutive invocations of the network, which is akin to considering a significantly larger DNN.

While attempts have been made to develop tools tailored for DRL system verification (e.g., [16], [32], [44]), two important challenges have yet to be addressed. First, existing verification approaches for DRL systems have focused on refuting properties, and not on proving that they hold; and second, existing approaches were not geared towards verifying reactive systems. As part of the *whiRL* project, we make an initial attempt at addressing these two challenges.

III. *whiRL 2.0*

Our contribution in this paper is the *whiRL 2.0* verification tool, which significantly extends our existing DRL verification engine, *whiRL 1.0*. The *whiRL 2.0* tool allows to verify complex queries on DRL systems, which were previously beyond our reach. Specifically, it supports the verification of safety and liveness properties of DRL systems using a *k-induction*-based approach. Additionally, it incorporates *invariant inference* techniques, which facilitate the verification of complex safety properties. *whiRL 2.0* uses an underlying verification engine as a black-box, and is hence compatible with many existing DNN verifiers.

Formalizing DRL Agents. DRL agents typically operate within reactive systems: they process a (possibly infinite) sequence of states, each representing a current snapshot of the environment observed by the agent. Each state is obtained from its predecessor by triggering the action outputted by the DRL agent, and allowing the environment to react.

In line with the formulation proposed in [16], we formalize the DRL verification problem by encoding the DRL system, as well as its environment, into a transition system $\mathcal{T} = \langle S, I, T \rangle$. Each state $s \in S$ in this transition system is a snapshot of the current observable environment; these states correspond to the inputs of the DNN agent. We use $I \subseteq S$ to denote the set of initial states. The transition relation, $T \subseteq S \times S$, is defined such that $\langle x_i, x_j \rangle \in T$ iff the system can transition from state x_i to state x_j ; i.e., when the DNN is presented with state x_i , it selects some action, to which the environment can respond

in a way that leads the system to state x_j . Although the DNN is deterministic, the environment is not necessarily so, and so T need not be deterministic. An *execution* of the system is defined as a sequence of states x_1, \dots, x_n , such that $x_1 \in I$, and for all $1 \leq i \leq n-1$ it holds that $T(x_i, x_{i+1})$. The process of encoding a DRL system as a transition system is supported by *whiRL 1.0*, via constructs for representing features common to DRL systems (e.g., inputs in the form of a “sliding window” over the recent history of observations) [16].

Example. As a running example, we focus on the *Aurora* DRL system [27], which implements a congestion control policy. In today’s Internet, different services (e.g., video streaming like Netflix and Amazon, VoIP services such as Skype) contend over the same network bandwidth, with aggregate demand for bandwidth often exceeding the available supply. If Internet traffic sources do not pace the rates at which their data is injected into the network, the network will become congested, resulting in data being lost or delayed, and, consequently, in bad user experience and even global Internet outages. Congestion control is the task of determining, for each individual Internet traffic source, how quickly its traffic should be injected into the network at any given point in time. Congestion control is thus a both fundamental and timely networking challenge.

Recently, researchers have proposed employing DRL for this purpose, and presented the *Aurora* congestion controller [27]. An *Aurora*-controlled traffic source uses a DNN to select the next rate at which to send traffic, based on observations regarding the implications of its past choices of sending rates. Specifically, *Aurora*’s inputs are t vectors v_{-t}, \dots, v_{-1} , containing performance-related statistics pertaining to the sender’s most recent t rate-change decisions. These incorporate information about what fraction of sent data packets were lost following each rate selection, how long it took the sent packets to reach the traffic’s destination, etc. The DNN’s output determines whether the current rate should be increased, kept steady, or decreased. Changing the sending rate can potentially affect the environment, e.g., an increase to the rate might lead to packet loss if the new rate exceeds network capacity. These changes to the environment, in turn, affect the future inputs to the DNN. See [27] for additional details.

In the formulation of *Aurora* as a verification challenge in [16], each state, which corresponds to a possible input to *Aurora*’s DNN, is represented by a t -tuple of statistics vectors. The state also contains the DNN’s (deterministic) output for the input it represents. This is required for defining good and bad states, as will be discussed later. Congestion controllers are expected to converge to “good” rate decisions from any starting point. Hence, we let the set of initial states be the set of all states. Recall that the input to the DNN represents a sliding window over t -long histories of statistics vectors. Thus, for each two consecutive states, $s_1 \xrightarrow{T} s_2$, it holds that s_2 is obtained from s_1 by augmenting the vectors in s_1 with a statistics vector associated with the DNN’s rate change at state s_1 , and discarding the vector in s_1 corresponding to the least recent of the t prior rate changes.

DRL System Specifications. Once the DRL system is formulated as a transition system, we can specify safety and liveness properties [11] that it should uphold. *Safety properties* indicate that the system never displays unwanted behavior, and these are often formulated through a predicate $P_B(s)$ that returns true iff $s \in S$ is a bad state, i.e., a state in which the property is violated. The safety verification problem then boils down to determining whether there is a reachable bad state in \mathcal{T} [4]. *Liveness properties* indicate that the system eventually displays desirable behavior, and these are often formulated through a predicate $P_G(s)$ that returns true iff $s \in S$ is a good state, i.e., a state in which the property is fulfilled. Verifying a liveness property is performed by checking that there are no infinite sequences of consecutive states in which only finitely many of the states are good [4]. For instance, a natural safety property with respect to Aurora is that when Aurora observes excellent network conditions (no packet loss, close-to-minimum packet delays), as reflected by the statistics vectors fed to the DNN, the DRL agent does not advise to decrease the sending rate in the *next time-step*. An example of a liveness property in this setting is that if excellent network conditions persist, Aurora should always *eventually* increase the sending rate.

K-Induction. Proving that safety or liveness properties hold (or finding counter-examples) involves traversing large transition system graphs. For modern DRL systems, this is often infeasible, in particular because the rich environments in which these systems operate can react in many ways after each action taken by the agent, resulting in high (or even infinite) out degrees for many states. In *whiRL 1.0*, this issue was addressed through the application of *bounded model checking* (BMC), an approach that explores only a small fraction of the transition system graph, namely, states within a k -step distance from an initial state. BMC can find safety and liveness violations (if they are reachable within k steps) as depicted in Fig. 2, but cannot prove the absence of such violations.

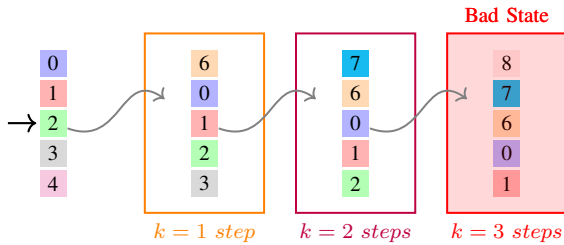


Fig. 2: BMC searches for violations of a safety property. Each vector represents a state, and encodes the statistics that Aurora observed in the past $t = 5$ time-steps. The unwanted state is surrounded by a red rectangle, and is reachable only after $k = 3$ steps from the initial state. Note that consecutive states have shared inputs shifted, and each time-step sample is depicted in a different color.

In *whiRL 2.0*, we address this important gap by adding the means for proving that safety and liveness properties hold. To this end, we employ the method of *k-induction* [11].

Intuitively, the idea in *k-induction* is to look for state sequences of length k , which can start from arbitrary states

in \mathcal{T} (not necessarily from initial states), and for which the property is violated. If a violating execution exists, it must contain an indicative k -long sequence of steps — a suffix of the execution that ends in the bad state for safety properties, or a sequence of non-good states for liveness properties. Thus, if a verifier finds that a k -induction query is UNSAT, we know that the corresponding property holds. If, however, it returns SAT with a counter-example that does not start at an initial state, we cannot conclude whether the property holds, and must increase k in search of a conclusive answer. Fig. 3 depicts a snapshot of the *k*-induction process used for proving a safety property.

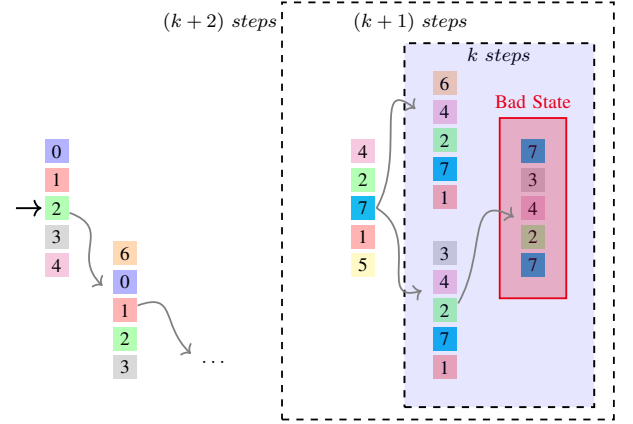


Fig. 3: Using *k*-induction to prove a safety property, i.e., that the system never reaches the bad state (surrounded by a red rectangle). Although there are k -long and $(k+1)$ -long execution sequences that end in the bad state, there is no such sequence of length $(k+2)$; and due to this and to BMC on the base cases, the property holds.

More formally, following the terminology in [4], verifying ω -regular liveness properties is reducible to checking persistence properties of the form “*eventually forever B*”, where B represents a “bad” state ($\exists s \text{ s.t. } B = \neg P_G(s)$). Using *k*-induction in the spirit of [6], [54], we can rule out the existence of k -long sequences of bad states for a given k (even ones not starting at an initial state). This is performed by formulating the following query:

$$\exists x_1, x_2, \dots, x_k. \left(\bigwedge_{i=1}^{k-1} T(x_i, x_{i+1}) \right) \wedge \left(\bigwedge_{i=1}^k \neg P_G(x_i) \right)$$

for increasingly large values of k . As soon as one such query returns UNSAT, we are guaranteed that the liveness property holds. A similar encoding can be used for proving safety properties.

We note that realizing *k*-induction in our case-studies entailed contending with challenges such as the need to encode verification queries that capture the system-environment interaction from *any* (possibly non-initial) state. An additional challenge was scalability; duplicating the network to encode k steps can induce an exponential blowup in running time. *whiRL 2.0* curtails the search space by using bound tightening mechanisms, and by enforcing certain dependencies between the inputs to the k duplicate networks encoded as part of a k -

induction query. Specifically, these k inputs typically represent the k recent observations of the agent’s environment, and can be restricted by requiring them to constitute a “sliding window”: each pair of consecutive inputs must agree on the $k - 1$ previous observations that appear in both inputs.

BMC and k -induction are related techniques; the former is geared towards refuting a property, and the latter is geared towards proving it. In *whiRL 2.0*, we take a portfolio approach, as depicted in Fig. 4: we alternate between BMC and k -induction queries, until we: (i) refute the property (BMC returns SAT); or (ii) prove the property (k -induction returns UNSAT); or (iii) hit a timeout threshold. When steps 1 and 2 both fail, we increment k by 1 and repeat the process. Thus, although we do not know in advance whether the property in question holds, we hope that one of the two techniques will either find a counter-example or prove the property.

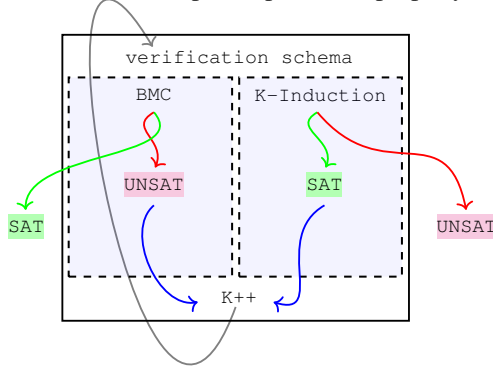


Fig. 4: *whiRL 2.0*’s verification schema.

Abstraction. In computer networking systems, such as the Aurora congestion controller, the system’s state is often a set of observations about the environment. Through close inspection of our considered case-studies, we observe that occasionally some of the input fields are irrelevant to the property being checked, in the sense that the property can be proved even when disregarding them. We thus integrate into *whiRL 2.0* abstraction capabilities [10] — the ability to strip off irrelevant input fields, as indicated by the user, when dispatching a verification query. The original transition system \mathcal{T} is thus changed into an abstract transition system, \mathcal{T}' , which over-approximates the original one. Specifically, the states of \mathcal{T}' are symbolic, each corresponding to multiple states of \mathcal{T} ; and $s'_1 \xrightarrow{\mathcal{T}'} s'_2$ if and only if some states s_1 and s_2 , to which s'_1 and s'_2 correspond, satisfy $s_1 \xrightarrow{\mathcal{T}} s_2$. If the verification engine concludes that the property holds for \mathcal{T}' (i.e., the negation of the property is UNSAT), it follows that it also holds for the original \mathcal{T} . However, a counter-example for \mathcal{T}' may be spurious, as it may not be valid for \mathcal{T} , in which case the original query may need to be solved to obtain a definite result.

For example, in Aurora, the DNN input represents performance-related statistics pertaining to the t most recent rate adjustments made by the sender. In Aurora’s implementation used for our evaluation, we chose $t = 10$ (as in [27]). In this context, abstraction might expose, for instance, that a

certain property holds regardless of what values are assigned to the fields not relating to the 5 most recent rate changes, indicating that the policy is, in essence, dependent only on the 5 most recently observed statistics vectors.

We leverage the fact that inputs to recently-proposed computer networked systems consist of fairly few fields with natural semantic meaning, thus leading to a limited number of actual combinations of input fields that are abstracted.

In Section VI we demonstrate how *whiRL 2.0*’s abstraction capabilities can shed light on the inner workings of the verified system, rendering the “black-box” policy learned by the DRL system somewhat more translucent.

IV. INVARIANT INFERENCE

Verifying DRL systems is difficult, as one must often reason about transitions across many states to establish that a property holds. BMC and k -induction can mitigate this issue to some extent, but sometimes this is not enough. To further boost the scalability of *whiRL 2.0*, we enhanced it with semi-automated invariant inference capabilities.

In the context of safety verification of a transition system graph, an *invariant* can be regarded as a partition of the state space S into two disjoint sets, S_1 and S_2 , such that no transition leads from one set to the other: $s_1 \in S_1 \wedge s_2 \in S_2 \Rightarrow \langle s_1, s_2 \rangle \notin T$. Invariants are useful if we know that $I \subseteq S_1$ (all initial states are in S_1) and $P_B(s) \Rightarrow s \in S_2$ (all bad states are in S_2). In this case, the existence of the invariant immediately guarantees that no bad states are reachable. Unfortunately, discovering such useful invariants is known to be undecidable in general, and very difficult to accomplish in practice [46].

As part of *whiRL 2.0*, we propose a heuristic for semi-automated invariant inference, which leverages common traits of communication networking systems. More precisely, we observe that many relevant properties in these systems can be regarded as *Boolean monotonic functions*; they tend to be satisfiable when the DNN’s input vectors are allowed to fluctuate extensively, but quickly become unsatisfiable when these input vectors are restricted. Often, finding the tipping point, i.e., the minimal input restrictions that cause the property to shift from SAT to UNSAT, constitutes an invariant that is useful for proving other properties, and which can also render the policy learned by the DNN more translucent to humans.

We demonstrate these notions on the Aurora congestion controller. Recall that Aurora’s output indicates whether the sending rate should be increased, maintained, or decreased. *whiRL 2.0* can search for an invariant that translates to the range of inputs for which the DNN outputs that the sending rate should be decreased. Such an invariant can assist in the verification of complex properties, and provide human engineers with comprehensible insights into the DRL system.

Technically, *whiRL 2.0* allows the user to specify the output property and mark the relevant input fields. For example, in Aurora’s case, “the sending rate should be decreased” as the output property, and a subset of the input statistics as the relevant fields. Then begins a binary search on the range of the inputs in order to find the minimal restrictions that render

the verification query UNSAT. At each step of the binary search, we invoke a black-box verification procedure to solve the resulting query. This allows us to locate the tipping point up to a prescribed precision. *whiRL 2.0* has built-in *templates* for input and output restrictions, which can be regarded as different strategies for conducting the aforementioned binary search. Each template takes into account either the DRL system’s input variables or output variables, and controls them by adjusting their bounds; tightening them to “push” the query towards the UNSAT region. Currently, these templates include (i) for a fixed output, tightening or loosening the bounds of the specified input variables, executing binary search until the point in which the query switches from SAT to UNSAT is discovered; and (ii) performing a similar operation, but this time on the bounds of the specified output variables, while fixing the inputs according to user-specified constants.

Fig. 5 illustrates an invariant search procedure. In this procedure, we have a candidate invariant (the middle blue line) that splits the search space into two parts. Ideally, the reachable states should all be on one side of the partition, and the bad states on the other side. Our binary search automatically adjusts the invariant candidate. In case an initial invariant candidate is too strong (there are reachable states on both sides), it is weakened, and the line is moved towards *B*. If, however, the initial invariant candidate is too weak (there are bad states on both sides), it is strengthened, and the line is moved towards *I*. Both kinds of adjustments are performed by tightening or loosening the bounds on the input or output variables.

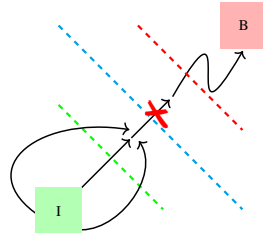


Fig. 5: Invariant search procedure. The initial states are the green square labeled *I*, and the bad states are the red square labeled *B*.

V. IMPLEMENTATION

We implemented *whiRL 2.0* as a Python framework that provides general functionality for verifying DRL systems. *whiRL 2.0* uses Marabou [31], a state-of-the-art SMT-based [5], [12], [14] DNN verifier, as a backend (although other verifiers could also be used). *whiRL 2.0* includes the following key modules, which did not exist in *whiRL 1.0*:

- 1) **K-Induction Query Verifier.** A module that allows the user to generate *k*-induction queries. The module can encode either a safety property or a liveness property, specified by their $P_B(s)$ and $P_G(s)$ predicates, respectively.
- 2) **Invariant Finder.** A module through which a user can instruct *whiRL 2.0* to search for an invariant. The user needs to provide the post-condition Q , and mark the variables to focus on. *whiRL 2.0* then performs the previously described semi-automated search procedure, and returns within the specified parameters a range for which the invariant holds, if such a range is found.
- 3) **Input Abstraction.** A module that allows the user to specify, for a given verification query, which input fields

TABLE I: *whiRL 2.0* features used in each case study.

	Aurora	Pensieve	DeepRM
<i>K-Induction</i>	✓	✓	✗
<i>Bounded Model Checking</i>	✓	✓	✓
<i>Invariant</i>	✓	✗	✓
<i>Abstraction</i>	✗	✓	✓

should be abstracted. When abstraction is applied, *whiRL 2.0* will either return UNSAT (if the abstract query returns UNSAT), or default to the original query if the abstract query returns a spurious counter-example.

Additionally, *whiRL 2.0* retains some of *whiRL 1.0*’s functionality, most notably its DNN loading interfaces and bounded model checking capabilities. The code for *whiRL 2.0*, alongside documentation and the experiments described in the paper, are all available online under a permissive license [1]. An appendix with the formulation of the verified properties is also available online [2].

VI. CASE STUDIES

We evaluate *whiRL 2.0* on three case studies of DRL systems: the *Aurora* [27] congestion controller, the *Pensieve* [43] adaptive video streamer, and the *DeepRM* [42] compute resource scheduler. All three case studies, which were used to illustrate the power of *whiRL 1.0* in [16], are from the domain of communication networks. We have identified such DRL systems as highly suitable candidates for evaluating DRL system verification techniques as they achieve state-of-the-art results despite being of moderate sizes, rendering verification tractable. Table I summarizes the *whiRL 2.0* capabilities applied in each case study. All experiments were conducted on an HP EliteDesk machine with six Intel *i5* – 8500 cores running at 3.00 GHz, and with a 32 GB memory.

A. The Aurora Congestion Controller

Aurora [27] is a state-of-the-art DRL system that acts as a congestion controller for data transmission [27]. *Aurora* receives an input vector of size $3t$, which consists of observations from the previous t time-steps. Specifically, the input consists of 3 distinct values representing performance-related statistics for each of the previous t rate changes outputted by the DNN: (i) *latency gradient*: the derivative of latency (packet delays) across time, as measured by the sender, following a change to the rate; (ii) *latency ratio*: the ratio of the average latency experienced by the sender, following a change to the rate, to the minimum past latency experienced. This value is never smaller than 1; and (iii) *sending ratio*: the ratio of the rate at which packets are injected into the network by the sender (i.e., the sending rate), to the rate at which the sent packets arrive at the receiver. We note that the latter rate can be strictly lower than the former rate if the network is congested, which can lead to sent packets being forced to wait in in-network buffers, or being dropped along the way. The sending ratio is never smaller than 1. Intuitively, simultaneous low latency gradient, latency ratio, and sending ratio are indicative

of excellent network conditions. Aurora has a single output value, which indicates whether the sending rate should be increased (positive output), decreased (negative output), or maintained (output is zero). When network conditions are good (low latency, no packet loss), this is indicative of the current rate not overshooting the network bandwidth. Hence, we expect the sending rate to increase so as to take over available bandwidth. In contrast, when network conditions are poor (high latency, high packet loss), this is indicative of network congestion, and so we expect Aurora to decrease the rate. See [16], [27] for additional details.

In line with previous work [16], [27], we set $t = 10$, i.e., the input size to Aurora’s DNN is of size $3t = 30$. Aurora’s DNN has a single hidden ReLU layer with 48 neurons, and a single neuron in its output layer.

Proving Liveness. In our previous work [16], two liveness properties of Aurora were formulated, but could not be verified using *whiRL 1.0*. Using *whiRL 2.0*, we successfully proved that both properties from [16] always hold. Details follow.

- **Property 1: excellent network conditions eventually imply rate increase.** When Aurora observes a history of excellent network conditions (low latency, no packet loss), the DRL system should *eventually increase* the sending rate, i.e., eventually output positive values. Using *whiRL 2.0*’s k -induction capabilities, we successfully proved that this property, as formulated in [16], indeed holds for any infinite run. The property was successfully proved, within a few seconds, for $k = 2$.
- **Property 2: poor network conditions eventually imply rate decrease.** Symmetrically to property 1, when Aurora observes a history of poor network conditions, the DRL system should *eventually decrease* the sending rate by outputting negative values. By performing k -induction with $k = 5$, we proved that this property, as formulated in [16], indeed holds for all infinite executions. This query took approximately 4.5 hours to solve.

Semi-Automatic Invariance Inference. Next, we used *whiRL 2.0*’s invariant inference capabilities to find invariants for proving safety properties of Aurora.

- **Invariant A: bounding the next-step decrease in sending rate for excellent network conditions.** When Aurora observes a history of excellent network conditions (low latency, no packet loss), the DRL agent’s output should be non-negative, i.e., should not imply a decrease to the sending rate. This safety property was shown to be violated in previous work [16]. Here, we utilize *whiRL 2.0*’s invariance inference techniques to prove a bound on this (undesirable) next-step decrease in sending rate, to provide visibility into the performance of the DRL system. *whiRL 2.0*’s method for producing the desired invariant appears in Alg. 1. The algorithm takes two user inputs: the *latency slack* ϵ , and the *precision* η . The ϵ input captures the notion of “excellent network conditions” encoded as inputs to the DNN: the observed latency gradient is restricted to

the range $[-\epsilon, \epsilon]$; and the observed latency ratio is restricted to the range $[1, 1 + \epsilon]$. Additionally, the sending ratio is set to 1 (indicating that sent traffic arrives at the receiver without being delayed or dropped within the network). The algorithm now performs a binary search over the DNN’s output space (leaving the prescribed input ranges for the DNN fixed). Specifically, the η input specifies the desired precision: the output of the algorithm will be an upper bound b on the DNN’s output, such that the output b is impossible, but $b + \eta$ is possible, given the aforementioned input restrictions. Recall that the upper bound b relates to the *negation* of the desired property, and so an upper bound of b implies that Aurora’s DNN will never decrease the sending rate by b or more when network conditions are excellent. This procedure terminates within a few seconds, returning an upper bound on the input for which the DNN verifier returns UNSAT. The algorithm’s correctness immediately follows from the underlying verifier’s soundness.

Algorithm 1 Finding Invariant A

Input: ϵ, η // latency slack, precision

Output: UB_{UNSAT} // worst-case output decrease bound

```

1:  $UB_{UNSAT} \leftarrow -\infty$  //  $-M$ , for some large constant  $M$ 
2:  $UB_{SAT} \leftarrow 0$ 
3:  $QUERY \leftarrow \text{DNN\_VERIFY}(\epsilon, \text{output} \leq 0)$ 
4: while ( $|UB_{SAT} - UB_{UNSAT}| \geq \eta$ ) do
5:    $OUT_{UPPER} \leftarrow \frac{1}{2} (UB_{UNSAT} + UB_{SAT})$ 
6:    $QUERY \leftarrow \text{DNN\_VERIFY}(\epsilon, \text{output} \leq OUT_{UPPER})$ 
7:   if  $QUERY$  is SAT then  $UB_{SAT} \leftarrow OUT_{UPPER}$ 
8:   if  $QUERY$  is UNSAT then  $UB_{UNSAT} \leftarrow OUT_{UPPER}$ 
9: return  $UB_{UNSAT}$ 

```

- **Invariant B: inferring when Aurora fails to decrease the next-step sending rate even though network conditions are poor.** We now wish to characterize poor network conditions in which Aurora does not decrease its sending rate, as expected of it. The procedure is described in Alg. 2. Now, the sending ratio is not fixed to 1, but is rather within the range $[1, P]$, for a user-specified P value. P represents a user-provided upper bound on ratio of the rate at which packets leave the sender (i.e., the sending rate) to the rate which these packets arrive at the receiver. For a slack ϵ , the procedure again restricts the latency gradient to the range $[-\epsilon, \epsilon]$ and the latency ratio to the range $[1, 1 + \epsilon]$. Intuitively, setting low values for ϵ while allowing sending ratios to be high corresponds to sending traffic across communication networks in which in-network buffers are very shallow. In such networks, packets cannot accumulate within the network, resulting in low latencies for packet delivery. However, since in-network buffers are shallow, packets are dropped once network bandwidth is even slightly exceeded, resulting in high sending ratios when the sending rate significantly overshoots the network’s capacity (and many packets are lost). The algorithm fixes the output’s lower bound to be non-negative, and executes a binary search on the input sending

ratio. Specifically, the algorithm returns, for any user-chosen value P , a lower bound (LB_{UNSAT}) such that Aurora always decreases the sending rate when its observations regarding past sending ratios all lie within the range $[LB_{UNSAT}, P]$. *whiRL 2.0* finds the invariant within a few seconds.

Algorithm 2 *Finding Invariant B*

Input: $P \geq 2$ // upper bound on the sending ratio

Output: LB_{UNSAT} // worst-case sending ratio bound

```

1:  $LB_{SAT}, SR_{LOWER} \leftarrow 1$ 
2:  $LB_{UNSAT}, SR_{UPPER} \leftarrow P$ 
3:  $QUERY \leftarrow \text{DNN VERIFY } (\epsilon, \text{output} \geq 0, SR_{LOWER}, SR_{UPPER})$ 
4: while ( $LB_{SAT} + 1 < LB_{UNSAT}$ ) do
5:    $SR_{LOWER} \leftarrow \frac{1}{2} (LB_{SAT} + LB_{UNSAT})$ 
6:    $QUERY \leftarrow \text{DNN VERIFY } (\epsilon, \text{output} \geq 0, SR_{LOWER}, SR_{UPPER})$ 
7:   if  $QUERY$  is SAT then  $LB_{SAT} \leftarrow SR_{LOWER}$ 
8:   if  $QUERY$  is UNSAT then  $LB_{UNSAT} \leftarrow SR_{LOWER}$ 
9: return  $LB_{UNSAT}$ 

```

Observing the bounds produced by Alg. 2 yielded surprising insights regarding the decision-making policy learned by Aurora. Specifically, to gain insight into what our discovered invariants reveal regarding the policies, we created multiple instances of Aurora agents, and trained them all on the same training data until achieving an averaged reward value similar to that of the original Aurora controller [27]. We then observed that for some of the Aurora instances, the discovered invariants depended only on the *proportion* between the sending ratio’s lower bound (SR_{LOWER}) and upper bound (SR_{UPPER}), as opposed to their *absolute* values. Specifically, for violating counter-examples (inputs to Aurora’s DNN) produced for these instances, the ratio between the highest and lowest past sending ratios was at least 2, with lower ratios giving rise to desirable behavior by Aurora. For other trained instances of Aurora, violating counter-examples only depended on the absolute values of the bounds; e.g., Aurora always decreases the rate for inputs to the DNN where all sending ratios lie in the range $[1, M]$ for some value M , but not when these lie in the range $[1, M + \delta]$ for some small δ . Our findings show that policies that yield the same expected reward on the training set might *generalize* very differently to inputs that lie outside this training set, and that our discovered invariants can shed light on the generalization strategies of different policies learned.

B. The Pensieve Video Streamer

Pensieve is a DRL system [43] for *adaptive bitrate* (ABR) selection. To provide high quality of experience for video clients, *Pensieve* continuously collects statistics about the client’s experience when downloading video chunks (e.g., was the video rebuffered? how long did it take to download the chunk?) to dynamically adapt the resolution at which the next video chunk is downloaded from the video server. Each video chunk represents a fixed-duration video segment (e.g., 4-second-long chunks in our experiments) encoded in one

of several possible resolutions (SD, HD, etc.), with higher resolutions corresponding to larger chunks, in terms of number of bits. When client-sensed network conditions are good, we expect the ABR algorithm to decide that the next video chunk will be downloaded in high resolution (HD); and when they are poor, we expect a low resolution (SD) to be selected, to avoid having the client not finish the download in time, which leads to video rebuffering. The input to *Pensieve*’s DNN consists of $(2t + M + 3)$ fields, where $t > 0$ represents the number of recent video chunk downloads considered, and $M > 0$ represents the number of available video resolutions. The input comprises: (i) the *bitrate* (1 field) in which the last video chunk was downloaded; (ii) the current *video buffer size* (1 field) of the client, reflecting the number of seconds of unwatched video stored at the client; (iii) network *throughput measurements* for video chunks downloaded in the past t time-steps (t fields); (iv) *download times* for the video chunks downloaded in the past t time-steps (t fields); (v) *resolution options* (M fields) to download the next chunk; and (vi) the number of *remaining chunks* to be downloaded (1 field). See [43] for a thorough exposition of *Pensieve*, and [16] for a formalism of the *Pensieve* verification challenge.

To maintain consistency with *Pensieve*’s original hyper-parameters, in our experiments $t = 8$ and $M = 6$. Due to the nature of an ABR algorithm, all executions are finite (downloads finish in finite time), and so all relevant properties are safety properties. In previous work [16], *whiRL 1.0* was applied to check two safety properties of *Pensieve*:

- **Property 1.** When the chunk download history represents *excellent conditions* (short download times, large client buffer size), the DRL system should *increase* the resolution at which chunks are requested before the download finishes.
- **Property 2.** When the download history represents *poor network conditions* (long download times, small client buffer size), the DRL system should *decrease* the resolution at which chunks are requested before the download finishes.

While Property 1 was shown not to hold [16], no counter-examples could previously be found for Property 2, and so it could neither be proved nor disproved using existing tools.

Using *whiRL 2.0*, we were able to prove that Property 2 indeed holds under certain, realistic, assumptions.¹ To achieve this, we applied k-induction, with $k = 1$. The result returned by the verifier indicated that the bad states are unreachable, and, hence, that the undesirable behavior cannot occur. These verification queries took approximately 20 minutes to solve.

C. The DeepRM Resource Manager

DeepRM [42] is a DRL-based resource manager, responsible for allocating various cluster compute resources (e.g., CPU, memory) to queued jobs, in order to optimize the cluster’s throughput. *DeepRM* receives the following as input: (i) the *current resource usage* in the system; (ii) a *queue* with up to

¹We assumed that chunks represent 4-second-long video segments. Considered chunk download times are between 4 to 15 seconds per chunk, which implies that downloading each chunk takes longer than consuming it.

Q pending jobs waiting to be scheduled; and (iii) a *backlog*, indicating the number of jobs waiting to be scheduled that are not yet in the queue. For a fixed Q -sized job queue, the DeepRM controller may output one of $(Q+1)$ possible actions: a *wait* action (i.e., no resources will be allocated at this time-step), or a *schedule_q* action for $1 \leq q \leq Q$, indicating that job q should be scheduled next. DeepRM’s output is interpreted as a probability distribution, assigning a certain probability to each of the $(Q+1)$ possible actions. We refer the reader to [42] for a thorough exposition of DeepRM, and to [16] for a formalism of the DeepRM verification challenge.

In our case study, as in [16], we used a DeepRM system trained with $R = 2$ resources: *CPU* and *memory units*, and a job queue of size $Q = 5$. Overall system resources consist of 10 CPUs and 10 memory units. We considered two kinds of jobs: *small* jobs, which require 1 CPU and 1 memory unit for a single time-step, and *large* jobs, which require 10 CPUs and 10 memory units, for $t = 20$ time-steps.

Previous work [16] considered the following safety properties for DeepRM:

- **Property 1.** When all resources are fully available, and the queue is filled with *small* jobs, DeepRM should never assign the highest probability to the *wait* action.
- **Property 2.** When no resources are available, and the queue is filled with *small* jobs, DeepRM should assign the highest probability to the *wait* action.
- **Property 3.** When no resources are available, and the queue is filled with *large* jobs, DeepRM should assign the highest probability to the *wait* action.

Using *whiRL 1.0*, it was shown [16] that Property 1 holds, and that there exist counter-examples for Properties 2 and 3. However, by using *whiRL 2.0* we were able to prove (within a few seconds) a stronger property that, in fact, generalizes properties 1, 2 and 3. By applying *whiRL 2.0*’s abstraction capabilities to both the inputs indicating resource utilization and the output indicating the recommended action, we proved that for *any* resource utilization level, when the queue is filled with identical jobs, the DRL system’s output assigns a higher probability to *schedule₂* than to *wait*. This immediately proves Property 1, and implies that Properties 2 and 3 cannot hold.

This finding sheds new light on previous results, and enhances our understanding of DeepRM: (i) the three original properties do not depend on the current resource utilization. Rather, due to the DRL system learning a suboptimal policy, it is biased towards scheduling a specific job (job #2), and may fail to select *wait* when appropriate; and (ii) the counter-examples found for Properties 2 and 3 are not outliers, but rather the general case. Indeed, we were able to use *whiRL 2.0* to prove that the inverses of both these properties always hold. These results demonstrate that, beyond proving or disproving specific properties, *whiRL 2.0* can shed light on the policy learned by the DRL system, and expose problematic issues.

VII. RELATED WORK

Due to the increasing use of DNNs, many DNN verification tools have been proposed in recent years; some are SMT-

based (e.g., [28], [31], [35], [47]), whereas others use different verification strategies, such as *abstract interpretation* [48], [56], [59], *mixed integer linear programming* (MILP) [52], and many others. Recently, these approaches were extended to verify systems with multi-step executions, such as Recurrent Neural Networks (RNNs) [26], [58] or hybrid systems [50].

In our evaluation of *whiRL 2.0*, we used *Marabou* [31], [57] as a black-box DNN verifier. To date, Marabou has mostly been applied for solving adversarial robustness queries [3], [8], [24], [29], and our work demonstrates that it is also applicable in the field of computer and networked systems. Marabou affords additional features, such as built-in abstraction [15], simplification [20], [36], repair [21] and optimization [49] techniques, which could also be applied to our case studies.

In addition to general DNN verification engines, methods have been devised to formally verify safety properties of DRL systems, which are the subject matter of this work. Such approaches include *shield synthesis* [33], and combining the verification process with *verified runtime monitoring* [18]. Other methods focus on finding adversarial attacks that pertain specifically to DRL agents, e.g., by using MILP [13].

In addition to the *whiRL* project, other approaches have been proposed for verifying DRL systems in the domain of communication networks. These include, e.g., *Verily* [32] and *Metis* [44]. Importantly, however, our focus is on verifying (as opposed to only refuting) various safety and liveness properties of these systems. To the best of our knowledge, this lies beyond the grasp of other existing tools.

VIII. CONCLUSION

DRL systems provide excellent performance in multiple settings, but suffer from severe vulnerabilities. Several verification tools have been developed to mitigate this concern, but these mostly refute, as opposed to prove, safety and liveness properties of interest. In this work, we presented *whiRL 2.0* — a novel verification engine that supports proving both safety and liveness properties of DRL systems. *whiRL 2.0* accomplishes this through semi-automatic invariance inference, alongside techniques such as k-induction and query abstraction. We demonstrated our tool’s capabilities through three case studies from the communication networks domain. In addition, we demonstrated how *whiRL 2.0* can provide insights into the inner workings of these systems, uncovering weaknesses that would otherwise remain unnoticed.

In the future, we plan to enhance our tool’s scalability by using improved search heuristics. Also, we intend to enrich the semi-automatic invariant inference templates to support searching for more complex invariants.

Acknowledgements. We thank Nathan Jay, Tomer Eliyahu and the anonymous reviewers for their contributions to this project. The project was partially supported by the Israel Science Foundation (grant number 683/18), the Binational Science Foundation (grant numbers 2017662 and 2019798), and the Center for Interdisciplinary Data Science Research at The Hebrew University of Jerusalem.

REFERENCES

- [1] G. Amir, M. Schapira, and G. Katz. Artifact Repository, 2021. <https://doi.org/10.5281/zenodo.4769612>.
- [2] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning, 2021. Technical Report. <https://arxiv.org/abs/2105.11931>.
- [3] 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.
- [4] C. Baier and J. Katoen. *Principles of Model Checking*. MIT press, 2008.
- [5] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.
- [6] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.
- [7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. <http://arxiv.org/abs/1604.07316>.
- [8] N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. <https://arxiv.org/abs/1709.10207>.
- [9] W. Chen, Y. Xu, and X. Wu. Deep reinforcement learning for multi-resource multi-machine job scheduling. *arXiv preprint arXiv:1711.07440*, 2017.
- [10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2000.
- [11] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*, volume 10. Springer, 2018.
- [12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [13] A. Dethise, M. Canini, and N. Narodytska. Analyzing learning-based networked systems with formal verification. *IEEE International Conference on Computer Communications (IEEE InfoCom)*, 2021.
- [14] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, 2(2):1–2, 2006.
- [15] Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 43–65, 2020.
- [16] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Annual Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2021.
- [17] H. F. Eniser, M. Christakis, and V. Wüstholtz. Raid: Randomized adversarial-input detection for neural networks. *arXiv preprint arXiv:2002.02776*, 2020.
- [18] N. Fulton and A. Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.
- [19] 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.
- [20] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.
- [21] 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.
- [22] C. Gongye, H. Li, X. Zhang, M. Sabbagh, G. Yuan, X. Lin, T. Wahl, and Y. Fei. New passive and active attacks on deep neural networks in medical applications. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.
- [23] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.
- [24] D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Assessing Robustness of Neural Networks. In *Proc. 16th. Int. Symposium on on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
- [25] 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.
- [26] 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)*, 2020.
- [27] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019.
- [28] 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.
- [29] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, pages 19–26, 2017.
- [30] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021. To appear.
- [31] 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.
- [32] Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 83–89, 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] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Proc. 26th Conf. on Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.
- [35] L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, 2018. Technical Report. <https://arxiv.org/abs/1801.05950>.
- [36] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification, 2021. Technical Report. <https://arxiv.org/abs/2105.13649>.
- [37] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent Convolutional Neural Networks for Text Classification. In *Proc. 29th AAAI Conf. on Artificial Intelligence*, 2015.
- [38] A. Lekharu, K. Moulili, A. Sur, and A. Sarkar. Deep learning based prediction model for adaptive video streaming. In *2020 International Conference on COMMunication Systems & NETWORKS (COMSNETS)*, pages 152–159. IEEE, 2020.
- [39] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering*, 6(3):445–458, 2018.
- [40] Y. Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.
- [41] R. Mammadli, A. Jannesari, and F. Wolf. Static neural compiler optimization via deep reinforcement learning. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 1–11. IEEE, 2020.
- [42] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.
- [43] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.
- [44] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.

- [45] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [46] O. Padon, N. Immerman, S. Shoham, A. Karbyshev, and M. Sagiv. Decidability of Inferring Inductive Invariants. In *Proc. 43th Symposium on Principles of Programming Languages (POPL)*, pages 217–231, 2016.
- [47] L. Pulina and A. Tacchella. Challenging smt solvers to verify neural networks. *Ai Communications*, 25(2):117–135, 2012.
- [48] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev. Fast and effective robustness certification. *NeurIPS*, 1(4):6, 2018.
- [49] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU networks, 2020. Technical Report. <http://arxiv.org/abs/2010.03258>.
- [50] X. Sun, K. H., and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.
- [51] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. <http://arxiv.org/abs/1312.6199>.
- [52] 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>.
- [53] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar. Learning to route with deep rl. In *NIPS Deep Reinforcement Learning Symposium*, 2017.
- [54] T. Wahl. The k-induction principle. *Northeastern University, College of Computer and Information Science*, pages 1–2, 2013.
- [55] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, pages 1599–1614, 2018.
- [56] L. Weng, H. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. Boning, and I. Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, pages 5276–5285. PMLR, 2018.
- [57] 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.
- [58] 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 Conf. of European Conference on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.
- [59] H. Zhang, T. Weng, P. Chen, C. Hsieh, and L. Daniel. Efficient neural network robustness certification with general activation functions, 2018.
- [60] 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 *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432, 2019.

Exploiting Isomorphic Subgraphs in SAT

Alexander Ivrii
IBM Haifa Research Lab, Israel
alexi@il.ibm.com



Ofer Strichman
Information System Engineering,
IE, Technion, Haifa, Israel
offers@ie.technion.ac.il



Abstract—While static symmetry breaking has been explored in the SAT community for decades, only as of 2010 research has focused on exploiting the same discovered symmetry dynamically, during the run of the SAT solver, by learning extra clauses. The two methods are distinct and not compatible. The former may prune solutions, whereas the latter does not – it only prunes areas of the search that are guaranteed not to have solutions, like standard conflict clauses. Both approaches, however, require what we call *full symmetry*, namely a propositionally-consistent mapping σ between the literals, such that $\sigma(\varphi) \equiv \varphi$, where here \equiv means syntactic equivalence modulo clause ordering and literal ordering within the clauses. In this article we show that such full symmetry is not a necessary condition for adding extra clauses: isomorphism between possibly-overlapping subgraphs of the colored incidence graph is sufficient. While finding such subgraphs is a computationally hard problem, there are many cases in which they can be detected a priori by analyzing the high-level structure of the problem from which the CNF was derived. We demonstrate this principle with several well-known problems.

I. INTRODUCTION: SYMMETRY, ALMOST SYMMETRY, AND E-CLAUSES

Symmetry breaking [22] is a well known technique for accelerating SAT solving, which originated decades ago by Puget [21] for CSP, and later by Crawford et al. [8] for CNF. Symmetry-breaking for CNF was implemented efficiently in the tool SHATTER [4] and later improved in BREAKID [11]. In a nutshell, it means that new predicates, called *symmetry-breaking* predicates, are added to the input formula φ , without changing its satisfiability. These predicates prune the search space and are likely to remove solutions, but without changing the satisfiability of the formula. The construction of those predicates is based on finding a mapping σ between the literals of the input formula φ , such that $\sigma(\varphi) \equiv \varphi$. Here ‘ \equiv ’ means syntactic equivalence modulo clause ordering and literal ordering within the clauses. The mapping has to be *propositionally-consistent*, which means that $\forall v_1, v_2 \in \text{var}(\varphi). \sigma(v_1) = v_2 \Rightarrow \sigma(\bar{v}_1) = \bar{v}_2$ and $\sigma(v_1) = \bar{v}_2 \Rightarrow \sigma(\bar{v}_1) = v_2$. If we find such a mapping, then it means that every satisfying solution α to φ has the property that $\sigma(\alpha)$ also satisfies φ . We can then add a constraint that prunes one of those solutions. As an example, consider

$$\varphi = (1 \text{ -}3)(2 \text{ -}3)(1 \ 2 \ 3)(\text{-}1 \ \text{-}2)$$

and the mapping $\sigma : 1 \mapsto 2, 2 \mapsto 1$ (by convention, each such mapping implies that the mapping of the negated literals is

also included in σ , e.g., $-1 \mapsto -2 \in \sigma$). We see that

$$\sigma(\varphi) = (2 \text{ -}3)(1 \text{ -}3)(2 \ 1 \ 3)(\text{-}2 \ \text{-}1),$$

and that $\sigma(\varphi) \equiv \varphi$. Indeed if we take any solution α to φ , we see that $\sigma(\alpha)$ is a solution as well. For example, for $\alpha = (1, 2, 3) \mapsto (T, F, F)$ we have $\alpha \models \varphi$, and indeed $\sigma(\alpha) \models \varphi$ as well, since $\sigma(\alpha) = (1, 2, 3) \mapsto (F, T, F)$. Crawford et al. showed how to add symmetry-breaking constraints, which we will not detail here. In this case it may amount to adding the clause $(\text{-}1 \ 2)$, which indeed in this case excludes the first solution without excluding the second one. Such pruning of solutions is in many cases helpful for shortening the overall run-time [4], [17].

Symmetry-breaking tools discover such mappings by analyzing the colored literals incidence graph¹ G with respect to multiple potential mappings Σ : if for $\sigma \in \Sigma$ it holds that $\sigma(G) \equiv G$ (this is called ‘automorphism’), then σ defines a symmetry. The isomorphism in this case is restricted such that for every two nodes, $n_1, n_2 \in G$, if $\sigma(n_1) = n_2$ then n_1 and n_2 must have the same color, i.e., clause nodes are mapped to clause nodes and literal nodes to literal nodes.

Another way to exploit symmetry is by adding clauses during search. Henceforth we will call such clauses ‘e-clauses’, for ‘Extra’ clauses. This option has mostly been researched in the CSP community, under the names *Symmetry breaking during search - SBDS* [5], [14], [15], [7] and *Symmetry Breaking by Dominance Detection - SBDD* [13]. In the SAT community this route was first explored via the Symmetrical Learning Scheme (SLS) [6], which adds new clauses during the search based on learned clauses and a pre-computed set of symmetry ‘generators’. SLS was later improved by Symmetry Propagation (SP) [9], which only adds such extra clauses if they lead to further (immediate) propagations, and several years later by Symmetric Explanation Learning (SEL) [10], which is integrated within BCP (it takes the reason clause of the propagation as the base for adding e-clauses). According to [10], SEL is the only one of those that is competitive with modern static symmetry breaking. Finally, [25] has a similar scheme in which e-clauses are only added if the learned clause has a low LBD. In [10] those methods were jointly called *dynamic symmetry handling*, to emphasize that

¹Such a graph is constructed from a CNF by introducing a vertex for each literal and each clause, connecting opposite literals with an edge, and connecting the literals to the clauses that they are part of. The clauses’ nodes have one color, and the literals’ nodes have a different color.

unlike *static symmetry breaking* they are based on an analysis during the search (hence ‘dynamic’), and that they do *not break symmetry*, as they do not remove solutions. We find this name inadequate, however, because symmetry does not need to be ‘handled’. A more proper name is *dynamic symmetry exploitation*, which is the name we will use in the rest of this article. Although static symmetry breaking and dynamic symmetry exploitation are based on the same data – the symmetries in the formula – they are not compatible. One cannot use dynamic symmetry exploitation if the symmetries it relies on are broken by added predicates.

Dynamic symmetry exploitation was also studied for the case of *almost symmetric* formulas (also called ‘weak symmetry’) [19], [7], formalized as follows. Let

$$\varphi \equiv \varphi_1 \cup \varphi_2, \quad (1)$$

where here we equate formulas $\varphi, \varphi_1, \varphi_2$ with sets of clauses. Let σ be a literal map of φ such that

$$\sigma(\varphi_2) \equiv \varphi_2. \quad (2)$$

This reflects a common scenario, where a few clauses – marked here by φ_1 – disrupt the symmetries in the formula. The main method that was suggested in these references is to add e-clauses based on φ_2 . That is, once a clause c is learned from φ_2 alone, add $\sigma(c)$ as well.

In this article we observe that the requirement of symmetry as used by all of those prior works on dynamic symmetry exploitation is a sufficient, yet not a *necessary* condition for adding e-clauses. We will need the following definitions for explaining this claim.

Definition 1 (*The refined colored incidence graph*): The *refined* version of a colored incidence graph assigns separate colors to clauses of different arity.

We will denote this graph by G , assuming the underlying formula is clear from the context (it can also include learned clauses).

Definition 2 (*The subgraph induced by a resolution sequence*): Given a resolution sequence c_1, \dots, c_n , its corresponding *induced* subgraph in G is comprised of the subgraphs induced by these clauses, and the edges between opposite literals that were resolved in the sequence.

Now, consider such a resolution sequence c_1, \dots, c_n that was used for learning a clause c (c itself is not part of the sequence), and its corresponding induced subgraph g . Consider also another subgraph g' of G that is color-isomorphic to g . It is not hard to see that g' reflects another possible resolution sequence in the formula, ending with a different clause, which we can add as an e-clause. This criterion is *ad-hoc* and does not require automorphism of the original formula or some pre-defined part of it as in almost-symmetries. In fact, it can be seen as an application of the SR-II inference rule suggested by Krishnamurthy in [18] already in 1985 (there was no indication, however, how a solver may exploit that rule in [18]). In some types of formulas, finding e-clauses based on this reasoning is computationally cheap, and can lead to improvements in the overall run-time of the solver. The

important point is that this technique can be applied even when there is no mapping σ such that $\sigma(\varphi) \equiv \varphi$, which implies that this technique can derive e-clauses that cannot be derived by the above-mentioned symmetry exploitation techniques.

In fact, this idea was implicitly used in the past by the second author [24] for adding e-clauses in the case of bounded-model checking problems, and by Say et al. for adding such clauses in the case of optimizing a planning process with neural networks [23]. Both references reported performance gains. In this article we give a general view that encompasses also these two references, and show that the potential for such clauses is present in many other types of formulas.

Example 1: Let φ be comprised of the following clauses:

$$\begin{array}{cccc} (1 \ 2 \ 3) & (-1 \ -2 \ -3) & (2 \ 3 \ 4) & (-2 \ -3 \ -4) \\ (3 \ 4 \ 5) & (-3 \ -4 \ -5) & (4 \ 5 \ 6) & (-4 \ -5 \ -6) \\ (5 \ 6 \ 7) & (-5 \ -6 \ -7) & (1 \ 3 \ 5) & (-1 \ -3 \ -5) \\ (2 \ 4 \ 6) & (-2 \ -4 \ -6) & (3 \ 5 \ 7) & (-3 \ -5 \ -7) \\ (1 \ 4 \ 7) & (-1 \ -4 \ -7) & & \end{array} \quad (3)$$

It happens to be the Van der Waerden formula (3,3; 7). We will describe this type of formulas later, in section III-A.

Symmetry breaking, as emitted by BREAKID, discovers the two mappings below (these are also called ‘generators’). To get to the full set of possible mappings one needs to also consider their compositions.

$$\begin{aligned} \sigma_1 &: [1 \ 7] [2 \ 6] [3 \ 5] \\ \sigma_2 &: [1 \ -1] [2 \ -2] [3 \ -3] \dots [7 \ -7] \end{aligned} \quad (4)$$

This representation is called ‘cycle form’, and should be interpreted as follows: in each line, every literal appears at most once; it should be replaced with the literal that comes next in the brackets, and if it is the last one then with the first literal in the brackets. In this example σ_1 implies that simultaneously swapping literals 1 and 7, 2 and 6, 3 and 5 (and correspondingly, their negated versions, -1 and -7, etc.) results in the same formula. Readers familiar with Van der Waerden formulas may notice that this symmetry corresponds to a reversal of the indices, i.e., the first variable becomes last, the second one becomes second to last, etc, and that σ_2 corresponds to a swap of the colors. In such formulas, regardless of their length, these are the only two possible symmetries.

Now suppose that we learn a new conflict clause $c = (1 \ 2 \ -5 \ 6)$, via the following resolution sequence:

$$(1 \ 2 \ 3), (-3 \ -4 \ -5), (2 \ 4 \ 6). \quad (5)$$

We can therefore add two e-clauses corresponding to the two generators:

$$\sigma_1(1 \ 2 \ -5 \ 6) = (7 \ 6 \ -3 \ 2) \quad \sigma_2(1 \ 2 \ -5 \ 6) = (-1 \ -2 \ 5 \ -6). \quad (6)$$

However, more e-clauses can be derived based on this conflict clause. We need to find a subgraph of G that is color-isomorphic to the one representing the sequence (5). Going back to our example, it is indeed not hard to see that $(2 \ 3 \ 4)$, $(-4 \ -5 \ -6)$, $(3 \ 5 \ 7)$, all of which are clauses in φ , give

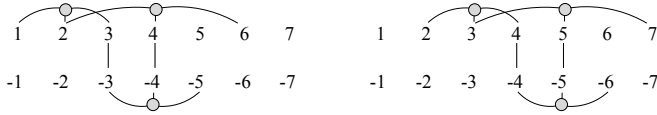


Fig. 1. Two isomorphic subgraphs of the same refined colored incidence graph corresponding to (3). The literals are nodes with a separate color than the clause nodes. All the clause nodes in this example are of the same arity, hence they have the same color.

us just that – see Fig. 1. Applying the same resolution steps yields a new e-clause (2 3 -6 7), which cannot be deduced by any composition of σ_1, σ_2 , simply because our inference is not based on the original CNF’s symmetry, rather it is inferred dynamically from the resolution process. ■

Since the subgraph isomorphism problem is NP-complete, we only focus on cases in which it can be indirectly inferred from analyzing the high-level structure of the original problem and controlling (or knowing) how it is encoded. Specifically, in such problems we derive a mapping between the literals, and adapt the solver to use this information in order to derive new e-clauses. Our implementation of this technique shows average overall improvement in terms of run-time.

To summarize, our contributions in this article are:

- 1) We show several problem domains in which this known principle can be exploited by the SAT solver for improving performance. So far it has only been used in bounded model checking and in neural network verification;
- 2) We show how this technique is superior to, and can be seen as an extension of, dynamic symmetry;
- 3) We show how to modify the SAT-solver in order to implement this technique, and suggest several techniques for filtering e-clauses (i.e., decide which ones to keep, in light of possibly having too many of them) and deletion of such clauses;
- 4) We present experimental results that show certain performance improvements (around 50% reduction in run time) due to this technique with domains in which it has not been used before.

Although any paper that mentions an open mathematical problem such as Van der Waerden numbers raises the expectation that it was able to solve it (i.e., find a new Van der Waerden number), this is not a result that can be found here: we only use it as one of several examples of problem domains in which the high-level structure can be used for improving run-time.

We continue in the next section by describing the method in detail. In Sec. III we will demonstrate how to apply it with several famous problems.

II. FINDING ADDITIONAL E-CLAUSES

Let us recap. *Symmetry* over φ is a propositionally-consistent map $\sigma : lits(\varphi) \mapsto lits(\varphi)$ such that $\sigma(\varphi) \equiv \varphi$. In this situation we can add symmetry-breaking constraints, and also use dynamic symmetry exploitation by adding e-clauses, but not both.

Almost symmetries refer to a situation where we have a formula $\varphi \equiv \varphi_1 \cup \varphi_2$ and a propositionally-consistent map

$\sigma : lits(\varphi_2) \mapsto lits(\varphi_2)$ such that $\sigma(\varphi_2) \equiv \varphi_2$. Here we *cannot* add symmetry-breaking constraints because of the φ_1 clauses, but we can still use dynamic symmetry exploitation by adding e-clauses that are based on φ_2 .

We now generalize almost symmetries as follows. Let

$$\varphi \equiv \varphi_1 \cup \varphi_2 \cup \varphi_3, \quad (7)$$

where $\varphi, \varphi_1, \dots$ are sets of clauses, possibly overlapping. Let $\sigma : lits(\varphi_2) \mapsto lits(\varphi_3)$ be a literal map such that

$$\sigma(\varphi_2) \equiv \varphi_3. \quad (8)$$

Our central claim is:

Proposition 1: Let c be a conflict clause that was learned from φ_2 ’s clauses, i.e., $\varphi_2 \models c$. Then φ and $\varphi \cup \sigma(c)$ have the same solutions.

Proof: Consider the resolution process by which c was inferred from φ_2 . The same resolution process can be applied to $\sigma(\varphi_2)$, and the result will be $\sigma(c)$. Hence $\sigma(\varphi_2) \models \sigma(c)$, and because of (8) we have $\varphi_3 \models \sigma(c)$. Therefore, $\varphi \models \sigma(c)$ and we can add the e-clause $\sigma(c)$ to φ without removing solutions. ■

The following table summarizes the discussion so far.

	Symmetry	Almost symmetry	e-clauses
Formula:	φ	$\varphi_1 \cup \varphi_2$	$\varphi_1 \cup \varphi_2 \cup \varphi_3$
Requires:	$\sigma(\varphi) \equiv \varphi$	$\sigma(\varphi_2) \equiv \varphi_2$	$\sigma(\varphi_2) \equiv \varphi_3$

For a given formula φ , the question is how to define φ_2, φ_3 and the corresponding mapping σ that satisfy (8). As we will see in the next section, for certain types of formulas it can be done in such a way that e-clauses can be added in linear time. In fact it can be done in multiple ways, i.e., many such mappings exist, and we can use all of them.

III. EXAMPLES

We will show here two example problems that received attention in the SAT community in recent years, and in which e-clauses can be added efficiently : Van der Waerden numbers, and Boolean Pythagorean triples. The long version of this article [1] includes additional examples: Bounded model checking, SAT-based Planning, a combinatorial problem called ‘Sweep’, and the anti-bandwidth problem.

A. Van der Waerden numbers (2 colors)

We begin with the following definition:

Definition 3: The Van der Waerden number $W(j, k)$ is the smallest integer n such that every 2-coloring of $1..n$ has a monochromatic arithmetic progression of length j of color 1, or of length k of color 2.

For example, the following coloring proves that $W(3, 3) > 8$, since there is no arithmetic progression of size 3 of either color:



However, there is no such coloring for $n = 9$, hence $W(3, 3) = 9$.

There is relatively little symmetry in such formulas. An obvious one is the symmetry between the colors, when $j = k$. Another type of symmetry is reversal (reading the sequence from the end). Reconsidering Example 1, σ_1, σ_2 of (4) break these two symmetries.

Given j, k and n , encoding the decision problem whether $W(j, k) > n$ with CNF is simple. Define n variables x_i for $1 \leq i \leq n$, indicating whether location i is assigned the color ‘1’. The constraints on the arithmetic progression are given by

$$\begin{aligned} & \{(x_i \vee x_{i+d} \vee \dots \vee x_{i+(j-1)d}) \mid i \in [1, n - (j-1)d], d \geq 1\} \\ & \cup \\ & \{(\bar{x}_i \vee \bar{x}_{i+d} \vee \dots \vee \bar{x}_{i+(k-1)d}) \mid i \in [1, n - (k-1)d], d \geq 1\} \end{aligned} \quad (9)$$

as was described, e.g., by Knuth in [17]. From here on we will use integers as representatives of literals.

Example 2: Consider the case of $j = k = 3, n = 10$. When a variable i is assigned true, it represents the decision to assign slot i the color ‘1’, and ‘0’ otherwise. Then no 3 slots...

- ... with gap 1 are all ‘0’: (1 2 3) (2 3 4) ... (8 9 10)
- ... with gap 2 are all ‘0’: (1 3 5) (2 4 6) ... (6 8 10)
- ... with gap 3 are all ‘0’: (1 4 7) (2 5 8) ... (4 7 10)
- ... with gap 4 are all ‘0’: (1 5 9) (2 6 10)

The same constraints, but with negated literals, are now added for the color ‘1’. For example, for gap 1, add $(-1, -2, -3) \dots (-8, -9, -10)$, etc. ■

The clauses as defined in (9) have what we call a *gliding symmetry*². This means that the same clause is replicated in the formula while shifting the variable index by a constant up to some bound, for example (1 2 3) is in φ , but also (2 3 4)...(8 9 10). Similarly $(-1 -2 -3)$ is replicated with a negative constant. For a clause c , let c_z^i denote the clause attained by taking i steps towards zero, and similarly let c_n^i denote the clause attained by taking i steps away from zero, i.e., towards n or $-n$. For example $(3 4 5)_z^1 = (2 3 4)$ and $(1 2 3)_n^1 = (2 3 4)$. As another example, this time focusing on the negative constraints, $(-1 -3 -5)_n^1 = (-3 -5 -7)_z^1 = (-2 -4 -6)$.

For each clause $c \in \varphi$, we save the *gliding bounds* $[i, j]$, where i, j are the maximal integers such that $c_z^i, c_n^j \in \varphi$. For example, for the clause $c = (2 3 4)$ of Example 2, we save the pair $[1, 6]$, because we can ‘glide’ by up to one step towards zero and by up to six steps towards $n = 10$ (giving us, respectively, (1 2 3) and (8 9 10)). As another example, the pair for the clause $(-4 -5 -6)$ is $[3, 4]$, because we can glide by up to three steps towards zero, and by up to four steps towards $-n = -10$. Denote by $c.z$ and $c.n$ the two bounds of a clause c , corresponding to i, j above, respectively.

So far we only considered the original clauses of the problem. We now consider the question of what are the bounds for the learned clauses. Let c_1, \dots, c_m be the antecedent clauses of a new learned clause c . We compute the gliding bounds of c as follows:

$$c.z = \min(c_1.z, \dots, c_m.z) \quad c.n = \min(c_1.n, \dots, c_m.n) \quad (10)$$

²Mathematicians use this term for describing a pattern that repeats itself by an operation of shifting in one dimension in space, e.g., ♠ ♠ ♠ ♠ ...

The rational of (10) is that we can only glide c towards zero (or away from zero) as much as we can glide all of its antecedents towards zero (or away from zero).

Given the gliding bounds of each clause, it is easy to use Proposition 1 for learning new e-clauses. Using the terminology of that proposition, the antecedents of c form φ_2 , and σ is a mapping that applies ‘gliding’ to them. Each amount of gliding is a separate mapping σ . The gliding bounds tell us the amount by which gliding each clause results in a clause that is still in φ – those new clauses are φ_3 in the proposition. In other words, those bounds define the mappings that we can use for deriving new e-clauses.

Example 3: Suppose φ includes the following clauses and respective bounds:

$$(3 \ 6 \ 10)[2, 0] \quad (-7 \ -5 \ -3)[2, 2] \quad (-7 \ -6 \ -5)[4, 2] \quad (11)$$

from which the solver inferred via resolution the clause $c = (-7 \ -5 \ 10)$. With (10) we compute the gliding bounds $[2, 0]$ for c . This means that we have two mappings:

- σ_1 maps each positive literal l to $l-1$ and negative literal $-l$ to $-l+1$
- σ_2 maps each positive literal l to $l-2$ and negative literal $-l$ to $-l+2$,

i.e., a glide by one and two towards 0. So we add the e-clauses $\sigma_1(c) = (-6 \ -4 \ 9)$ and $\sigma_2(c) = (-5 \ -3 \ 8)$. Indeed, if we apply σ_1 to the clauses in (11), we get three clauses in φ , from which we can infer $\sigma_1(c)$:

$$\begin{aligned} \sigma_1(3 \ 6 \ 10) &= (2 \ 5 \ 9) & \sigma_1(-7 \ -5 \ -3) &= (-6 \ -4 \ -2) \\ \sigma_1(-7 \ -6 \ -5) &= (-6 \ -5 \ -4) \end{aligned}$$

Finally, we should compute the gliding bounds of the e-clauses themselves, because they may participate in further learning. For this, we shift the bounds of the conflict clause by the same amount as dictated by the mapping σ , while recalling that any step towards zero is a step away from n (or $-n$ if it is a negative literal), and vice versa.

Example 4: Reconsider c of Example 3. Its bounds are $[2, 0]$. We computed $\sigma_1(c)$ by gliding c towards zero by 1. Hence the bounds of $\sigma_1(c)$ are $[2-1, 0+1] = [1, 1]$. ■

B. Boolean Pythagorean triples

We conclude with an example that shows that e-clauses are not necessarily tied to gliding symmetry.

Three positive integers a, b, c are called a Pythagorean triple if they satisfy $a^2 + b^2 = c^2$. The challenge is:

Definition 4: For a given $n \in \mathbb{N}$, can $1..N$ be separated into two sets, such that no set contains a Pythagorean triple?

As an example, for $n = 17$ if we choose the subset of integers that is here marked with an underline: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17, it proves that for $n = 17$ the answer is yes.

The general question of whether there exists an n for which the answer is negative was open for many years. The celebrated result of Heule et al. [16] a few years ago proved, with the help of a SAT solver, that the answer is positive.

The encoding of the problem in Def. 4 with CNF is very simple: define n variables, where the Boolean values in the satisfying assignment separate the values naturally to the two requested sets. For example, the encoding for $n = 17$ is

$$\begin{array}{cccc} (3 \ 4 \ 5) & (-3 \ -4 \ -5) & (5 \ 12 \ 13) & (-5 \ -12 \ -13) \\ (6 \ 8 \ 10) & (-6 \ -8 \ -10) & (8 \ 15 \ 17) & (-8 \ -15 \ -17) \\ (9 \ 12 \ 15) & (-9 \ -12 \ -15) & & \end{array}$$

Denote by φ_n this formula for a given n . In the discussion that follows we will overload the multiplication and division signs, \cdot and $/$ to operate on clauses and sets of clauses: the operation is simply applied to each of the literals. For example, $2 \cdot (3 \ 4 \ 5) = (6 \ 8 \ 10)$ and $(6 \ 8 \ 10)/2 = (3 \ 4 \ 5)$.

We begin with two simple observations:

Observation 1: Pythagorean triples are closed under multiplication:

$$\forall a, b, c, i \in N. \ a^2 + b^2 = c^2 \Rightarrow (a \cdot i)^2 + (b \cdot i)^2 = (c \cdot i)^2.$$

Observation 2: Let $|_d$ denote ‘divisible by d ’. When applied to a set of numbers, then it means that all the set’s members are divisible by d . Then for all n ,

$$(a \ b \ c) \in \varphi_n \wedge (a \ b \ c)|_d \Rightarrow \frac{(a \ b \ c)}{d} \in \varphi_n. \quad (12)$$

The second observation is simply the other side of the first one (dividing rather than multiplying), but it also states that the divided clause must be in φ_n . For example, if $n = 80$ then $(30 \ 72 \ 78) \in \varphi_{80}$, which implies that also $(30 \ 72 \ 78)/2 = (15 \ 36 \ 39) \in \varphi_{80}$.

For each clause c , we define recursively

$$c.gcd = \begin{cases} gcd(\{l \mid l \in c\}) & c \text{ is original} \\ gcd(\{c_i.gcd \mid c_i \in S\}) & c \text{ is inferred from a clause set } S \end{cases} \quad (13)$$

where $gcd()$ is the greatest common divider function. Observe that if c is original, then $c.gcd$ is the greatest common divider of its own variables, and otherwise of the variables in the core of original clauses that derived it, which we will denote by $core(c)$. This recursive definition gives us an immediate method to implement it in a SAT solver: the base case corresponds to the original clauses, and the step to the learning that is done during conflict analysis.

Given a conflict clause c , we can see that for $i \in [1, bound(n)]$ ($bound(n)$ will be defined shortly), we have

$$i \cdot \frac{core(c)}{c.gcd} \subseteq \varphi_n. \quad (14)$$

This is a direct result of the two observations above: From Observation 2 we know that $\frac{core(c)}{c.gcd} \subseteq \varphi_n$, and from Observation 1 we know that any multiplication of this clause is a Pythagorean triple. Whether it is part of φ_n depends on the value of i , which brings us to the problem of computing $bound(n)$. To compute it, we need to know the largest variable

that participates in $core(c)$. For each clause c , we define recursively

$$c.maxvar = \begin{cases} \max(\{l \mid l \in c\}) & c \text{ is original} \\ \max(\{c_i.maxvar \mid c_i \in S\}) & c \text{ is inferred from a set } S \end{cases}$$

Hence, for each clause c , $c.maxvar$ denotes the largest variable that appears in $core(c)$. In (14) we considered clauses $i \cdot \frac{core(c)}{c.gcd}$. For these clauses to be part of φ_n , the following relation should hold:

$$i \cdot \frac{c.maxvar}{c.gcd} \leq n.$$

Isolating i gives us the bound: $bound(n) = \frac{n \cdot c.gcd}{c.maxvar}$. Finally, observe the implication of (14): since $i \cdot \frac{core(c)}{c.gcd} \subseteq \varphi_n$, then

$$\varphi_n \models i \cdot \frac{c}{c.gcd}, \text{ for } i \in [1, bound(n)]. \quad (15)$$

This means that $i \cdot \frac{c}{c.gcd}$ can be added safely as e-clauses to φ_n , without removing solutions. In other words, using the terminology of Sec. II, each $i \in [1, bound(n)]$ defines us a separate mapping for a conflict clause c :

$$\sigma_i(c) = i \cdot \frac{c}{c.gcd}. \quad (16)$$

IV. IMPLEMENTATION DETAILS

Recall that according to (7) the formula may contain a non-empty set of clauses φ_1 , that cannot participate in generating e-clauses. In our implementation we mark those clauses at the beginning (such clauses are expected to be given in a separate input file), and then also each learned clause that one of its antecedents is marked that way. For simplicity let us call these clauses *non-symmetric* and the rest *symmetric*.

To keep track of these dependencies, we altered the solver. This is a non trivial task because logical dependency between clauses is created in many different parts of a modern solver. In particular, our implementation is based on MAPLE_LCM_DIST_CHRONOBT [20] (we will abbreviate its name to CHRONO from hereon), the winner of the SAT competition in 2018, which in itself is built on top of multiple generations of optimizations that were added to it over the years, all the way up to MINISAT-2.2 [12]. In particular, dependency is created during conflict analysis in the process of learning a new clause, but also during clause minimization, binary-resolution minimization, learnt-clause simplifications, var elimination and propagation at decision level 0³. We maintain a single bit in the header of each clause that determines whether it is symmetric or not. Since CHRONO, like all MINISAT-based solvers, do not maintain unit clauses, we maintain a separate list of variables that their value is determined at level 0 based on non-symmetric clauses.

Next, we need to maintain problem-specific information that is necessary for deriving e-clauses. For example, for Van der Waerden formulas – see Secs. III-A – we need to keep for

³These are implemented in the following functions in CHRONO: analyze, LitRedundant, binResMinimize, simplifyLearnt, eliminateVar, propagate

each clause its gliding bounds. For the Boolean Pythagorean triples problem – see Sec. III-B – we maintain the greatest common divider (gcd) of the literals in the clause and all clauses that participated in deriving it, and the max variable in those clauses. As in the case of the symmetry bit described above, here too we need to update this information in every location in which dependency is created.

Our implementation accumulates e-clauses and then adds them to the clause database at the nearest restart. This is a different strategy than the ones mentioned in the introduction in the context of symmetric explanation learning [10] and dynamic symmetry handling [10], [25], where such clauses are added during BCP, hence affecting the current search branch (we implemented both, and the results are rather similar, with a small advantage to the technique described here). To reduce side-effects, upon adding a new e-clause we do *not* increase the counter of conflict clauses, since that counter affects various other heuristics, such as the frequency of applying simplifications and clause deletion.

The above-mentioned prior works describe various filtering methods: adding clauses only if they conflict the current state or lead to further propagation, or, in the case of [25], if the conflict clause itself has a low LBD. Several filtering and deletion strategies that we experimented with are described in the long version of this article [1]. Briefly, the ones we settled on as best in our experiments are (1) add an e-clause only if up to 3 literals are not false under the current partial assignment, and (2) do not add e-clauses larger than 20. As for deletion strategies, we (1) gave a separate initial activity score of 0.8 for e-clauses and (2) set the deletion ratio to 0.8, i.e., a more aggressive deletion comparing to the default of 0.5. We left this deletion ratio also for the experiments without e-clauses, for a fair comparison.

V. RESULTS

We implemented this method for Van der Waerden numbers and Boolean Pythagorean triples. Since there is no standard benchmark sets for these problems, we generated instances, and took all of those that can be solved with at least one configuration in less than 30 min., and with at least one configuration in more than 1 min. For the Van der Waerden problems, this resulted in 30 benchmarks (16 unsat, 14 sat). The benchmarks, full tables of results, and the implementation are available from [3]. We used the HBENCH benchmarking system [2] to conduct the experiments and data collection.

In the results tables below, timed-out benchmarks contribute the values they had at the timeout point to the various columns, other than the **par-2** column, where the timeout is added twice, to be consistent with the ranking method of the SAT competitions. Our goal was mostly to measure the number of e-clauses that can be found based on isomorphic subgraphs, beyond what can be found with dynamic symmetry exploitation. We have evidence from multiple previous works, e.g., [10], [25], [24] (see Sec. I), that such clauses can help in reducing the run time. Our results below show not only that indeed many more such clauses can be generated, but also that

when combined with the right filtering and deletion methods, it reduces the run time on average.

The results for the Van der Waerden problems are summarized in Table I, sorted by performance. The ‘-waerden’ flag indicates that e-clauses are added as described in Sec. III-A. The ‘-dyn-sym-exploit’ flag indicates that e-clauses based on dynamic symmetry exploitation were added. ‘native’ means that the solver was run in its default configuration other than the deletion ratio – see Sec. IV. ‘static-sym-breaking’ indicates that we solved the formula with static symmetry-breaking constraints, as provided by BREAKID, while the solver is in the same configuration as ‘native’. For these benchmarks static symmetry breaking turns out to be better than dynamic symmetry exploitation, based on the same data (even when considering the unsat cases on their own).

On average each conflict clause learned while solving these benchmarks results in over 20 e-clauses with the -waerden flag (this clearly depends on the value of n), and less than 1 with the -dyn-sym-exploit. The latter is expected, since BREAKID generates a single generator for these benchmarks (see text after Def. 3). The top part of the table does not reflect these numbers, however, because it refers to runs in which we applied aggressive filtering as mentioned before. With these filters, the number of e-clauses added is typically less than 5% of the total number of clauses. Hence the potential for e-clauses is large, and perhaps future research into filtering techniques will be able to exploit this unused potential. The overhead of generating the e-clauses is marginal (the ‘Overhead’ column). The overhead of running BREAKID, a necessary step for applying both -dynamic-symmetry and -symmetry-breaking, was a few seconds and not included in the ‘Time’ column.

We can see a run-time reduction of 42% comparing to a native run for the case of Van der Waerden formulas, and of 55% for the case of Pythagorean triples. In both cases the technique as described in III-A is better than adding e-clauses based on data derived from static symmetry, and better than combining these two sources of data. Cactus plots for both families appear in Figs. 2 and 3.

We also checked how active the e-clauses are in deriving new clauses. For this measure we define as e-clauses, recursively, the set of clauses that we add directly and the clauses that were learned based on at least one e-clause premise. Activity of clauses is updated in the solver in the usual way, based on their participation in deriving other clauses. Since clause deletion is based on this activity, the ratio between the average number of ‘live’ clauses (i.e., that were not deleted) and the total number of learned clauses is an indication of how active they are. This ratio for e-clauses and normal conflict clauses appear in the last two columns of the table. It is surprising to see that the e-clauses are more active, especially since we initiate the activity score of e-clauses with a lower value in comparison to the value given to conflict clauses.

For the Boolean Pythagorean triples problem, we generated 21 satisfiable instances (the first unsatisfiable instance takes weeks to solve — see [16]) with the same selection criteria

Configuration	Timed-out	Time	Time (par-2)	Conflicts	e-clauses	Over-head	Active -E-	Active -C-
-waarden	0	111.2	111.2	1,079,719	30568	6	0.017	0.015
-static-sym-breaking	1	149.8	211.2	2110472	0	0		
(native)	1	190.4	251.7	2,112,666	0	0		0.011
-waarden -dyn-sym-exploit	2	198.5	317.7	1,963,104	50618	10	0.014	0.008
-dyn-sym-exploit	3	233.2	418.6	2,477,840	6,729	3	0.011	0.008
-waarden	6	476.5	841.9	750,453	16,556,216	29	0.013	0.013
-dyn-sym-exploit	1	119.8	181.3	1,248,573	1,290,402	13	0.008	0.011

TABLE I

AVERAGE RESULTS FOR THE VAN DER WAERDEN PROBLEM, OVER 30 BENCHMARKS. TIME IS IN SECONDS. THE LAST TWO ROWS REFER TO RUNS WITHOUT ANY FILTERING OF THE E-CLAUSES.

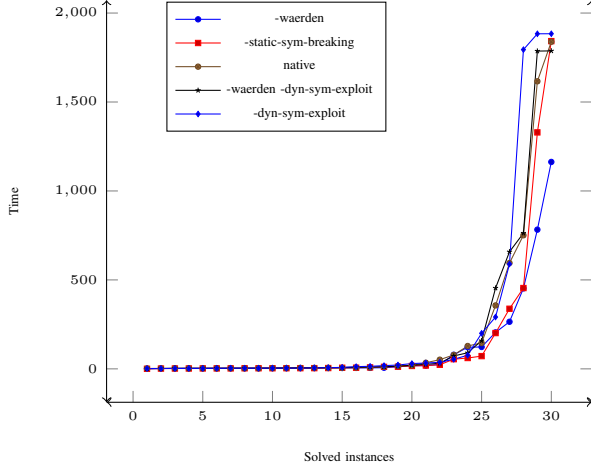


Fig. 2. Results for the Van der Waerden benchmarks.

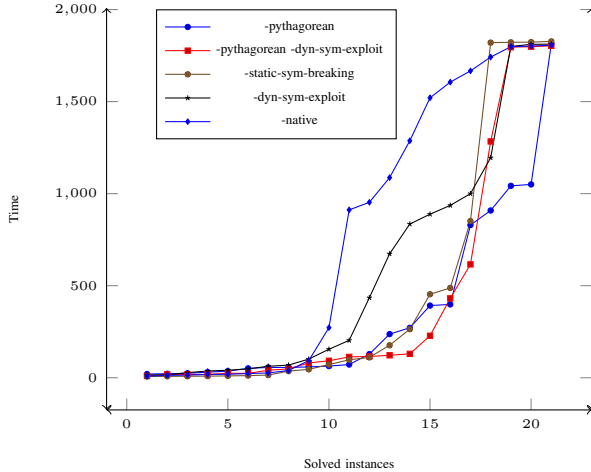


Fig. 3. Results for the Pythagorean-triples benchmarks.

as described above. The results appear in Table II, also in ascending performance order. Here the native solver turns out to be improved-upon in each of the configurations, including static symmetry breaking.

VI. CONCLUSIONS AND FUTURE WORK

We presented a general condition for adding what we call e-clauses, right after conflict analysis. We showed how this technique generalizes ‘symmetry’ and ‘almost symmetry’, and that indeed this method can add far more clauses than dynamic symmetry exploitation and related methods that are solely based on such symmetries. We showed several known problems for which this is relevant, and mentioned cases in which it was already done in the past with empirical success.

There are three lines of future work that we consider important. First, it is important to classify additional problems as having the property that they are amenable to adding e-clauses, and check whether it can assist in accelerating their solving. Second, we foresee a dedicated SAT solver that maintains and reasons about *clause generators*. That is, instead of adding many e-clauses as normal clauses, just keep the base learned clause with its bounds. It can be faster than the alternative of adding all e-clauses and does not suffer from the necessity to delete most of them. In a sense, this way the e-clauses are generated lazily, on demand, and then immediately erased. There are many implementation details that need to be developed for this. For example, one can add the generator to the watch list of all the literals that would have watched one of its generated e-clauses. In BCP, that literal tells us how to apply the unit implication rule to the generator. The reason clause can be maintained as a pair of a reference to the generator and an instantiation index. Many other details still need to be worked out.

A third direction, is to control the BCP order, such that it works first on ‘normal’ clauses and only if it terminates without a conflict, continue to propagate through the e-clauses, based on the assumption that the latter are less likely to cause a conflict at the current branch. One can also envision a SAT solver that splits BCP on normal and e-clauses between two threads. A possible high-level architecture is one in which the main thread, T , works on ‘normal’ clauses and then on e-clauses, and the other, T_e , in the other direction. The first that finds a conflict terminates the other, or, alternatively, the solver chooses the better conflict clause based on its LBD and backtracking level.

Configuration	Timed-out	Time	Time (par-2)	Conflicts	e-clauses	Over-head	Active -E-	Active -C-
-pythagorean	1	360.1	446.5	1,973,404	60303.3	0.2	0.006	0.006
-pythagorean -dyn-sym-exploit	3	419.5	676.6	1,864,767	55264.4	36.0	0.007	0.006
-static-sym-breaking	4	474.1	821.4	3,132,118	0	0		
-dyn-sym-exploit	3	579.1	837.4	2,558,436	388.4	50.5	0.004	0.007
(native)	3	795.6	1053.7	3,901,308	0	0		0.007
-pythagorean	4	578.0	1008.9	3,045,218.4	214,993.7	0.4	0.004	0.054
-dyn-sym-exploit	9	931.7	1714.1	1,813,843.8	3,541,747.0	568.9	0.005	0.007

TABLE II

RESULTS FOR THE BOOLEAN PYTHAGOREAN TRIPLES PROBLEM, OVER 21 BENCHMARKS. THE BOTTOM TWO CONFIGURATIONS ARE WITHOUT FILTERING.

REFERENCES

- [1] Exploiting isomorphic subgraphs in SAT (long). <https://arxiv.org/abs/2103.10267>.
- [2] Hbench. <https://strichman.net.technion.ac.il/hbench/>.
- [3] Statistics. https://technionmail-my.sharepoint.com/:f/g/personal/ofers_technion_ac_il/EIPLTu98GFNGidn8MHqydHYBzKlKJe1WFKS0-g8s78wV0w?e=17Id5A.
- [4] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 22(9):1117–1137, 2003.
- [5] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. *Constraints An Int. J.*, 7(3-4):333–349, 2002.
- [6] B. Benhamou, T. Nabhani, R. Ostrowski, and M. R. Saidi. Enhancing clause learning by symmetry in sat solvers. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 329–335, 2010.
- [7] Geoffrey Chu, Maria Garcia de la Banda, Christopher Mears, and Peter J. Stuckey. Symmetries, almost symmetries, and lazy clause generation. *Constraints An Int. J.*, 19(4):434–462, 2014.
- [8] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, Cambridge, Massachusetts, USA, November 5-8, 1996, pages 148–159. Morgan Kaufmann, 1996.
- [9] J. Devriendt, B. Bogaerts, B. d. Cat, M. Denecker, and C. Mears. Symmetry propagation: Improved dynamic symmetry breaking in sat. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, volume 1, pages 49–56, 2012.
- [10] Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017, Proceedings*, volume 10491 of LNCS, pages 83–100. Springer, 2017.
- [11] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2016.
- [12] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of LNCS, pages 502–518. Springer, 2003.
- [13] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2001.
- [14] Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints: Symmetry breaking during search. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 415–430. Springer, 2002.
- [15] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In Werner Horn, editor, *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000*, pages 599–603. IOS Press, 2000.
- [16] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.
- [17] Donald Knuth. *The Art of Computer Programming: Satisfiability*, volume 4b, Fascicle 6. 2015.
- [18] Balakrishnan Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22(3):253–275, 1985.
- [19] Roland Martin. The challenge of exploiting weak symmetries. In Brahim Hnich, Mats Carlsson, Fran(c)ois Fages, and Francesca Rossi, editors, *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and Invited Papers*, volume 3978 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2005.
- [20] Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *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*, volume 10929 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 2018.
- [21] Jean-Francois Puget. On the satisfiability of symmetrical constrained satisfaction problems. In Henryk Jan Komorowski and Zbigniew W. Ras, editors, *Methodologies for Intelligent Systems, 7th International Symposium, ISMIS '93, Trondheim, Norway, June 15-18, 1993, Proceedings*, volume 689 of *Lecture Notes in Computer Science*, pages 350–361. Springer, 1993.
- [22] Karem Sakallah. *Symmetry and Satisfiability*, chapter 10, pages 289–338. IOS press, 2009.
- [23] Buser Say, Jo Devriendt, Jakob Nordström, and Peter J. Stuckey. Theoretical and experimental results for planning with learned binarized neural network transition models. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 917–934. Springer, 2020.
- [24] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Proc. 12th Intl. Conference on Computer Aided Verification (CAV'00)*, LNCS. Springer-Verlag, 2000.
- [25] Rodrigue Konan Tchinda and Clémentin Tayou Djamégni. Enhancing static symmetry breaking with dynamic symmetry handling in CDCL SAT solvers. *Int. J. Artif. Intell. Tools*, 28(3):1950011:1–1950011:32, 2019.

On Decomposition of Maximal Satisfiable Subsets

Jaroslav Bendík 

Max Planck Institute for Software Systems

Kaiserslautern, Germany

xbendik@mpi-sws.org

Abstract—In many areas of computer science, we are given an unsatisfiable formula F in CNF, i.e., a set of clauses, with the goal to analyze the unsatisfiability. A kind of such analysis is to identify Minimal Correction Subsets (MCSes) of F , i.e., minimal subsets of clauses that need to be removed from F to make it satisfiable. Equivalently, one might identify the complements of MCSes, i.e., Maximal Satisfiable Subsets (MSSes) of F . The more MSSes (MCSes) of F are identified, the better insight into the unsatisfiability can be obtained. Hence, there were proposed many algorithms for complete MSS (MCS) enumeration. Unfortunately, the number of MSSes can be exponential w.r.t. $|F|$, which often makes the complete enumeration practically intractable.

In this work, we attempt to cope with the intractability of complete MSS enumeration by initiating the study on *MSS decomposition*. In particular, we propose several techniques that often allows for decomposing the input formula F into several subformulas. Subsequently, we explicitly enumerate all MSSes of the subformulas, and then combine those MSSes to form MSSes of the original formula F . An extensive empirical study demonstrates that due to the MSS decomposition, the number of MSSes that need to be explicitly identified is often exponentially smaller than the total number of MSSes. Consequently, we are able to improve upon a scalability of contemporary MSS enumeration approaches by many orders of magnitude.

I. INTRODUCTION

Boolean formulas in the Conjunctive Normal Form (CNF), wherein we are given a set $F = \{c_1, \dots, c_n\}$ of Boolean clauses, have been widely adopted as a suitable representation language to model the behaviour of systems and properties. In case we are given an unsatisfiable CNF formula F , the goal is usually to analyze the unsatisfiability. To perform such an analysis, two concepts are often used: a *Minimal Unsatisfiable Subset* (MUS) of F , and a *Minimal Correction Subset* (MCS) of F . Intuitively, an MUS represents a minimal reason for the unsatisfiability, whereas an MCS is a minimal subset of clauses that need to be removed from F to make it satisfiable. A dual notion to an MCS is that of a Maximal Satisfiable Subset (MSS), i.e., a satisfiable subset M of F such that for every clause $c \in F \setminus M$ the set $M \cup \{c\}$ is unsatisfiable. It holds that every MSS is a complement of an MCS of F and vice versa, i.e., MSSes and MCSes represent the same information.

MCSes (MSSes) find many practical applications in various areas of computer science. For instance, in the context of belief update and argumentation, MCSes are used during an update of the belief in the presence of an incoming contradictory belief [16], [21]. Similarly, in the field of diagnosis of constraint systems [5], [37], [49], MCSes represent the constraints that need to be relaxed for the system to be conflict-free. Another application of MSSes arises in the context of

the maximum satisfiability problem (MaxSAT), since MSSes with the maximum cardinality correspond to the solutions of MaxSAT. Yet other applications of MCSes can be found, e.g., during model based diagnosis [7], ontology debugging, or axiom pinpointing [1].

Often, it is the case that finding just a single MCS is sufficient. However, in many applications, the task of enumerating several or even all MCSes (MSSes) is crucial for properly understanding the underlying sources of the unsatisfiability. For example, enumeration of minimal correction subsets is essential in software fault localization [30]. In the context of MaxSAT solving, a restricted MSS enumeration is effective in approximately solving the problem if finding the exact solution is intractable [41]. In the domain of diagnosis, there have been proposed many diagnosis metrics that are based on complete enumeration and counting of MSSes and MCSes (see, e.g., [26], [52]). Moreover, there are several computational problems, such as enumeration of minimal unsatisfiable subsets [37], prime implicants [28], and maximal and minimal models [39], that can be reduced to MSS enumeration.

In the past decades, there have been proposed many approaches for enumeration of MSSes (see e.g., [5], [9], [11], [22], [35], [39], [44], [51]). However, the complete MSS enumeration is still often practically intractable [11]. One of the reasons is that the identification of the individual MSSes naturally subsumes checking several subsets of F for satisfiability, and these checks are very expensive (NP-complete). Another issue is that there can be in general exponentially many MSSes of F w.r.t. the number $|F|$ of clauses of F .

In spirit, the intractability of complete MSS enumeration is very similar to the intractability that was dealt with in the context of the Boolean model counting problem. That is, given a Boolean formula H , count all models (satisfying assignments) of H . The earliest approaches for model counting were based on a complete model enumeration, however, since the number of models can be exponential w.r.t. the number of variables of H , the complete model enumeration is often practically intractable. Fortunately, due to an extensive research in the past decades (e.g., [6], [43], [50], [53]), the model counting problem is often practically feasible even for formulas with exponentially many models. A substantial ingredient of contemporary model counters is *decomposition*; in particular, the counters are often able to decompose the input formula H into several independent sub-formulas, then count models of the sub-formulas, and multiply the sub-counts to get the model count for the whole H . At this point, one

might wonder *whether it is possibly to perform some kind of a decomposition in the context of MSS enumeration?*

In this paper, we initiate the study on the problem of MSS decomposition, and provide an affirmative answer to the above question. In particular, we propose two decomposition techniques that are applicable to some kinds of formulas. The first technique attempts to *directly* decompose the input formula F into several independent components (i.e., disjoint subsets of clauses) based on literals in the individual clauses. Due to the decomposition, we can first identify all MSSes of the individual components (using any existing MSS enumerator), and then form the MSSes of F by just cheaply composing the MSSes of the components. Note that the sum of the MSSes in the individual components can be exponentially smaller than the total number of MSSes of F that we obtain from the composition. The second technique is applicable when the input formula F is not *directly decomposable*. In such a case, we first attempt to identify a suitable *cut* K for F , i.e., a subset K of F such that the formula $F \setminus K$ can be directly decomposed. In this case, we can divide the MSSes of F into two groups: 1) MSSes that are subsets of $F \setminus K$, and 2) the remaining MSSes of F . The former group can be decomposed and solved via the first decomposition technique, whereas the latter group can be identified via any existing MSS enumerator.

Based on the two decomposition techniques, we build a novel MSS enumeration algorithm and experimentally compare it with other contemporary MSS enumeration tools. Out of 1491 benchmarks, the best contemporary approach can solve only 415 benchmarks, whereas our approach solves 788 benchmarks. Moreover, whereas contemporary approaches scale only to instances with at most 10^8 MSSes, our approach can handle even benchmarks with 10^{22} MSSes.

Outline. The rest of the paper is organized as follows. Section II introduces preliminaries and Section III discusses related work. The two decomposition techniques are introduced in Section IV, and our MSS enumeration algorithm is presented in Section V. Section VI provides results of our experimental evaluation. Finally, Section VII discusses practical limitations of our approach, and Section VIII concludes.

II. PRELIMINARIES

Standard definitions for propositional (Boolean) logic are assumed. A Boolean formula F is built over a set $\text{Vars}(F)$ of Boolean variables. A *literal* l is either a variable $x \in \text{Vars}(F)$ or its negation $\neg x$, and $\text{Lits}(F)$ denotes the set of all literals used in F . A *clause* $c = \{l_1, \dots, l_k\}$ is a set of literals. A Boolean formula in conjunctive normal form $F = \{c_1, \dots, c_n\}$, shortly a *CNF formula*, is a set of clauses.

Given a CNF formula F , a *valuation* π of $\text{Vars}(F)$ is a mapping $\pi : \text{Vars}(F) \rightarrow \{1, 0\}$. The valuation π *satisfies* a clause $c \in F$ iff there exists a variable x such that $x \in c$ and $\pi(x) = 1$ or $\neg x \in c$ and $\pi(x) = 0$. Moreover, π *satisfies* F if it satisfies every clause $c \in F$; such a valuation π is called a *model* of F . Finally, F is *satisfiable* if it has a model, and otherwise, F is *unsatisfiable*.

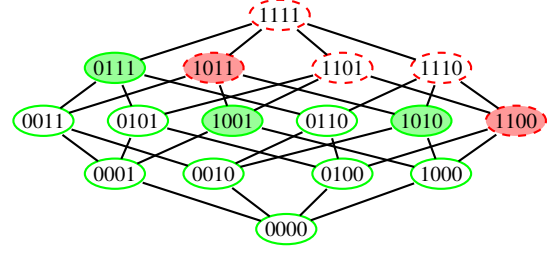


Fig. 1: Illustration of $\mathcal{P}(F)$ from the Example 1. We denote individual subsets of F as bit-vectors, e.g., $\{c_1, c_3\}$ is written as 1010. The subsets with a dashed border are the unsatisfiable subsets, and the others are satisfiable subsets. The MUSes and MSSes are filled with a background color.

Throughout the whole paper, we use $F = \{c_1, \dots, c_n\}$ to denote the input unsatisfiable CNF formula of interest. Moreover, we write just *formula* instead of *CNF formula*. Finally, given a set X , we write $\mathcal{P}(X)$ to denote the power-set of X , and $|X|$ to denote the cardinality of X .

Definition 1 (MSS). A set N , $N \subseteq F$, is a maximal satisfiable subset (MSS) of F iff N is satisfiable and for every $c \in F \setminus N$ the set $N \cup \{c\}$ is unsatisfiable.

Definition 2 (MCS). A set N , $N \subseteq F$, is a minimal correction subset (MCS) of F iff $F \setminus N$ is satisfiable and for every $c \in N$ the set $F \setminus (N \setminus \{c\})$ is unsatisfiable. Equivalently, N is an MCS of F iff $F \setminus N$ is an MSS of F .

Definition 3 (MUS). A set N , $N \subseteq F$, is a minimal unsatisfiable subset (MUS) of F iff N is unsatisfiable and for every $c \in N$ the set $N \setminus \{c\}$ is satisfiable.

Note that the maximality (minimality) concept used here is a *set maximality (minimality)*, and not a *maximum (minimum) cardinality* as, e.g., in the MaxSAT problem. Consequently, there can be MSSes (MUSes) with different cardinalities, and in general, there can be up to $\mathcal{O}(2^{|F|})$ MSSes (MUSes) of F (intuitively, there are exponentially many pair-wise incomparable subsets of F (w.r.t. the subset inclusion) and all of them can be MSSes (MUSes)). Given a formula N , we write MSS_N , MCS_N , and MUS_N , to denote the set of all MSSes, MCSes, and MUSes of N , respectively. Moreover, given a subset K of N , we write MSS_N^K to denote the set of all MSSes of N that contain at least a single clause from K , i.e., $\text{MSS}_N^K = \{M \in \text{MSS}_N \mid M \cap K \neq \emptyset\}$.

Example 1. We illustrate the concepts on a simple example, depicted in Figure 1. Assume that $F = \{c_1 = \{x_1\}, c_2 = \{\neg x_1\}, c_3 = \{x_2\}, c_4 = \{\neg x_1, \neg x_2\}\}$. There are two MUSes: $\text{MUS}_F = \{\{c_1, c_2\}, \{c_1, c_3, c_4\}\}$, three MSSes: $\text{MSS}_F = \{\{c_1, c_4\}, \{c_1, c_3\}, \{c_2, c_3, c_4\}\}$, and three MCSes: $\text{MCS}_F = \{\{c_2, c_3\}, \{c_2, c_4\}, \{c_1\}\}$.

By the definition, MCSes are exactly the complements of MSSes, and hence finding MSSes is the same as finding MCSes. Both these concepts are used in the literature, since in some situations, it is more suitable to talk about *corrections*,

and in other situations about *maximal satisfiability*. In the rest of the paper, we will stick just to the notion of MSSes and focus on the following problem:

Problem 1. *Given an unsatisfiable CNF formula F , identify the set MSS_F of all MSSes of F .*

When searching for MSSes of a given formula N , it is often possible to *reduce the search-space* via the concepts of *autark variables* and *lean kernel*. A set $A \subseteq \text{Vars}(N)$ is an *autark set* for N iff there exists a valuation of A such that every clause of N that uses a variable from A is satisfied by the valuation [42]. Note that a union of two autark sets is also an autark set, and hence there exists a unique maximum autark set of N [31], [32]. The *lean kernel* of N is the set of all clauses of N that do not contain any variable from the maximum autark set. Let L be the lean kernel of N . It is well-known that the set $N \setminus L$ is a subset of every MSS of N (see, e.g., [14], [31], [32]). Furthermore, the following observation holds¹:

Observation 1. *Let N be a formula and L its lean kernel. Then $\text{MSS}_N = \{(N \setminus L) \cup M \mid M \in \text{MSS}_L\}$.*

Proof. Let A be the autarky set that corresponds to L , and let π be a valuation of A that satisfies $N \setminus L$.

\supseteq : Given $M \in \text{MSS}_L$, we show that $(N \setminus L) \cup M \in \text{MSS}_N$. First, note that $(N \setminus L) \cup M$ is satisfiable: since $A \cap \text{Vars}(M) = \emptyset$, we can combine π with a model π' of M to get a model of $(N \setminus L) \cup M$. Second, by contradiction, assume that there is a clause $c \in L \setminus M$ such that $(N \setminus L) \cup M \cup \{c\}$ has a model ϕ (i.e., $(N \setminus L) \cup M \notin \text{MSS}_N$). However, such ϕ is necessarily also a model of $M \cup \{c\}$ which contradicts that $M \in \text{MSS}_L$.

\subseteq : Given $M' \in \text{MSS}_N$, we show that $M = M' \setminus (N \setminus L) \in \text{MSS}_L$. Since $M' \supseteq M$ and M' is satisfiable, then M is also satisfiable. Now, by contradiction, assume that $M \notin \text{MSS}_L$, i.e., there exists $c \in L \setminus M$ such that $M \cup \{c\}$ is satisfiable with a model ϕ . However, since $\text{Vars}(M \cup \{c\}) \cap A = \emptyset$, we can combine ϕ with π to get a model of $M' \cup \{c\}$ which contradicts that $M' \in \text{MSS}_N$. \square

In other words, instead of searching for MSSes of the whole N , we can just search for MSSes of the lean kernel of N . If the lean kernel is relatively small, then working just with the kernel can bring a significant runtime and memory improvement.² There have been proposed several efficient algorithms for finding maximum autarky sets and the corresponding lean kernels (see, e.g., [33], [40]).

III. RELATED WORK

The problem of MSS (MCS) enumeration was extensively studied in the past decades and many various techniques for the complete enumeration were proposed, e.g., [5], [11], [22],

[35], [36], [39], [44], [46]–[48], [51]. Below, we just briefly describe the work-flow of contemporary approaches (for a more detailed overview, please refer to [8]).

Contemporary MSS enumeration approaches gradually explore the power-set of F ; *explored subsets* are those whose satisfiability is already determined by the algorithm, and *unexplored* are the other ones. When finding each subsequent MSS M , an MSS enumeration algorithm needs to ensure two things: 1) that M is so far unexplored, and 2) that M is indeed an MSS. Both these tasks are usually carried out via several calls to a SAT solver, and these SAT solver queries are the most time-consuming part of the computation. Despite the fact that extracting just a single MSS is in $\text{FP}^{\text{NP}}[\log]$ [29] (i.e., requiring $\log |F|$ calls to a SAT solver), contemporary MSS enumerators usually need to perform *just* around 1-5 SAT solver calls per MSS (see [11]). Yet, in cases where the number of MSSes is relatively large (or even exponential), the overall number of SAT solver calls is still too high, which makes the complete enumeration practically intractable.

Alternatively, one can identify all MCSes (MSSes) by exploiting the so-called *minimal hitting set duality* [17], [49] between MCSes and MUSes. The duality states that every $M' \in \text{MCS}_F$ is a minimal hitting set of MUS_F . Hence, one can first identify the set MUS_F via an MUS enumeration approach (e.g., [3]–[5], [9], [10], [12], [18], [24], [25], [35], [37], [44], [46], [51]), and then compute the minimal hitting sets of MUS_F to get all MCSes of F . However, due to potentially exponentially many MUSes w.r.t. $|F|$, the complete MUS enumeration is also often practically intractable.

Recently, we have initiated a study [14] on the problem of counting the number $|\text{MSS}_F|$ of MSSes of a given formula F . In particular, we proposed the first MSS counting technique that does not rely on a complete explicit MSS enumeration. Briefly, given a formula F , we defined two Boolean formulas W and R such that $|\text{MSS}_F| = M_W - M_R$, where M_W and M_R are the number of models of the two formulas, respectively. Therefore, we were able to determine the MSS count via two calls to a model counting tool. Crucially, contemporary model counters often need to explicitly identify just a fraction of the models, i.e., the model-counter somehow *decomposes* the task of identifying/counting MSSes. However, this decomposition is performed on the level of the model counting, whereas in this work, we propose a decomposition scheme that works natively on the structure of MSSes.

Finally, let us note that there were proposed several single MSS extractors, e.g. [2], [20], [23], [41], that are often used as subroutines of contemporary MSS enumerators. Also, there have been proposed several caching techniques, e.g. [47], [48], that can be used to speed up MSS enumerators.

IV. DECOMPOSITION OF MSSES

In this section, we provide several observations and propose several techniques that can be used to decompose the MSS enumeration problem into multiple easier sub-problems. Subsequently, in Section V, we utilize these techniques to build an efficient MSS enumeration algorithm.

¹We believe that this observation is also well-known in the community, however, we did not find any work that explicitly formulates and proves it.

²Note that we have seen many industrial benchmarks where the lean kernel is indeed relatively small. However, there are also many industrial benchmarks where the lean kernel is the whole formula; in such cases, the extraction of the lean kernel is not useful.

Definition 4 (Decomposition Graph). *Given a formula N , the decomposition graph of N , denoted $\mathcal{G}(N)$, is an undirected graph with:*

- vertices N (a vertex per clause),
- and edges $E \subseteq \{\{c_1, c_2\} \mid c_1, c_2 \in N\}$ such that $\{c_1, c_2\} \in E$ iff there exists $l \in c_1$ with $\neg l \in c_2$.

Definition 5 (Decomposition). *Given a formula N , the decomposition of N , denoted $\mathcal{D}(N)$, is the set of connected components of $\mathcal{G}(N)$ (i.e., $c_1, c_2 \in N$ belong to the same component iff there exists a path between c_1 and c_2 in $\mathcal{G}(N)$).*

Our crucial observation here is that if $|\mathcal{D}(N)| > 1$, then the problem of finding MSSes of N can be solved as follows. First, we identify the MSSes of the individual components in $\mathcal{D}(N)$. Second, we compose the MSSes of the individual components via a *compositional operator* \sqcup into MSSes of the whole N . The compositional operator and our compositional observation is formalized as follows.

Definition 6 (\sqcup). *Let $\Omega = \{\mathcal{M}_1, \dots, \mathcal{M}_p\}$ be a collection of sets of formulas. By $\sqcup(\Omega)$, we denote the set of formulas $\sqcup(\Omega) = \{M_1 \cup \dots \cup M_p \mid M_1 \in \mathcal{M}_1 \wedge \dots \wedge M_p \in \mathcal{M}_p\}$.*

Proposition 1. *Given a formula N , it holds that $\text{MSS}_N = \sqcup(\{\text{MSS}_C \mid C \in \mathcal{D}(N)\})$.*

Proof. Let $\mathcal{D}(N) = \{C_1, \dots, C_p\}$ and assume a set $M = M_1 \cup \dots \cup M_p$ such that $M_1 \in \text{MSS}_{C_1} \wedge \dots \wedge M_p \in \text{MSS}_{C_p}$.

\supseteq : Assuming $M \in \sqcup(\{\text{MSS}_C \mid C \in \mathcal{D}(N)\})$, we show $M \in \text{MSS}_N$. Let π_1, \dots, π_p be models of M_1, \dots, M_p , respectively. W.l.o.g, assume that for every $1 \leq k \leq p$ and every literal $l \in \text{Lits}(M_k)$ such that $\neg l \notin \text{Lits}(M_k)$, it holds that π_k satisfies l . By Definition 4, there are no two distinct M_i, M_j with clauses $c_i \in M_i, c_j \in M_j$ such that there exists a literal $l \in c_i$ with $\neg l \in c_j$. Consequently, for every two π_i and π_j it holds that they agree on common variables. Hence, we can compose π_1, \dots, π_p to form a model of M . To see that M is an MSS of N , assume by contradiction a clause $c \in N \setminus M$ such that $M \cup \{c\}$ is satisfiable. However, this means that there exists $1 \leq k \leq p$ such that $c \in C_k$ and $M_k \cup \{c\}$ is satisfiable, which contradicts that M_k is an MSS of C_k .

\subseteq : Assuming $M \in \text{MSS}_N$, we show $M \in \sqcup(\{\text{MSS}_C \mid C \in \mathcal{D}(N)\})$. Since M is satisfiable, then all individual M_1, \dots, M_p are also satisfiable. Now, by contradiction, assume an M_i that is not an MSS of C_i , i.e., there exists a clause $c \in C_i \setminus M_i$ such that $M_i \cup \{c\}$ has a model π_i . Furthermore, let $\pi_1, \dots, \pi_{i-1}, \pi_{i+1}, \dots, \pi_p$ be models of $M_1, \dots, M_{i-1}, M_{i+1}, \dots, M_p$. W.l.o.g, assume that for every $1 \leq k \leq p$ and every literal $l \in \text{Lits}(C_k)$ such that $\neg l \notin \text{Lits}(C_k)$, it holds that π_k satisfies l . Same as in \supseteq : above, we can compose π_1, \dots, π_p to form a model of $M \cup \{c\}$ which contradicts that M is an MSS of N . \square

Example 2. Let $N = \{c_1 = \{x_1\}, c_2 = \{\neg x_1\}, c_3 = \{x_2\}, c_4 = \{\neg x_2\}, c_5 = \{\neg x_1, \neg x_2\}, c_6 = \{y_1\}, c_7 = \{\neg y_1\}, c_8 = \{y_2\}, c_9 = \{\neg y_1, \neg y_2\}\}$. Here, $\mathcal{D}(N) = \{C_1, C_2\}$, where $C_1 = \{c_1, c_2, c_3, c_4, c_5\}$ and $C_2 = \{c_6, c_7, c_8, c_9\}$. $\text{MSS}_{C_1} =$

$\{\{c_2, c_3, c_5\}, \{c_2, c_4, c_5\}, \{c_1, c_4, c_5\}, \{c_1, c_3\}\}$ and $\text{MSS}_{C_2} = \{\{c_7, c_8, c_9\}, \{c_6, c_8\}, \{c_6, c_9\}\}$. Thus, the whole N has 12 MSSes.

As witnessed in Example 2, due to Proposition 1, we can substantially reduce the number of MSSes that need to be *explicitly* identified to obtain the whole set MSS_N . Theoretically, it might be even the case that we need to explicitly identify just logarithmically many MSSes w.r.t. $|\text{MSS}_N|$ (assume that N contains $\log_2 |\text{MSS}_N|$ components with 2 MSSes per component). However, from the practical point of view, how often is it the case that we can actually achieve such a reduction? And, moreover, what if $|\mathcal{D}(N)| = 1$, i.e., when Proposition 1 cannot be applied? *Can we still do some decomposition when $|\mathcal{D}(N)| = 1$?* We provide an affirmative answer to this question by finding *decomposition cuts* for N .

Definition 7 (decomposition cut). *Given a formula N such that $|\mathcal{D}(N)| = 1$, a set $K \subsetneq N$ is a decomposition cut for N iff $|\mathcal{D}(N \setminus K)| \geq 2$.*

Note that decomposition cuts for a formula N correspond to *graph cuts* in the decomposition graph $\mathcal{G}(N)$. Our crucial observation about decomposition cuts is stated in Proposition 2 and Corollary 1.

Proposition 2. *Let N be a formula and K its subset. Then $\text{MSS}_N = \text{MSS}_N^K \cup \{M \in \text{MSS}_{N \setminus K} \mid \forall M' \in \text{MSS}_N^K. M \not\subseteq M'\}$.*

Proof. Let us by MSS_N^K denote the set of all MSSes of N that do not contain any clause from K . Clearly, $\text{MSS}_N = \text{MSS}_N^K \cup \text{MSS}_N^K$. To prove Proposition 2, we show that $\text{MSS}_N^K = \{M \in \text{MSS}_{N \setminus K} \mid \forall M' \in \text{MSS}_N^K. M \not\subseteq M'\}$.

\subseteq : Assume $M \in \text{MSS}_N^K$, hence for all $c \in (N \setminus M)$ the set $M \cup \{c\}$ is unsatisfiable, and hence $M \in \text{MSS}_{(N \setminus K)}$. Furthermore, since M is an MSS of N , there cannot exist any $M' \in \text{MSS}_N^K$ with $M \subsetneq M'$.

\supseteq : Given $M \in \text{MSS}_{N \setminus K}$ such that $\forall M' \in \text{MSS}_N^K. M \not\subseteq M'$, we show $M \in \text{MSS}_N^K$. By contradiction, assume that $M \notin \text{MSS}_N^K$, i.e., there exists $c \in N \setminus M$ such that $M \cup \{c\}$ is satisfiable. Since $M \in \text{MSS}_{N \setminus K}$, then $c \in K$, however, that means that there exists $M' \in \text{MSS}_N^K$ such that $M' \supseteq M \cup \{c\}$. \square

Corollary 1. *Let N be a formula and $K \subsetneq N$ a decomposition cut for N . Then $\text{MSS}_N = \text{MSS}_N^K \cup \{M \in \sqcup(\{\text{MSS}_C \mid C \in \mathcal{D}(N \setminus K)\}) \mid \forall M' \in \text{MSS}_N^K. M \not\subseteq M'\}$.*

Proof. A direct consequence of Propositions 1 and 2. \square

Finally, let us note that graph structures similar to the decomposition graph have been already used in several MUS and MSS related studies (see e.g. the work on *model rotation* [54] or *MUS counting* [13], [15]).

V. DECOMPOSITION-BASED MSS ENUMERATION

In this section, we present a novel MSS enumeration algorithm that is based on the *MSS decomposition* observations introduced in the previous section. Moreover, we exploit the concept of the lean kernel which was introduced in Section II.

A. Main Procedure

The main procedure of our algorithm is shown in Algorithm 1. The input is a formula F and the output is the set MSS_F of all MSSes F . The computation starts by calling a procedure $\text{getKernel}(F)$ that identifies the lean kernel L of F . Based on Observation 1, we can now restrict ourselves just to searching for MSSes of L and then *enlarge* the MSSes of L to MSSes of the whole F . To find MSSes of L , we first use a procedure $\text{getComponents}(L)$ that determines the decomposition $\mathcal{D}(L)$ of L . Subsequently, we iteratively identify all MSSes of the individual components. In particular, each component $N \in \mathcal{D}(L)$ is first checked for satisfiability via a SAT solver (denoted $\text{isSAT}(N)$). If N is satisfiable, then N is the only MSS of N . Otherwise, we use the procedure $\text{processComponent}(N)$ to identify all MSSes of N . We store the sets of MSSes of individual components into an auxiliary set $LMSS_{\text{parts}}$. After processing all the components, we exploit Proposition 1 and build the MSSes MSS_L of L by composing the MSSes of the individual components (stored in $LMSS_{\text{parts}}$). Finally, based on Observation 1, we form the set MSS_F of all MSSes of F by adding the complement $F \setminus L$ of the lean kernel L to the individual MSSes of L .

To implement the procedure $\text{getKernel}(F)$ that identifies a lean kernel of a given formula F , we employ an approach proposed in [40]. To implement the procedure $\text{getComponents}(L)$ that finds the decomposition $\mathcal{D}(L)$ of L , we build the decomposition graph $\mathcal{G}(L)$ and identify its connected components (any graph algorithm for finding connected components can be used). Finally, the procedure $\text{processComponent}(N)$ is more involved and it is described in the following subsection.

B. Processing a Component

The procedure $\text{processComponent}(N)$ (Algorithm 2) starts by computing the lean kernel I of N . Then, we identify a decomposition cut K for I via a procedure $\text{findCut}(I)$. Subsequently, following Corollary 1, we identify all MSSes of I .

In particular, first, we employ an existing MSS enumeration algorithm, denoted $\text{getMSSes}(I, K)$, to identify the set MSS_I^K of all MSSes of I that contain at least a single clause from K . Subsequently, we use the procedure $\text{getComponents}(I \setminus K)$ to obtain the decomposition $\mathcal{D}(I \setminus K)$ of $I \setminus K$. Then, we iteratively identify all MSSes of individual components $P \in \mathcal{D}(I \setminus K)$ and store the sets of the MSSes into an auxiliary set $IKMSS_{\text{parts}}$. Once we process all the components, we can form the MSSes of $I \setminus K$ as $\sqcup(IKMSS_{\text{parts}})$ (Proposition 1). Consequently, following Corollary 1, we can obtain MSS_I by combining MSS_I^K and $\sqcup(IKMSS_{\text{parts}})$ (line 8). Finally, to obtain the MSSes of the input set N , we enlarge individual MSSes from MSS_I by the set $N \setminus I$ (Observation 1).

The procedure $\text{findCut}(I)$ is described in the following subsection. To conclude this subsection, we explain how to implement the procedure $\text{getMSSes}(A, B)$ that identifies all MSS of a formula A that contain at least a single clause from a set B . When $A = B$ (i.e., we look for all MSSes of A (line 7)),

we can implement $\text{getMSSes}(A, B)$ by an arbitrary existing MSS enumeration algorithm. In the other case, when $B \subsetneq A$, the situation is more complicated. We are not aware of any existing MSS enumeration tool that would directly allow the user to specify sets A and B and then identify the MSSes of A that contain at least a single clause from B . However, there exist several MSS enumeration algorithms, e.g., [11], [39], that allow the user to specify a subset $B' \subsetneq A$ of *hard clauses* and then identify all MSSes of A that contain *all* clauses in B' . We observe that we can reduce the former task to the latter:

Proposition 3. *Let A and B be formulas such that $B \subsetneq A$. Furthermore, let $A' = A \cup \{c_B\}$ where $c_B = \bigcup_{b \in B} b$. Then $MSS_A^B = \{M \setminus \{c_B\} \mid M \in MSS_{A'}^{\{c_B\}}\}$.*

Proof. \subseteq : If $M \setminus \{c_B\} \in MSS_A^B$, then there exists a clause $c \in M \cap B$, and since $M \setminus \{c_B\}$ is satisfiable and $c \subseteq c_B$, then also M is satisfiable. Now, by contradiction, assume that M is not an MSS of MSS_A , i.e., there exists $d \in A \setminus M$ such that $M \cup \{d\}$ is satisfiable, hence $(M \cup \{d\}) \setminus \{c_B\}$ is satisfiable (which contradicts that $M \setminus \{c_B\} \in MSS_A^B$).

\supseteq : If $M \in MSS_{A'}^{\{c_B\}}$, then there necessarily exists a clause $c \subseteq c_B$ such that $c \in B \cap M$. Furthermore, since M is satisfiable, then $M \setminus \{c_B\}$ is also satisfiable. Now, by contradiction, assume that $M \setminus \{c_B\} \notin MSS_A^B$, i.e., there exists a clause $d \in A \setminus (M \setminus \{c_B\})$ such that $(M \setminus \{c_B\}) \cup \{d\}$ has a model π . Since $c \subseteq c_B$, then π also satisfies $M \cup \{d\}$ which contradicts that $M \in MSS_{A'}^{\{c_B\}}$. \square

Informally, the task of finding MSSes of A that contain at least a single clause from B can be reduced to the task of finding MSSes of A' that contain the hard clause c_B . Namely, in our implementation, we employ the contemporary MSS enumeration tool RIME [11] to carry out $\text{getMSSes}(A, B)$.

Finally, let us note that instead of using an external MSS enumerator to implement $\text{getMSSes}(A, B)$, we could possibly make a recursive call of $\text{processComponent}(\dots)$ (with some minor modifications) to get the MSSes. That is, we could recursively decompose the input formula into smaller and smaller parts. The reason why we do not do that is explained later in Observation 2. Briefly, every *usable* cut requires existence of two disjoint MUSES in the formula, and based on our empirical experience, industrial benchmarks usually do not contain many disjoint MUSES.

C. Finding a Suitable Decomposition Cut

Recall that finding a decomposition cut K for I with $|\mathcal{D}(I)| = 1$ equals to finding a *graph cut* in the decomposition graph $\mathcal{G}(I)$. Hence, we could use any existing algorithm for finding *cuts in a graph* to find K . However, here we need to find a *suitable* decomposition cut. In the following, we will first describe three properties of a suitable decomposition cut: *Minimality*, *Balance*, and *Necessity*. Subsequently, we describe how to find a decomposition cut with such properties.

For the ease of the presentation, assume that we identify a decomposition cut K for I such that $|\mathcal{D}(I \setminus K)| = 2$, and let us

Algorithm 1: DecExact(F)

```

1  $L \leftarrow \text{getKernel}(F)$ 
2  $\mathcal{D}(L) \leftarrow \text{getComponents}(L)$ 
3  $LMSS_{\text{parts}} \leftarrow \emptyset$ 
4 for  $N \in \mathcal{D}(L)$  do
5   if  $\text{isSAT}(N)$  then
6      $LMSS_{\text{parts}} \leftarrow LMSS_{\text{parts}} \cup \{N\}$ 
7   else
8      $LMSS_{\text{parts}} \leftarrow$ 
        $LMSS_{\text{parts}} \cup \{\text{processComponent}(N)\}$ 
9  $MSS_L \leftarrow \sqcup(LMSS_{\text{parts}})$ 
10 return  $\{(F \setminus L) \cup M \mid M \in MSS_L\}$ 

```

Algorithm 2: processComponent(N)

```

1  $I \leftarrow \text{getKernel}(N)$ 
2  $K \leftarrow \text{findCut}(I)$ 
3  $MSS_I^K \leftarrow \text{getMSSes}(I, K)$ 
4  $\mathcal{D}(I \setminus K) \leftarrow \text{getComponents}(I \setminus K)$ 
5  $IKMSS_{\text{parts}} \leftarrow \emptyset$ 
6 for  $P \in \mathcal{D}(I \setminus K)$  do
7    $IKMSS_{\text{parts}} \leftarrow IKMSS_{\text{parts}} \cup \{\text{getMSSes}(P, P)\}$ 
8  $MSS_I \leftarrow MSS_I^K \cup \{M \in \sqcup(IKMSS_{\text{parts}}) \mid \forall M' \in$ 
   $MSS_I^K. M \not\subseteq M'\}$ 
9 return  $\{(N \setminus I) \cup M \mid M \in MSS_I\}$ 

```

by C_1 and C_2 denote the two components of $\mathcal{D}(I \setminus K)$. Hence, in Algorithm 2, it holds that $IKMSS_{\text{parts}} = \{MSS_{C_1}, MSS_{C_2}\}$.

Minimality Recall that in Algorithm 2, line 8, we build the set MSS_I as $MSS_I^K \cup MSS_I^{\bar{K}}$, where $MSS_I^{\bar{K}} = \{M \in \sqcup(\{MSS_{C_1}, MSS_{C_2}\}) \mid \forall M' \in MSS_I^K. M \not\subseteq M'\}$. Note that whereas the set MSS_I^K is computed via an external explicit MSS enumerator, i.e., relatively expensively, the set $MSS_I^{\bar{K}}$ is computed via the decomposition, i.e., relatively cheaply. Consequently, we should attempt to find a decomposition cut K such that $|MSS_I^K|$ is relatively small (compared to $|MSS_I^{\bar{K}}|$). Now, observe that since MSS_I^K contains the MSSes of I that include at least a single clause from K , it holds that the smaller $|K|$ is, the smaller is the maximum possible cardinality of MSS_I^K . Consequently, we should minimize $|K|$.

Balance By Proposition 1, $|\sqcup(\{MSS_{C_1}, MSS_{C_2}\})| = |MSS_{C_1}| \times |MSS_{C_2}|$. Observe that to maximize $|\sqcup(\{MSS_{C_1}, MSS_{C_2}\})|$ while minimizing the number $|MSS_{C_1}| + |MSS_{C_2}|$ of MSSes that are needed to build $\sqcup(\{MSS_{C_1}, MSS_{C_2}\})$, we should ideally find a decomposition cut K such that $|MSS_{C_1}|$ and $|MSS_{C_2}|$ are roughly equal. However, since we do not know in advance what are the MSSes of I , we cannot (cheaply) find a decomposition cut that balances $|MSS_{C_1}|$ and $|MSS_{C_2}|$. Instead, we will just try to find a decomposition cut such that $|C_1|$ and $|C_2|$ are roughly equal (and thus the maximal possible number of MSSes in C_1 and C_2 is roughly equal).

Necessity Note in order to ensure that $|\sqcup(\{MSS_{C_1}, MSS_{C_2}\})| > |MSS_{C_1}| + |MSS_{C_2}|$, it has to hold that $|MSS_{C_1}| > 1$ and $|MSS_{C_2}| > 1$. Furthermore, observe that:

Observation 2. *Given a formula X , it holds that $|MSS_X| > 1$ iff X is unsatisfiable.*

Therefore, for a suitable decomposition cut K , it should hold that both the components C_1 and C_2 are unsatisfiable. All the above three conditions can be straightforwardly generalized for a cut K that yields more than two components.

To find a decomposition cut K with the above three properties, we build a *weighted partial MaxSAT (WPM)* [34] instance and solve it with a MaxSAT solver. In WPM, we are given a tuple $(H, S, w : S \rightarrow \mathbb{N}_+)$, where H is a set of *hard clauses*, S is a set of *soft clauses*, and w is a weight function that assigns to every soft clause a positive weight. A *solution* of the WPM is a valuation π of $\text{Vars}(H \cup S)$ such that π satisfies all hard clauses and maximizes the sum of the weights of satisfied soft clauses.

In our case, we build $H \cup S$ using two sets of Boolean variables: $P = \{p_1, \dots, p_{|I|}\}$ and $Q = \{q_1, \dots, q_{|I|}\}$. Note that every valuation π of $P \cup Q$ corresponds to the subsets $\pi_{P,I}$ and $\pi_{Q,I}$ of I defined as $\pi_{P,I} = \{c_i \in I \mid \pi(p_i) = 1\}$ and $\pi_{Q,I} = \{c_i \in I \mid \pi(q_i) = 1\}$. Furthermore, we write π_K to denote the set $I \setminus (\pi_{P,I} \cup \pi_{Q,I})$. We define a WPM instance $(H, S, w : S \rightarrow \mathbb{N}_+)$ in such a way that for every one of its solutions π it holds that: 1) π_K is a decomposition cut for I , and 2) the clauses in $\pi_{P,I}$ and $\pi_{Q,I}$ are disconnected in $\mathcal{G}(I \setminus \pi_K)$, i.e., they *witness* that π_K is a decomposition cut for I . To ease the presentation, we express H and S below as plain propositional formulas using the standard Boolean connectives of conjunction (\wedge), disjunction (\vee) and implication (\rightarrow). One can use the Tseitin transformation to convert the formulas to sets of clauses.

The formula (hard clauses) H is divided into three sub-formulas, $H = \text{cut} \wedge \text{unsat} \wedge \text{minimal}$. The formula cut (Equation 1) expresses that π_K is a decomposition cut, and encodes this property via two sub-formulas: disj and discn . The formula disj expresses that $\pi_{P,I} \cap \pi_{Q,I} = \emptyset$, whereas discn encodes that there are no two clauses $c_i \in \pi_{P,I}$ and $c_j \in \pi_{Q,I}$ such that there exists a literal $l \in c_i$ with $\neg l \in c_j$ (i.e. that c_i and c_j are connected in $\mathcal{G}(\pi_K)$). Consequently, the clauses from $\pi_{P,I}$ and $\pi_{Q,I}$ do not belong to a same component of $\mathcal{G}(I \setminus \pi_K)$, and hence, by Definition 7, π_K is a decomposition cut for I . Note that cut does not enforce that $|\mathcal{D}(I \setminus \pi_K)| = 2$, i.e., $\pi_{Q,I}$ and/or $\pi_{P,I}$ can be fragmented into multiple components in $\mathcal{D}(I \setminus \pi_K)$.

$$\begin{aligned}
\text{cut} &= \text{disj} \wedge \text{discn}, \text{ where} \\
\text{disj} &= \left(\bigwedge_{c_i \in I} \neg p_i \vee \neg q_i \right), \text{ and} \\
\text{discn} &= \bigwedge_{c_i \in I} \left(\bigwedge_{l \in c_i} \left(\bigwedge_{c_j \in \{c_j \in I \mid \neg l \in c_j\}} \neg p_i \vee \neg q_j \right) \right)
\end{aligned} \tag{1}$$

The formula unsat (Equation 2) attempts to encode that both $\pi_{P,I}$ and $\pi_{Q,I}$ are unsatisfiable, i.e., to fulfil the **Necessity**

condition. To ensure this property, we first attempt to identify a pair of disjoint MUSes of I , denoted by M_1 and M_2 . Equation 2 expresses that $\pi_{P,I} \supseteq M_1$ and $\pi_{Q,I} \supseteq M_2$, and hence $\pi_{P,I}$ and $\pi_{Q,I}$ are unsatisfiable. To find M_1 and M_2 , we enumerate a sequence X_1, X_2, \dots of MUSes of I using an MUS enumerator, and for each MUS X_z we check whether $I \setminus X_z$ is unsatisfiable. If there is such an MUS X_z , we use X_z as M_1 , and we *shrink* $I \setminus X_z$ to the MUS M_2 via a single MUS extractor. We enumerate only a subset of MUSes of I (limited via a user-definable time limit), and hence, we might fail to identify disjoint MUSes even if there are some. Also, it might be the case that I does not contain disjoint MUSes. In such cases, we set `unsat` to 1 (*True*), i.e., we do not ensure satisfaction of the Necessity condition.

$$\text{unsat} = \left(\bigwedge_{c_i \in M_1} p_i \right) \wedge \left(\bigwedge_{c_i \in M_2} q_i \right) \quad (2)$$

The formula `minimal` (Equation 3) targets the **Minimality** condition. We express that for every $c \in \pi_K$ the set $\pi_K \setminus \{c\}$ is not a decomposition cut for I . Note that the minimality is the minimality in the subset inclusion sense, and not in the cardinality sense. The formula states that every clause $c \in \pi_K$ is connected (in $\mathcal{G}(I \setminus \pi_K)$) to a clause in $\pi_{P,I}$ and to a clause in $\pi_{Q,I}$. Consequently, adding c to $\pi_{P,I}$ ($\pi_{Q,I}$), i.e., flipping the assignment $\pi(p_i)$ ($\pi(q_i)$) to 1, would violate the formula `discn`.

$$\begin{aligned} \text{minimal} = & \bigwedge_{c_i \in I} (-p_i \wedge -q_i) \rightarrow \\ & \left(\left(\bigvee_{l \in c_i} \left(\bigvee_{c_j \in \{c_j \in I \mid -l\}} p_i \right) \right) \wedge \left(\bigvee_{l \in c_i} \left(\bigvee_{c_j \in \{c_j \in I \mid -l\}} q_i \right) \right) \right) \end{aligned} \quad (3)$$

Finally, the *soft formula* (clauses) $S = S_1 \wedge S_2$ is divided into two sub-formulas. S_1 (Equation 4) expresses that every $c \in I$ belongs either to $\pi_{P,I}$ or to $\pi_{Q,I}$, i.e., that π_K is empty. The weight assigned to the clauses of S_1 is $3 \cdot |I|$, which ensures that every solution π of the WPM minimizes $|\pi_K|$. Hence, S_1 further strengthens the **Minimality** condition. S_2 (Equation 5) attempts to fulfil the **Balance** condition. In particular, for every $c_i \in I$, we add two soft clauses, p_i and q_i , and with an equal probability (0.5) we randomly set the weights $w(p_i) = 1$ and $w(q_i) = 2$ or vice versa. Intuitively, the formula `disj` enforces that at most one of p_i and q_i holds, and the weights for S_2 attempt to randomly *push* c_i either towards $\pi_{P,I}$ or $\pi_{Q,I}$.

$$S_1 = \bigwedge_{c_i \in I} (p_i \vee q_i) \quad (4)$$

$$S_2 = \left(\bigwedge_{c_i \in I} p_i \right) \wedge \left(\bigwedge_{c_i \in I} q_i \right) \quad (5)$$

Finally, let us note even if by solving the WPM we obtain a decomposition cut K such that $|\sqcup(\{\text{MSS}_C \mid C \in \mathcal{D}(I \setminus K)\})|$ is very large, there is no guarantee that $|\{M \in \sqcup(\{\text{MSS}_C \mid C \in \mathcal{D}(I \setminus K)\}) \mid \forall M' \in \text{MSS}_I^K. M \not\subseteq M'\}| > 0$, i.e., the decomposition might not be helpful. Therefore, the three conditions

on finding a suitable decomposition cut should be seen as heuristics.

D. Towards Partial MSS Enumeration

Few words are in order concerning the practical tractability of running Algorithm 2. As discussed above, the lean kernel I of the input formula N can possibly contain exponentially many MSSes. Hence the MSS enumeration might be beyond the reach of contemporary MSS enumerators (which usually perform around 1-5 SAT solver calls per MSS [8]). To cope with this intractability, we decompose I into several components, and we hope that the MSSes count for the individual components will be relatively small and thus tractable for a contemporary MSS enumerator. However, note that if there is a component which is still intractable for a contemporary enumerator (calls of `getMSSes(...)`, lines 3 and 7), then Algorithm 2 does not terminate in a reasonable time.

Here, we propose a slight modification of Algorithm 2 that deals with such an intractability. When running `getMSSes(A, B)`, we instruct the underlying MSS enumerator to return at most k MSSes of A , where k can be specified by the user of our algorithm. Consequently, if k is reasonably small, the calls of `getMSSes(A, B)` become tractable and Algorithm 2 terminates. After such a modification, the sets MSS_I^K and IKMSSparts might be incomplete, and thus the set MSS_I formed on line 8 can be also incomplete (and hence also the overall set of MSSes returned by Algorithm 1). However, besides the incompleteness, the set MSS_I might not be sound, i.e., it can contain elements that are not MSSes of I .

In particular, we add to MSS_I every $M \in \sqcup(\text{IKMSSparts})$ such that $\forall M' \in \text{MSS}_I^K. M \not\subseteq M'$. Provided that MSS_I^K is complete, passing the check $\forall M' \in \text{MSS}_I^K. M \not\subseteq M'$ ensures that M is an MSS of I (Proposition 2). However, if MSS_I^K is incomplete, then 1) every M that *does not pass* the check *is not* an MSS of I , and 2) every M that *does pass* the check *can be* an MSS of I . Thus, in the case when MSS_I^K is incomplete, we first check for every M whether it satisfies $\forall M' \in \text{MSS}_I^K. M \not\subseteq M'$, and if yes, then we also verify that M is an MSS of I using a SAT solver. Such a verification can be performed using a single call of a SAT solver [14] (we check whether $M \wedge (\bigvee_{c \in I \setminus M} c)$ is satisfiable).

VI. EXPERIMENTAL EVALUATION

We have implemented our novel approach for MSS/MCS enumeration in a python-based tool using the MSS enumerator RIME [11] to implement the procedure `getMSSes`, the library PySAT [27] for maintaining CNF formulas, Minisat [19] (accessed via PySAT) as a SAT solver, and UWMaxSat [45] as a MaxSAT solver. The tool is available at:

<https://github.com/jar-ben/MSSDecomposition>

Here we provide results of our experimental evaluation. We write `DecExact` to denote the *complete* MSS enumeration approach as described in Algorithms 1 and 2, and `DecApprox` to denote the *partial* MSS enumeration version as described in Section V-D. For `DecApprox`, we set the parameter k to 100000, i.e., every call of `getMSSes` identifies at most 100000

MSSes. Moreover, we evaluate three contemporary MSS/MCS enumeration algorithms: MARCO³ [36], FLINT⁴ [44], and RIME⁵ [11]. In all cases, we used the original implementations of the algorithms with their best (default) settings.

As benchmarks, we used a collection of 1491 Boolean CNF formulas that were used in several recent MSS or MUS related studies. Out of the 1491 formulas, 1200 instances⁶ are randomly generated formulas that were first used in [38], and the remaining 291 benchmarks were taken from the MUS track of the SAT Competition 2021⁷. The former benchmarks contain from 100 to 1000 clauses, use from 50 to 996 variables, and have from 2 to at least 10^{22} MSSes (the highest MSS count revealed in our evaluation). The latter benchmarks contain from 70 to 16 million clauses, use from 26 to 4.4 million variables, and have from 2 to at least 10^8 MSSes. We run all experiments on an AMD EPYC 7371 16-Core Processor, 1 TB memory machine running Debian Linux. We used 20 GB memory limit and 3600 seconds (1 hour) time limit per benchmark.

A. Research Questions

We focus on answering the following research questions.

- RQ1:** Our first research question simply asks: *Can our novel MSS enumeration technique complete the enumeration for more benchmarks than the contemporary approaches?*
- RQ2:** As discussed above, the proposed MSS decomposition technique can, in a theory, exponentially reduce the number of MSSes that need to be *explicitly* identified. Hence, our novel approach might be able to handle benchmarks with a very large number of MSSes. Our second RQ is thus: *what is the scalability of the evaluated algorithms w.r.t. the number of MSSes in the individual benchmarks?*
- RQ3:** Finally, we also examine the manifestation of the MSS decomposition in our approach. Our third RQ is: *what is the ratio between the number of explicitly identified MSSes and the total number of identified MSSes for the individual benchmarks.*

B. RQ1: Number of Solved Benchmarks

In Figure 2, we show the number of benchmarks for which individual algorithms finished their computation (within the time limit). In particular, a point with coordinate $[x, y]$ means that there are x benchmarks that were finished by the algorithm in at most y seconds. FLINT, RIME, and MARCO were able to identify all MSSes *only* for 364, 376, and 415 benchmarks, respectively. On the other hand, DecExact identified all MSSes for 788 benchmarks, i.e., solving two times as many benchmarks as its competitors. Finally, DecApprox finished the computation for 1240 benchmarks, however, in many cases, it identified only a portion of all MSSes (due to the limit of

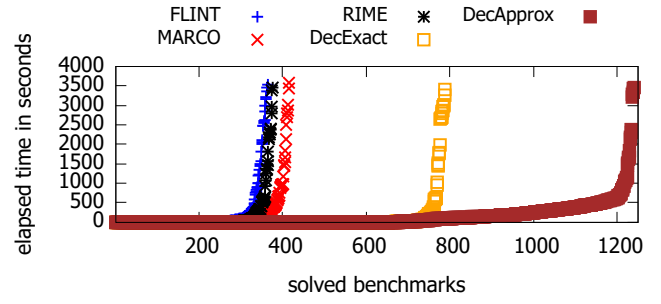


Fig. 2: Number of solved benchmarks.

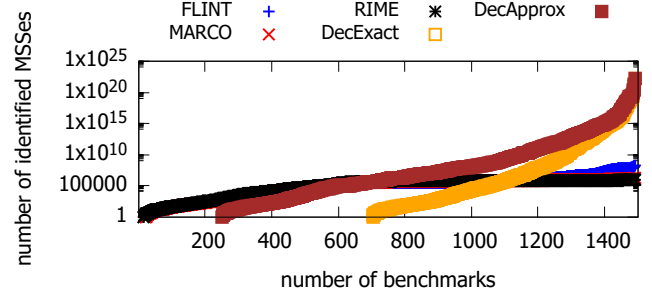


Fig. 3: Scalability w.r.t. the MSS Count

100000 MSS per getMSSes call). In particular, DecApprox identified all MSSes for 742 benchmarks, and at least some MSSes for 498 benchmarks.

We observed that the tractability of the benchmarks highly correlates with their size (number of clauses). In particular, there are only 16 benchmarks that contain more than 10000 clauses and were solved by at least one of the tools (excluding the incomplete tool DecApprox). Moreover, FLINT, RIME, and MARCO scale better w.r.t. this criterion than DecExact since there are 10 benchmarks that contain more than 500000 clauses (but only up to 20000 MSSes) and were solved by these tools. On the other hand, the largest benchmark solved by DecExact contains only 13236 clauses. We further discuss this bottleneck of our approach in Section VII.

C. RQ2: Scalability W.R.T. the MSS Count

In Figure 3, we compare the scalability of the evaluated algorithms w.r.t. the number of MSSes in the input formulas. In particular, a point with coordinates $[x, y]$ denotes that there are x benchmarks where the corresponding algorithm identified fewer than y MSSes. You can see that MARCO and RIME were able to identify at most only around 10^6 MSSes. FLINT performed slightly better w.r.t. this criterion since for some benchmarks, it identified around 10^8 MSSes. On contrary, both DecExact and DecApprox were able to identify up to 10^{22} MSSes in a benchmark. This witnesses that the use of our MSS decomposition techniques allow us to substantially improve the scalability of existing approaches.

³<https://sun.iwu.edu/~mliffito/marco/>

⁴The implementation of FLINT was kindly provided to us by its author, Nina Narodytska.

⁵<https://github.com/jar-ben/rime>

⁶https://github.com/luojie-sklsde/MUS_Random_Benchmarks

⁷<http://www.satcompetition.org/>

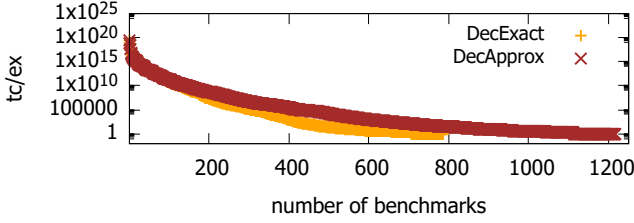


Fig. 4: The ratio between the total number of MSSes and the number of explicitly identified MSSes.

D. RQ3: Number of Explicitly Identified MSSes

Finally, the third research question concerns just our two algorithms, DecExact and DecApprox. Given a formula F , we examine the ratio $\frac{tc}{ex}$, where tc is the total number of identified MSSes of F (i.e., $|MSS_F|$ and an under-approximation of $|MSS_F|$ for DecExact and DecApprox, respectively) and ex is the number of MSSes identified via the calls of `getMSSes`. A point with coordinates $[x, y]$ in Figure 4 denotes that for the corresponding algorithm, there are x benchmarks where the ratio was at least y . Note that we show the ratio only for the 788 and 1240 benchmarks where DecExact and DecApprox finished the computation.

Recall that `getMSSes` is implemented via an *explicit MSS enumerator*, i.e., it identifies individual MSSes one by one using sequence of SAT solver calls, i.e., identification of these MSSes is the most expensive part of our algorithm(s). On the other hand, the tc MSSes are identified extremely cheaply since they are built by just composing the MSSes identified via `getMSSes`. Therefore, the ratio $\frac{tc}{ex}$ actually represents the (maximum possible) speed-up of the MSS enumeration when using DecExact and DecApprox compared to using the *explicit enumerators* FLINT, MARCO, and RIME.

VII. LIMITATIONS AND PRACTICAL APPLICABILITY

Even though our novel approaches, DecExact and DecApprox, solved in our evaluation substantially more benchmarks than contemporary MSS enumerators, the practical efficiency of our approaches remains to be unclear. Here, we discuss two main bottlenecks of our approaches and propose ways how to deal with them.

The first bottleneck of our MSS decomposition technique is its reliance on a MaxSAT solver (which is used to find a suitable cut). The size of the formula cut (Equation 1) depends on the number $|F|$ of clauses in the input formula F . Hence, for larger input formulas F , solving the MaxSAT problem for cut easily becomes practically intractable. A possible way how to deal with this limitation is to use just an *approximate* MaxSAT solver. In particular, recall that our approach for finding a suitable cut via the formula cut is just a heuristic, i.e., there is no guarantee that it will indeed find a suitable cut. Using an approximate MaxSAT solver instead of an exact one might increase the scalability of our approach w.r.t. $|F|$.

The second bottleneck of our MSS decomposition technique was stated in Observation 2. In particular, recall there exists

a usable cut for a given formula F only if F contains a disjoint pair of MUSEs. Based on our empirical experience, there are many applications where the input formula does not contain a disjoint pair of MUSEs and hence our approach cannot be applied. Yet, we have also witnessed many industrial benchmarks where disjoint MUSEs naturally appear (for instance, there is a SAT encoding of the graph coloring problem where disjoint MUSEs correspond to disjoint non-colorable subgraphs). Hence, one might initially check whether the input formula F contains disjoint MUSEs and employ our approach only if it is the case.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we focused on the problem of enumeration of Maximal Satisfiable Subsets of a given CNF formula F . Despite the fact that the enumeration problem was extensively studied in the past decades, contemporary enumerators are still often unable to finish the computation within a reasonable time limit. The problem is that there can be up to exponentially many MSSes w.r.t. $|F|$ and contemporary approaches usually need to perform a sequence of SAT solver queries to obtain individual MSSes. To combat the combinatorial explosion, we proposed a novel MSS enumeration approach that decomposes F into several smaller sub-formulas, identifies their MSSes, and then compose the MSSes of the sub-formulas to form MSSes of the whole F . Our experimental evaluation witnessed that the decomposition in some cases allows us to identify exponentially more MSSes than other contemporary approaches. Yet, as described in Section VII, the class of benchmarks where our approach can be applied is limited.

We see several directions for future work. A crucial ingredient of our algorithm is the ability to identify a suitable decomposition cut K . The approach for finding K we proposed seems to be quite good, i.e., indeed allowing for a decomposition. However, we believe that there might be even better approaches how to find a suitable decomposition cut. Another direction for future work would be to improve upon the partial MSS enumeration approach (DecApprox). In particular, instead of limiting the number of MSSes returned by `getMSSes`, one might try to either interleave or parallelize the computation of MSSes of individual components and compose the MSSes on-the-fly. Finally, since our approach is applicable only to a specific class of benchmarks, it might be worth building a portfolio approach.

ACKNOWLEDGEMENT

This research was funded in part by the Deutsche Forschungsgemeinschaft project 389792660-TRR 248 and by the European Research Council under the Grant Agreement 610150 (ERC Synergy Grant IMPACT).

REFERENCES

- [1] M. Fareed Arif, Carlos Mencía, and João Marques-Silva. Efficient axiom pinpointing with EL2MCS. In *KI*, volume 9324 of *LNCS*, pages 225–233. Springer, 2015.
- [2] Fahiem Bacchus, Jessica Davies, Maria Tsimpoukelli, and George Katsirelos. Relaxation search: A simple way of managing optional clauses. In *AAAI*, pages 835–841. AAAI Press, 2014.

- [3] Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV* (2), volume 9207 of *LNCS*, pages 70–86. Springer, 2015.
- [4] Fahiem Bacchus and George Katsirelos. Finding a collection of MUSes incrementally. In *CPAIOR*, volume 9676 of *LNCS*, pages 35–44. Springer, 2016.
- [5] James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186. Springer, 2005.
- [6] Roberto J Bayardo Jr and Joseph Daniel Pehoushek. Counting models using connected components. In *AAAI/IAAI*, pages 157–162, 2000.
- [7] Rachel Ben-Eliyahu and Rina Dechter. On computing minimal models. In *AAAI*, pages 2–8. AAAI Press / The MIT Press, 1993.
- [8] Jaroslav Bendík. *Minimal Sets over a Monotone Predicate: Enumeration and Counting*. PhD thesis, Masaryk University, 2021.
- [9] Jaroslav Bendík, Nikola Beneš, Ivana Černá, and Jiří Barnat. Tunable online MUS/MSS enumeration. In *FSTTCS*, volume 65 of *LIPIcs*, pages 50:1–50:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
- [10] Jaroslav Bendík and Ivana Černá. Replication-guided enumeration of minimal unsatisfiable subsets. In *CP*, volume 12333 of *LNCS*, pages 37–54. Springer, 2020.
- [11] Jaroslav Bendík and Ivana Černá. Rotation based MSS/MCS enumeration. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 120–137. EasyChair, 2020.
- [12] Jaroslav Bendík, Ivana Černá, and Nikola Beneš. Recursive online enumeration of all minimal unsatisfiable subsets. In *ATVA*, volume 11138 of *LNCS*, pages 143–159. Springer, 2018.
- [13] Jaroslav Bendík and Kuldeep S. Meel. Approximate counting of minimal unsatisfiable subsets. In *CAV* (1), volume 12224 of *LNCS*, pages 439–462. Springer, 2020.
- [14] Jaroslav Bendík and Kuldeep S. Meel. Counting maximal satisfiable subsets. In *AAAI*, pages 3651–3660. AAAI Press, 2021.
- [15] Jaroslav Bendík and Kuldeep S. Meel. Counting minimal unsatisfiable subsets. In *CAV*, pages 313–336. Springer, 2021.
- [16] Philippe Besnard, Éric Grégoire, and Jean-Marie JM Lagniez. On computing maximal subsets of clauses that must be satisfiable with possibly mutually-contradictory assumptive contexts. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
- [17] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.
- [18] Maria J. García de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *PPDP*, pages 32–43. ACM, 2003.
- [19] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
- [20] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62, 2012.
- [21] Eduardo L. Fermé and Sven Ove Hansson. AGM 25 years - twenty-five years of research in belief change. *J. Philos. Log.*, 40(2):295–331, 2011.
- [22] Éric Grégoire, Yacine Izza, and Jean-Marie Lagniez. Boosting mcscs enumeration. In *IJCAI*, pages 1309–1315. ijcai.org, 2018.
- [23] Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure. An experimentally efficient method for (MSS, CoMSS) partitioning. In *AAAI*, pages 2666–2673. AAAI Press, 2014.
- [24] Benjamin Han and Shie-Jue Lee. Deriving minimal conflict sets by cs-trees with mark set in diagnosis from first principles. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 29(2):281–286, 1999.
- [25] Aimin Hou. A theory of measurement in diagnosis from first principles. *AI*, 65(2):281–328, 1994.
- [26] Anthony Hunter and Sébastien Konieczny. Measuring inconsistency through minimal inconsistent sets. In *KR*, pages 358–366. AAAI Press, 2008.
- [27] Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat: A python toolkit for prototyping with SAT oracles. In *SAT*, volume 10929 of *LNCS*, pages 428–437. Springer, 2018.
- [28] Saïd Jabbour, João Marques-Silva, Lakhdar Sais, and Yakoub Salhi. Enumerating prime implicants of propositional formulae in conjunctive normal form. In *JELIA*, volume 8761 of *LNCS*, pages 152–165. Springer, 2014.
- [29] Mikoláš Janota and Joao Marques-Silva. On the query complexity of selecting minimal sets for monotone predicates. *Artificial Intelligence*, 233:73–83, 2016.
- [30] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, pages 437–446. ACM, 2011.
- [31] Hans Kleine Büning and Oliver Kullmann. Minimal unsatisfiability and autarkies. In *Handbook of Satisfiability*, volume 185 of *FAIA*, pages 339–401. IOS Press, 2009.
- [32] Oliver Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.
- [33] Oliver Kullmann and João Marques-Silva. Computing maximal autarkies with few and simple oracle queries. In *SAT*, volume 9340 of *LNCS*, pages 138–155. Springer, 2015.
- [34] Chu Min Li and Felip Manyà. Maxsat, hard and soft constraints. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 613–631. IOS Press, 2009.
- [35] Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In *CPAIOR*, volume 7874 of *LNCS*, pages 160–175. Springer, 2013.
- [36] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2):223–250, 2016.
- [37] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *JAR*, 40(1):1–33, 2008.
- [38] Shaofan Liu and Jie Luo. FMUS2: An efficient algorithm to compute minimal unsatisfiable subsets. In *AISC*, volume 11110 of *LNCS*, pages 104–118. Springer, 2018.
- [39] João Marques-Silva, Federico Heras, Mikoláš Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *IJCAI*, pages 615–622. IJCAI/AAAI, 2013.
- [40] João Marques-Silva, Alexey Ignatiev, António Morgado, Vasco M. Manquinho, and Inês Lynce. Efficient autarkies. In *ECAI*, volume 263 of *FAIA*, pages 603–608. IOS Press, 2014.
- [41] Carlos Mencía, Alessandro Previti, and João Marques-Silva. Literal-based MCS extraction. In *IJCAI*, pages 1973–1979. AAAI Press, 2015.
- [42] Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, 1985.
- [43] Christian Muise, Sheila A McIlraith, J Christopher Beck, and Eric I Hsu. D sharp: fast d-dnnf compilation with sharpsat. In *Canadian Conference on Artificial Intelligence*, pages 356–361. Springer, 2012.
- [44] Nina Narodytska, Nikolaj Bjørner, Maria-Cristina Marinescu, and Mooly Sagiv. Core-guided minimal correction set and core enumeration. In *IJCAI*, pages 1353–1361. ijcai.org, 2018.
- [45] Marek Piotrów. Uwrmaxsat: Efficient solver for maxsat and pseudo-boolean problems. In *ICTAI*, pages 132–136. IEEE, 2020.
- [46] Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *AAAI*. AAAI Press, 2013.
- [47] Alessandro Previti, Carlos Mencía, Matti Järvisalo, and João Marques-Silva. Improving MCS enumeration via caching. In *SAT*, volume 10491 of *LNCS*, pages 184–194. Springer, 2017.
- [48] Alessandro Previti, Carlos Mencía, Matti Järvisalo, and João Marques-Silva. Premise set caching for enumerating minimal correction subsets. In *AAAI*, pages 6633–6640. AAAI Press, 2018.
- [49] Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.
- [50] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In *IJCAI*, pages 1169–1176. ijcai.org, 2019.
- [51] Roni Tzvi Stern, Meir Kalech, Alexander Feldman, and Gregory M. Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI*. AAAI Press, 2012.
- [52] Matthias Thimm. On the evaluation of inconsistency measures. *Measuring Inconsistency in Information*, 73, 2018.
- [53] Marc Thurley. sharpsat-counting models with advanced component caching and implicit bcp. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 424–429. Springer, 2006.
- [54] Sier Wieringa. Understanding, improving and parallelizing MUS finding using model rotation. In *CP*, volume 7514 of *Lecture Notes in Computer Science*, pages 672–687. Springer, 2012.

Designing Samplers is Easy: The Boon of Testers

Priyanka Golia

Indian Institute of Technology Kanpur
National University of Singapore

Mate Soos

National University of Singapore

Sourav Chakraborty

Indian Statistical Institute, Kolkata

Kuldeep S. Meel

National University of Singapore

Abstract—Given a formula φ , the problem of uniform sampling seeks to sample solutions of φ uniformly at random. Uniform sampling is a fundamental problem with a wide variety of applications. The computational intractability of uniform sampling has led to the development of several samplers that heavily rely on heuristics and are not accompanied by theoretical analysis of their distribution. Recently, Chakraborty and Meel (2019) designed the first scalable sampling tester, Barbarik, based on a grey-box sampling technique for testing if the distribution, according to which the given sampler is sampling, is close to the uniform or far from uniform. While the theoretical analysis of Barbarik provides only unconditional soundness guarantees, the empirical evaluation of Barbarik did show its success in determining that some of the off-the-shelf samplers were far from a uniform sampler.

The availability of Barbarik has the potential to spur development of samplers techniques such that developers can design sampling methods that can be accepted by Barbarik even though these samplers may not be amenable to a detailed mathematical analysis. In this paper, we present the realization of this aforementioned promise. Based on the flexibility offered by CryptoMiniSat, we design a sampler CMSGen that promises the achievement of sweet spot of the quality of distributions and runtime performance. In particular, CMSGen achieves significant runtime performance improvement over the existing samplers. We conduct two case studies, and demonstrate that the usage of CMSGen leads to significant runtime improvements in the context of combinatorial testing and functional synthesis.

A salient strength of our work is the simplicity of CMSGen, which stands in contrast to complicated algorithmic schemes developed in the past that fail to attain the desired quality of distributions with practical runtime performance.

I. INTRODUCTION

Given a formula φ , the problem of uniform sampling seeks to sample solutions of φ uniformly at random. Uniform sampling has emerged as an essential technique in the context of constrained-random simulation [33], constraint-based fuzzing [5], [19], [22], configuration testing [13], [23], bug synthesis [36], and the like. For example, in the context of constrained-random simulation, uniform sampling is employed to generate test cases that satisfy the set of constraints encoding domain knowledge from sources such as designers, end-users, and the like.

The widespread applications of uniform sampling have led to several algorithmic proposals over the years with varying theoretical guarantees and empirical scalability. Chakraborty, Meel, and Vardi introduced the first practical almost-uniform sampler, UniGen [11], [12], which has since been improved

to UniGen3 [9], [39]. Recently, Sharma et al. proposed a knowledge compilation-based approach [37], called KUS, that can perform uniform sampling. While UniGen3 and KUS can scale to hundreds of thousands of variables for some problems, their performance still falls short of the desired scale for some real-world instances. The need for scalability has led to the development of several tools that seek to achieve scalability at the cost of theoretical guarantees. The underlying techniques for such tools cover a broad spectrum ranging from adapted BDD-based techniques [26], random seeding of DPLL-based SAT solvers [32], Markov Chain Monte Carlo-based (MCMC) methods [24], [43], interval propagation and belief networks-based methods [14], [20], MaxSAT-based techniques [16].

The lack of guarantees for various samplers leads their designers to illustrate the quality of samples generated via computation of statistics for generated distributions over a small set of benchmarks. Such demonstrations, however, do not generalize to many classes of benchmarks, and it is often the case that subsequent studies tend to demonstrate cases where previously proposed samplers generate distributions far away from uniform. While the theoretical guarantees of uniformity can be viewed as a holy grail, much of the software engineering progress owes to the development of testing methodologies. These methodologies employed both to validate the system and find bugs by the developers themselves in the form of test-driven development (TDD) and to build trust with the end-users; all without requiring the developers to supply a formal proof of correctness.

A major contributing factor to the dramatic improvement in the robustness and scalability of SAT solvers has been the development of the DRAT proof format and associated proof checker drat-trim [44]. The availability of drat-trim allows SAT solver developers to find bugs that would be hard to discover owing to the complex architecture of state-of-the-art SAT solvers. While the problem of checking whether a given formula is UNSAT is *merely* Co-NP, the problem of testing whether a sampler is a uniform requires $\Omega(2^n)$ samples given black-box access to the sampler [3], [8], where n is the number of variables.

Recently, Chakraborty and Meel proposed the first scalable sampler test framework, Barbarik [8]. This framework distinguishes whether the distribution generated by the given sampler is ε -close to uniform (Accept) or η -far from uniform (Reject), while the number of samples required depends only

on ε and η , and is independent of n . The core idea of the Barbarik is to reduce testing of uniformity over the entire solution space of φ to the testing of uniformity over solutions space of another formula, $\hat{\varphi}$ constructed over two randomly chosen solutions of φ (observe that $\hat{\varphi} \rightarrow \varphi$). The subroutine to construct $\hat{\varphi}$ is called Kernel. The analysis of Barbarik states that if Barbarik Rejects a sampler, the distribution generated by sampler is indeed (probabilistically) far from uniform, but if Barbarik Accepts a sampler, the sampler’s distribution is close to uniform under the assumption of *non-adversality* with respect to Kernel. Informally, the *non-adversality* assumption with respect to Kernel dictates that given φ , the conditional distribution of the sampler over the solutions of $\hat{\varphi}$ is same as the distribution of the sampler with $\hat{\varphi}$ as input. Note that this allows some samplers to behave in an *adversarial* manner, i.e., such samplers may not generate uniform distribution over φ , however may generate uniform distributions for $\hat{\varphi}$. In such a case, causing Barbarik will return Accept for such samplers. At this point, it is worth remarking that given the strong lower bounds on black-box testing, the usage of such an assumption is a *practical* necessity.

Empirically, Barbarik was able to return Reject for all the state of the art samplers without rigorous mathematical analysis certifying (almost)-uniformity of the generated distributions. In particular, Barbarik was demonstrated to Accept UniGen3 while rejecting the state of the art samplers STS [18] and QuickSampler [16]. It is worth noting that the three samplers, UniGen3, QuickSampler, and STS, were found to be statistically indistinguishable by the usage of simple metrics such as KL-divergence [27] after a small number of samples.

The availability of Barbarik, however, has potential to allow development of samplers, whose algorithmic frameworks may not be amenable to mathematical analysis but can be accepted by Barbarik. The primary contribution of this paper is realization of the promise of Barbarik via development of a new state of the art sampler, CMSGen. In particular, we make following contributions:

A. CMSGen: A State of the Art Sampler

- 1) We design a new sampler, CMSGen, by modifying the existing state-of-the-art Conflict-Driven Clause Learning (CDCL) SAT solver CryptoMiniSat¹ [41].
- 2) Since understanding the behavior of CDCL itself is an open problem, we can not provide an unconditional analysis of the distribution produced by CMSGen. We rely on the availability of Barbarik, and observe that surprisingly, Barbarik returns Accept for all the benchmarks. Barbarik’s failure to Reject CMSGen stands in sharp contrast to its ability to Reject other samplers without guarantees, such as QuickSampler. Furthermore, we perform empirical comparisons of runtime performance via-a-vis UniGen3, the state-of-the-art sampler with theoretical guarantees. We observe that CMSGen

significantly improves upon UniGen3 in terms of runtime performance.

B. Case Studies: Combinatorial Testing and Functional Synthesis

- 3) At this point, one may wonder whether there are practical applications of CMSGen. We next focus on applications that are beyond the reach of UniGen3, and for such cases, one has to rely on the heuristics-based samplers. In particular, we perform two case studies: (1) combinatorial testing, and (2) functional synthesis; two problems with a long history of sustained interest in formal methods and software engineering community. For both the case studies, we observe that the usage of CMSGen leads to significant performance improvements in comparison to usage of other competing samplers UniGen3 and QuickSampler.

It is worth remarking that a salient strength of CMSGen is the simplicity of its design. We find it exciting that a sampler with such a simple design could outperform sophisticated state of the art samplers. Based on our empirical analysis, one would remark that CMSGen aims to achieve the sweet spot of scalability and uniformity. In particular, CMSGen is significantly more scalable than samplers with guarantees and, at the time, achieves distributions of higher quality than samplers without guarantees. The runtime performance combined with the quality of distribution as certified by Barbarik makes CMSGen the ideal choice for applications such as combinatorial testing and functional synthesis where scalability and quality of distribution are equally crucial.

The rest of the paper is organized as follows: In Section II, we present the formal definitions and also present a brief description of the sampler verifier Barbarik. In Section III we present the new sampler CMSGen and in Section IV we present the evaluation of CMSGen both by comparing its runtime performance with other samplers and also its performance against Barbarik. Then in Section V we demonstrate the usefulness of CMSGen with two case studies on problems of fundamental importance to formal methods community: functional synthesis and combinatorial testing. Finally, we conclude in Section VI.

II. NOTATION AND BACKGROUND

A literal is a Boolean variable or its negation. Let φ be a Boolean formula in conjunctive normal form (CNF), and let X be the set of variables appearing in φ . The set X is called the *support* of φ , denoted by $Supp(\varphi)$. Given an array \mathbf{a} , $\mathbf{a}[i : j]$ represents the sub-array consists of all the elements of \mathbf{a} between indices i and j . A *satisfying assignment* or *witness*, denoted by σ , is an assignment of truth values to variables in its support such that φ evaluates to true. A satisfying assignment is also represented as a set of literals. For $S \subseteq Supp(\varphi)$, we use $\sigma_{\downarrow S}$ to indicate the projection of σ over the set of variables S . We denote the set of all witnesses of φ as $sol(\varphi)$. For notational convenience, whenever the formula

¹Available at <https://github.com/msoos/cryptominisat>

φ and/or the set $S \subseteq \text{Supp}(\varphi)$ is clear from the context, we omit mentioning them.

A. Samplers

Definition 1: Given a Boolean formula φ , a *CNF-sampler* (or simply *sampler*) \mathcal{G} of φ is a probabilistic algorithm that generates a random element in $\text{sol}(\varphi)$. We will assume that a sampler takes as input a CNF-formula φ , a set $S \subseteq \text{Supp}(\varphi)$ and an integer k . It generates k elements $\sigma_1, \dots, \sigma_k$ from $\text{sol}(\varphi)$ and outputs $\sigma_{1 \downarrow S}, \dots, \sigma_{k \downarrow S}$. When the integer k and the set $S \subseteq \text{Supp}(\varphi)$ is clear from the context (or is not important) we will drop them and use $\mathcal{G}(\varphi)$ or $\mathcal{G}(\varphi, S)$ to denote the sampler.

We use $p_{\mathcal{G}}(\varphi, \sigma)$ (or $p_{\mathcal{G}}(\varphi, \sigma, S)$) to denote the probability that $\mathcal{G}(\varphi, \cdot, \cdot)$ (or $\mathcal{G}(\varphi, S, \cdot)$) generates σ (or $\sigma_{\downarrow S}$). And, we use $\mathcal{D}_{\mathcal{G}(\varphi)}$ (and $\mathcal{D}_{\mathcal{G}(\varphi, S)}$) to denote the distribution induced by \mathcal{G} over the set $\text{sol}(\varphi)$ (and $\text{sol}(\varphi)_{\downarrow S}$). For a set $T \subseteq \text{sol}(\varphi)$, we use $\mathcal{D}_{\mathcal{G}(\varphi)} \downarrow T$ to denote the distribution $\mathcal{D}_{\mathcal{G}(\varphi)}$ conditioned on set T .

Definition 2: Given a Boolean formula φ , A *uniform sampler* $\mathcal{G}^u(\varphi)$ is a sampler that given φ guarantees

$$\forall y \in \text{sol}(\varphi), \Pr[\mathcal{G}^u(\varphi) = y] = 1/|\text{sol}(\varphi)|, \quad (1)$$

Definition 3: Given a Boolean formula φ and tolerance parameter ε , $\mathcal{G}^{AAU}(\varphi, \varepsilon)$ is an additive *almost-uniform generator* (AAU) if the following holds:

$$\forall y \in \text{sol}(\varphi), \frac{1 - \varepsilon}{|\text{sol}(\varphi)|} \leq \Pr[\mathcal{G}^{AAU}(\varphi, \varepsilon) = y] \leq \frac{1 + \varepsilon}{|\text{sol}(\varphi)|} \quad (2)$$

A sampler is allowed to occasionally “fail” in the sense that no element may be returned even if $\text{sol}(\varphi)$ is non-empty. The failure probability for such generators must be bounded by a constant strictly less than 1.

Definition 4: Given a Boolean formula φ and an intolerance parameter η an generator $\mathcal{G}(\varphi, \cdot)$ is η -far from uniform generator if the ℓ_1 -distance (or, twice the variation distance) of $\mathcal{D}_{\mathcal{G}(\varphi)}$ from uniform is at least η . That is,

$$\sum_{x \in \text{sol}(\varphi)} \left| p_{\mathcal{G}(\varphi, x)} - \frac{1}{|\text{sol}(\varphi)|} \right| \geq \eta$$

B. Sampler Tester

Given a sampler \mathcal{G} , one would like to test if the sampler is indeed correct. Or in other words, one would like to test the following:

- 1) Does the sampler always output a satisfying assignment?
- 2) On any CNF-formula φ , is $\mathcal{G}(\varphi)$ an additive almost-uniform generator?

While the first point is very easy to test, testing the second point is quite challenging. Standard verification techniques or black box sampling techniques would need exponential time/samples and thus are very inefficient.

Chakraborty and Meel [8] designed the tester Barbarik that would accept if the sampler is an additive almost-uniform generator on any input and reject if the sampler is far from a uniform generator on some input under certain assumptions

discussed below. The idea of Barbarik comes from the world of property testing, where the sample complexity for testing whether a distribution is a uniform is studied. While it was known from classical sample complexity [3] that an exponential number of samples are required to distinguish a uniform distribution from a distribution that is η -from uniform, in [7] it was observed that if given access to conditional samples only a constant number of samples suffice. Conditional samples from a distribution \mathcal{D} means for a subset T of the domain Ω , drawing samples from the conditional distribution $\mathcal{D}|_T$. The algorithm for checking whether a given distribution \mathcal{D} over domain Ω is uniform or η -far from uniform, consists of following steps:

- 1) Draw one sample σ_1 according to the distribution \mathcal{D} .
- 2) Draw one sample σ_2 according to the uniform distribution over Ω .
- 3) Check if the distribution $\mathcal{D}|_T$ is uniform or “far”-from uniform, where $T = \{\sigma_1, \sigma_2\}$.

The last point of the above algorithm can be performed using only a constant number of conditional samples. It can also be shown that the above algorithm, with non-trivial probability, will Accept if \mathcal{D} is uniform and Reject if \mathcal{D} is η -far from uniform, by repeating this algorithm a certain number of times, one can boost the success probability.

While the algorithm is theoretically interesting, applying it to design a sampler test framework required several hurdles to cross. Firstly, for Step 2 of the algorithm, one needs to run a uniform sampler. This is not too much of a hurdle as one can use a non-efficient uniform sampler, since the sampler tester is only to be used a few times to certify if a sampler is good.

The second problem is that the algorithms, as such, could only distinguish between a uniform distribution, and a distribution “far” from a uniform distribution, while a sample tester should also Accept samplers that are “close” to uniform samplers (and not necessarily just uniform samplers).

Finally, the main concern was how to obtain conditional samples. In [8] this was achieved by constructing a new formula $\hat{\varphi}$ on a larger number of variables such that the satisfying assignments of $\hat{\varphi}$ restricted to the original set of variables is either σ_1 and σ_2 . In fact if $S = \text{Supp}(\varphi)$, then

$$\Pr_{\sigma \sim (\text{sol}(\hat{\varphi}))}[\sigma_{\downarrow S} = \sigma_1] = \Pr_{\sigma \sim (\text{sol}(\hat{\varphi}))}[\sigma_{\downarrow S} = \sigma_2] = \frac{1}{2}$$

where $\mathcal{U}(\text{sol}(\hat{\varphi}))$ denotes uniform distribution over $\text{sol}(\hat{\varphi})$ The new formula $\hat{\varphi}$ is obtained from φ by using a subroutine Kernel that uses the chain formula technique from [10].

The goal of the construction of $\hat{\varphi}$ is such that the following two conditions are satisfied:

- 1) If the sampler $\mathcal{G}(\varphi)$ was ϵ -additive almost-uniform generator then the distribution $\mathcal{D}_{\mathcal{G}(\hat{\varphi}, S)}$ is “close” to the uniform distribution on the set $\{\sigma_1, \sigma_2\}$.
- 2) If the sampler $\mathcal{G}(\varphi)$ was η -far from the uniform sampler in the ℓ_1 distance then the distribution $\mathcal{D}_{\mathcal{G}(\hat{\varphi}, S)}$ is “far” from the uniform distribution on the set $\{\sigma_1, \sigma_2\}$.

Now, if the sampler \mathcal{G} is additive almost-uniform generator on any input φ the first condition would be satisfied. But

for the second condition to hold some more assumptions are necessary. This assumption is called the *non-adversarial assumption* in [8].

Definition 5: The **non-adversarial sampler assumption** states that if $(\hat{\varphi}, \hat{S})$ is the output obtained from $\text{Kernel}(\varphi, S, \sigma_1, \sigma_2, N)$ then

- $S \subseteq \hat{S}$
- the output of $\mathcal{G}(\hat{\varphi}, S, N)$ is N independent samples from the conditional distribution $\mathcal{D}_{\mathcal{G}(\varphi, S)|_T}$, where $T = \{\sigma_1, \sigma_2\}$.

Thus Barbarik has the following guarantees.

Theorem 1: Given a sampler \mathcal{G} , tolerance parameter ϵ , intolerance parameter η and correctness parameter δ ,

- 1) If for all φ , $\mathcal{G}(\varphi)$ is ϵ -additive almost-uniform generator then Barbarik will Accept with probability $(1 - \delta)$.
- 2) If for some φ the sampler $\mathcal{G}(\varphi)$ is η -far from the uniform sampler in the ℓ_1 distance and the sampler satisfies the **non-adversarial sampler assumption** then Barbarik will Reject with probability $(1 - \delta)$.

For the implementation, the subroutine Kernel is designed in an attempt to fool the sampler into satisfying the *non-adversarial assumption*. The idea being that the new CNF-formula $\hat{\varphi}$ would be “hard” to distinguish from φ and hence one would expect

$$p_{\mathcal{G}}(\hat{\varphi}, \sigma_1, S) = \frac{p_{\mathcal{G}}(\varphi, \sigma_1, S)}{p_{\mathcal{G}}(\varphi, \sigma_1, S) + p_{\mathcal{G}}(\varphi, \sigma_2, S)}$$

C. Experimental Setup

All our experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2690 v3 CPU with 24 cores and 96GB of RAM, with a memory limit set to 4GB per core.

III. FROM CryptoMiniSat TO CMSGen

The naive technique to design a sampler is to pick a random assignment of variables, check if it satisfies the CNF formula, and, if so, output the assignment as a witness; otherwise, pick another random assignment and start over again. Using an unbiased random coin for the assignments, it is trivial to see that the technique leads to a uniform sampler. Such a proposal is, however, very inefficient as with a very high probability, every picked assignment is likely not to satisfy the formula.

One way to make such a sampler into an efficient one is by not starting with a complete assignment but build the partial assignment up the variable by variable, set all variables that are implied by the current partial assignment, and if a partial assignment is incorrect, record and learn from the failure. The concept of learning from failure is captured by the well-known conflict-driven clause-learning (CDCL) framework used by most state-of-the-art SAT solvers. We refer the reader to Chapter 4 of [4] for a detailed exposition on CDCL. We present an extension that seeks to combine the CDCL framework with randomization in the choice of partial assignments in Algorithm 1, called UniformLikeWitness. UniformLikeWitness is essentially a randomized variation on the CDCL framework,

with a randomized heuristic for what variable to assign next, a randomized heuristic for variable polarities, and without restarts.

Algorithm 1 UniformLikeWitness(F, seed)

```

1: while true do
2:    $x \leftarrow$  pick an unassigned variable at random
3:    $\text{assigns}[x] \leftarrow$  pick 0 or 1 uniformly at random
4:    $\text{conflict}, \text{assigns} \leftarrow$  perform unit propagation
5:   if assigns is full then return assigns
6:   if conflict is found then
7:      $\text{back\_lvl}, \text{conf\_clause} \leftarrow$  Conflict-Analysis [32]
8:     if conf_clause is empty then return NULL
9:     Update assigns as per back_lvl
10:     $F \leftarrow F \cup \text{conf\_clause}$ 
11:    if F is too large then
12:      Perform Learnt Clause Deletion [2]
```

One major problem of the above process is that the sampler, just like an SAT solver, may get stuck in the corner of the space where there are no satisfying solutions. Once stuck, it can take much time to record the relevant conflicts before it can escape this part of the search space. In modern SAT solvers, such an escaping is enabled by performing restarts. The idea of a restart is to stop the current search procedure, keeping conflict clause and heuristic data such as polarities, variable activities in the line, but otherwise starting afresh, resetting the assignment state. The idea of performing a restart is to reduce the chance of getting stuck in a non-fruitful part of the search space. Performing regular, frequent restarts is a core component of all state-of-the-art SAT solvers.

CMSGen² is a sampler that exploits the flexibility CryptoMiniSat to implement the behaviour of UniformLikeWitness. We use the restart policy based on the number of conflicts, i.e., we perform a restart after the pre-determined number of conflicts, which is set to 100. Hence, the final set of options passed to CryptoMiniSat turn off the features unrelated to CDCL (such as bounded variable elimination [17], local search [6], or symmetry breaking [15]), and set the options that control variable branching and polarity picking to match Algorithm 1, and set the restart interval to 100. Note that while it is possible that other CDCL SAT solvers could be adjusted to generate samples as well as CMSGen, the newer and more performant glucose-based SAT solvers [2] tend to be highly tuned without any command-line options to change or turn off heuristics.

We would like to emphasize that we do not claim that CMSGen is expected to generate uniform distributions over all the formulas as it is possible to construct worst case scenarios where CMSGen would not work well. At this point, it is worthwhile to note that, to the best of our knowledge, the current techniques are insufficient to analyse the kind of formulas for which UniformLikeWitness would behave like

²CMSGen is available at <https://github.com/meelgroup/cmsgen>

a uniform sampler given their limitations to understand the behaviour of CDCL itself. Traditionally, the proposal of a new sampler is accompanied by theoretical analysis, but in our case, we seek to rely on the testing framework of Barbarik to analyse the behavior of CMSGen.

IV. THE POWER OF CMSGen

As mentioned above, instead of taking a conventional route focusing on the theoretical analysis of CMSGen, we seek to employ Barbarik to test whether CMSGen is a uniform sampler or not. In addition, we seek to understand the runtime behavior of CMSGen in comparison to other state of the art techniques. We conducted an extensive evaluation of diverse public domain benchmarks employed in prior studies [8], [40].

A comment on the choice of benchmarks for the two studies: For the first study, we selected the same 50 benchmarks that were employed in the evaluation of Barbarik so as to situate the results with prior context [8]. Since Barbarik needs to sample up to 1.835×10^3 solutions, the choice of benchmarks in [8] was restricted to instances for which generating samples is easy. On the other hand, these benchmarks are not meaningful for runtime performance comparison as all the tools finish on them very quickly. To this end, we relied on 70 benchmarks employed in prior sampling studies [38], [39] for runtime performance comparison.

The objective of our evaluation was two-fold:

- RQ1** To understand the behavior of Barbarik in terms of the frequency of outputs Accept and Reject with CMSGen as sampler under test.
- RQ2** To evaluate the runtime performance of CMSGen vis-a-vis the state of the art sampler with guarantees of almost-uniformity, UniGen3.

In summary, we observe that Barbarik, somewhat surprisingly, returns Accept for CMSGen and UniGen3 on all the 50 instances while returning Reject for all the 50 instances for QuickSampler [16], and for 36 instances for STS [18], the state-of-the-art samplers without guarantees. At the same time, comparison in terms of runtime for over 70 benchmarks arising from different application domains, we observe that CMSGen is significantly faster than UniGen3.

A. Testing CMSGen with Barbarik

For experimentation evaluations with Barbarik, we used the default parameters suggested by the authors: In particular, we set tolerance parameter ϵ , intolerance parameter η , and confidence δ to be 0.3, 1.8, and 0.1 respectively. For our chosen parameters, the number of samples required to return Accept for a given sampler under test is 1.836×10^3 , and to maintain consistency with evaluation setup of Barbarik, we selected benchmarks (50 in total) that were used in evaluation of QuickSampler and UniGen3 for which Barbarik terminates within 2 hours. To test uniformity of distributions generated by CMSGen and other samplers, we employed Barbarik augmented with SPUR [1] as the underlying uniform sampler. We present the results of our evaluation in Table I, where the four columns present results corresponding to QuickSampler,

TABLE I: Analysis of different samplers with Barbarik over 50 benchmarks. Parameters $\epsilon : 0.3, \eta : 1.8, \delta : 0.1$, and samples required to return Accept 1.836×10^3 .

	QuickSampler	STS	UniGen3	CMSGen
Accept	0	14	50	50
Reject	50	36	0	0

STS, UniGen3, and CMSGen respectively. The first and second rows indicate the number of instances for which Barbarik returned Accept and Reject respectively. We first note that while Barbarik returned Reject for QuickSampler and STS for the 50 and 36 instances respectively, it returned Accept for both CMSGen and UniGen3 for all the instances. It is worth highlighting that UniGen3 provides guarantees of almost-uniformity.

Remark 1: At this point, it is worth highlighting that we arrived at the choice of parameters of CMSGen, such as when to restart via an iterative process where we would run Barbarik for the given choice of parameters and change them based on the number of instances rejected by Barbarik. In this context, it is rather encouraging that such an iterative process led us to design a sampler, CMSGen, which could not be distinguished from UniGen3 by Barbarik while significantly improving upon UniGen3 in terms of runtime performance. This highlights the advantages of a TDD-style design approach.

B. Runtime Comparison

Upon observing that Barbarik returns Accept for all the 50 instances for both CMSGen and UniGen3, a natural question is whether the runtime performance of CMSGen is comparable to that of UniGen3. To this end, we compared CMSGen with UniGen3, STS and QuickSampler on 70 benchmark instances arising from a wide range of application areas of uniform sampling, such as probabilistic reasoning, Bounded Model Checking [37], [40]; these instances had been previously employed in empirical studies focused on the comparison of sampling techniques [38], [39].

For each of the instances, we invoke each of the sampler to generate 1000 solutions within a timeout of 7200 seconds. Figure 1 shows the cactus plot for CMSGen, UniGen3, STS and QuickSampler. We present the number of benchmarks on the x-axis and the time taken on the y-axis. A point (x, y) implies that for a x benchmark, the sampler took less than or equal to y seconds to generate 1000 solutions of x . With a timeout of 7200 seconds, UniGen3 and CMSGen were able to sample 1000 solutions of 51 and 52 benchmarks respectively, whereas STS and QuickSampler generated samples for *merely* 37 and 33 instances respectively. Figure 1 clearly shows that for all the benchmarks that were sampled 1000 times by both UniGen3 and CMSGen, CMSGen outperformed UniGen3 with a geometric speedup of over $420\times$.

Table II represent the runtime performance for QuickSampler, STS, UniGen3 and CMSGen for a representative set of 20 benchmarks. As shown in Table II, there are instances (18 out of 70) for which UniGen3 is able to samples 1000 solutions

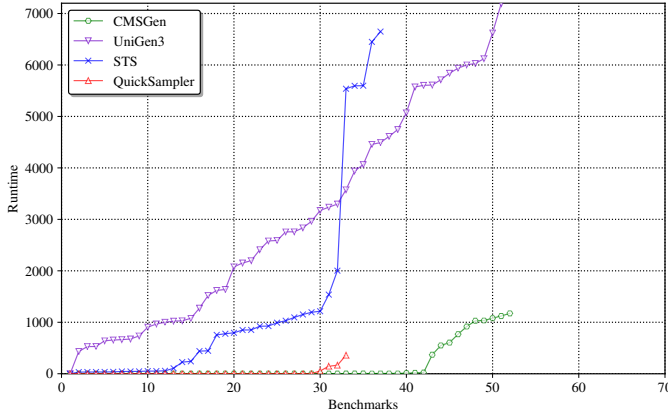


Fig. 1: Cactus plot showing runtime performance of UniGen3, STS, QuickSampler and CMSGen to generate 1000 samples. Timeout: 7200s

TABLE II: Runtime performance of different samplers to generate 1000 solutions for a representative set of benchmarks. Timeout (TO): 7200s.

Benchmarks	QuickSampler	STS	UniGen3	CMSGen
or-70-5-5-UC-20	0.07	36.39	3173.45	0.29
or-60-20-10-UC-20	0.07	43.53	4065.0	0.31
or-100-20-8-UC-40	0.09	51.25	2152.01	0.4
tire-2	1.09	226.01	TO	0.48
or-50-10-7-UC-10	0.06	33.28	2196.98	0.95
b12_2_linear	TO	1214.73	1520.01	2.08
b14_2_linear	TO	926.18	1220.01	2.18
squaring41	TO	5595.0	6002.0	2.8
squaring60	TO	TO	TO	4.52
s15850a_15_7	359.37	TO	675.33	5.58
b12_even2_linear	TO	TO	TO	15.52
isolateRightmost	TO	TO	432.73	21.66
modexp8-5-4	TO	TO	6122.0	550.9
modexp8-6-4	TO	TO	TO	1034.27
modexp8-6-3	TO	TO	6624.0	1079.82
modexp8-6-8	TO	TO	TO	1173.64
prod-20	TO	TO	1274.42	TO
04B-1	TO	5598.0	2410.61	TO
06B-1	TO	6449.0	2835.64	TO
hash-10-7	TO	TO	5610.0	TO

whereas CMSGen could not sample. Similarly, there are 19 instances for which CMSGen is able to sample solutions but UniGen3 could not.

V. CASE STUDIES: FUNCTIONAL SYNTHESIS AND COMBINATORIAL TESTING

Having established that the quality of distribution generated by CMSGen is significantly better than QuickSampler, one wonders about the practical utility of CMSGen. The significant gap between runtime performance of CMSGen and UniGen3 argues for the usage of CMSGen in applications where the quality and runtime performance of samplers are key determining factors.

To this end, we focused on two such application domains: Combinatorial testing and Boolean functional synthesis. The

state of the art techniques for each of these domains crucially rely on underlying uniform samplers; in fact the sampler QuickSampler was proposed in the context of combinatorial testing. For each of these case studies, we substitute the three samplers CMSGen, QuickSampler, and UniGen in the state of the art techniques, and analyse their performance on the resulting tool.

A. Combinatorial Testing

Combinatorial testing is considered as a powerful paradigm for testing configurable software. The primary task of a test generator is the generation of a test suite that maximizes t -wise coverage. t -wise coverage is measured as the fraction of feature combinations appearing in the test set out of the possible valid feature combinations. Uniform sampling is considered one of the promising approach to have higher t -wise coverage [31], [34], [35]. Therefore, a natural question is whether CMSGen can serve as a good test suite generator. To this end, we performed a comparative study of CMSGen vis-a-vis UniGen3, STS and QuickSampler on the set of 110 publicly available benchmarks that have been employed in prior comparative studies of sampling techniques in the context of combinatorial testing [25], [29], [35]³. It is worth emphasizing that UniGen3, STS and QuickSampler are viewed as a state of the art test suite generation techniques in the presence of constraints as witnessed by empirical study by Plazar et al. [35].

In our comparative study of sampling techniques of their efficiency in achieving higher t -wise coverage, we focus on the case of $t = 2$ as is standard in the most empirical studies in combinatorial testing. To this end, for every benchmark, we generate 1000 samples from each of the four samplers: CMSGen, STS, QuickSampler, and UniGen3. We used a timeout of 3600 seconds for sampling. UniGen3 is, however, unable to sample for all but six benchmarks. Therefore, we exclude UniGen3 from further analysis.

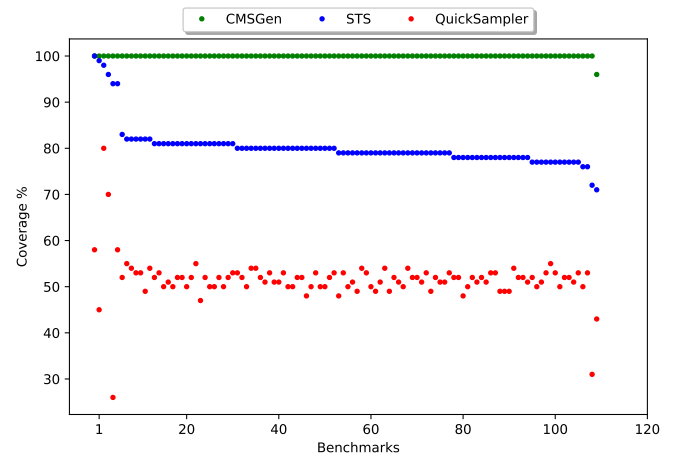


Fig. 2: Plot to show 2-wise coverage% for 110 benchmarks with 1000 samples. Sampling timeout: 3600s.

³Benchmarks are available at <https://zenodo.org/record/4022395>

TABLE III: Analysis for 2-wise coverage with QuickSampler, STS, and CMSGen.

Benchmark	# Feature Combinations	QuickSampler		STS		CMSGen	
		# combination observed	Coverage	# combination observed	Coverage	# combination observed	Coverage
busybox_1_28_0	1965023	513565	0.26	1849127	0.94	1964962	1.0
ecos-icse11	2910229	898195	0.31	2104721	0.72	2910078	1.0
financial	917150	392381	0.43	649279	0.71	876356	0.96
buildroot	621270	278254	0.45	613184	0.99	621252	1.0
vads	2896324	1360422	0.47	2348489	0.81	2895931	1.0
mpc50	2719748	1354164	0.5	2078077	0.76	2719508	1.0
XSEngine	2974825	1498239	0.5	2383688	0.8	2974448	1.0
ocelot	2986129	1519047	0.51	2344079	0.78	2986002	1.0
dreamcast	2908040	1523501	0.52	2253050	0.77	2907734	1.0
refidt334	3022264	1557688	0.52	2356854	0.78	3021978	1.0
integrator_arm7	2957100	1566676	0.53	2275664	0.77	2956958	1.0
pc_i82559	2977432	1582402	0.53	2384286	0.8	2977280	1.0
p2106	2887921	1544728	0.53	2282100	0.79	2887653	1.0
skmb91302	2755776	1451902	0.53	2133950	0.77	2755538	1.0
cma28x	2694432	1419911	0.53	2156230	0.8	2694257	1.0
ipaq	2897450	1576622	0.54	2305020	0.8	2897153	1.0
axtls	16212	9381	0.58	15264	0.94	16212	1.0
uClinux	3013528	1751212	0.58	3013456	1.0	3013528	1.0
toybox	256494	180332	0.7	246484	0.96	256494	1.0
FM-3.6.1-refined	3151	2518	0.8	3075	0.98	3151.0	1.0

Figure 2 shows the experimental results with STS, QuickSampler and CMSGen. We present the number of benchmarks on the x-axis and pair-wise coverage % on the y-axis. A point (x, y) implies that x benchmarks had $y\%$ pair-wise coverage. Benchmarks are ordered in the decreasing order of coverage achieved with the samples produced by STS. Figure 2 shows that almost all the benchmarks had nearly 100% pair-coverage with samples generated by CMSGen, on the other hand, the average pair-wise coverage with samples from QuickSampler and STS is 51.5% and 80.15%. One should view the significant performance improvement due to CMSGen over QuickSampler in light of the fact that the primary motivation behind the proposal of QuickSampler was to achieve higher coverage.

Table III represents the analysis for 2-wise coverage with CMSGen, STS and QuickSampler for representative 20 benchmarks. In table III, Column 2 present the possible valid feature combinations. Column 3, 5 and 7 present the feature combinations appearing in test set generated by QuickSampler, STS and CMSGen respectively, and Column 4,6 and 8 is for the corresponding coverage. As shown in Table III, the test set generated with CMSGen is able to cover *all* possible feature combinations for all the benchmarks.

B. Boolean Functional Synthesis

Given a formula $\exists Y F(X, Y)$, the problem of Boolean functional synthesis seeks to compute a function φ such that $\exists Y F(X, Y) \equiv F(X, \varphi(X))$. Typically, we view F as a specification and φ as the function that implements the specification φ . Boolean functional synthesis is a fundamental problem with wide variety of applications ranging from logic synthesis [28], cryptography [30], program synthesis [42], and the like. For example, Boolean functional synthesis encompasses program synthesis, where φ can be viewed as the desired program.

Consequently, there has been a sustained interest in the design of efficient algorithmic techniques for Boolean functional synthesis. The current state of the art approach, Manthan, was proposed recently and builds on the advances in sampling techniques, automated reasoning, and machine learning [21]. Manthan was demonstrated to solve 70 more benchmarks than the next best technique. In this regard, Manthan serves as a good test-bed to compare different sampling techniques.

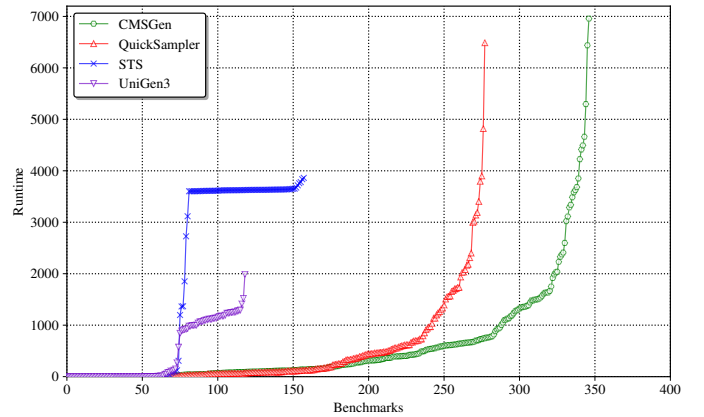


Fig. 3: Cactus plot to show the impact of different sampler on functional synthesis engine, Manthan. Timeout: 7200s

We sought to compare CMSGen vis-a-vis UniGen3, STS and QuickSampler in their impact on the performance of Manthan. We set the timeout of 3600 seconds for the sampling phase of Manthan. To this end, we augment the sampling step of Manthan with the corresponding samplers. We perform the empirical analysis of the same 609 benchmarks⁴ that were employed in the analysis of Manthan [21]. We present a

⁴Benchmarks are available at <https://zenodo.org/record/3892859>

summary of our analysis in the form of cactus plot in Figure 3: the number of instances are shown on the x-axis and the time taken on the y-axis; a point (x, y) implies that Manthan augmented with the corresponding sampler took less than or equal to y seconds to solve x instances.

Table IV shows the time taken to synthesize Boolean functions with samples generated from different samplers for a representative set of 20 benchmarks.

Few observations are in order:

- 1) Manthan augmented with UniGen3 could solve only 118 instances due to UniGen3’s inability to sample for all but 220 instances. Similarly, Manthan with STS could solve only 157 instances.
- 2) Manthan augmented with CMSGen solves 345 instances while Manthan augmented with QuickSampler could solve only 275 instances.

TABLE IV: Runtime analysis of Manthan with QuickSampler, STS, UniGen3, and CMSGen. Timeout (TO): 7200s.

Benchmarks	QuickSampler	STS	UniGen3	CMSGen
kenflashp02	9.55	1367.12	573.69	26.77
kenoopp1	25.96	1852.07	TO	28.88
bobsynth00neg	114.66	3621.66	TO	74.06
bobtuint04neg	58.62	3636.39	1276.1	109.29
small-swap1-fix-4	TO	TO	TO	148.15
pdpmsrotate32	TO	TO	TO	279.6
exquery_query42	254.17	TO	TO	281.5
GuidanceService2	529.16	TO	TO	290.71
subtraction256	699.09	3836.48	TO	321.35
IssueServiceImpl	1567.23	TO	TO	424.77
query55_query42	6488.93	TO	TO	766.98
rankfunc48_s_64	TO	TO	TO	775.42
sortnetsort7.006	732.42	TO	TO	785.13
LoginService	TO	TO	TO	1108.0
query30_query42	1134.6	TO	TO	1126.53
ethernet-fixpoint-4	TO	TO	TO	1752.18
query44_query26	TO	TO	TO	2037.54
small-equiv-fix-8	TO	TO	TO	2231.22
pi-fixpoint-2	535.74	3674.9	TO	2373.72
sortnetsort9.010	3795.4	TO	TO	4414.56

Therefore, in conclusion, Manthan augmented with CMSGen solves significantly more instances than Manthan augmented with UniGen3, STS, or QuickSampler.

VI. CONCLUSION

Motivated by the availability of Barbarik, a tester for samplers, we sought to design a sampler for which Barbarik would return Accept. We succeeded in our task by a simple but careful tweaking of the existing state-of-the-art SAT solver, CryptoMiniSat. Our resulting sampler CMSGen is not only accepted by Barbarik but achieves better runtime performance than state-of-the-art samplers with theoretical guarantees. We then show that the resulting sampler, CMSGen, can significantly improve the performance of applications that utilize samplers. It is perhaps worth reiterating that we view the simplicity of CMSGen as its salient strength. The simplicity of CMSGen stands in stark contrast to complicated algorithmic schemes developed in the past that fail to attain the desired quality of distributions with practical runtime performance.

We now turn our attention back to Remark 1; the design of CMSGen was an iterative process with Barbarik in loop. A natural direction of future work would be the development of a tester that provides a quantitative analysis instead of a qualitative answer of Accept or Reject to measure the quality of samplers. The significant runtime improvements in the context of functional synthesis and combinatorial testing due to CMSGen motivate us to study the impact of CMSGen in other application domains; to this end, we will release CMSGen open-source upon publication of our manuscript.

Acknowledgments: This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore: <https://www.nscg.sg>

REFERENCES

- [1] D. Achlioptas, Z. S. Hammoudeh, and P. Theodoropoulos, “Fast sampling of perfectly uniform satisfying assignments,” in *Proc. of SAT*, 2018.
- [2] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers,” in *Proc. of IJCAI*, 2009.
- [3] T. Batu, S. Dasgupta, R. Kumar, and R. Rubinfeld, “The complexity of approximating the entropy,” *Proc. SIAM Journal on Computing*, 2005.
- [4] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.
- [5] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based grey-box fuzzing as markov chain,” *Proc. of ACM SIGSAC*, 2016.
- [6] S. Cai, C. Luo, and K. Su, “CCAnr: A configuration checking based local search solver for non-random satisfiability,” in *SAT 2015*, ser. LNCS, M. Heule and S. A. Weaver, Eds., vol. 9340. Springer, 2015, pp. 1–8.
- [7] S. Chakraborty, E. Fischer, Y. Goldhirsh, and A. Matsliah, “On the power of conditional samples in distribution testing,” *Proc. of SIAM Journal on Computing*, 2016.
- [8] S. Chakraborty and K. S. Meel, “On testing of uniform samplers,” in *Proc. of AAAI*, 2019.
- [9] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, “On parallel scalable uniform SAT witness generation,” in *Proc. of TACAS*, 2015.
- [10] S. Chakraborty, D. Fried, K. S. Meel, and M. Y. Vardi, “From weighted to unweighted model counting,” in *Proc. of AAAI*, 2015.
- [11] S. Chakraborty, K. S. Meel, and M. Y. Vardi, “A scalable and nearly uniform generator of sat witnesses,” in *Proc. of CAV*, 2013.
- [12] —, “Balancing scalability and uniformity in SAT witness generator,” in *Proc. of DAC*, 2014.
- [13] L. A. Clarke, “A program testing system,” in *Proc. of ACM*, 1976.
- [14] R. Dechter, K. Kask, E. Bin, R. Emek *et al.*, “Generating random solutions for constraint satisfaction problems,” in *Proc. of AAAI*, 2002.
- [15] J. Devriendt and B. Bogaerts, “BreakID: Static symmetry breaking for ASP (system description),” *CoRR*, vol. abs/1608.08447, 2016.
- [16] R. Dutra, K. Laeuffer, J. Bachrach, and K. Sen, “Efficient sampling of SAT solutions for testing,” in *Proc. of ICSE*, 2018.
- [17] N. Eén and A. Biere, “Effective preprocessing in SAT through variable and clause elimination,” in *SAT 2005*, ser. LNCS, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, pp. 61–75.
- [18] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman, “Uniform solution sampling using a constraint solver as an oracle,” in *Proc. of UAI*, 2012.
- [19] G. Fraser and A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software,” in *Proc. of ESEC/FSE*, 2011.
- [20] V. Gogate and R. Dechter, “A new algorithm for sampling CSP solutions uniformly at random,” in *Proc. of CP*, 2006.
- [21] P. Golia, S. Roy, and K. S. Meel, “Manthan: A data-driven approach for Boolean function synthesis,” in *Proc. of CAV*, 2020.
- [22] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments,” in *Proc. of USENIX Security 12*, 2012.
- [23] J. C. King, “Symbolic execution and program testing,” *Comm. ACM*, 1976.

- [24] N. Kitchen, “Markov chain monte carlo stimulus generation for constrained random simulation,” Ph.D. dissertation, UC Berkeley, 2010.
- [25] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer, “Is there a mismatch between real-world feature models and product-line research?” in *Proc. of ESEC/FSE*, 2017.
- [26] J. H. Kukula and T. R. Shiple, “Building circuits from relations,” in *Proc. of CAV*, 2000.
- [27] S. Kullback and R. A. Leibler, “On information and sufficiency,” *Proc. of Ann. Math. Statist.*, 1951.
- [28] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, “Complete functional synthesis,” 2010.
- [29] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, “Sat-based analysis of large real-world feature models is easy,” in *Proc. of SPLC*, 2015.
- [30] F. Massacci and L. Marraro, “Logical cryptanalysis as a SAT problem,” *Journal of Automated Reasoning*, 2000.
- [31] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, “A comparison of 10 sampling algorithms for configurable systems,” in *Proc. of ICSE*, 2016.
- [32] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient SAT solver,” in *Proc. of DAC*, 2001.
- [33] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, “Constraint-based random stimuli generation for hardware verification,” *Proc. of AI magazine*, 2007.
- [34] J. Oh, P. Gazzillo, and D. Batory, “t-wise coverage by uniform sampling,” in *Proc. of SPLC*, 2019.
- [35] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, “Uniform sampling of sat solutions for configurable systems: Are we there yet?” in *Proc. of ICST*, 2019.
- [36] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, “Bug synthesis: Challenging bug-finding tools with deep faults,” in *Proc. of ESEC/FSE*, 2018.
- [37] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, “Knowledge compilation meets uniform sampling,” in *Proc. of LPAR*, 2018.
- [38] —, “Knowledge compilation meets uniform sampling,” in *Proc. of LPAR*, 2018.
- [39] M. Soos, S. Gocht, and K. S. Meel, “Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling,” in *Proc. of CAV*, 2020.
- [40] M. Soos and K. S. Meel, “Bird: Engineering an efficient CNF-XOR sat solver and its applications to approximate model counting,” in *Proc. of the AAAI*, 2019.
- [41] M. Soos, K. Nohl, and C. Castelluccia, “Extending SAT solvers to cryptographic problems,” in *Proc. of SAT*, 2009.
- [42] S. Srivastava, S. Gulwani, and J. S. Foster, “Template-based program verification and program synthesis,” *STTT*, 2013.
- [43] W. Wei and B. Selman, “A new approach to model counting,” in *Proc. of SAT*, 2005.
- [44] N. Wetzler, M. J. H. Heule, and W. A. Hunt, “DRAT-trim: Efficient checking and trimming using expressive clausal proofs,” in *Proc. of SAT*, 2014.

SAT-Inspired Eliminations for Superposition

Petar Vukmirović¹ , Jasmin Blanchette^{1,2} , Marijn J.H. Heule³ 

¹Vrije Universiteit Amsterdam, Amsterdam, the Netherlands

²Université de Lorraine, CNRS, Inria, LORIA, Nancy, France

³Carnegie Mellon University, Pittsburgh, Pennsylvania, United States

Abstract—Optimized SAT solvers not only preprocess the clause set, they also transform it during solving as inprocessing. Some preprocessing techniques have been generalized to first-order logic with equality. In this paper, we port inprocessing techniques to work with superposition, a leading first-order proof calculus, and we strengthen known preprocessing techniques. Specifically, we look into elimination of hidden literals, variables (predicates), and blocked clauses. Our evaluation using the Zipperposition prover confirms that the new techniques usefully supplement the existing superposition machinery.

I. INTRODUCTION

Automated reasoning tools have become much more powerful in the last few decades thanks to procedures such as conflict-driven clause learning (CDCL) [1] for propositional logic and superposition [2] for first-order logic with equality. However, the effectiveness of these procedures crucially depends on how the input problem is represented as a clause set. The clause set can be optimized beforehand (*preprocessing*) or during the execution of the procedure (*inprocessing*). In this paper, we lift several preprocessing and inprocessing techniques from propositional logic to clausal first-order logic and demonstrate their usefulness in a superposition prover.

For many years, SAT solvers have used inexpensive clause simplification techniques such as hidden literal and hidden tautology elimination [3], [4] and failed literal detection [5, Sect. 1.6]. We generalize these techniques to first-order logic with equality (Sect. III). Since the generalization involves reasoning about infinite sets of literals, we propose restrictions to make them usable.

Variable elimination, based on Davis–Putnam resolution [6], has been studied in the context of both propositional logic [7], [8] and quantified Boolean formulas (QBFs) [9]. The basic idea is to resolve all clauses with negative occurrences of a propositional variable (i.e., a nullary predicate symbol) against clauses with positive occurrences and delete the parent clauses. Eén and Biere [10] refined the technique to identify a subset of clauses that effectively define a variable and use it to further optimize the clause set. This latter technique, *variable elimination by substitution*, has been an important preprocessor component in many SAT solvers since its introduction in 2004.

Specializing second-order quantifier elimination [11], [12], Khasidashvili and Korovin [13] adapted variable elimination to preprocess first-order problems, yielding a technique we call *singular predicate elimination*. We extend their work along two axes (Sect. IV): We generalize Eén and Biere’s refinement

to first-order logic, resulting in *defined predicate elimination*, and explain how both types of predicate elimination can be used during the proof search as inprocessing.

The last technique we study is *blocked clause elimination* (Sect. V). It is used in both SAT [14] and QBF solvers [15]. Its generalization to first-order logic has produced good results when used as a preprocessor, especially on satisfiable problems [16]. We explore more ways to use blocked clause elimination on satisfiable problems, including using it to establish equisatisfiability with an empty clause set or as an inprocessing rule. Unfortunately, we find that its use as inprocessing can compromise the refutational completeness of superposition.

All techniques are implemented in the Zipperposition prover (Sect. VI), allowing us to ascertain their usefulness (Sect. VII). The best configuration solves 160 additional problems on benchmarks consisting of all 13 495 first-order TPTP theorems [17]. The raw experimental data are publicly available.¹ More details, including all the proofs, can be found in a technical report [18].

II. PRELIMINARIES

A. Clausal First-Order Logic

Our setting is many-sorted, or many-typed, first-order logic [19] with interpreted equality and a distinguished type (or sort) o . Each variable x is assigned a non-Boolean type, and each symbol f is assigned a tuple $(\tau_1, \dots, \tau_n, \tau)$ where $n \geq 0$, τ_i are non-Boolean types, and τ is the *result type*. We distinguish between *predicate symbols*, with o as the result type, and *function symbols*. Nullary function symbols are called *constants*. Terms are either variables x or well-typed applications $f(t_1, \dots, t_n)$, or f if $n = 0$. A term is *ground* if it contains no variables. We assume standard definitions and notations for positions, subterms, and contexts [20]. We abbreviate a vector (a_1, \dots, a_n) to \vec{a}_n or \vec{a} , and write $f^i(s)$ for the i -fold application of an unary symbol f (e.g., $f^3(x) = f(f(f(x)))$).

An atom is an equation $s \approx t$ corresponding to an unordered pair $\{s, t\}$. A literal is an equation $s \approx t$ or a disequation $s \not\approx t$. For every predicate symbol p , $p(\vec{s})$ abbreviates $p(\vec{s}) \approx \top$, and $\neg p(\vec{s})$ abbreviates $p(\vec{s}) \not\approx \top$, where \top is a distinguished constant of type o . We distinguish between *predicate literals* $(\neg)p(\vec{s})$ and *functional literals* $s \approx t$, where s and t are not of type o . Given a literal L , we overload notation and write $\neg L$ to denote its complement. A clause C is a multiset of literals,

¹<https://doi.org/10.5281/zenodo.4552499>

written as $L_1 \vee \dots \vee L_n$ and interpreted disjunctively. Clauses are often defined as sets of literals, but superposition needs multisets; with multisets, an instance $C\sigma$ always has the same number of literals as C , a most convenient property. Given a clause set N , $N \downarrow_2$ denotes the subset of its binary clauses: $N \downarrow_2 = \{L_1 \vee L_2 \mid L_1 \vee L_2 \in N\}$.

B. Superposition Provers

Superposition [2] is a calculus for clausal first-order logic that extends ordered resolution [21] with equality reasoning. It is refutationally complete: Given a finite, unsatisfiable clause set, it will eventually derive the empty clause. It is parameterized by a *selection function* that influences which of a clause's literals are eligible as the target of inferences. Moreover, it is compatible with the *standard redundancy criterion*, which can be used to delete a clause C while preserving completeness of the calculus.

The redundancy criterion relies on an order \succ that compares terms, literals, or clauses. The order is used to determine whether clauses can be deleted. If N is ground, C can be deleted if it is entailed by \prec -smaller clauses in N . This definition is lifted to nonground sets N . The criterion can be used to delete a clause that is *subsumed* by another clause (e.g., $p(a) \vee q$ by $p(x)$) or to *simplify* a clause C into C' , which amounts to adding C' and then deleting C as redundant with respect to $N \cup \{C'\}$. Subsumption and simplification are the main inprocessing mechanisms available to superposition provers. Some provers also implement clause splitting [22]–[24].

Superposition provers saturate the input problem with respect to the calculus's inference rules using the *given clause procedure* [25], [26]. It partitions the proof state into a passive set \mathcal{P} and an active set \mathcal{A} . All clauses start in \mathcal{P} . At each iteration of the procedure's main loop, the prover chooses a clause C from \mathcal{P} , simplifies it, and moves it to \mathcal{A} . Then all inferences between C and active clauses are performed. The resulting clauses are again simplified and put in \mathcal{P} .

III. HIDDEN-LITERAL-BASED ELIMINATION

In propositional logic, binary clauses from a clause set N can be used to efficiently discover literals L, L' for which the implication $L' \rightarrow L$ is entailed by N 's binary clauses—i.e., $N \downarrow_2 \models L' \rightarrow L$. Heule et al. [4] introduced the concept of *hidden literals* to capture such implications.

Definition 1: Given a propositional literal L and a propositional clause set N , the set of *propositional hidden literals* for L and N is $\text{HL}_p(L, N) = \{L' \mid L' \hookrightarrow_p^* L\} \setminus \{L\}$, where \hookrightarrow_p is defined such that $\neg L_1 \hookrightarrow_p L_2$ whenever $L_1 \vee L_2 \in N$. Moreover, $\text{HL}_p(L_1 \vee \dots \vee L_n, N) = \bigcup_{i=1}^n \text{HL}_p(L_i, N)$.

Heule et al. used a fixpoint computation, but our definition based on the reflexive transitive closure is equivalent. Intuitively, a hidden literal can be added to or removed from a clause without affecting its semantics in models of N . By eliminating hidden literals from C , we simplify it. By adding hidden literals to C , we might get a tautology C' (i.e., a valid clause: $\models C'$), meaning that $N \downarrow_2 \models C$, thereby enabling us to delete C . Note that $\text{HL}_p(L, N)$ is finite for a finite N .

Definition 2: Given $L' \vee L \vee C \in N$, if $L' \in \text{HL}_p(L, N)$, *hidden literal elimination* (HLE) replaces N by $(N \setminus \{L' \vee L \vee C\}) \cup \{L \vee C\}$. Given $C \in N$, $\{L_1, \dots, L_n\} = \text{HL}_p(C, N)$, and $C' = C \vee L_1 \vee \dots \vee L_n$, if C' is a tautology, *hidden tautology elimination* (HTE) replaces N by $N \setminus \{C\}$.

Theorem 3: The result of applying HLE or HTE to a clause set N is equivalent to N .

Proof: For HLE, if $L' \in \text{HL}_p(L, N)$, $N \downarrow_2 \models \neg L' \vee L$. Then, subsumption resolution yields shortened clause $L \vee C'$ from Definition 2. For HTE, it can be shown that $N' \models C$ if and only if $C \vee L'$, where $L' \in \text{HL}_p(C, N)$. By transitivity of equivalence, we get the desired result. ■

We generalize hidden literals to first-order logic with equality by considering substitutivity of variables as well as congruence of equality.

Definition 4: Given a literal L and a clause set N , the set of *hidden literals* for L and N is $\text{HL}(L, N) = \{L' \mid L' \hookrightarrow^* L\} \setminus \{L\}$, where \hookrightarrow is defined so that (1) $\neg L' \sigma \hookrightarrow L \sigma$ if $L' \vee L \in N$ and σ is a substitution; (2) $s \approx t \hookrightarrow u[s] \approx u[t]$ for all terms s, t and contexts $u[\]$; and (3) $u[s] \not\approx u[t] \hookrightarrow s \not\approx t$ for all terms s, t and contexts $u[\]$. Moreover, $\text{HL}(L_1 \vee \dots \vee L_n, N) = \bigcup_{i=1}^n \text{HL}(L_i, N)$.

The generalized definition also enjoys the key property that $L' \in \text{HL}(L, N)$ implies $N \downarrow_2 \models L' \rightarrow L$. However, $\text{HL}(L, N)$ may be infinite even for predicate literals; for example, $p(f^i(x)) \in \text{HL}(p(x), \{p(x) \vee \neg p(f(x))\})$ for every i .

Based on Definition 4, we can generalize hidden literal elimination and support a related technique:

$$\frac{L' \vee L \vee C}{L \vee C} \text{HLE} \quad \text{if } L' \in \text{HL}(L, N)$$

$$\frac{L \vee C}{C} \text{FLE} \quad \text{if } L', \neg L' \in \text{HL}(\neg L, N)$$

Double lines denote *simplification rules*: When the premises appear in the clause set, the prover can use the redundancy criterion to replace them by the conclusions. The second rule is called *failed literal elimination*, inspired by the SAT technique of asserting $\neg L$ if L is a *failed literal* [5]. It is easy to see that rule HLE is sound. From $L' \in \text{HL}(L, N)$ we have $N \models L' \rightarrow L$ (i.e., $\neg L' \vee L$). Performing subsumption resolution [21] between $L' \vee L \vee C$ and $\neg L' \vee L$ yields the conclusion, which is therefore entailed by N . For FLE, the condition $L', \neg L' \in \text{HL}(\neg L, N)$ means that $N \downarrow_2 \models \{\neg L' \vee \neg L, L' \vee \neg L\} \models \neg L$.

Example 5: Consider the clause set $N = \{p(x) \vee \neg p(f(x)), p(f(f(x))) \vee a \approx b\}$ and the clause $C = f(a) \not\approx f(b) \vee p(x)$. The first clause in N induces $p(f(x)) \hookrightarrow p(x)$, $p(f(f(x))) \hookrightarrow p(f(x))$, and hence $p(f(f(f(x)))) \hookrightarrow^* p(x)$. Together with the second clause in N , it can be used to derive $a \not\approx b \hookrightarrow^* p(x)$. Finally, using rule (3) of Definition 4, we derive $f(a) \not\approx f(b) \hookrightarrow^* p(x)$ —that is, $f(a) \not\approx f(b) \in \text{HL}(p(x), N)$. This allows us to remove C 's first literal using HLE.

Two special cases of HLE exploit equality congruence as embodied by conditions (2) and (3) of Definition 4 without requiring to compute the HL set:

$$\frac{\frac{s \approx t \vee u[s] \approx u[t] \vee C}{u[s] \approx u[t] \vee C} \text{CONGHLE}^+}{\frac{s \not\approx t \vee u[s] \not\approx u[t] \vee C}{s \not\approx t \vee C} \text{CONGHLE}^-}$$

Hidden literals can be combined with unit clauses L' to remove more literals:

$$\frac{L' \quad L \vee C}{L' \quad C} \text{UNITHLE} \quad \text{if } L'\sigma \in \text{HL}(\neg L, N)$$

Given a unit clause $L' \in N$, the rule uses it to discharge $L'\sigma$ in $N \models L'\sigma \rightarrow \neg L$. As a result, we have $N \models \neg L$, making it possible to remove L from $L \vee C$.

Example 6: Consider the clause set $N = \{p(x) \vee q(f(x)), \neg q(f(a)) \vee f(b) \approx g(c), f(x) \not\approx g(y)\}$ and the clause $C = \neg p(a) \vee \neg q(b)$. The first clause in N induces $\neg q(f(a)) \hookrightarrow p(a)$, whereas the second one induces $f(b) \not\approx g(c) \hookrightarrow \neg q(f(a))$. Thus, we have $f(b) \not\approx g(c) \hookrightarrow^* p(a)$ —that is, $f(b) \not\approx g(c) \in \text{HL}(p(a), N)$. By applying the substitution $\{x \mapsto b, y \mapsto c\}$ to the third clause in N , we can fulfill the conditions of UNITHLE and remove C 's first literal.

Next, we generalize hidden tautologies to first-order logic.

Definition 7: A clause C is a *hidden tautology* for a clause set N if there exists a finite set $\{L_1, \dots, L_n\} \subseteq \text{HL}(C, N)$ such that $C \vee L_1 \vee \dots \vee L_n$ is a tautology.

Example 8: In general, hidden tautologies are not redundant and cannot be deleted during saturation. Consider the unsatisfiable set $N = \{\neg a, \neg b, a \vee c, b \vee \neg c\}$, the order $a \prec b \prec c$, and the empty selection function. The only possible superposition inference from N is between the last two clauses, yielding the hidden tautology $a \vee b$ (after simplifying away $\top \not\approx \top$), which is entailed by the larger clauses $a \vee c$ and $b \vee \neg c$. If this clause is removed, the prover could enter an infinite loop, forever generating and deleting the hidden tautology.

To delete hidden tautologies during saturation, the prover could check that all the relevant clause instances encountered along the computation of HL are \prec -smaller than a given hidden tautology. However, this would be expensive and seldom succeed, given that superposition creates lots of nonredundant hidden tautologies. Instead, we propose to simplify hidden tautologies using the following rules:

$$\frac{L \vee L' \vee C}{L \vee L'} \text{HTR} \quad \text{if } \neg L' \in \text{HL}(L, N) \text{ and } C \neq \perp$$

$$\frac{L \vee C}{L} \text{FLR} \quad \text{if } L', \neg L' \in \text{HL}(L, N) \text{ and } C \neq \perp$$

We call these techniques *hidden tautology reduction* and *failed literal reduction*, respectively. Both rules are sound. As with hidden literals, unit clauses L' can be exploited:

$$\frac{L' \quad L \vee C}{L' \quad L} \text{UNITHTR} \quad \text{if } L'\sigma \in \text{HL}(L, N) \text{ and } C \neq \perp$$

We give the simplification rules above the collective name of *hidden-literal-based elimination* (HLBE). Yet another use of hidden literals is for *equivalent literal substitution* [3]: If both $L' \in \text{HL}(L, N)$ and $L \in \text{HL}(L', N)$, we can often simplify $L'\sigma$ to $L\sigma$ in N if $L'\sigma \succ L\sigma$. We want to investigate this further.

Theorem 9: The rules HLE, FLE, CONGHLE⁺, CONGHLE⁻, UNITHLE, HTR, FLR, and UNITHTR are sound simplification rules.

IV. PREDICATE ELIMINATION

For propositional logic, variable elimination [10] is one of the main preprocessing and inprocessing techniques. Following Gabbay and Ohlbach's ideas [11], Khasidashvili and Korovin [13] generalized variable elimination to first-order logic with equality and demonstrated that it is effective as a preprocessor. We propose an improvement that makes this applicable in more cases and show that, with a minor restriction, it can be integrated in a superposition prover without compromising its refutational completeness.

A. Singular Predicates

Khasidashvili and Korovin's preprocessing technique removes singular predicates (which they call “non-self-referential predicates”) from the problem using so-called flat resolution.

Definition 10: A predicate symbol is called *singular* (or “non-self-referential”) for a clause set N if it occurs at most once in every clause contained in N .

Definition 11: Let $C = p(\vec{s}_n) \vee C'$ and $D = \neg p(\vec{t}_n) \vee D'$ be clauses with no variables in common. The clause $s_1 \not\approx t_1 \vee \dots \vee s_n \not\approx t_n \vee C' \vee D'$ is a *flat resolvent* of C and D on p .

Given two (possibly identical) clause sets M, N , predicate elimination iteratively replaces clauses from N containing the symbol p with all flat resolvents against clauses in M . Eventually, it yields a set with no occurrences of p .

Definition 12: Let M, N be clause sets and p be a singular predicate for M . Let \rightsquigarrow be the following relation on clause set pairs and clause sets:

- 1) $(M, \{(\neg)p(\vec{s}) \vee C'\} \uplus N) \rightsquigarrow (M, N' \cup N)$ if N' is the set that consists of all clauses (up to variable renaming) that are flat resolvents with $(\neg)p(\vec{s}) \vee C'$ on p and a clause from M as premises. The premises' variables are renamed apart.
- 2) $(M, N) \rightsquigarrow N$ if N has no occurrences of p .

The *resolved set* $M \bowtie_p N$ is the clause set N' such that $(M, N) \rightsquigarrow^* N'$.

The relation \rightsquigarrow is confluent up to variable renaming. Thanks to the singularity constraint on M , it also terminates on

finite sets because the following ordinal measure decreases: $\nu(\{D_1, \dots, D_n\}) = \omega^{\nu(D_1)} \oplus \dots \oplus \omega^{\nu(D_n)}$, where $\nu(D)$ counts the occurrences of p in D , ω is the first infinite ordinal, and \oplus is the Hessenberg, or natural, sum, which is commutative. For every transition $(M, \{C\} \cup N) \rightsquigarrow (M, N' \cup N)$, we have $\nu(\{C\}) = \omega^{\nu(C)} > \omega^{\nu(C)-1} \cdot |N'| = \nu(N')$.

Next, it is useful to partition clause sets into subsets based on the presence and polarity of a singular predicate.

Definition 13: Let N be a clause set and p be a singular predicate for N . Let N_p^+ consist of all clauses of the form $p(\vec{s}) \vee C' \in N$, let N_p^- consist of all clauses of the form $\neg p(\vec{s}) \vee C' \in N$, let $N_p = N_p^+ \cup N_p^-$, and let $\bar{N}_p = N \setminus N_p$.

Definition 14: Let N be a clause set and p be a singular predicate for N . *Singular predicate elimination* (SPE) of p in N replaces N by $\bar{N}_p \cup (N_p^+ \bowtie_p N_p^-)$.

The result of SPE is satisfiable if and only if N is satisfiable [13, Theorem 1], justifying SPE's use in a preprocessor. However, eliminating singular predicates aggressively can dramatically increase the number of clauses. To prevent this, Khasidashvili and Korovin suggested to replace N by N' only if $\lambda(N') \leq \lambda(N)$ and $\mu(N') \leq \mu(N)$, where $\lambda(N)$ is the number of literals in N and $\mu(N)$ is the sum for all clauses $C \in N$ of the square of the number of distinct variables in C .

Compared with what modern SAT solvers use, this criterion is fairly restrictive. We relax it to make it possible to eliminate more predicates, within reason. Let $K_{\text{tol}} \in \mathbb{N}$ be a tolerance parameter. A predicate elimination step from N to N' is allowed if $\lambda(N') < \lambda(N) + K_{\text{tol}}$ or $\mu(N') < \mu(N)$ or $|N'| < |N| + K_{\text{tol}}$.

B. Defined Predicates

SPE is effective, but an important refinement has not yet been adapted to first-order logic: variable elimination by substitution. Eén and Biere [10] discovered that a propositional variable x can be eliminated without computing all resolvents if it is expressible as an equivalence $x \leftrightarrow \varphi$, where φ , the “gate,” is an arbitrary formula that does not reference x . They partition a set N into a definition set G , essentially the clausification of $x \leftrightarrow \varphi$, and $R = N_p \setminus G$, the remaining clauses containing p . To eliminate x from N while preserving satisfiability, it suffices to resolve clauses from G against clauses from R , effectively substituting φ for x in R . Crucially, we do not need to resolve pairs of clauses from G or pairs of clauses from R . We generalize this idea to first-order logic.

Definition 15: Let G be a clause set, p be a predicate symbol, and \vec{x} be distinct variables. The set G is a *definition set* for p if (1) p is singular for G , (2) G consists of clauses of the form $(\neg)p(\vec{x}) \vee C'$ (up to variable renaming), (3) the variables in C' are all among \vec{x} , (4) all clauses in $G_p^+ \bowtie_p G_p^-$ are tautologies, and (5) $E(\vec{c})$ is unsatisfiable, where the *environment* $E(\vec{x})$ consists of all subclauses C' of any $(\neg)p(\vec{x}) \vee C' \in G$ and \vec{c} is a tuple of distinct fresh constants substituted in for \vec{x} .

A definition set G corresponds intuitively to a definition by cases in mathematics—e.g.,

$$p(\vec{x}) = \begin{cases} \top & \text{if } \varphi(\vec{x}) \\ \perp & \text{if } \psi(\vec{x}) \end{cases}$$

Part (4) states that the case conditions are mutually exclusive (e.g., $\neg\varphi(\vec{x}) \vee \neg\psi(\vec{x})$), and part (5) states that they are exhaustive (e.g., $\nexists \vec{c} \neg\varphi(\vec{c}) \wedge \neg\psi(\vec{c})$). Given a quantifier-free formula $p(\vec{x}) \leftrightarrow \varphi(\vec{x})$ with distinct variables \vec{x} such that $\varphi(\vec{x})$ does not contain p , any reasonable clausification algorithm would produce a definition set for p .

Example 16: Given the formula $p(x) \leftrightarrow q(x) \wedge (r(x) \vee s(x))$, a standard clausification algorithm [27] produces $\{\neg p(x) \vee q(x), \neg p(x) \vee r(x) \vee s(x), p(x) \vee \neg q(x) \vee \neg r(x), p(x) \vee \neg q(x) \vee \neg s(x)\}$, which qualifies as a definition set for p .

Definition sets generalize Eén and Biere's gates. They can be recognized syntactically for formulas such as $p(\vec{x}) \leftrightarrow \bigvee_i q_i(\vec{s}_i)$ or $p(\vec{x}) \leftrightarrow \bigwedge_i q_i(\vec{s}_i)$, or semantically: Condition (4) can be checked using the congruence closure algorithm, and condition (5) amounts to a propositional unsatisfiability check.

The key result about propositional gates carries over to definition sets.

Definition 17: Let N be a clause set, p be a predicate symbol, $G \subseteq N$ be a definition set for p , and $R = N_p \setminus G$. *Defined predicate elimination* (DPE) of p in N replaces N by $\bar{N}_p \cup (G_p \bowtie_p R_p)$.

Theorem 18: The result of applying DPE to a clause set N is satisfiable if and only if N is satisfiable.

Since there will typically be at most only a few defined predicates in the problem, it makes sense to fall back on SPE when no definition is found.

Definition 19: Let N be a clause set and p be a predicate symbol. If there exists a definition set $G \subseteq N$ for p , *portfolio predicate elimination* (PPE) on p in N replaces N with $\bar{N}_p \cup (G_p \bowtie_p R_p)$, where $R = N_p \setminus G$. Otherwise, if p is singular in N , it results in $\bar{N}_p \cup (N_p^+ \bowtie_p N_p^-)$. In all other cases, it is not applicable.

C. Refutational Completeness

Hidden-literal-based techniques fit within the traditional framework of saturation, because they delete or reduce a clause based on the *presence* of other clauses. In contrast, predicate elimination relies on the *absence* of clauses from the proof state. We can still integrate it with superposition as follows: At every k th iteration of the given clause procedure, perform predicate elimination on $\mathcal{A} \cup \mathcal{P}$, and add all new clauses to \mathcal{P} .

One may wonder whether such an approach preserves the refutational completeness of the calculus. The answer is no. To see why, consider the following *binary splitting* rule based on Riazanov and Voronkov [22]:

$$\frac{C \vee D}{p \vee C \quad D \vee \neg p} \text{BS}$$

Provisos: C and D have no free variables in common, p is fresh, and p is \prec -smaller than C and D . Since the conclusions are smaller than the premise, the rule can be applied aggressively as a simplification. But notice that the effect of

splitting can be undone by singular predicate elimination, possibly giving rise to loops BS,SPE,BS,SPE,... This breaks completeness.

Our solution is to curtail the entailment relation used by the redundancy criterion to disallow splitting-like simplifications. Weak entailment \models^b is defined via an ad hoc nonclassical logic so that $\{p \vee C, \neg p \vee C\} \not\models^b \{C\}$ and yet $\models^b \{p \vee \neg p\}$. More precisely, this logic is defined via an encoding: $M \models^b N$ if and only if $M^b \models N^b$, where $p(\vec{i})^b = p(\vec{i}) \not\approx \perp$, $\neg p(\vec{i})^b = p(\vec{i}) \not\approx \top$, and $L^b = L$ otherwise. Moreover, the type o may be interpreted as any set of cardinality at least 2, and \perp must be a distinguished symbol interpreted differently from \top .

The standard redundancy criterion Red^b based on \models^b supports all the familiar deletion and simplification techniques except splitting. Using Red^b not only prevents looping, but it also enables the use of the given clause procedure, because any redundant inference according to Red^b remains redundant after SPE or DPE. As usual, the devil is in the details, and the details are in the report [18].

V. SATISFIABILITY BY CLAUSE ELIMINATION

The main approaches to show satisfiability of a first-order problem are to produce either a finite Herbrand model or a saturated clause set. Saturations rarely occur except for very small problems or within decidable fragments. In this section, we explore an alternative approach that establishes satisfiability by iteratively removing clauses while preserving unsatisfiability, until the clause set has been transformed into the empty set. So far, this technique has been studied only for QBF [28]. We show that *blocked clause elimination* (BCE) can be used for this purpose. It can efficiently solve some problems for which the saturated set would be infinite. However, it can break the refutational completeness of a saturation prover. We conclude with a procedure that transforms a finite Herbrand model into a sequence of clause elimination steps ending in the empty clause set, thereby demonstrating the theoretical power of clause elimination.

Kiesl et al. [16] generalized blocked clause elimination to first-order logic. Their generalization uses flat L -resolvents, an extension of flat resolvents that resolves a single literal L against m literals of the other clause.

Definition 20: Let $C = L \vee C'$ and $D = L_1 \vee \dots \vee L_m \vee D'$, where (1) $m \geq 1$, (2) the literals L_i are of opposite polarity to L , (3) L 's atom is $p(\vec{s}_n)$, (4) L_i 's atom is $p(\vec{t}_i)$ for each i , and (5) C and D have no variables in common. The clause $(\bigvee_{i=1}^m \bigvee_{j=1}^n s_j \not\approx t_{ij}) \vee C' \vee D'$ is a *flat L -resolvent* of C and D .

Definition 21: A clause $C = L \vee C'$ is (*equality*-) *blocked* by L in a clause set N if all flat L -resolvents between C and clauses in $N \setminus \{C\}$ are tautologies.

Removing a blocked clause from a set preserves unsatisfiability [16]. Kiesl et al. evaluated the effect of removing all blocked clauses as a preprocessing step and found that it increases prover's success rate.

In fact, there exist satisfiable problems that cannot be saturated in finitely many steps regardless of the calculus's

parameters but that can be reduced to an empty, vacuously satisfiable problem through blocked clause elimination.

Example 22: Consider the clause set N consisting of $C = p(x, x)$ and $D = \neg p(y_1, y_3) \vee p(y_1, y_2) \vee p(y_2, y_3)$. Note that if no literal is selected, all literals are eligible for superposition. In particular, the superposition of $p(x, x)$ into D 's negative literal eventually needs to be performed regardless of the chosen selection function or term order, with the conclusion $E_1 = p(z_1, z_2) \vee p(z_2, z_1)$. Then, superposition of E_1 into D yields $E_2 = p(z_1, z_2) \vee p(z_2, z_3) \vee p(z_3, z_1)$. Repeating this process yields infinitely many clauses $E_i = p(z_1, z_2) \vee \dots \vee p(z_i, z_{i+1}) \vee p(z_{i+1}, z_1)$ that cannot be eliminated using standard redundancy-based techniques.

In the example above, the clause D is blocked by its second or third literal. If we delete D , C becomes blocked in turn. Deleting C leaves us with the empty set, which is vacuously satisfiable. The example suggests that using BCE during saturation might help focus the proof search. Indeed, Kiesl et al. ended their investigations by asking whether BCE can be used as an inprocessing technique in a saturation prover. Unfortunately, in general the answer is no.

Example 23: Consider the unsatisfiable set $N = \{C_1, \dots, C_6\}$, where

$$\begin{array}{lll} C_1 = \neg c \vee e \vee \neg a & C_2 = \neg c \vee \neg e & C_3 = b \vee c \\ C_4 = \neg b \vee \neg c & C_5 = a \vee b & C_6 = c \vee \neg b \end{array}$$

Assume the simplification ordering $a \prec b \prec c \prec d \prec e$ and the selection function that chooses the last negative literal of a clause as presented. Gray boxes indicate literals that can take part in superposition inferences. Only two superposition inferences are possible: from C_3 into C_4 , yielding the tautology $C_7 = b \vee \neg b$, and from C_5 into C_6 , yielding $C_8 = a \vee c$. Clause C_7 is clearly redundant, whereas C_8 is blocked by its first literal. If we allow removing blocked clauses, the prover enters a loop: C_8 is repeatedly generated and deleted. Thus, the prover will never generate the empty clause for this unsatisfiable set.

As with hidden tautologies, removing blocked clauses breaks the invariant of the given clause procedure that all inferences between clauses in \mathcal{A} are redundant. To see this, assume the setting of Example 23, and let $\mathcal{P} = N$ and $\mathcal{A} = \emptyset$. Assume C_1, C_2, C_3 are moved to the active set. As there are no possible inferences between them, the proof state becomes $\mathcal{A} = \{C_1, C_2, C_3\}$ and $\mathcal{P} = \{C_4, C_5, C_6\}$. After C_4 is moved to \mathcal{A} , the conclusion C_7 is computed, but it is not added to \mathcal{P} as it is redundant. Moving C_5 to \mathcal{A} produces no new conclusions, but after C_6 is moved, C_8 is produced. However, if we allow eliminating blocked clauses, it will not be added to \mathcal{P} as it is blocked. The prover then terminates with $\mathcal{A} = N$ and $\mathcal{P} = \emptyset$, even though the original set N is unsatisfiable.

Although using BCE as inprocessing breaks the completeness of superposition in general, it is conceivable that a well-behaved fragment of BCE might exist. This could be investigated further.

Not only can BCE prevent infinite saturation (Example 22), but it can also be used to convert a finite Herbrand model into a certificate of clause set satisfiability. The certificate uses only blocked clause elimination and addition, in conjunction with a transformation to reduce the clause set to an empty set. This theoretical result explores the relationship between Herbrand models and satisfiability certificates based on clause elimination and addition. It is conceivable that it can form the basis of an efficient way to certify Herbrand models.

In propositional logic, *asymmetric literals* can be added to or removed from clauses, retaining the equivalence of the resulting clause set with the original one. Kiesel and Suda [29] described an extension of this technique to first-order logic. Their definition of asymmetric literals can be relaxed to allow the addition of more literals, but the resulting set is then only equisatisfiable to the original one, not equivalent. This in turn allows us to show that a problem is satisfiable by reducing it to an empty problem, as is done in some SAT solvers.

For the rest of this section, we work with clausal first-order logic without equality. We use Herbrand models as canonical representatives of first-order models, recalling that every satisfiable set has a Herbrand model [30, Sect. 5.4].

Definition 24: A literal L is a *global asymmetric literal* (GAL) for a clause C and a clause set N if for every ground instance $C\sigma$ of C , there exists a ground instance $D\varrho \vee L'\varrho$ of $D \vee L' \in N \setminus \{C\}$ such that $D\varrho \subseteq C\sigma$ and $\neg L'\varrho = L\sigma$.

Every asymmetric literal is GAL, but the converse does not hold:

Example 25: Consider a clause $C = p(x, y)$ and a clause set $N = \{q \vee p(a, a)\}$. Then, $\neg q$ is not an asymmetric literal for C and N , but it is a GAL for C and N .

Adding and removing GALs maintains preserves and reflects satisfiability:

Theorem 26: If L is a GAL for the clause C and the clause set N , then the set $(N \setminus \{C\}) \cup \{C \vee L\}$ is satisfiable if and only if N is satisfiable.

For first-order logic without equality, a clause $L \vee C$ is blocked if all its L -resolvents are tautologies [16]. The L -resolvent between $L \vee C$ and $\neg L_1 \vee \dots \vee \neg L_n \vee D$ is $(C \vee D)\sigma$, where σ is the most general unifier of the literals L, L_1, \dots, L_n [21]. Given a Herbrand model \mathcal{J} of a problem, the following procedure removes all clauses while preserving satisfiability:

- 1) Let q be a fresh predicate symbol. For each atom $p(\vec{s})$ in the Herbrand universe: If $\mathcal{J} \models p(\vec{s})$, add the clause $q \vee p(\vec{s})$; otherwise, add $q \vee \neg p(\vec{s})$. Adding either clause preserves satisfiability as both are blocked by q .
- 2) Since \mathcal{J} is a model, for each ground instance $C\sigma$, there exists a clause $q \vee L$ with $L \in C\sigma$. We can transform $C \in N$ into $C \vee \neg q$, since $\neg q$ is a GAL for C and N .
- 3) Consider the clause $q \vee L$ added by step 1. Since L is ground and no clause $q \vee \neg L$ was added (since \mathcal{J} is a model), the only L -resolvents are against clauses added by step 2. Since all of those clauses contain $\neg q$, the

resolvents are tautologies. Thus, each $q \vee L$ is blocked and can be removed in turn.

- 4) The remaining clauses all contain the literal $\neg q$. They can be removed by BCE as well.

The procedure is limited to the first-order logic without equality, since step 3 is justified only if L is a predicate literal. (Otherwise, L cannot block clause $q \vee L$ [16].) The procedure also terminates only for finite Herbrand models.

Example 27: Consider the satisfiable clause set $N = \{r(x) \vee s(x), \neg r(a), \neg s(b)\}$ and a Herbrand model \mathcal{J} over $\{a, b, r, s\}$ such that $r(b)$ and $s(a)$ are the only true atoms in \mathcal{J} . We show how to remove all clauses in N using \mathcal{J} by following the procedure above.

Let $N_{\mathcal{J}} = \{q \vee \neg r(a), q \vee r(b), q \vee s(a), q \vee \neg s(b)\}$. We set $N \leftarrow N \cup N_{\mathcal{J}}$. This preserves satisfiability since all clauses in $N_{\mathcal{J}}$ are blocked. It is easy to check that $\neg q$ is GAL for every clause in $N \setminus N_{\mathcal{J}}$. The only substitutions that need to be considered are $\{x \mapsto a\}$ and $\{x \mapsto b\}$ for $r(x) \vee s(x)$. So we set $N \leftarrow \{\neg q \vee r(x) \vee s(x), \neg q \vee \neg r(a), \neg q \vee \neg s(b)\} \cup N_{\mathcal{J}}$. Clearly, all clauses in $N_{\mathcal{J}}$ are blocked, so we set $N \leftarrow N \setminus N_{\mathcal{J}}$. All clauses remaining in N have a literal $\neg q$ and can be removed, leaving N empty as desired.

VI. IMPLEMENTATION

Hidden-literal-based, predicate, and blocked clause elimination all admit efficient implementations in a superposition prover. In this section, we describe how to implement the first two sets of techniques. For BCE, we refer to Kiesel et al. [16]. All techniques are implemented in the Zipperposition prover [31]. Zipperposition is designed for fast prototyping of improvements to superposition, but it implements many of the most successful heuristics from the E prover [32] and has recently become quite competitive [33].

A. Hidden-Literal-Based Elimination

For HLBE, an efficient representation of $HL(L, N)$ is crucial. Because this set may be infinite, we underapproximate it by restricting the length of the transitive chains via a parameter K_{len} . Given the current clause set N , the finite map $\text{Imp}[L']$ associates with each literal L' a set of pairs (L, M) such that $L' \xleftrightarrow{k} L$, where $k \leq K_{\text{len}}$ and M is the multiset of clauses used to derive $L' \xleftrightarrow{k} L$. Moreover, we consider only transitions of type (1) (as per Definition 4). The following algorithm maintains Imp dynamically, updating it as the prover derives and deletes clauses. It depends on the global variable Imp and the parameters K_{len} and K_{imp} .

procedure ADDIMPLICATION(L_a, L_c, C)

if $\text{Imp}[L_a\sigma] \neq \emptyset$ for some renaming σ **then**
 $(L_a, L_c) \leftarrow (L_a\sigma, L_c\sigma)$

if there are no L', M, σ such that $(L', M) \in \text{Imp}[L]$,

- 5 $L\sigma = L_a$, and $L'\sigma = L_c$ **then**

for all (σ, M) such that $(L_c\sigma, M) \in \text{Imp}[L_a\sigma]$ **do**
 erase all (L', M') such that $M \subseteq M'$ from $\text{Imp}[L_a\sigma]$

for all L such that $(L', M) \in \text{Imp}[L]$

```

    and  $L_a\sigma = L'$  for some  $\sigma$  do
10  if  $|M| < K_{\text{len}}$  then
     $\text{Imp}[L] \leftarrow \text{Imp}[L] \cup \{(L_c\sigma, M \uplus \{C\})\}$ 
    for all  $L$  such that  $\text{Imp}[L] \neq \emptyset$ 
    and  $L\sigma = L_c$  for some  $\sigma$  do
     $\text{Concl} \leftarrow \{(L'\sigma, M \uplus \{C\}) \mid$ 
15   $(L', M) \in \text{Imp}[L], |M| < K_{\text{len}}\}$ 
     $\text{Imp}[L_a] \leftarrow \text{Imp}[L_a] \cup \text{Concl}$ 
     $\text{Congr} \leftarrow \{(s \approx t, \{C\}) \mid \exists u. L_c = u[s] \approx u[t]\}$ 
     $\text{Imp}[L_a] \leftarrow \text{Imp}[L_a] \cup \{(L_c, \{C\})\} \cup \text{Congr}$ 

procedure TRACKCLAUSE( $C$ )
20  if  $C = L_1 \vee L_2$  then
    ADDIMPLICATION( $\neg L_1, L_2, C$ )
    ADDIMPLICATION( $\neg L_2, L_1, C$ )
    if  $L_2 = \neg L_1\sigma$  for some nonidempotent  $\sigma$  then
    for all  $i \leftarrow 1$  to  $K_{\text{imp}}$  do
25   $L_2 \leftarrow L_2\sigma$ 
    ADDIMPLICATION( $\neg L_1, L_2, C$ )

procedure UNTRACKCLAUSE( $C$ )
    for all  $L_a, L_c, M$  such that  $(L_c, M) \in \text{Imp}[L_a]$  do
    if  $C \in M$  then
30  erase  $(L_c, M)$  from  $\text{Imp}[L_a]$ 

```

The algorithm views a clause $L \vee L'$ as two implications $\neg L \rightarrow L'$ and $\neg L' \rightarrow L$. It stores only one entry for all literals equal up to variable renaming (line 2). Each implication $L_a \rightarrow L_c$ represented by the clause is stored only if its generalization is not present in Imp (line 4). Conversely, all instances of the implication are removed (line 6).

Next, the algorithm finds each implication stored in Imp that can be linked to $L_a \rightarrow L_c$: Either L_c becomes the new consequent (line 9) or L_a becomes the new antecedent (line 13). If L_c can be decomposed into $u[s] \approx u[t]$, rule (3) of Definition 4 allows us to store $s \approx t$ in $\text{Imp}[L_a]$ (line 18). This is an exception to the idea that transitive chains should use only rule (1). The application of rule (3) does not count toward the bound K_{len} . If L_a is of the form $u[s] \approx u[t]$, then Imp could be extended so that $\text{Imp}[s \approx t] = \text{Imp}[L_a]$, but this would substantially increase Imp 's memory footprint.

In first-order logic, different instances of the same clause can be used along a transitive chain. For example, the clause $C = \neg p(x) \vee p(f(x))$ induces $p(x) \leftrightarrow^i p(f^i(x))$ for all i . The algorithm discovers such self-implications (line 23): For each clause C of the form $\neg L \vee L\sigma$, where σ is some nonidempotent substitution, the entire $(L\sigma^2, \{C\}), \dots, (L\sigma^{K_{\text{imp}}+1}, \{C\})$ are added to $\text{Imp}[L]$, where K_{imp} is a parameter.

To track and untrack clauses efficiently, we implement the mapping Imp as a nonperfect discrimination tree [34]. Given a query literal L , this indexing data structure efficiently finds all literals L' such that for some σ , $L'\sigma = L$ and $\text{Imp}[L'] \neq \emptyset$. We can use it to optimize all lookups except the one on line 9. For this remaining lookup, we add an index Imp^{-1} that inverts Imp , i.e., $\text{Imp}^{-1}[L] = \{L' \mid \text{Imp}[L'] = (L, M) \text{ for some } M\}$. To avoid sequentially going through all entries in Imp when the prover deletes them, for each clause C we keep track of each

literal L such that C appears in $\text{Imp}[L]$. Finally, we limit the number of entries stored in $\text{Imp}[L]$ – by default, up to 48 pairs in each $\text{Imp}[L]$ are stored.

Rules HLE and HTR have a simple implementation based on Imp lookups. To implement UNITHLE and UNITHTR, we maintain the index Unit , containing literals $L_c\sigma$, such that $(L_c, M) \in \text{Imp}[L_a]$ for some M and L_a and σ is the most general unifier of L' and L_a , for some unit clause $\{L'\}$. The implementation of FLE and FLR also uses Unit : When (L', M) is added to $\text{Imp}[L]$, we check if $(\neg L', M') \in \text{Imp}[L]$ for some M' . If so, $\neg L$ is added to Unit .

In propositional logic, the conventional approach constructs the *binary implication graph* for the clause set N [4], with edges $(\neg L, L')$ and $(\neg L', L)$ whenever $L \vee L' \in N$. To avoid traversing the graph repeatedly, solvers rely on timestamps to discover connections between literals. This relies on syntactic literal comparisons, which is very fast in propositional logic but not in first-order logic, because of substitutions and congruence.

B. Predicate Elimination

To implement portfolio predicate elimination, we maintain a record for each predicate symbol p occurring in the problem with the following fields: set of definition clauses for p , set of nondefinition clauses in which p occurs once, and set of clauses in which p occurs more than once. These records are kept in a priority queue, prioritized by properties such as presence of definition sets and number of estimated resolutions. If p is the highest-priority symbol that is eligible for SPE or DPE, we eliminate it by removing all the clauses stored in p 's record from the proof state and by adding flat resolvents to the passive set. Eliminating a symbol might make another symbol eligible.

As an optimization, predicate elimination keeps track only of symbols that appear at most K_{occ} times in the clause set. For inprocessing, we use signals that the prover emits whenever a clause is added to or removed from the proof state and update the records. At the beginning of the 1st, $(K_{\text{iter}} + 1)$ st, $(2K_{\text{iter}} + 1)$ st, ... iteration of the given clause procedure's loop body, predicate elimination is systematically applied to the entire proof state. The first application of inprocessing amounts to preprocessing. By default, $K_{\text{occ}} = 512$ and $K_{\text{iter}} = 10$. The same ideas and limits apply for blocked clause elimination.

The most important novel aspect of our predicate elimination implementation is recognizing the definition clauses for symbol p in a clause set N , which is performed as follows:

- 1) Let $G = \{C \mid C = (\neg)p(\vec{x}) \vee C', C \in N, \text{no variable repeats in } \vec{x}, \text{ and variables of } C' \text{ are among } \vec{x}\}$. If G is empty, report failure; otherwise continue.
- 2) Rename all clauses in G so that their only variables are \vec{x} .
- 3) Let $[a]$ be a function that assigns a propositional variable to each atom a . This function is lifted to literals by assigning $[\neg a] = \neg x$, if $[a] = x$, and to clauses pointwise. Furthermore, let $E = \{[C'] \mid (\neg)p(\vec{x}) \vee C' \in G\}$. If E is satisfiable, report failure. Else, let E' be the unsatisfiable

core of E and G' the set of corresponding first-order clauses and continue.

- 4) If all resolvents in $G'_p \times_p G'_{\neg p}$ are tautologies, then G' is the definition set for symbol p . Else, report failure.

The invalidity of set E from step 3 is checked using a SAT solver, which is already integrated in Zipperposition. As modern theorem provers (such as E or Vampire) also use SAT solvers, the method can easily be implemented.

During experimentation, we noticed that recognizing definitions of symbols that occur in the conjecture often harms performance. Thus, Zipperposition recognizes definitions only for the remaining symbols.

VII. EVALUATION

We measure the impact of our elimination techniques for various values of their parameters. As a baseline, we use Zipperposition's first-order portfolio mode, which runs the prover in 13 configurations of heuristic parameters in consecutive time slices. None of these configurations use our new techniques. To evaluate a given parameter value, we fix it across all 13 configurations and compare the results with the baseline.

The benchmark set consists of all 13495 CNF and FOF TPTP 7.3.0 theorems [17]. The experiments were carried out on StarExec servers [35] equipped with Intel Xeon E5-2609 CPUs clocked at 2.40 GHz. The portfolio mode uses a single CPU core with a CPU time limit of 180 s. The base configuration solves 7897 problems. The values in the tables indicate the number of problems solved minus 7897. Thus, positive numbers indicate gains over the baseline. The best result is shown in bold.

A. Hidden-Literal-Based Elimination

The first experiments use all implemented HLBE rules. To avoid overburdening Zipperposition, we can enable an option to limit the number of tracked clauses for hidden literals. Once the limit has been reached, any request for tracking a clause will be rejected until a tracked clause is deleted. We can choose which kind of clauses are tracked: only clauses from the active set \mathcal{A} , only clauses from the passive set \mathcal{P} , or both. We also vary the maximal implication chain length K_{len} and the number of computed self-implications K_{imp} .

In Zipperposition, every lookup for instances or generalizations of $s \approx t$ must be done once for each orientation of the equation. To avoid this inefficiency, and also because the implementation of hidden literals does not fully exploit congruence, we can disable tracking clauses with at least one functional literal. Clauses containing functional literals can then still be simplified.

Figures 1 and 2 show the results, without and with functional literal tracking enabled, for $K_{\text{len}} = 2$ and $K_{\text{imp}} = 0$. The columns specify different limits on the number of tracked clauses, with ∞ denoting that no limit is imposed. The rows represent different kinds of tracked clauses. The results suggest that tracking functional literals is not worth the effort but that tracking predicate literals is. The best improvement is observed when both active and passive clauses are tracked. Normally

	Tracked clauses			
	250	500	1000	∞
Active	-14	-16	-8	-12
Passive	+7	+10	+5	-35
Both	+12	+10	+7	-45

Fig. 1. Impact of the number and kinds of tracked clauses on HLBE performance, when only predicate literals are tracked

	Tracked clauses			
	250	500	1000	∞
Active	-10	-14	-8	-18
Passive	-5	-5	-14	-71
Both	+2	-1	-8	-79

Fig. 2. Impact of the number and kinds of tracked clauses on HLBE performance, when all literals are tracked

DISCOUNT-loop provers [26] such as Zipperposition do not simplify active clauses using passive clauses, but here we see that this can be effective. Figure 3 shows the impact of varying K_{len} and K_{imp} , when 500 clauses from the entire proof state are tracked. These results suggest that computing long implication chains is counterproductive.

B. Predicate and Blocked Clause Elimination

For defined predicate elimination, the number of resolvents grows exponentially with the number of occurrences of p . To avoid this expensive computation, we limit the applicability of PPE to proof states for which p is singular. According to our informal experiments, full PPE, without this restriction, generally performs less well.

Predicate elimination can be done using Khasidashvili and Korovin's criterion (K&K) or using our relaxed criterion with different values of K_{tol} . Figure 4 shows the results for SPE and PPE used as preprocessors. Our numbers corroborate Khasidashvili and Korovin's findings: SPE with K&K proves 70 more problems than the base, a 0.9% increase, comparable to the 1.8% they observe when they combine SPE with additional preprocessing. Remarkably, the number of additional proved problems more than doubles when we use our criterion with $K_{\text{tol}} > 0$, for both SPE and PPE.

Although this is not evident in Figure 4, varying K_{tol} substantially changes the set of problems solved. For example, when $K_{\text{tol}} = 0$, SPE proves 60 theorems not proved using $K_{\text{tol}} = 50$. The effect weakens as K_{tol} grows. When $K_{\text{tol}} = 100$, SPE proves only 13 problems not found when $K_{\text{tol}} = 200$. Similarly, the set of problems proved by SPE and PPE differs: When $K_{\text{tol}} = 25$, 14 problems are proved by PPE but missed by SPE. Recognizing definition sets is useful: PPE outperforms SPE regardless of the criterion.

Performing BCE and variable elimination until fixpoint increases the performance of SAT solvers [14]. We can check whether the same holds for superposition provers. In this experiment, we use the relaxed criterion with $K_{\text{tol}} = 25$ and HLBE which tracks up to 500 clauses from any clause set, $K_{\text{len}} = 2$, and $K_{\text{imp}} = 0$. We use each technique as preprocessing and inprocessing.

	Chain length K_{len}			
	1	2	4	8
$K_{imp} = 0$	+9	+10	+7	+5
$K_{imp} = 1$	+5	+11	+7	+4
$K_{imp} = 2$	+6	+11	+8	+8

Fig. 3. Impact of the parameters K_{len} and K_{imp} on HLBE performance

	K&K	Relaxed with K_{tol}				
		0	25	50	100	200
SPE preproc.	+70	+117	+154	+160	+154	+158
PPE preproc.	+71	+124	+160	+164	+165	+162

Fig. 4. Impact of the choice of criterion on predicate elimination performance

The results are summarized in Figure 5, where the + sign denotes the combination of techniques. We confirm the results obtained by Kiesl et al. about the performance of BCE as preprocessing: It helps prove 30 more problems from our benchmark set, increasing the success rate by roughly 0.4%. The same percentage increase was obtained Kiesl et al. Using BCE as inprocessing, however, hurts performance, presumably because of its incompatibility with the redundancy criterion.

For preprocessing, the combinations SPE+BCE and PPE+BCE performed roughly on a par with SPE and PPE, respectively. This stands in contrast to the situation with SAT solvers, where such a combination usually helps. It is also worth noting that the inprocessing techniques never outperform their preprocessing counterparts. The last column shows that combining HLBE with other elimination techniques overburdens the prover.

C. Satisfiability by Blocked Clause Elimination

Kiesl et al. found that blocked clause elimination is especially effective on satisfiable problems. To corroborate their results and ascertain whether a combination of predicate elimination and blocked clause elimination increases the success rate, we evaluate BCE on all 2273 satisfiable or TPTP FOF and CNF problems. The hardware and CPU time limits are the same as in the experiments above. Figure 6 presents the results.

The baseline establishes the satisfiability of 856 problems. We consider only preprocessing techniques, since BCE compromises refutational completeness—a saturation does not guarantee that the original problem was satisfiable. We note that recognizing definition sets makes almost no difference on satisfiable problems. The sets of problems solved by BCE and PPE differ—30 problems are solved by BCE and not by PPE.

VIII. CONCLUSION

We adapted several preprocessing and inprocessing elimination techniques implemented in modern SAT solvers so that they work in a superposition prover. This involved lifting the techniques to first-order logic with equality but also tailoring them to work in tandem with superposition and its redundancy criterion. Although SAT solvers and superposition provers embody radically different philosophies, we found that the lifted SAT techniques provide valuable optimizations.

	BCE	SPE	SPE +BCE	PPE	PPE +BCE	HLBE +PPE +BCE
Preprocessing	+30	+154	+159	+160	+166	+162
Inprocessing	−48	+140	+127	+146	+131	+127

Fig. 5. Performance of predicate and blocked clause elimination

	BCE	SPE	SPE +BCE	PPE	PPE +BCE	HLBE +PPE +BCE
Preprocessing	+29	+46	+60	+47	+59	+55

Fig. 6. Performance of predicate and blocked clause elimination for establishing satisfiability

We see several avenues for future work. First, the implementation of hidden literals could be extended to exploit equality congruence. Second, although inprocessing blocked clause elimination is incomplete in general, we hope to achieve refutational completeness for a substantial fragment of it. Third, predicate and blocked clause elimination, which thrives on the absence of clauses from the proof state, could be enhanced by tagging and ignoring generated clauses that have not yet been used to subsume or simplify untagged clauses. Fourth, predicate and blocked clause elimination could be extended to work with functional literals. Fifth, more SAT techniques could be adapted, including bounded variable addition [36] and blocked clause addition [37]. Sixth, the techniques we covered could be adapted to work with other first-order calculi, or generalized further to work with higher-order calculi such as combinatory superposition [38] and λ -superposition [39].

A. Acknowledgment

We are grateful to the maintainers of StarExec for letting us use their service. Uwe Waldmann participated in the search for a counterexample to completeness of BCE as inprocessing and confirmed that Example 23 is correct. He also suggested major simplifications and helped us debug the proofs of the claims about predicate elimination. Anne Baanen helped us define the nonclassical logic used to disallow splitting. Ahmed Bhayat, Armin Biere, Mathias Fleury, Benjamin Kiesl, and the anonymous reviewers made some useful comments on our manuscript, and Mark Summerfield suggested many textual improvements. We thank them all.

Vukmirović and Blanchette’s research has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Blanchette’s research has also received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward). Heule is supported by the National Science Foundation (NSF) under grant CCF-2015445.

REFERENCES

- [1] J. P. Marques-Silva, I. Lynce, and S. Malik, “Conflict-driven clause learning SAT solvers,” in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 185, pp. 131–153.
- [2] L. Bachmair and H. Ganzinger, “Rewrite-based equational theorem proving with selection and simplification,” *J. Log. Comput.*, vol. 4, no. 3, pp. 217–247, 1994.
- [3] M. J. H. Heule, M. Järvisalo, and A. Biere, “Clause elimination procedures for CNF formulas,” in *LPAR-17*, ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Springer, 2010, pp. 357–371.
- [4] —, “Efficient CNF simplification based on binary implication graphs,” in *SAT 2011*, ser. LNCS, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 201–215.
- [5] J. W. Freeman, “Improvements to propositional satisfiability search algorithms,” Ph.D. dissertation, University of Pennsylvania, 1995.
- [6] M. Davis and H. Putnam, “A computing procedure for quantification theory,” *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.
- [7] S. Subbarayan and D. K. Pradhan, “NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances,” in *SAT 2004*, ser. LNCS, H. H. Hoos and D. G. Mitchell, Eds., vol. 3542. Springer, 2004, pp. 276–291.
- [8] P. Chatalic and L. Simon, “ZRES: The old Davis–Putnam procedure meets ZBDD,” in *CADE-18*, ser. LNCS, D. A. McAllester, Ed., vol. 1831. Springer, 2000, pp. 449–454.
- [9] A. Biere, “Resolve and expand,” in *SAT 2004*, ser. LNCS, H. H. Hoos and D. G. Mitchell, Eds., vol. 3542. Springer, 2004, pp. 59–70.
- [10] N. Eén and A. Biere, “Effective preprocessing in SAT through variable and clause elimination,” in *SAT 2005*, ser. LNCS, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75.
- [11] D. M. Gabbay and H. J. Ohlbach, “Quantifier elimination in second-order predicate logic,” in *KR ’92*, B. Nebel, C. Rich, and W. R. Swartout, Eds. Morgan Kaufmann, 1992, pp. 425–435.
- [12] H. J. Ohlbach, “SCAN—elimination of predicate quantifiers,” in *CADE-13*, ser. LNCS, M. A. McRobbie and J. K. Slaney, Eds., vol. 1104. Springer, 1996, pp. 161–165.
- [13] Z. Khasidashvili and K. Korovin, “Predicate elimination for preprocessing in first-order theorem proving,” in *SAT 2016*, ser. LNCS, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 361–372.
- [14] M. Järvisalo, A. Biere, and M. Heule, “Blocked clause elimination,” in *TACAS 2010*, ser. LNCS, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 129–144.
- [15] A. Biere, F. Lonsing, and M. Seidl, “Blocked clause elimination for QBF,” in *CADE-23*, ser. LNCS, N. Bjørner and V. Sofronie-Stokkermans, Eds., vol. 6803. Springer, 2011, pp. 101–115.
- [16] B. Kiesel, M. Suda, M. Seidl, H. Tompits, and A. Biere, “Blocked clauses in first-order logic,” in *LPAR-21*, ser. EPiC Series in Computing, T. Eiter and D. Sands, Eds., vol. 46. EasyChair, 2017, pp. 31–48.
- [17] G. Sutcliffe, “The TPTP problem library and associated infrastructure—from CNF to TH0, TPTP v6.4.0,” *J. Autom. Reason.*, vol. 59, no. 4, pp. 483–502, 2017.
- [18] P. Vukmirović, J. Blanchette, and M. J. H. Heule, “SAT-inspired eliminations for superposition (technical report),” Technical report, 2021, https://matryoshka-project.github.io/pubs/satelimsup_report.pdf.
- [26] J. Avenhaus, J. Denzinger, and M. Fuchs, “DISCOUNT: A system for distributed equational deduction,” in *RTA-95*, ser. LNCS, J. Hsiang, Ed., vol. 914. Springer, 1995, pp. 397–402.
- [19] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Wiley, 1987.
- [20] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.
- [21] L. Bachmair and H. Ganzinger, “Resolution theorem proving,” in *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds. Elsevier and MIT Press, 2001, vol. I, pp. 19–99.
- [22] A. Riazanov and A. Voronkov, “Splitting without backtracking,” in *IJCAI 2001*, B. Nebel, Ed. Morgan Kaufmann, 2001, pp. 611–617.
- [23] A. Fietzke and C. Weidenbach, “Labelled splitting,” *Ann. Math. Artif. Intell.*, vol. 55, no. 1–2, pp. 3–34, 2009.
- [24] A. Voronkov, “AVATAR: The architecture for first-order theorem provers,” in *CAV 2014*, ser. LNCS, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 696–710.
- [25] W. McCune and L. Wos, “Otter—the CADE-13 competition incarnations,” *J. Autom. Reason.*, vol. 18, no. 2, pp. 211–220, 1997.
- [27] A. Nonnengart and C. Weidenbach, “Computing small clause normal forms,” in *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds. Elsevier and MIT Press, 2001, vol. I, pp. 335–367.
- [28] M. Heule, M. Seidl, and A. Biere, “A unified proof system for QBF preprocessing,” in *IJCAR 2014*, ser. LNCS, S. Demri, D. Kapur, and C. Weidenbach, Eds., vol. 8562. Springer, 2014, pp. 91–106.
- [29] B. Kiesel and M. Suda, “A unifying principle for clause elimination in first-order logic,” in *CADE-26*, ser. LNCS, L. de Moura, Ed., vol. 10395. Springer, 2017, pp. 274–290.
- [30] M. Fitting, *First-Order Logic and Automated Theorem Proving*, 2nd ed., ser. Graduate Texts in Computer Science. Springer, 1996.
- [31] S. Cruanes, “Superposition with structural induction,” in *FroCoS 2017*, ser. LNCS, C. Dixon and M. Finger, Eds., vol. 10483. Springer, 2017, pp. 172–188.
- [32] S. Schulz, S. Cruanes, and P. Vukmirović, “Faster, higher, stronger: E 2.3,” in *CADE-27*, ser. LNCS, P. Fontaine, Ed., vol. 11716. Springer, 2019, pp. 495–507.
- [33] G. Sutcliffe, “The CADE-27 Automated Theorem Proving System Competition—CASC-27,” *AI Commun.*, vol. 32, no. 5–6, pp. 373–389, 2020.
- [34] I. V. Ramakrishnan, R. C. Sekar, and A. Voronkov, “Term indexing,” in *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001, vol. II, pp. 1853–1964.
- [35] A. Stump, G. Sutcliffe, and C. Tinelli, “StarExec: A cross-community infrastructure for logic solving,” in *IJCAR 2014*, ser. LNCS, S. Demri, D. Kapur, and C. Weidenbach, Eds., vol. 8562. Springer, 2014, pp. 367–373.
- [36] N. Manthey, M. Heule, and A. Biere, “Automated reencoding of Boolean formulas,” in *HVC 2012*, ser. LNCS, A. Biere, A. Nahir, and T. E. J. Vos, Eds., vol. 7857. Springer, 2012, pp. 102–117.
- [37] O. Kullmann, “On a generalization of extended resolution,” *Discr. Appl. Math.*, vol. 96–97, pp. 149–176, 1999.
- [38] A. Bhayat and G. Reger, “A combinator-based superposition calculus for higher-order logic,” in *IJCAR 2020, Part I*, ser. LNCS, N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12166. Springer, 2020, pp. 278–296.
- [39] A. Bentkamp, J. Blanchette, S. Tourret, P. Vukmirović, and U. Waldmann, “Superposition with lambdas,” *J. Autom. Reason.*, to appear.

SAT Solving in the Serverless Cloud

Alex Ozdemir[§] , Haoze Wu[§] , and Clark Barrett[§] 

Stanford University, USA.

{aozdemir, haozewu, barrett}@cs.stanford.edu

Abstract—In recent years, cloud service providers have sold computation in increasingly granular units. Most recently, “serverless” executors run a single executable with restricted network access and for a limited time. The benefit of these restrictions is scale: thousand-way parallelism can be allocated in seconds, and CPU time is billed with sub-second granularity. To exploit these executors, we introduce **gg-SAT**: an implementation of divide-and-conquer SAT solving. Infrastructurally, **gg-SAT** departs substantially from previous implementations: rather than handling process or server management itself, **gg-SAT** builds on the **gg** framework, allowing computations to be executed on a configurable backend, including serverless offerings such as AWS Lambda. Our experiments suggest that when run on the same hardware, **gg-SAT** performs competitively with other D&C solvers, and that the 1000-way parallelism it offers (through AWS Lambda) is useful for some challenging SAT instances.

Index Terms—parallel SAT, serverless computing, divide and conquer.

I. INTRODUCTION

Modern Boolean satisfiability (SAT) solvers have been successfully applied to important practical and theoretical domains, such as hardware verification, planning, and mathematics. Progress in the scalability of these tools has come from both algorithmic improvements and better leveraging of multi-processing hardware. While the number of processors on a single machine is limited, and maintaining a warm cluster to run occasional tasks is expensive, cloud-computing is a promising approach for leveraging on-demand parallelism at low cost.

Recent cloud-computing services are offered at increasingly fine granularity and low latency. Instead of renting a server or a cluster, one can now rent state-free executors, which can be rapidly and plentifully provisioned at a low price—a paradigm referred to as *serverless computing*. Serverless executors generally have restricted network access, limited memory, and limited runtime. For example, Amazon’s Lambda service rents a Linux container to run arbitrary x86-64 executables for up to 15 minutes, with less than a second of startup time and no charge when idle. Similar services are offered by Google, Microsoft, Alibaba, and IBM. Previous research has used serverless computing as a “burstable supercomputer” for video processing [2], neural network training [25], and more [13]–[15], [33]. These successes beg the question: “can serverless computing be leveraged for massively parallel SAT-solving?”

There are two traditional parallel SAT-solving paradigms: 1) the portfolio approach, where each thread runs a different

SAT solver on the same instance; and 2) the divide-and-conquer (D&C) approach, where a problem is partitioned into independent sub-problems to be solved in parallel. While the former approach in combination with clause-sharing leads to surprisingly good performance for small portfolio sizes, the benefits decrease as parallel computing power increases, and this approach is also not well aligned with the runtime and communication limitations of serverless executors. In this paper, we follow the second approach and present **gg-SAT**, a divide-and-conquer (D&C) SAT solver compatible with serverless computing. **gg-SAT** makes black-box use of a *solver* (e.g., CaDiCaL [8]) and a *divider* (e.g., march [28]) to solve and partition the problems, respectively. Problem division is performed throughout the search, whenever a sub-problem reaches a timeout imposed by either the user or the cloud-service. Infrastructurally, **gg-SAT** differs substantially from previous D&C implementations: rather than handling process or server management itself, **gg-SAT** builds on top of the **gg** framework for parallel computation. By expressing D&C search using **gg**, **gg-SAT** can execute that search on any mixture of user-specified backends; supported backends currently include local processes, remote machines, and serverless cloud-services such as AWS Lambda and Google Cloud Functions. To implement **gg-SAT**, we designed and built **pygg**, a novel and idiomatic Python interface to **gg**. We expect that **pygg** will be independently useful for other future projects, perhaps including parallel SMT solving.

We evaluate **gg-SAT** using local processes and AWS Lambda as backends. Local experiments suggest that **gg-SAT** performs competitively with the original Cube-and-Conquer prototype [19], a recent reimplement of it [18], and a portfolio solver **PLingeling** [7], on benchmarks taken from [18], [19]. Cloud experiments suggest that **gg-SAT** unlocks levels of parallelism which are useful for solving some challenging instances from the 2020 SAT Competition.

II. BACKGROUND & RELATED WORK

A. Parallel SAT

Propositional satisfiability is an old problem; we refer the reader to the handbook of satisfiability [9] for an introduction. Parallel SAT-solving also has a lengthy history, with two main approaches.

The first approach is *portfolio solving*, pioneered in [16], [22], [34]. In a portfolio solver, each thread runs a different solver or configuration on the same original formula. An instance is solved as quickly as the best individual solver for that instance. Portfolio solvers include: ManySAT

[§]Equal contribution

[17], CryptoMinisat [32], PLingeling [7], Syrup [3], HordSAT [6], and Painless [26]. Some portfolio solvers also use *clause sharing* [11], [31]: sharing learnt clauses among the different solvers.

Another approach to parallelizing SAT is *divide-and-conquer* (D&C). D&C solvers attempt to divide a SAT instance into easier SAT instances, which can then be solved in parallel by a base solver. Typically, D&C solvers divide instances by partitioning the search space. The important questions—how and when to divide—are answered heuristically, typically with heuristics derived from look-ahead solvers and CDCL solvers. There has been substantial work on D&C SAT solving [10], [23], [24], including: Psato [35], Painless [27], and AMPHAROS [29]. One prominent approach, “cube-and-conquer” [19] uses a lookahead solver to divide instances and a CDCL solver to solve subproblems; this approach has been successful for large mathematical problems [21].

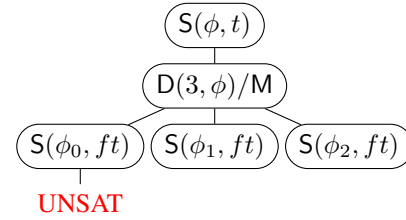
B. Distributed SAT

A number of systems attempt parallel SAT solving using a cluster of computers, possibly rented from the cloud. Most of these systems (Qsat [30], HordSAT [6], TopoSAT [12], SLIME [20]) follow the portfolio approach. One recent system (Paracooba [18]) follows the D&C approach. All of these systems operate in the “cluster” computational model, in which long-running processes on each node communicate over the network.

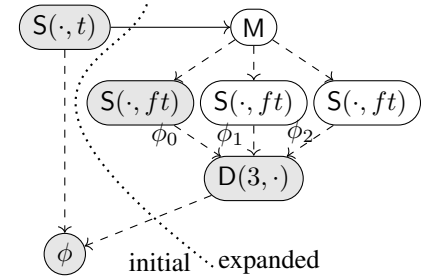
C. Serverless Computing

Cloud service providers, such as Microsoft Azure, rent out computational resources including compute, storage, and accelerators. Over the past decade, service providers have rented compute with increasing granularity, scale, and availability. Their recent offerings include *serverless* services, which run a single executable for a limited time, with limited memory and restricted network access. While restricted, serverless computing has strengths: it offers massive parallelism that can be rapidly provisioned, with fine-grained billing. For example, AWS Lambda [4] runs executables for up to 15 minutes, with 3GB of memory and 500MB of disk space; the runs are billed at sub-second granularity, and a thousand executors can be provisioned in seconds.

While serverless computing was designed for operational convenience, recent work has explored using it as a “burstable supercomputer-on-demand” [13], for tasks such as video processing [2], ray tracing [14], and machine learning [25]. One system, gg [13], provides a general framework for leveraging minimal executors (including serverless ones). It uses a configurable backend (such as a local machine, remote machines, or serverless executors) to evaluate a programmer-defined dependency graph of *thunks*: programs that take files as inputs. Thunks can output files or new thunks; the latter causes the dependency graph to dynamically grow. Dynamic dependency graphs can express many applications; gg has been used for tasks such as neural network verification [33], compilation [13], and video encoding [15].



(a) The D&C search tree. ϕ 's solve query times out and is split into three sub-problems, one of which has been solved.



(b) The gg dependency graph. Dashed arrows denote dependencies; if a node produces multiple outputs, the dependency edges are labelled. The solid arrow denotes a thunk that returns another thunk. Shaded thunks have been evaluated.

Fig. 1: A D&C search snapshot and its corresponding dependency graph. In both diagrams, S, M, and D denote solve, merge, and divide, respectively.

III. DESCRIPTION

A. Algorithm

gg-SAT uses a D&C algorithm with multiplicatively growing timeouts. It is parameterized by a *base solver* and a *divider*. The base solver can be any SAT solver. The divider's job is to partition a problem into a requested number of sub-problems such that the disjunction of the sub-problems is equisatisfiable with the original problem. Other parameters to the algorithm include the timeout t , the timeout growth factor f , the number of initial partitions p_i , and the number of partitions for each sub-problem, p_s .

Figure 1a illustrates the solving of formula ϕ as a tree, with $p_i = 1$ and $p_s = 3$. The number of initial divisions is 1, so the base solver first attempts the original problem ϕ with timeout t . This times out, so the divider runs and splits ϕ into sub-problems (ϕ_0, ϕ_1, ϕ_2) , each of which is attempted with timeout ft . The sub-problem ϕ_0 is determined to be UNSAT; other sub-problems have yet to be solved, and may be divided again. The process ends when all sub-problems are determined to be UNSAT or any sub-problem is determined to be SAT.

B. Implementation

To apply D&C to SAT, we must instantiate its primitive notions (sub-problems, solving, and dividing) for SAT. We follow previous work [19], [24] by using a lookahead solver (*march*) to build sub-problems described by *cubes* (lists of asserted literals) and by using a CDCL solver (CaDiCaL [8]) to attempt to solve problems and sub-problems. *march* can

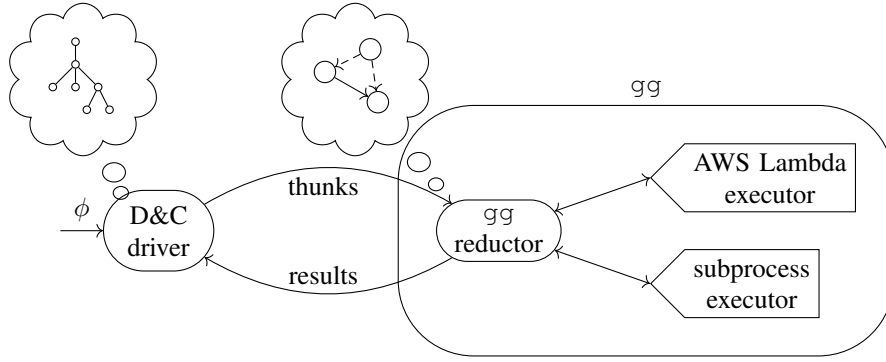


Fig. 2: `gg-SAT` expresses D&C search as a dynamically expanding dependency graph and uses `gg` to evaluate that graph using a back-end of the user’s choice.

produce a large number of cubes (e.g., millions) and can take a long time. This was appropriate for cube-and-conquer (which ran `march` exactly once per problem) but is inappropriate for divide-and-conquer (which runs `march` many times seeking a small number of sub-problems each time). To address this, we configure `march` with a maximum cube length, which substantially reduces its runtime.

Our D&C implementation uses the `gg` framework for parallel execution [13]. Recall (§II) that using `gg` requires the computation to be expressed as a dependency graph of *thunks*, each of which is an individual executable. For D&C, there are three kinds of *thunks*. *Solve thunks* run the base solver; if it returns a result, the thunk returns that result as well; otherwise, the solve thunk returns a *merge thunk*, which combines the solutions to sub-problems that are produced by a *divide thunk*, which runs the divider. Figure 1 illustrates the relationship between an in-progress D&C search and the `gg` dependency graph. When D&C attempts to solve $S(\phi, t)$, the dependency graph contains only the nodes left of the dotted line. However, when that query times out, the corresponding thunk returns 5 new *thunks*: a divide thunk to create 3 sub-problems, three solve *thunks* to (attempt to) solve them, and a merge thunk, whose output should be taken as the output of the original S thunk.

By expressing D&C search as a `gg` dependency graph, we can use `gg` to execute that search using a back-end (or combination of back-ends) of the user’s choice. Figure 2 visualizes the different runtime components of the system. Our driver translates the D&C search tree into a graph. The reductor analyzes this graph, searching for *thunks* whose dependencies are fully evaluated; it sends these to a configured backend. When an executor returns values or subgraphs, the reductor updates its graph. When the graph is reduced to a single value, the reductor returns that value to the driver. For more details about the execution process, see [13].

To ease the development of `gg-SAT`, we built `pygg`, a python library for building dynamic `gg` dependency graphs. While `gg` is conceptually simple, using it typically requires programmers to write many different shell scripts for tasks such as embedding values in the `gg` graph, creating different

kinds of *thunks*, and reformatting files for different solvers. With `pygg`, the entire computation can be expressed as a single python script. Different kinds of *thunks* are just different python functions, each of which can return basic python values, one or more files, or the output of some combination of other *thunks*. With `pygg`, our D&C implementation fits in a single python script of less than 200 lines. `pygg` has been merged upstream into the `gg` project.

IV. EXPERIMENTS

`gg-SAT` is the first SAT solver targeting serverless computation, so we cannot compare with previous tools on our infrastructure of interest. Nonetheless, we perform two experiments. First, we compare `gg-SAT` with other multithreaded solvers on a single multicore machine, to validate the general architecture and performance of `gg-SAT`. Second, we use 1000 serverless executors to attempt unsolved benchmarks from the SAT 2020 competition, showing the utility of the massive parallelism that `gg-SAT` unlocks.

A. Local experiment

We compare with the default configurations of three parallel solvers: 1) the original Cube-and-Conquer prototype (denoted `CnC`)¹ [19]; 2) `Paracooba`² [18], a recent Cube-and-Conquer re-implementation that is optimized for distributed computing; 3) `Treengeling`³ [8], a divide-and-conquer SAT solver; and 4) `PLingeling` [8], a state-of-the-art portfolio SAT solver. We evaluate on the benchmarks reported in [18], [19]. We run `gg-SAT` with $p_i = 64$, $p_s = 4$, $t = 10$, and $f = 1.5$, a set of parameters empirically determined to work well. For the other four solvers, we use the default parameters except that the number of threads is set to 64. Our testbed machines have two 2.70GHz Xeon Platinum 8280 CPUs, running CentOS 7. Each job is run with a 256 GB memory limit, and a 1-hour wall-clock timeout.

Table I shows the solvers’ wall-clock runtime for each benchmark. Given the small set of benchmarks, we can

¹<https://github.com/marijnheule/CnC/tree/ee8f8aab3729b46bc92dc>

²<https://github.com/maximal/Paracooba/tree/d905b67304eb780>

³<https://github.com/arminbiere/lingeling/tree/7d5db72420b95ab> (same for `PLingeling`)

TABLE I: Runtime (s) of gg-SAT, CnC, Paracooba, Treengeling, and PLingeling on the benchmarks reported in [18], [19]

benchmark	Result	gg-SAT	CnC	Paracooba	Treengeling	PLingeling
9dlx_vliw_at_b_iq8	UNSAT	850	—	966	—	155
9dlx_vliw_at_b_iq9	UNSAT	2830	—	1302	—	222
AProVE07-25	UNSAT	599	—	2091	1596	—
cruxmiter32.cnf	UNSAT	717	496	—	2078	—
dated-5-19-u	UNSAT	1723	436	1819	891	1030
eq.atree.braun.12	UNSAT	466	170	465	384	605
eq.atree.braun.13	UNSAT	3225	826	—	1615	1517
gss-24-s100	SAT	1166	—	—	1618	335
gss-26-s100	SAT	3509	—	—	560	—
gus-md5-14	—	—	—	—	—	—
ndhf_xits_09_UNK	UNSAT	948	—	—	—	1633
rbcl_xits_09_UNK	UNSAT	629	—	—	—	2965
rpoc_xits_09_UNK	UNSAT	331	—	—	—	1267
sortnet-8-ipc5-h19	SAT	—	—	3008	—	225
total-10-17-u	UNSAT	1098	388	919	310	666
total-5-15-u	UNSAT	—	1440	—	3253	—

draw only limited conclusions. Nonetheless, the results suggest gg-SAT’s performance is reasonable. It solves more benchmarks than the other three divide-and-conquer solvers, corroborating past research [1] that interleaving look-ahead with CDCL can be beneficial. It also solves more than PLingeling, suggesting that the divide-and-conquer approach can be preferable to the portfolio approach in some cases. Note, however, that each other solver can solve at least one benchmark that gg-SAT cannot, suggesting that the approaches are complementary.

B. Serverless experiment

Our second experiment demonstrates the utility of the thousand-way parallelism that gg-SAT makes convenient. We find that with this parallelism, gg-SAT can solve challenging instances that are out of reach for solvers running at lower levels of parallelism.

We sample 8 instances from the Cloud track of the SAT Competition 2020 [5], none of which were solved during the competition.⁴ As summarized in Table II, four of the five solvers from the previous section (using the same configurations) are unable to solve any of these instances within 4 hours. Treengeling solves one instance, Steiner-81-21-bce, in 9331 seconds. However, with gg-SAT running on AWS Lambda with 1000-way parallelism, we find that three instances: Steiner-81-21-bce, bv-term-small-rw_350.smt2, and mulhs16.smt2 are UNSAT in 2559, 1455, and 2866 seconds respectively. For AWS Lambda, we configure gg-SAT with $p_i = 1024$, $p_s = 8$, $t = 10$, and $f = 1.5$.⁵

⁴Steiner-81-21-bce, abw-I-ash85.mtx-w24, ccp-s8-facto4, bv-term-small-rw_350.smt2, Steiner-405-71-bce, mulhs16.smt2, LED_round_29-32_faultAt_29_fault_injections_5_seed_1579630418, PRESENT_round_1-32_faultAt_30_fault_injections_10_seed_1579630418

⁵Our experiment is incomparable with the results of the 2020 SAT cloud track. The competition environment differs substantially from our testbed; it uses 1600 cores, 20 minutes, and different hardware.

TABLE II: Solver performance on 8 hard instances from the SAT Competition 2020

Solver	Executor	Parallelism	Time Limit (h)	Solved
CnC	local threads	64	4	0
Paracooba	local threads	64	4	0
Treengeling	local threads	64	4	1
PLingeling	local threads	64	4	0
gg-SAT	local threads	64	4	0
gg-SAT	AWS Lambda	1000	1	3

V. DISCUSSION

We have presented gg-SAT, a parallel D&C SAT solver compatible with serverless-computing. gg-SAT is built on top of gg, an infrastructure for evaluating parallel computations. gg-SAT appears competitive with other parallel SAT solvers, and easily unlocks ad-hoc large-scale parallelism through execution on serverless cloud-services. This massive parallelism appears to be effective in solving some challenging instances. To implement gg-SAT, we also built pygg, a novel python interface to gg, which we hope will be useful for other applications, such as parallel SMT solving.




Future Work: gg-SAT itself could be substantially improved. Currently, its search strategy (e.g., how many sub-problems to create, when to re-divide) is independent of the number of idle workers and the number of unsolved problems. This can cause one of two undesirable dynamics: most workers sitting idle while a few tackle challenging sub-problems (that would ideally be immediately divided) or too much time being spent re-dividing (even though all workers are already busy). In the future, we hope to adjust the search strategy depending on the current workload of the system, dividing more when workers are idle, and less when they are not. We suspect that this will improve performance while also reducing the number of parameters for the system.

Other future directions for gg-SAT include proof-generation, new dividers, and trying to retain useful clauses from failed base solver attempts.

REFERENCES

- [1] T. Ahmed, O. Kullmann, and H. Snevily. On the van der Waerden numbers $w(2; 3, t)$. *Discrete Applied Mathematics*, 174:27–51, 2014.
- [2] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, 2018.
- [3] G. Audemard and L. Simon. Lazy clause exchange policy for parallel sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 197–205. Springer, 2014.
- [4] AWS lambda. <https://docs.aws.amazon.com/lambda/index.html>.
- [5] T. Balyo, N. Froleyks, M. J. Heule, M. Iser, M. Järvisalo, and M. Suda. Proceedings of sat competition 2020: Solver and benchmark descriptions. 2020.
- [6] T. Balyo, P. Sanders, and C. Sinz. Hordesat: A massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172. Springer, 2015.
- [7] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In T. Balyo, M. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.
- [8] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
- [9] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.
- [10] W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.
- [11] W. Chrabakh and R. Wolski. Gridsat: Design and implementation of a computational grid application. *Journal of Grid Computing*, 4(2):177, 2006.
- [12] T. Ehlers and D. Nowotka. Tuning parallel sat solvers. *Proceedings of Pragmatics of SAT*, 59:127–143, 2019.
- [13] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019*, pages 475–488, 2019.
- [14] S. Fouladi and B. Shacklett. R2-t2. <https://github.com/r2t2-project/r2t2>.
- [15] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalarao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, Mar. 2017. USENIX Association.
- [16] C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
- [17] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010.
- [18] M. Heisinger, M. Fleury, and A. Biere. Distributed cube and conquer with paracooba. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 114–122. Springer, 2020.
- [19] M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haija Verification Conference*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2011.
- [20] M. J. Heule, M. Järvisalo, and M. Suda. Proceedings of sat race 2019: Solver and benchmark descriptions. 2019.
- [21] M. J. H. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.
- [22] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.
- [23] A. E. Hyvärinen, T. Junttila, and I. Niemelä. A distribution method for solving sat in grids. In *International conference on theory and applications of satisfiability testing*, pages 430–435. Springer, 2006.
- [24] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Partitioning sat instances for distributed solving. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 372–386, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [25] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99th. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.
- [26] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Painless: a framework for parallel sat solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 233–250. Springer, 2017.
- [27] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Modular and efficient divide-and-conquer sat solver on top of the painless framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 135–151. Springer, 2019.
- [28] S. Mijnders, B. De Wilde, and M. Heule. Symbiosis of search and heuristics for random 3-sat. *arXiv preprint arXiv:1402.4455*, 2014.
- [29] S. Nejati, Z. Newsham, J. Scott, J. H. Liang, C. Gebotys, P. Poupart, and V. Ganesh. A propagation rate based splitting heuristic for divide-and-conquer solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 251–260. Springer, 2017.
- [30] Y. Ngoko, C. Cérin, and D. Trystram. Solving sat in a distributed cloud: a portfolio approach. *International Journal of Applied Mathematics and Computer Science*, 29(2):261–274, 2019.
- [31] C. Sinz, W. Blochinger, and W. Küchlin. Pasat—parallel sat-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001.
- [32] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.
- [33] H. Wu, A. Ozdemir, A. Zeljić, K. Julian, A. Irfan, D. Gopinath, S. Fouladi, G. Katz, C. Pasareanu, and C. Barrett. Parallelization techniques for verifying neural networks. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 128–137. IEEE, 2020.
- [34] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.
- [35] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

Induction with Recursive Definitions in Superposition

Márton Hajdu^{*} , Petra Hozzová^{*} , Laura Kovács^{*}  and Andrei Voronkov[†]

^{*}TU Wien

[†]University of Manchester and EasyChair

Abstract—Functional programs over inductively defined data types, such as lists, binary trees and naturals, can naturally be defined using recursive equations over recursive functions. In first-order logic, function definitions can be considered as universally quantified equalities. Verifying functional program properties therefore requires inductive reasoning with both theories and quantifiers. In this paper we propose new extensions and generalizations to automate induction with recursive functions in saturation-based first-order theorem proving, using the superposition calculus. Instead of using function definitions as first-order axioms, we introduced new simplification rules for treating function definitions as rewrite rules. We guide inductive reasoning and strengthen induction schema using recursively defined functions. Our experimental results show that handling recursive definitions in superposition reasoning significantly improves automated reasoning with induction.

I. INTRODUCTION

Automated reasoning has become the backbone of formal software development [1]. Automating inductive reasoning is of increasing importance for emerging applications in software verification, in particular in the context of functional programming and inductive/algebraic data types (also called term algebras), such as natural numbers, lists and binary trees. Functional programs can be typically described by recursive equations/functions over algebraic data types, as illustrated in Figure 1. On the other hand, algebraic data types are, for example, commonly used in security applications to encode uniqueness of hash functions [2] or to express non-interference properties preventing information flow between private/public channels [3]. Formalizing such properties requires full first-order logic with theories, and automating their validation requires inductive reasoning.

Previous works on automating induction mainly focus on inductive theorem proving [4], [5], [6], [7], [8], [9], [10], [11]: deciding when induction should be applied and what induction axiom should be used. Further restrictions are made on the logical expressiveness, for example induction over only universal properties [7], [9], [6], term algebras [12] or Horn clauses [13]. Recent advances related to automating inductive reasoning, such as first-order reasoning with inductively defined data types [14], inductive strengthening of SMT properties [15], structural induction in first-order theorem proving [16], [17], [18], [12], open up new possibilities for automating induction. *In this paper we focus on first-order theorem proving and automate induction by integrating it directly into the proof search algorithm of first-order theorem proving.* The program

assertions from lines 17–18 of Figure 1 show what we strive for: validating first-order properties over algebraic data types, such as binary trees, lists and naturals, involving additional recursive function definitions and predicates, such as even, mul, app, flat and aflat. We prove such and similar inductive properties by using saturation-based proof search based on the superposition calculus [19], which is the leading technology in automated theorem proving [20], [21], [22].

Reasoning about inductively defined data types with recursive definitions. Our work targets full and efficient automation of induction with recursive function reasoning, as illustrated in a toy ML-like functional program of Figure 1. Lines 1–3 of Figure 1 declare respectively the algebraic data types of natural numbers `nat`, lists `list` and binary trees `bt`, using constructors. In first-order logic, these data types correspond to term algebras [14]. Functional programs over data types can be defined by recursive equations, for example lines 4–5 of Figure 1 define the addition `add` of two natural numbers x, y (in first-order logic, function definitions can be considered as universally quantified equalities). Verifying the correctness of Figure 1 requires then to prove the formulas of lines 17–18, which asserts the equivalence of two functions over binary trees (line 17) and even properties of naturals (line 18). Automating reasoning about properties of inductively defined data types like `nat`, `list` and `bt` needs to handle acyclicity already for equational properties (which, in general, is not finitely axiomatizable) and induction. Our recent results on reasoning with inductively defined data types and induction [14], [18] enable induction in superposition-based theorem proving, yet only by applying induction over one clause at a time. Our work builds upon these results and brings novel extensions for handling recursive functions and (generalized) induction on arbitrarily many clauses simultaneously.

Our contributions. This paper brings the following contributions.

- We introduce an induction formula generation method, utilizing unification and recursive function definitions over algebraic data types (Section IV). We propose inductive strengthening and generalization methods well-suited for saturation-based approaches.
- We propose new inference rules for induction in superposition by treating recursive function definitions over algebraic data types as rewrite rules in superposition (Section V). Moreover, we make use of induction hypotheses with specialized inference rules. Applications of induc-

<pre> 1 <u>datatype</u> nat = zero s <u>of</u> nat 2 <u>datatype</u> list = nil cons <u>of</u> nat list 3 <u>datatype</u> bt = leaf node <u>of</u> bt nat bt 4 add zero y = y 5 add (s x) y = s (add x y) 6 mul zero y = zero 7 mul (s x) y = add (mul x y) y 8 even zero 9 ¬even (s zero) </pre>	<pre> 10 even (s (s x)) ↔ even x 11 app nil z = z 12 app (cons x y) z = cons x (app y z) 13 flat leaf = nil 14 flat (node x y z) = app (flat x) (cons y (flat z)) 15 aflat leaf u = u 16 aflat (node x y z) u = aflat x (cons y (aflat z u)) 17 <u>assert</u> (∀x, y)(app (flat x) y = aflat x y) 18 <u>assert</u> (∀x, y)(even y → even (mul x y)) </pre>
--	--

Fig. 1. Motivating example with recursive definitions over algebraic data types.

tion become inference rules of the saturation process, adding instances of appropriate induction schemata.

- We extend superposition-based equational reasoning with new inference rules capturing inductive steps over multiple clauses and optimize saturation-based proof search with induction (Section VI). Unlike [16], our results do not necessarily depend on the AVATAR clause splitting framework [23]. Contrarily to [12], we are not limited to induction over term algebras with the subterm ordering and we stay in a standard saturation framework.
- We implemented our approach in the VAMPIRE theorem prover [22] and evaluated it on a large collection of examples, including 327 examples from the SMT-LIB repository [24] and 3,397 mathematical properties over naturals, lists and binary trees (Section VII).
- Our experiments show the potential of our new approach, by solving 527 problems that other systems automating induction could not prove (Section VII).

Structure of the paper. The rest of the paper is organized as follows. We illustrate the challenges of automating induction with recursive definitions in superposition reasoning in Section II. We present our induction formula generation method in Section IV. Section V describes inductive reasoning with recursive definitions, whereas Section VI generalizes our work to induction with multiple premises. After summarizing our experimental findings in Section VII, we overview related work in Section VIII. We conclude the paper in Section IX.

II. MOTIVATING EXAMPLE

We first motivate our work using the functional program of Figure 1 over naturals, lists and binary trees.

Example 1 (Inductive reasoning with lists and binary trees). Using the recursive function definition `app` over lists, and recursive function definitions `flat` and `aflat` over binary trees (lines 11–16 of Figure 1), we first focus on proving the equivalence of functions `flat` and `aflat` flattening binary trees to lists, specified as an assertion at line 17 of Figure 1. For easing readability, we write this assertion in infix notation as below:

$$\forall u, v. \text{app}(\text{flat}(u), v) = \text{aflat}(u, v) \quad (1)$$

Proving (1) requires induction over binary trees, using for example the structural induction formula

$$(F[\text{leaf}] \wedge \forall x, y, z. ((F[x] \wedge F[z]) \rightarrow F[\text{node}(x, y, z)])) \rightarrow \forall u. F[u], \quad (2)$$

where $F[x]$ denotes a first-order formula over x . By instantiating (2), proving (1) reduces to proving two formulas: the base case and the step case. The base case,

$$\forall v. \text{app}(\text{flat}(\text{leaf}), v) = \text{aflat}(\text{leaf}, v), \quad (3)$$

holds by the recursive definitions at lines 11, 13 and 15 of Figure 1. For the step case, we strengthen the hypotheses by replacing v with fresh universally quantified variables v_0, v_1 :

$$\forall x, y, v. (\forall v_0. \text{app}(\text{flat}(x), v_0) = \text{aflat}(x, v_0) \wedge \quad (4)$$

$$\forall v_1. \text{app}(\text{flat}(y), v_1) = \text{aflat}(y, v_1) \rightarrow \quad (5)$$

$$\text{app}(\text{flat}(\text{node}(x, y, v)), v) = \text{aflat}(\text{node}(x, y, v), v)) \quad (6)$$

For proving (6), we first use the recursive definitions at lines 14 and 16 of Figure 1 to obtain (omitting (4), (5) and implicit universal quantification):

$$\text{app}(\text{app}(\text{flat}(x), \text{cons}(y, \text{flat}(z))), v) = \text{aflat}(x, \text{cons}(y, \text{aflat}(z, v))) \quad (7)$$

By rewriting (7) with (4) and (5), we are left with proving:

$$\text{app}(\text{app}(\text{flat}(x), \text{cons}(y, \text{flat}(z))), v) = \text{app}(\text{flat}(x), \text{cons}(y, \text{app}(\text{flat}(z), v))) \quad (8)$$

By replacing `flat`(x) with a fresh variable w in (8), we obtain

$$\text{app}(\text{app}(w, \text{cons}(y, \text{flat}(z))), v) = \text{app}(w, \text{cons}(y, \text{app}(\text{flat}(z), v))) \quad (9)$$

which is a generalized/stronger formula than (8). By applying the structural induction formula over lists

$$(F[\text{nil}] \wedge \forall x, y. (F[y] \rightarrow F[\text{cons}(x, y)])) \rightarrow \forall z. F[z]$$

over w in (9), we derive the validity of (9) by also using the definition of `app` from lines 11–12 in Figure 1. We thus conclude that (1) holds, and hence the assertion at line 17 of Figure 1 is valid.

While the proof above is quite natural for humans, it is very difficult for saturation-based first-order provers using the superposition calculus. For example, the state-of-the-art solvers supporting induction CVC4 [15], ZIPPERPOSITION [16] and VAMPIRE [17] fail proving (1). To organize proof search, saturation-based theorem provers, intuitively speaking, disallow rewriting small terms into big terms w.r.t. some ordering. In most (simplification) orderings used by these provers, the terms `flat` and `aflat` in (6) cannot be expanded using their recursive definitions, as the right-hand sides of these definitions are heavier/bigger¹ than their left-hand sides. Moreover, deciding the order in which induction hypotheses should be applied, such as (4) and (5), is as difficult as doing the proof itself. In this paper, we *extend superposition reasoning with special treatment of recursive definitions, guiding the generation of induction formulas during saturation (Section IV). We use rewrite rules for terms occurring in recursive definitions and inductive hypotheses (Section V).* Thanks to this extension, our work can easily validate (1). \square

Another challenging aspect of induction with recursive definitions comes with generalizing and adjusting induction formulas over recursively defined terms and multiple premises, as illustrated next.

Example 2 (Inductive reasoning with naturals). Using the recursive function and predicate definitions of `add`, `mul`, and `even` from lines 4–10 of Figure 1, the assertion at line 18 encodes the following first-order formula over naturals:

$$\forall x, y. \text{even}(y) \rightarrow \text{even}(\text{mul}(x, y)) \quad (10)$$

Similarly as in Example 1, proving (10) requires instantiating a structural induction formula for naturals as below:

$$(F[\text{zero}] \wedge \forall z. (F[z] \rightarrow F[s(z)])) \rightarrow \forall x. F[x] \quad (11)$$

and thereby proving the following two formulas:

$$\forall y. \text{even}(y) \rightarrow \text{even}(\text{mul}(\text{zero}, y)) \quad (12)$$

$$\forall z, y. ((\text{even}(y) \rightarrow \text{even}(\text{mul}(z, y))) \rightarrow (\text{even}(y) \rightarrow \text{even}(\text{mul}(s(z), y)))) \quad (13)$$

Validity of the formula (12) follows from the recursive function definitions in lines 6 and 8 of Figure 1. By using the recursive definition in line 7 of Figure 1, formula (13) reduces to

$$\forall z, y. (\text{even}(\text{mul}(z, y)) \rightarrow \text{even}(\text{add}(\text{mul}(z, y), y))) \quad (14)$$

The antecedent of (14) cannot however be used for proving its conclusion. We overcome this limitation by replacing/generalizing `mul(z, y)` in (14) with a fresh new variable `u` and instantiating the following variant of (11):

$$(F[\text{zero}] \wedge F[s(\text{zero})] \wedge \forall z. (F[z] \rightarrow F[s(s(z))])) \rightarrow \forall x. F[x] \quad (15)$$

While (11) cannot be used to prove (14), note that (15) enables the application of the recursive definition of `even` in line 10

of Figure 1. As such, proving the generalized version of (14) reduces to proving the three formulas:

$$\text{even}(\text{zero}) \rightarrow \text{even}(\text{add}(\text{zero}, y)) \quad (16)$$

$$\text{even}(s(\text{zero})) \rightarrow \text{even}(\text{add}(s(\text{zero}), y)) \quad (17)$$

$$\forall z. ((\text{even}(z) \rightarrow \text{even}(\text{add}(z, y))) \rightarrow (\text{even}(s(s(z))) \rightarrow \text{even}(\text{add}(s(s(z)), y)))) \quad (18)$$

All three formulas can be proven by applying the recursive function definitions of `add` and `even` from Figure 1 and using induction with multiple premises over (18) (Section VI). In this paper, we *generate induction formula variants, such as (15), based on recursive function/predicate definitions (Section IV) and support induction with multiple premises (Section VI),* proving for example (10). \square

While relatively simple, Figure 1 illustrates the key challenges in automating induction with recursive definitions in superposition: (i) *strengthening and creating induction formulas* using recursive definitions (Section IV); (ii) *rewriting recursively defined terms* by their (function/predicate) definitions (Section V); and (iii) *applying induction with multiple premises* (Section VI). In what follows, we describe our solutions for these challenges.

III. PRELIMINARIES

We assume familiarity with *standard multi-sorted first-order logic with equality*. Functions are denoted with f, g, h , predicates with p, q, r , variables with x, y, z, u, v, w , and Skolem constants with σ , all possibly with indices. A term is *ground* if it contains no variables. By \bar{x} and \bar{t} we denote tuples of variables and terms, respectively.

We use the standard logical connectives $\neg, \vee, \wedge, \rightarrow$ and \leftrightarrow , and quantifiers \forall and \exists . A *literal* is an atom or its negation. For a literal L , we write \bar{L} to denote its complementary literal. A disjunction of literals is a *clause*. We reserve the symbol \square for the *empty clause* which is logically equivalent to \perp . We denote the *clausal normal form* of a formula F by $\text{cnf}(F)$. We call every term, literal, clause or formula an *expression*. We use the notation $s \trianglelefteq t$ to denote that s is a *subterm* of t and $s \triangleleft t$ if s is a *proper subterm* of t .

We use the words *sort* and *type* interchangeably. We distinguish special sorts called *inductive sorts*, function symbols for inductive sorts called *constructors* and *destructors*. We distinguish *recursive constructors*, which have at least one argument of the same sort as their return sort, from *base constructors*, which do not have any arguments of the same type as their return sort. We call the ground terms built from the constructor symbols of a sort its *term algebra*.

We axiomatise term algebras using their *injectivity*, *distinctness*, *exhaustiveness* and *acyclicity* axioms [14]. In this paper, we refer to term algebras also as algebraic data types or inductively defined data types.

We write $E[s]$ to denote that expression E contains k distinguished occurrence(s) of the term s , with $k \geq 0$. For simplicity, $E[t]$ means that these occurrences of s are replaced by the term t . Further, $E[t]_{p_1 \dots p_k}$, with $p_1 \dots p_k \in \{0, 1\}^k$,

¹W.r.t. orderings of first-order provers.

is the expression obtained by replacing i th distinguished occurrence of s by t in $E[s]$ iff $p_i = 1$. We abbreviate $E[t_1] \dots [t_n]$ with $E[\bar{t}]$.

A *substitution* θ is a mapping from variables to terms. A substitution θ is a *unifier* of two terms s and t if $s\theta = t\theta$, and is a *most general unifier (mgu)* if for every unifier η of s and t , there exists substitution μ s.t. $\eta = \theta\mu$. We denote the mgu of s and t with $\text{mgu}(s, t)$.

A. Saturation-based proof search

First-order theorem provers work with clauses, rather than with arbitrary formulas. Given a set S of input clauses, first-order provers *saturate* S by computing all logical consequences of S with respect to a sound inference system \mathcal{I} . The saturated set of S is called the *closure* of S and process of computing the closure of S is called *saturation* [22]. If the closure contains the empty clause \square , the original set S of clauses is unsatisfiable. A simplified saturation algorithm for inference system \mathcal{I} is given below with a clausified goal F and clausified assumptions A as input:

```

1  passive :=  $A \cup \{\neg F\}$ , active :=  $\emptyset$ 
2  while passive  $\neq \emptyset$ :
3     $G := \text{select}(\text{passive})$ 
4    derive consequences  $\mathcal{C}$  of  $G$  and active w.r.t.  $\mathcal{I}$ 
5    passive :=  $(\text{passive} \cup \mathcal{C}) \setminus G$ 
6    active := active  $\cup \{G\}$ 
7  if  $\square \in \text{passive}$  then return UNSAT
8  return SAT

```

Completeness and efficiency of saturation-based reasoning rely heavily on properties of *select* and \mathcal{I} (lines 3 and 4). The *superposition calculus* [19] (denoted Sup) is the most common inference system employed by saturation-based first-order theorem provers, such as E [20], VAMPIRE [22] and ZIPPERPOSITION [16]. The superposition calculus is *sound* and *refutationally complete*: for any unsatisfiable formula, the empty clause can be derived as a logical consequence. To organize saturation, first-order provers use simplification *orderings* on terms, which are extended to orderings over literals and clauses; for simplicity, we write \succ for both the term ordering and its clause ordering extension. We write $s \doteq t$ to mean that the orientation of the equality $s = t$ is fixed (i.e., either $s \succ t$ or $t \succ s$).

We make use of the following inference rules of Sup in this paper:

Binary resolution:

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\theta}$$

where θ is the mgu of A and B .

Superposition:

$$\frac{l = r \vee C \quad s[l'] \neq t \vee D}{(s[r] \neq t \vee C \vee D)\theta} \quad \frac{l = r \vee C \quad s[l'] = t \vee D}{(s[r] = t \vee C \vee D)\theta}$$

where θ is the mgu of l and l' , $r\theta \not\prec l\theta$ and $t\theta \not\prec s[l']\theta$. There are special cases of these rules, imposing more restrictions on

the premises. One such case is when one of the premises of superposition is a unit clause, yielding the so-called *demodulation* rules, as given in Section V.

Given an ordering \succ , a clause C is *redundant* with respect to a set S of clauses if there exists a subset S' of S such that S' is smaller than $\{C\}$ (i.e., $C \succ S'$) and S' implies C . Redundant clauses can be eliminated during proof search without destroying completeness; *simplification and deletion rules* are used to remove redundant clauses.

IV. INDUCTION FORMULAS OVER RECURSIVE DEFINITIONS IN SUPERPOSITION

We now describe our solution for generating induction formulas in saturation-based theorem proving. Unlike [7], [4], [16], [10], [11], [25], [26], we integrate induction directly in the saturation-based theorem proving using the superposition calculus. For doing so, we rely on [17], [18] and use the following sound *inference rule of induction*:

$$\frac{\bar{L}[\bar{t}] \vee C}{\text{cnf}(F \rightarrow \forall \bar{y}. L[\bar{y}])} \text{ (Ind)},$$

where L is a ground literal, C is a clause, and $F \rightarrow \forall \bar{y}. L[\bar{y}]$ is a valid induction formula. Further, \bar{y} is a tuple of variables and \bar{t} is a tuple of induction terms, of the same size.

In [17], [18], the inference rule (Ind) has been used by considering the induction formulas as instances of mathematical and structural induction. In this paper, we go beyond these works and utilise recursive function/predicate definitions to derive induction formulas to be used in (Ind). For doing so, we first select terms in recursive definitions over which induction formulas will be generated in Section IV-A and strengthened in Section IV-B. Further, in Section VI we extend (Ind) to induction formulas with multiple premises.

A. Generating Induction Formulas over Recursive Definitions

A recursive function/predicate definition has a number of branches, characterized by one or more clauses. We assume that (i) a function definition clause contains exactly one equality with a fixed orientation, i.e., $f(\bar{s}) \doteq t \vee C$. Similarly, (ii) a predicate definition axiom contains one marked literal, i.e., $(\neg)\hat{p}(\bar{s}) \vee D$, where \hat{p} denotes that p is marked/selected. Two clauses $f(\bar{s}_1) \doteq t_1 \vee C$ and $f(\bar{s}_2) \doteq t_2 \vee D$ belong to the same branch of f if $f(\bar{s}_1)$ and $f(\bar{s}_2)$ are variants of each other. Similarly, two clauses $(\neg)\hat{p}(\bar{s}_1) \vee C$ and $(\neg)\hat{p}(\bar{s}_2) \vee D$ belong to the same branch of p if $p(\bar{s}_1)$ and $p(\bar{s}_2)$ are variants of each other. We therefore characterize a recursive definition branch with its *characteristic term* $f(\bar{s})$ or *characteristic atom* $p(\bar{s})$. We write “branch $f(\bar{s})$ ” and “branch $p(\bar{s})$ ” to refer to the branches with the characteristic term $f(\bar{s})$ and characteristic atom $p(\bar{s})$, respectively. We denote the set of variable disjoint branches of a function f and predicate p with \mathcal{B}_f and \mathcal{B}_p , respectively.

Definition 1 (Recursive Calls of Recursive Definitions). Let f be a recursive function and p a recursive predicate. The *set of*

recursive calls corresponding, respectively, to the branch $f(\bar{s})$ and the branch $p(\bar{s})$ are defined as:

$$\begin{aligned}\mathcal{R}_{f(\bar{s})} &:= \bigcup_{f(\bar{s}') \doteq t \vee C} \{f(\bar{s}')\theta \mid f(\bar{s}') \leq t, f(\bar{s}')\theta = f(\bar{s})\} \\ \mathcal{R}_{p(\bar{s})} &:= \bigcup_{\hat{p}(\bar{s}') \vee C} \{p(\bar{s}')\theta \mid p(\bar{s}') \in C, p(\bar{s}')\theta = p(\bar{s})\}\end{aligned}\quad \square$$

The rest of this section only details the generation of induction formulas using recursive function definitions; recursive predicates are handled similarly. Given a recursive function f , we categorize its argument positions similarly to [16].

Definition 2 (Active Positions, Accumulators). If for any branch $f(\bar{s}) \in \mathcal{B}_f$ and $f(\bar{s}') \in \mathcal{R}_{f(\bar{s})}$:

- (1) if $s'_i \triangleleft s_i$, then i is an *active* argument position of f
- (2) if s_i is a variable and $s_i \neq s'_i$, then i is an *accumulator* argument position of f

We denote the set of active and accumulator argument positions of f with I_f . \square

Example 3. Based on the functions `app`, `flat` and `aflat` from Figure 1 lines 11-16, we have:

$$\mathcal{B}_{\text{app}} = \{\text{app}(\text{nil}, z_0), \text{app}(\text{cons}(x, y), z_1)\}$$

$$\mathcal{B}_{\text{flat}} = \{\text{flat}(\text{leaf}), \text{flat}(\text{node}(x, y, z))\}$$

$$\mathcal{B}_{\text{aflat}} = \{\text{aflat}(\text{leaf}, u_0), \text{aflat}(\text{node}(x, y, z), u_1)\}$$

While $\mathcal{R}_{\text{app}(\text{nil}, z_0)} = \mathcal{R}_{\text{flat}(\text{leaf})} = \mathcal{R}_{\text{aflat}(\text{leaf}, u_0)} = \emptyset$, the second branches of the three functions have the following sets of recursive calls:

$$\mathcal{R}_{\text{app}(\text{cons}(x, y), z_1)} = \{\text{app}(y, z_1)\}$$

$$\mathcal{R}_{\text{flat}(\text{node}(x, y, z))} = \{\text{flat}(x), \text{flat}(z)\}$$

$$\mathcal{R}_{\text{aflat}(\text{node}(x, y, z), u_1)} = \left\{ \begin{array}{l} \text{aflat}(x, \text{cons}(y, \text{aflat}(z, u_1))), \\ \text{aflat}(z, u_1) \end{array} \right\}$$

$I_{\text{app}} = \{1\}$, since y is a proper subterm of $\text{cons}(x, y)$ but the second argument is not an accumulator since it remains z_1 in the only recursive call. The only argument position of `flat` is active, and therefore $I_{\text{flat}} = \{1\}$. Finally, `aflat` has one active and one accumulator argument position, hence $I_{\text{aflat}} = \{1, 2\}$. \square

Definition 3 (Induction Terms from Active and Accumulator Positions). Consider a recursive function f of arity n and a ground term $f(\bar{c})$. The term $f(\bar{c}')$ is a *generator term* iff (i) \bar{c}' coincides with \bar{c} in all positions from $\{1 \leq i \leq n\} \setminus I_f$, and (ii) \bar{c}' contains fresh variables on positions from I_f .

The *induction case* of $f(\bar{c})$ over branch $f(\bar{s}) \in \mathcal{B}_f$ is the two-tuple:

$$(\theta, \{\text{mgu}(f(\bar{c}'), f(\bar{s}')\theta) \mid f(\bar{s}') \in \mathcal{R}_{f(\bar{s})}\})$$

where $\theta := \text{mgu}(f(\bar{c}'), f(\bar{s}))$.

The *case distinction* $\Theta_{f(\bar{c})}$ of $f(\bar{c})$ is the set of induction cases of $f(\bar{c})$ over each branch of f . We call $\{c_i \mid i \in I_f\}$ the *induction terms* of $f(\bar{c})$. \square

Induction Formula over Active and Accumulator Terms.

Using Definition 3, we guide induction formula generation over active and accumulator terms, as follows. Given a literal $L[\bar{c}]$ with zero or more occurrences of the terms \bar{c} , we generate and add the following *induction formula over active and accumulator terms* to saturation-based proving:

$$(\forall) \bigwedge_{(\theta, R) \in \Theta_{f(\bar{c})}} \left(\bigwedge_{\theta' \in R} L[\bar{c}']\theta' \rightarrow L[\bar{c}']\theta \right) \rightarrow L[\bar{c}] \quad (19)$$

Since (19) is a valid induction formula, using it in the conclusion of (Ind) yields a sound (Ind) inference.

Example 4. For proving the assertion of line 17 from Figure 1 in a saturation-based framework, we consider its negation:

$$\text{app}(\text{flat}(\sigma_0), \sigma_1) \neq \text{aflat}(\sigma_0, \sigma_1) \quad (20)$$

Using Definition 3 and I_{flat} (Example 3), the generator term of `flat`(σ_0) is $t := \text{flat}(v)$. Moreover, by $\mathcal{B}_{\text{flat}}$ from Example 3, we obtain

$$\theta_1 = \text{mgu}(t, \text{flat}(\text{leaf})) = \{v \mapsto \text{leaf}\}$$

$$\theta_2 = \text{mgu}(t, \text{flat}(\text{node}(x, y, z))) = \{v \mapsto \text{node}(x, y, z)\}$$

Applying the unifier θ_2 on the recursive calls of $\mathcal{R}_{\text{flat}(\text{node}(x, y, z))}$ from Example 3 is a no-op, since the recursive calls do not contain v and we derive

$$\theta_{2.1} = \text{mgu}(t, \text{flat}(x)) = \{v \mapsto x\}$$

$$\theta_{2.2} = \text{mgu}(t, \text{flat}(z)) = \{v \mapsto z\}$$

Using the case distinction

$$\Theta_{\text{flat}(\sigma_0)} = \{(\theta_1, \emptyset), (\theta_2, \{\theta_{2.1}, \theta_{2.2}\})\} \quad (21)$$

we derive the following induction formula:

$$\begin{aligned} & \forall x, y, z, u. \\ & \left(\left(\text{app}(\text{flat}(\text{leaf}), \sigma_1) = \text{aflat}(\text{leaf}, \sigma_1) \wedge \right. \right. \\ & \quad \left(\text{app}(\text{flat}(x), \sigma_1) = \text{aflat}(x, \sigma_1) \wedge \right. \\ & \quad \left. \text{app}(\text{flat}(z), \sigma_1) = \text{aflat}(z, \sigma_1) \rightarrow \right. \\ & \quad \left. \left. \text{app}(\text{flat}(\text{node}(x, y, z)), \sigma_1) = \text{aflat}(\text{node}(x, y, z), \sigma_1) \right) \right) \\ & \rightarrow \text{app}(\text{flat}(u), \sigma_1) = \text{aflat}(u, \sigma_1) \end{aligned} \quad \square \quad (22)$$

B. Strengthening Induction over Recursive Definitions

Induction hypotheses of induction formulas might not be strong enough to prove the corresponding induction step. A common technique to overcome such limitations is to strengthen the induction hypotheses: replace some terms in the hypotheses with universally quantified fresh variables, yielding thus logically stronger versions of induction hypotheses. Introducing universally quantified variables during saturation can however negatively impact the performance of the prover (e.g., yielding more unifications/rewriting steps). As a remedy to this practical burden in the context of recursive function definitions f , we utilize the *accumulator argument positions* from I_f in Definition 3, which supersede the need for introducing universally quantified variables by implicitly instantiating these variables to the terms that will be matched by the recursive calls of f .

Example 5. The induction formula (22) is not strong enough to prove (20) and strengthening its induction hypotheses by replacing σ_1 with a universally quantified fresh variable – as in (4) and (5) from Example 1, – is inefficient. Instead, we use the term $\text{aflat}(\sigma_0, \sigma_1)$ from (20) with the generator term $t' := \text{aflat}(v, w)$ and induction terms $\{\sigma_0, \sigma_1\}$. We obtain the following unifiers:

$$\begin{aligned}\theta'_1 &= \text{mgu}(t', \text{aflat}(\text{leaf}, u_0)) = \{v \mapsto \text{leaf}, w \mapsto u_0\} \\ \theta'_2 &= \text{mgu}(t', \text{aflat}(\text{node}(x, y, z), u_1)) \\ &= \{v \mapsto \text{node}(x, y, z), w \mapsto u_1\}\end{aligned}$$

Applying θ'_2 is once again a no-op on the recursive calls $\mathcal{R}_{\text{aflat}(\text{node}(x, y, z), u_1)}$, and we get the unifiers:

$$\begin{aligned}\theta'_{2,1} &= \text{mgu}(t', \text{aflat}(x, \text{cons}(y, \text{aflat}(z, u_1)))) \\ &= \{v \mapsto x, w \mapsto \text{cons}(y, \text{aflat}(z, u_1))\} \\ \theta'_{2,2} &= \text{mgu}(t', \text{aflat}(z, u_1)) = \{v \mapsto z, w \mapsto u_1\}\end{aligned}$$

Thus we obtain the induction formula with the required induction hypothesis with term $\text{cons}(y, \text{aflat}(z, u_1))$ that matches the conclusion after simplification:

$$\begin{aligned}& \forall x, y, z, u_0, u_1, v, w. \\ & ((\text{app}(\text{flat}(\text{leaf}), u_0) = \text{aflat}(\text{leaf}, u_0) \wedge \\ & (\text{app}(\text{flat}(x), \text{cons}(y, \text{aflat}(z, u_1))) = \\ & \quad \text{aflat}(x, \text{cons}(y, \text{aflat}(z, u_1))) \wedge \\ & \text{app}(\text{flat}(z), u_1) = \text{aflat}(z, u_1) \rightarrow \\ & \text{app}(\text{flat}(\text{node}(x, y, z), u_1) = \text{aflat}(\text{node}(x, y, z), u_1)) \\ & \rightarrow \text{app}(\text{flat}(v), u_1) = \text{aflat}(v, u_1))\end{aligned} \quad (23)$$

After skolemizing x, y, z, u_0 and u_1 during clausification, binary resolving with (20), with v and w bound to σ_0 and σ_1 , respectively, we get the following ground induction hypotheses literals and ground conclusion literal from (23):

$$\text{app}(\text{flat}(\sigma_0), \text{cons}(\sigma_3, \text{aflat}(\sigma_4, \sigma_5))) = \text{aflat}(\sigma_0, \text{cons}(\sigma_3, \text{aflat}(\sigma_4, \sigma_5))) \quad (24)$$

$$\text{app}(\text{flat}(\sigma_4), \sigma_5) = \text{aflat}(\sigma_4, \sigma_5) \quad (25)$$

$$\text{app}(\text{flat}(\text{node}(\sigma_0, \sigma_3, \sigma_4)), \sigma_5) \neq \text{aflat}(\text{node}(\sigma_0, \sigma_3, \sigma_4), \sigma_5) \quad (26)$$

Further, the hypotheses of (23) are strong enough to prove (20), as shown in Section V. \square

In summary, we use Definition (3) to generate induction formulas over the active and accumulator terms from $I_{\mathcal{F}}$. To further limit and guide the generation of induction formulas, we devised *heuristics* similar to [16]. Foremost, we only generate induction formulas from function/predicate terms with active occurrences.

Definition 4 (Active Term Occurrences). An occurrence of a term t in literal L is an *active occurrence* if (i) t is L , or (ii) L is an equality $l = r$ and t is l or r , or (iii) the immediate superterm s of t is an active occurrence and the occurrence of t is in an active argument position of s . \square

As described in [18], apart from generalizing over complex terms as seen in Example (1), we can also generalize over active term occurrences. For example, we can refine the

induction formula (19) to induct upon only certain occurrences of an induction term t with k occurrences in literal L , by using any bit vector $p \in \{0, 1\}^k$ and $L[t]_p$ instead of $L[t]$.

V. REFUTING INDUCTIVE PROPERTIES WITH RECURSIVE DEFINITIONS

Automating inductive reasoning not only requires finding useful induction formulas, but also comes with the task of proving inductive properties. Section IV detailed our approach towards finding useful induction formulas over recursive definitions. As a next step, we now present our solution towards (more) efficient refutation of inductive properties over recursive definitions.

A. Rewriting with Recursive Function Definitions

We extend superposition reasoning with two inference rules in support of rewriting recursive functions by their definitions.

First, we focus on a *simplification inference* implementing rewriting by unit equalities, called also demodulation [22]. We adjust demodulation to handle unit clauses describing recursive function definitions, as follows:

$$\frac{f(\bar{s}) \doteq t \quad \underline{L[f(\bar{s})\theta]} \vee \mathcal{D}}{L[t\theta] \vee \mathcal{D}} \quad (\text{DemF})$$

where $f(\bar{s})\theta \succ t\theta$ and $L[f(\bar{s})\theta] \vee \mathcal{D} \succ f(\bar{s})\theta = t\theta$.

Second, we introduce a *generating inference* rule as an instance of superposition rules. Namely, we enable rewriting arbitrary recursive functions with their definitions, as follows:

$$\frac{f(\bar{s}) \doteq t \vee C \quad \underline{L[f(\bar{s})\theta]} \vee \mathcal{D}}{L[t\theta] \vee C\theta \vee \mathcal{D}} \quad (\text{ParF})$$

Note that (ParF) has no side conditions restricting which terms can be rewritten. As such, (ParF) allows to expand function headers, yet at the cost that small terms may be rewritten into bigger terms w.r.t. the underlining term ordering \succ of a superposition prover. As a result, the simplification ordering constraints of \succ are violated by (ParF), yielding an incomplete extension of superposition. On the other hand, soundness of superposition implies soundness of our new inference rules.

Theorem 1 (Soundness of Rewriting). The inference rules (DemF) and (ParF) are sound. \square

B. Rewriting Induction Hypotheses

Upon clausifying the induction formula (19) introduced in Section IV, for each step case $\bigwedge_{1 \leq i \leq m} L[t_i] \rightarrow L[t]$ we obtain a set of *induction hypothesis literals* $L[t_i]$ and an *induction conclusion literal* $L[t]$. Intuitively, we extend these notions such that any literal resulting from the rewriting or simplification of induction hypothesis or induction conclusion literals is also an induction hypothesis or induction conclusion literal, respectively.

We introduce an *induction hypothesis rewriting rule*, in short (IndHRW), to (i) rewrite one side of an induction conclusion literal with one of its induction hypothesis literals (against

ordering constraints) and (ii) apply induction on the rewritten induction conclusion literal without adding it to the search space:

$$\frac{l = r \vee D \quad s[l] \neq t \vee C}{\text{cnf}(F \rightarrow \forall \bar{y}. (s[r] = t)[\bar{y}])} \text{ (IndHRW)}$$

where $s \neq t$ is an induction conclusion literal with corresponding induction hypothesis literal $l = r$, $l \not\preceq r$, and $F \rightarrow \forall \bar{y}. (s[r] = t)[\bar{y}]$ is a valid induction formula. By soundness of (Ind), we conclude soundness of (IndHRW).

Theorem 2 (Soundness of Induction Hypothesis Rewriting). The inference rule (IndHRW) is sound. \square

Note that (IndHRW) allows rewriting only with induction hypothesis literals that are positive equalities. Hence, the induction conclusion literal must be a disequality ($s \neq t$). We further stress that rewriting using the premises of (IndHRW) yields $s[r] \neq t \vee C \vee D$, which is binary resolved against the resulting induction formula clauses of (19) and not added to the search space.

Example 6. Continuing Example 5, rewriting (26) with (ParF) results in a new induction conclusion literal:

$$\text{app}(\text{app}(\text{flat}(\sigma_2), \text{cons}(\sigma_3, \text{flat}(\sigma_4))), \sigma_5) \neq \text{aflat}(\sigma_2, \text{cons}(\sigma_3, \text{aflat}(\sigma_4, \sigma_5))) \quad (27)$$

By rewriting the right-hand side of (27) with the corresponding hypotheses literals (24) and (25), we obtain the intermediate induction conclusion literal

$$\text{app}(\text{app}(\text{flat}(\sigma_2), \text{cons}(\sigma_3, \text{flat}(\sigma_4))), \sigma_5) \neq \text{app}(\text{flat}(\sigma_2), \text{cons}(\sigma_3, \text{app}(\text{flat}(\sigma_4), \sigma_5))) \quad (28)$$

By applying induction with (IndHRW) with case distinction $\Theta_{\text{app}(\text{flat}(\sigma_2), \text{cons}(\sigma_3, \text{flat}(\sigma_4)))}$ and induction term $\text{flat}(\sigma_2)$, we obtain the induction formula:

$$\begin{aligned} & \forall x, y, \dots \\ & \left((\text{app}(\text{app}(\text{nil}, \text{cons}(\sigma_3, \text{flat}(\sigma_4))), \sigma_5) = \right. \\ & \quad \left. \text{app}(\text{nil}, \text{cons}(\sigma_3, \text{app}(\text{flat}(\sigma_4), \sigma_5))) \wedge \right. \\ & \quad (\text{app}(\text{app}(y, \text{cons}(\sigma_3, \text{flat}(\sigma_4))), \sigma_5) = \\ & \quad \left. \text{app}(y, \text{cons}(\sigma_3, \text{app}(\text{flat}(\sigma_4), \sigma_5))) \rightarrow \right. \\ & \quad \text{app}(\text{app}(\text{cons}(x, y), \text{cons}(\sigma_3, \text{flat}(\sigma_4))), \sigma_5) = \\ & \quad \left. \text{app}(\text{cons}(x, y), \text{cons}(\sigma_3, \text{app}(\text{flat}(\sigma_4), \sigma_5)))) \right) \\ & \rightarrow \text{app}(\text{app}(\text{flat}(\sigma_2), \text{cons}(\sigma_3, \text{flat}(\sigma_4))), \sigma_5) = \\ & \quad \text{app}(\text{flat}(\sigma_2), \text{cons}(\sigma_3, \text{app}(\text{flat}(\sigma_4), \sigma_5))) \end{aligned} \quad (29)$$

The resulting clauses – after binary resolving with the intermediate unit clause (28) – can be finally refuted using the definitions at lines 11 and 12 of Figure 1. We thus validate correctness of the assertion on line 17 in Figure 1. \square

VI. MULTI-CLAUSE INDUCTION IN SUPERPOSITION

The induction rule (Ind) does not allow inducting on multiple literals, limiting for example the use of (Ind) over (14) in Example 2. Moreover, when (Ind) is used together with the induction formula (19), clausification introduces new Skolem constants, making it impossible to use ground assumptions or previous induction hypotheses containing different ground

subterms. To address this issue, in this section we revise the induction inference rule (Ind) with only one premise to an *induction rule with multiple premises*, as follows.

We extend (Ind) for a given literal \bar{L} (the *main literal*) to also incorporate other literals L_i (the *side literals*) that are relevant for proving \bar{L} , as follows:

$$\frac{L_1[\bar{t}] \vee C_1 \quad \dots \quad L_n[\bar{t}] \vee C_n \quad \bar{L}[\bar{t}] \vee C}{\text{cnf}(F \rightarrow \forall \bar{y}. (\bigwedge_{1 \leq i \leq n} L_i[\bar{y}] \rightarrow \bar{L}[\bar{y}]))} \text{ (IndMC)}$$

where \bar{L} and L_i are ground literals, C and C_i are clauses, and $F \rightarrow \forall \bar{y}. (\bigwedge_{1 \leq i \leq n} L_i[\bar{y}] \rightarrow \bar{L}[\bar{y}])$ is a valid induction formula. Further, \bar{y} and \bar{t} are tuples of variables and induction terms, respectively. Soundness of (IndMC) follows then from soundness of (Ind).

Theorem 3 (Soundness of Multi-clause Induction). The rule (IndMC) is sound. \square

We note that after the application of (IndMC), binary resolution can be applied on each resulting clause with the main and side literals, yielding $\text{cnf}(\neg F) \vee \bigvee_{1 \leq i \leq n} C_i \vee C$.

Multi-Clause Induction Formula over Active and Accumulator Terms. For generating valid induction formulas to be used in (IndMC), we proceed as in Section IV. Yet, we adjust the generation of (19), by using Definition 3 over the active and accumulator terms of $\bigwedge_{k=1}^n L_k[\bar{c}'] \rightarrow L[\bar{c}']$ (rather than just $L[\bar{c}]$). As a result, for a given case distinction $\Theta_{\mathbf{f}(\bar{c})}$, we generate the following *multi-clause induction formula over active and accumulator terms* in saturation-based proving:

$$\begin{aligned} & (\forall) \quad \bigwedge_{(\theta, R) \in \Theta_{\mathbf{f}(\bar{c})}} \left(\bigwedge_{\theta' \in R} (\bigwedge_{k=1}^n L_k[\bar{c}'] \theta' \rightarrow L[\bar{c}'] \theta') \rightarrow \right. \\ & \quad \left. (\bigwedge_{k=1}^n L_k[\bar{c}'] \theta \rightarrow L[\bar{c}'] \theta) \right) \rightarrow (\bigwedge_{k=1}^n L_k[\bar{c}'] \rightarrow L[\bar{c}']) \end{aligned} \quad (30)$$

Since (30) is a valid induction formula, using it in the conclusion of (IndMC) yields a sound (IndMC) inference.

Example 7. Negating and clausifying the assertion on line 18 of Figure 1, we obtain the two unit clauses:

$$\text{even}(\sigma_1) \quad (31)$$

$$\neg \text{even}(\text{mul}(\sigma_0, \sigma_1)) \quad (32)$$

Inducting on (32) using $\Theta_{\text{mul}(\sigma_0, \sigma_1)}$ and induction term σ_0 , we get the following clauses:

$$\neg \text{even}(\text{mul}(\text{zero}, \sigma_1)) \vee \text{even}(\text{mul}(\sigma_2, \sigma_1))$$

$$\neg \text{even}(\text{mul}(\text{zero}, \sigma_1)) \vee \neg \text{even}(\text{mul}(\text{s}(\sigma_2), \sigma_1))$$

By function and predicate definitions of mul and even , the base case reduces to false and we are left with the unit clauses

$$\text{even}(\text{mul}(\sigma_2, \sigma_1)) \quad (33)$$

$$\neg \text{even}(\text{add}(\text{mul}(\sigma_2, \sigma_1), \sigma_1)) \quad (34)$$

The hypothesis literal in (33) and the conclusion literal in (34) cannot be binary resolved with each other to solve the step case but they share the term $\text{mul}(\sigma_2, \sigma_1)$. We can use (33)

and (34) in (IndMC) as side and main literals, respectively, with induction term $\text{mul}(\sigma_2, \sigma_1)$ and the case distinction:

$$\Theta_{\text{even}(\text{mul}(\sigma_2, \sigma_1))} = \left\{ \begin{aligned} &(\{z \mapsto \text{zero}\}, \emptyset), (\{z \mapsto \text{s}(\text{zero})\}, \emptyset), \\ &(\{z \mapsto \text{s}(\text{s}(x))\}, \{\{z \mapsto x\}\}) \end{aligned} \right\}$$

We get the following induction formula:

$$\begin{aligned} \forall x, . &((\text{even}(\text{zero}) \rightarrow \text{even}(\text{add}(\text{zero}, \sigma_1))) \wedge \\ &(\text{even}(\text{s}(\text{zero})) \rightarrow \text{even}(\text{add}(\text{s}(\text{zero}), \sigma_1))) \wedge \\ &((\text{even}(x) \rightarrow \text{even}(\text{add}(x, \sigma_1))) \rightarrow \\ &(\text{even}(\text{s}(\text{s}(x))) \rightarrow \text{even}(\text{add}(\text{s}(\text{s}(x)), \sigma_1)))) \\ &\rightarrow (\text{even}(\text{mul}(\sigma_2, \sigma_1)) \rightarrow \text{even}(\text{add}(\sigma_2, \sigma_1)))) \end{aligned} \quad (35)$$

After clausifying (35), and binary resolving the resulting clauses against (33) and (34), using function and predicate definitions and the unit clause (31), we arrive at the empty clause, thus validating the assertion at line 18 in Figure 1. \square

We conclude this section by noting that the (IndMC) inference rule might use an arbitrary number of side literals, slowing down the practical efficiency of saturation-based proving with multi-clause induction. As a remedy, the following two heuristics could be used to choose the literal L from clause $L \vee C$ as a side literal of (IndMC): (i) if L is $\text{p}(\bar{s})$ for some predicate p , and L is an induction hypotheses to the main literal $\text{p}(\bar{t})$, and \bar{s} and \bar{t} share some non-Skolem (complex) term with an active occurrence, or (ii) if neither L nor the main literal are derived from a clausified induction formula and they share some common term with an active occurrence.

VII. EXPERIMENTS

Implementation. We implemented our approach to automating induction with recursive definitions in superposition-based theorem prover VAMPIRE. We extended VAMPIRE’s induction framework [18] with recursive definitions and hypothesis strengthening, as described in Section IV. This can be enabled with `--structural_induction_kind rec_def`. Rewriting with induction hypotheses and function definitions, as presented in Section V, can be switched on using `--induction_hypothesis_rewriting on` and `--function_definition_rewriting on`, respectively. The multi-clause induction rule from Section VI is enabled by `--induction_multiclaue on`. All together, our implementation consists of around 5,000 lines of C++ code and is available at <https://github.com/vprover/vampire/tree/induction-recursive-functions>.

Experimental setup. To experimentally evaluate our approach, we used the benchmarking tool BENCHEXEC [27], [28] and two benchmark sets²: (i) the UFDTLIA examples from SMT-LIB [24], consisting of 327 problems over algebraic data types; and (ii) our new set `dtv_RD` of 3,397 inductive examples with recursive definitions, as described in [30]. We used the keyword `define-fun-rec` for defining recursive functions in the examples from our `dtv_RD` dataset. Moreover,

²While some examples from the TIP library [29] are included in SMT-LIB, most of the TIP examples are parametric and not yet supported by VAMPIRE.

	UFDTLIA 327 problems	dtv_RD 3,397 problems
VAMPIRE	180 (0)	1,641 (0)
VAMPIRE*	259 (30)	3,223 (497)
ZIPPERPOSITION	174 (0)	2,534 (21)
CVC4	235 (12)	165 (0)

Fig. 2. Numbers of problems solved by respective solvers in our experiments. The number in parentheses is the number of problems solved uniquely compared to the other solvers.

we also converted examples from the UFDTLIA set to explicitly use `define-fun-rec`, detecting this way recursive definitions in UFDTLIA.

We also combined our inductive approach in VAMPIRE with recent developments in first-order reasoning [18], [31], [32], creating this way various VAMPIRE configurations for automating induction with recursive definitions. The *default options* we used for these configurations are: `--induction_gen on` `--induction_on_complex_terms on` enabling inductive generalizations and induction on complex terms [18]; `--newcnf on` to select the `cnf` method in [31]; and `--theory_split_queue on` `--theory_split_queue_cutoffs 0,8` and `--theory_split_queue_ratios 20,10,1` to control theory reasoning with split queues [32]. As a result, we designed a new VAMPIRE portfolio mode for inductive reasoning, which can be switched on by `--mode portfolio --schedule struct_induction`.

Experimental comparison. In what follows, VAMPIRE refers to the (default) version of VAMPIRE, as in [18]. By VAMPIRE* we denote our new version of VAMPIRE, using induction with recursive definitions and the aforementioned options. We compared our work in VAMPIRE* against VAMPIRE, as well as against the superposition prover ZIPPERPOSITION³ [16] and the SMT solver CVC4 [33].

Since the default mode of VAMPIRE and VAMPIRE* only occasionally solves unique problems with respect to their portfolio mode counterpart, we omitted the former results. Note that we used the same portfolio schedule `struct_induction` for VAMPIRE as well. Since in portfolio mode VAMPIRE ignores the new options and most of the schedule is not specific to VAMPIRE*, the results obtained for VAMPIRE give a meaningful baseline. We used ZIPPERPOSITION in the default mode, while for CVC4 we used the parameters `--conjecture-gen` `--quant-ind`. Each prover was given 300 seconds of time and 16 GB of memory per problem. The experiments were ran on computers with 32 cores (AMD Epyc 7502, 2.5 GHz) and 1 TB RAM.

Experimental results. We summarize our experimental results in Figure 2. For each solver, listed in the first column of

³ZIPPERPOSITION has a non-official option `--input tip` to parse benchmarks in a variant of SMT-LIB. In order to parse UFDTLIA benchmarks, we converted them to this variant.

VAMPIRE* forced option	UFDTLIA 327 problems	dtv_RD 3,397 problems
default	259 (1)	3223 (3)
-indmc off	237 (0)	3259 (33)
-indhrw off	242 (0)	3192 (4)
-fnrw off	237 (3)	3001 (0)
-sik one	200 (1)	962 (0)

Fig. 3. Numbers of problems solved by VAMPIRE* with different new features disabled. The number in parentheses is the number of problems solved uniquely compared to the other configurations.

Figure 2, we indicate the total number of examples the solver proved from the respective benchmark category; the values in parentheses show the number of uniquely solved problems compared to the other solvers. Figure 2 shows that while VAMPIRE performs reasonably well on both benchmark sets, it cannot solve more problems than CVC4 in the UFDTLIA set and than ZIPPERPOSITION in the dtv_RD set, where the latter two perform the best. VAMPIRE*, on the other hand, is able to solve many more problems than the other solvers in both sets, suggesting that combining the state-of-the-art techniques of superposition with induction over recursive definition can perform much better than SMT solvers and superposition provers with only structural induction. All together, **VAMPIRE* solved 527 new problems that the other automated solvers could not prove.** It is also worth noting that while VAMPIRE* dominates the uniquely solved problems w.r.t. the dtv_RD set, its dominance is only marginal compared to the uniquely solved problems of CVC4 in the UFDTLIA set. Looking at the problems uniquely solved by CVC4, we found that these problems mostly contain either some nested structure that current techniques in VAMPIRE* cannot handle and require non-trivial lemma generation or recursive definitions that cannot be used with our induction formula generation as their well-foundedness is not based on the subterm relation.

In addition to comparing to other solvers, we compared VAMPIRE* to itself with different techniques from the paper disabled, overriding the portfolio options during these runs. Our results are shown in Figure 3.

For UFDTLIA, the default run still performs best but we can see different deviations from this value with each disabled technique. We argue that the relatively small differences obtained by turning off induction hypothesis rewriting (-indhrw off) and function definition rewriting (-fnrw off) can be attributed to combinations of options that together may simulate these techniques. In comparison, multi-clause induction cannot be simulated with other techniques in VAMPIRE, so the relatively small difference obtained by turning off this technique (-indmc off) for UFDTLIA is probably due to the lack of non-unit induction needed in most of this set. For dtv_RD, the decrease in solved problems when this feature is turned on needs further investigation. The greatest difference to the default is obtained by using

structural induction (-sik one, see [17]) instead of inferring induction formulas from recursive function definitions. We can conclude with the observation that each configuration solved problems uniquely which suggests the portfolio schedule can be improved.

VIII. RELATED WORK

Generation of induction formulas, as presented in Section IV, although similar to *recursion analysis* of [7] and *recursion induction* of [10], utilizes unification and generates non-trivial induction hypotheses. Our work complements these techniques by integrating induction in saturation: rather than replacing inductive goals by sub-goals/other formulas, we generate induction formulas over recursive definitions and add these induction formulas as additional properties to the search space.

When compared to superposition approaches treating certain *E*-theories [19] or function definitions as rewrite rules [16], we note that our method designs new induction inference rules as simplification rules in superposition and strengthens induction hypotheses during saturation-based inductive reasoning. Our approach extends [17] by handling recursive definitions as rewrite rules and multiple clauses in a single induction step; the latter is often required when assumptions are supported in universally quantified conjectures. Unlike [16], our technique generalizes to scenarios where multiple induction steps are needed to refute non-equality literals. Contrarily to [12], we are not limited to induction over term algebras as most of these techniques work for e.g. mathematical induction as well.

While our approach often does not need auxiliary lemmas due to generalizations over (complex) term occurrences and strengthened induction hypotheses, extending our work towards lemma generation would be beneficial. In particular, theory exploration and lemma generation approaches from [8], [15], [10], [34], [35], [13] could complement our method, ranging from randomly generating terms by iterative deepening to analysing failed induction steps and even circumventing the need for auxiliary lemmas by using predicates.

IX. CONCLUSION

We introduce a new approach for automating induction with recursive definition in first-order theorem proving. We design new inference rules for rewriting with function definitions as well as induction hypotheses in superposition-based proving. We generate induction formulas based on recursive function definitions and extend our work to support multi-clause induction. Our experiments show that induction with recursive definitions in superposition allows us to solve many new problems that other automated reasoners failed to prove.

ACKNOWLEDGMENTS

This work was partially funded by the ERC CoG ARTIST 101002685, the ERC StG SYMCAR 639270, the EPSRC grant EP/P03408X/1, the FWF grant LogiCS W1255-N23, the Amazon ARA 2020 award FOREST and the TU Wien SecInt DK.

REFERENCES

- [1] B. Cook, “Formal Reasoning About the Security of Amazon Web Services,” in *CAV*, H. Chockler and G. Weissenbacher, Eds. Springer, 2018, pp. 38–47.
- [2] “SHA-2 Cryptographic Hash Standard,” National Institute of Standards and Technology, 2002. [Online]. Available: <https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2withchangenotice.pdf>
- [3] J. A. Goguen and J. Meseguer, “Security Policies and Security Models,” in *S&P*, 1982, pp. 11–20.
- [4] A. Bundy, “The Automation of Proof by Mathematical Induction,” in *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds., 2001, vol. I, ch. 13, pp. 845–911.
- [5] J. S. Moore and C.-P. Wirth, “Automation of Mathematical Induction as part of the History of Logic,” *CoRR*, vol. abs/1309.6226, 2013. [Online]. Available: <http://arxiv.org/abs/1309.6226>
- [6] W. Sonnex, S. Drossopoulou, and S. Eisenbach, “Zeno: An automated prover for properties of recursive data structures,” in *TACAS*, C. Flanagan and B. König, Eds. Springer, 2012, pp. 407–421.
- [7] R. S. Boyer and J. S. Moore, *A Computational Logic Handbook*. Academic Press, 1988.
- [8] A. Bundy, D. Basin, D. Hutter, and A. Ireland, *Rippling: Meta-Level Guidance for Mathematical Reasoning*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005.
- [9] G. O. Passmore, S. Cruanes, D. Ignatovich, D. Aitken, M. Bray, E. Kagan, K. Kanishev, E. Maclean, and N. Mometto, “The Imandra Automated Reasoning System (System Description),” in *IJCAR*, N. Peltier and V. Sofronie-Stokkermans, Eds. Springer, 2020, pp. 464–471.
- [10] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, “Automating Inductive Proofs Using Theory Exploration,” in *CADE*, M. P. Bonacina, Ed. Springer, 06 2013, pp. 392–406.
- [11] I. L. Valbuena and M. Johansson, “Conditional Lemma Discovery and Recursion Induction in Hipster,” *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 72, 2015. [Online]. Available: <https://doi.org/10.14279/tuj.eceasst.72.1009>
- [12] M. Echenheim and N. Peltier, “Combining Induction and Saturation-Based Theorem Proving,” *J. Automated Reasoning*, vol. 64, pp. 253–294, 2020.
- [13] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, “Removing Algebraic Data Types from Constrained Horn Clauses Using Difference Predicates,” in *IJCAR*, N. Peltier and V. Sofronie-Stokkermans, Eds. Springer, 2020, pp. 83–102.
- [14] L. Kovács, S. Robillard, and A. Voronkov, “Coming to Terms with Quantified Reasoning,” in *POPL*, G. Castagna and A. D. Gordon, Eds., 2017, pp. 260–270.
- [15] A. Reynolds and V. Kuncak, “Induction for SMT Solvers,” in *VMCAI*, D. D’Souza, A. Lal, and K. G. Larsen, Eds. Springer, 2015, pp. 80–98.
- [16] S. Cruanes, “Superposition with Structural Induction,” in *FroCoS*, C. Dixon and M. Finger, Eds. Springer, 2017, pp. 172–188.
- [17] G. Reger and A. Voronkov, “Induction in saturation-based proof search,” in *CADE*, P. Fontaine, Ed. Springer, 2019, pp. 477–494.
- [18] P. Hozzová, M. Hajdú, L. Kovács, J. Schoisswohl, and A. Voronkov, “Induction with Generalization in Superposition Reasoning,” in *CICM*, C. Benz Müller and B. Miller, Eds. Springer, 2020, pp. 123–137.
- [19] R. Nieuwenhuis and A. Rubio, “Paramodulation-Based Theorem Proving,” in *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds., 2001, vol. I, ch. 7, pp. 371–443.
- [20] S. Schulz, S. Cruanes, and P. Vukmirović, “Faster, Higher, Stronger: E 2.3,” in *CADE*, P. Fontaine, Ed. Springer, 2019, pp. 495–507.
- [21] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischniewski, “SPASS Version 3.5,” in *CADE*, R. A. Schmidt, Ed. Springer, 2009, pp. 140–145.
- [22] L. Kovács and A. Voronkov, “First-Order Theorem Proving and Vampire,” in *CAV*, N. Sharygina and H. Veith, Eds. Springer, 2013, pp. 1–35.
- [23] A. Voronkov, “AVATAR: The Architecture for First-Order Theorem Provers,” in *CAV*, A. Biere and R. Bloem, Eds. Springer, 2014, pp. 696–710.
- [24] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” www.SMT-LIB.org, 2016.
- [25] L. Dixon and J. Fleuriot, “IsaPlanner: A Prototype Proof Planner in Isabelle,” in *CADE*, F. Baader, Ed. Springer, 2003, pp. 279–283.
- [26] C. Walther, “Computing Induction Axioms,” in *LPAR*, A. Voronkov, Ed. Springer, 1992, pp. 381–392.
- [27] D. Beyer, S. Löwe, and P. Wendler, “Reliable Benchmarking: Requirements and Solutions,” *Int. J. on Software Tools for Technology Transfer*, vol. 21, no. 1, pp. 1–29, 2019.
- [28] D. Beyer, “Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016),” in *TACAS*, M. Chechik and J.-F. Raskin, Eds. Springer, 2016, pp. 887–904.
- [29] N. Smallbone, M. Johansson, and K. Claessen, “Tons of Inductive Problems (TIP),” tip-org.github.io, Springer, pp. 333–337, 2015.
- [30] M. Hajdu, P. Hozzová, L. Kovács, J. Schoisswohl, and A. Voronkov, “Inductive Benchmarks for Automated Reasoning,” in *CICM*, 2021, to appear. [Online]. Available: <https://easychair.org/publications/preprint/gGb9>
- [31] G. Reger, M. Suda, and A. Voronkov, “New Techniques in Clausal Form Generation,” in *GCAI*, C. Benz Müller, G. Sutcliffe, and R. Rojas, Eds., 2016, pp. 11–23.
- [32] B. Gleiss and M. Suda, “Layered Clause Selection for Theory Reasoning,” in *IJCAR*, N. Peltier and V. Sofronie-Stokkermans, Eds. Springer, 2020, pp. 402–409.
- [33] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *CAV*, G. Gopalakrishnan and S. Qadeer, Eds. Springer, 2011, pp. 171–177.
- [34] E. Singher and S. Itzhaky, “Theory Exploration Powered By Deductive Synthesis,” *ArXiv*, vol. abs/2009.04826, 2020.
- [35] A. Murali, L. Peña, C. Löding, and P. Madhusudan, “Synthesizing Lemmas for Inductive Reasoning,” *ArXiv*, vol. abs/2009.10207, 2020.

Fair and Adventurous Enumeration of Quantifier Instantiations

Mikoláš Janota

Czech Technical University in Prague

Prague, Czech Republic



Haniel Barbosa

Universidade Federal de Minas Gerais

Belo Horizonte, Brazil



Pascal Fontaine

University of Liège

Liège, Belgium



Andrew Reynolds

University of Iowa

USA



Abstract—SMT solvers generally tackle quantifiers by instantiating their variables with tuples of terms from the ground part of the formula. Recent enumerative approaches for quantifier instantiation consider tuples of terms in some heuristic order. This paper studies different strategies to order such tuples and their impact on performance. We decouple the ordering problem into two parts. First is the order of the sequence of terms to consider for each quantified variable, and second is the order of the instantiation tuples themselves. While the most and least preferred tuples, i.e. those with all variables assigned to the most or least preferred terms, are clear, the combinations in between allow flexibility in an implementation. We look at principled strategies of complete enumeration, where some strategies are more fair, meaning they treat all the variables the same but some strategies may be more adventurous, meaning that they may venture further down the preference list. We further describe new techniques for discarding irrelevant instantiations which are crucial for the performance of these strategies in practice. These strategies are implemented in the SMT solver *cvc5*, where they contribute to the diversification of the solver’s configuration space, as shown by our experimental results.

Index Terms—SMT, quantifier instantiation, enumeration

I. INTRODUCTION

While SMT (satisfiability modulo theory) solvers [5] are used successfully as decision procedures to automatically discharge quantifier-free proof obligations for many applications, there is an increasing need for tools that can furthermore handle quantifiers. Quantified languages however are most often undecidable, or have prohibiting complexity. Quantifier handling within SMT solving is thus a challenge and requires good heuristics.

Quantifier reasoning in SMT builds on the strength of SMT solvers, that is, their ability to efficiently reason on ground formulas, and relies on instantiation: ground consequences of quantified formulas are generated, and the ground reasoner’s view of the problem is gradually refined with these instances, to embed knowledge from the quantified formula into ground reasoning. The terms to generate instances may be generated using mostly syntactic methods, e.g., E-matching [6], or semantic techniques like model-based quantifier instantiation [7]. But plain enumeration, done in a principled manner, can give surprisingly good results, particularly in combination with other instantiation techniques [8].

A crucial aspect, when using enumeration-based instantiation, is to prioritize the numerous, often infinite, potential

instantiations. When instantiating just one variable, this is essentially a matter of prioritizing smaller terms that are already present in the original formula, according to some order. Quantified assertions however most often have many quantified variables, and there is a lot of freedom on the order on tuples of terms to instantiate those. We here investigate a few strategies based on different tuple orders, some favoring fairness, some being more adventurous, and show that they are valuable in a portfolio of enumerative instantiation strategies. In Section IV, we also present an elimination technique for redundant instantiations that significantly contributes to the improvement of enumeration-based instantiation.

II. BACKGROUND

Originally, SMT solvers were essentially decision procedures for ground (i.e., quantifier-free) problems in a combination of decidable languages, containing e.g., operators to handle arrays, linear arithmetic expressions, bitvectors, and uninterpreted predicates and functions. They excel at deciding the satisfiability of large formulas in these languages. As a toy example, consider the (satisfiable) conjunctive set of formulas

$$\{R(a), \neg S(b), a = b\}.$$

It belongs to the quantifier-free fragment of first-order logic, and as such, is decided by many SMT solvers. Quantifier reasoning in modern SMT solvers builds on this. The input formula, possibly after a pre-processing phase, is first given to the ground solver. From the point of view of this ground solver, each quantified formula is abstracted into a distinct propositional variable. As an example, the conjunctive set

$$\{R(a), \neg S(b), a = b, \forall x. R(x) \Rightarrow S(x)\}$$

is understood by the ground solver as the previous ground set, augmented with an abstract proposition Q corresponding to $\forall x. R(x) \Rightarrow S(x)$. Then the ground solver provides a satisfying assignment for the ground part of the formula, including a valuation of the propositional variables abstracting the quantified formulas (in our case Q must be true). The instantiation module recovers the quantified formulas associated to these variables, and generates new instances of the quantified formulas to the ground reasoner (Figure 1). In our toy example such an instance could be

$$Q \Rightarrow (R(a) \Rightarrow S(a)),$$

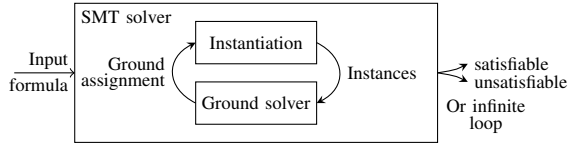


Fig. 1. The SMT instantiation loop.

which would render the problem unsatisfiable at the ground level. In general, the instantiation loop is iterated until the ground reasoner is able to conclude that the formula is unsatisfiable, a time out is reached, or no instance can be deduced anymore. In this paper, we focus on refutations only and will not consider the last case.

Thanks to the Herbrand Theorem (see e.g., [8]), with fair enumeration of instances using all possible terms built on the appropriate set of symbols, SMT solving is refutationally complete for satisfiability modulo well-behaved first-order theories. Since typical SMT inputs contain hundreds of quantified formulas with many nested quantifiers, on a language with often infinitely many terms, the number of possible instances is very large, and most often infinite. It is crucial to quickly find out the right instances, otherwise the ground solver will be overwhelmed by the amount of instances. For a quantified formula $\forall x_1 \dots x_n. \varphi$ with n variables, this boils down to ordering n -tuples of ground terms to prioritize instantiation.

III. ENUMERATION STRATEGIES

We start by the assumption that for each variable x_i there is a sequence of terms $\mathcal{T}_i = t_i^1, t_i^2, \dots$, which are the possible candidates for instantiation into the variable x_i . We further assume that this sequence of terms is sorted by some given preference, i.e., that t_i^j is more likely to yield a useful instantiation than the candidate $t_i^{j'}$ with $j < j'$. This lets us focus on the indices into the sequences of terms, rather than on the terms themselves. An instantiation, i.e., a tuple of terms, is uniquely represented as an n -tuple of indices.

While this setup already assumes a given order on the terms for the individual variables, it does not tell us how to order the actual tuples. Clearly, the tuple of indices $(0, \dots, 0)$ is the most advantageous and $(|\mathcal{T}_1| - 1, \dots, |\mathcal{T}_n| - 1)$ is the least advantageous one. However, it is unclear whether $(0, 1, 1)$ is more advantageous than $(0, 0, 2)$, or the other way around. This motivates our quest for different enumeration strategies. A general notion from multi-objective optimization is useful: *Pareto-optimal* solutions are such that improving any criterion worsens some other.

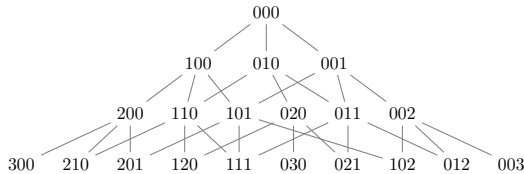


Fig. 2. Pareto graph for 3 variables with 4 candidate terms for each.

Definition 1 (Pareto dominates). Let $t_1 = (a_1, \dots, a_n)$ and $t_2 = (b_1, \dots, b_n)$ be n -tuples of integers. We say that t_1 Pareto dominates t_2 , if and only if $t_1 \neq t_2$ and $a_i \leq b_i$ for all $i \in 1..n$.

We focus on traversals of the graph of tuples where traversing an edge increases one of the indices. Hence, there is an edge from tuple t_1 to tuple t_2 iff t_2 is obtained by increasing either of the digits of t_1 by 1; see Figure 2. This graph anchors our initial motivation that the order on the terms pertaining to a single variable represents preference. Indeed, following down any edge in this graph means going to a less preferred tuple. We call this graph the *Pareto graph*.

So what does differentiate one traversal from another? In graph theory vernacular, a traversal is broad or deep. In our context, a broad traversal is more *fair* since it alters terms of different variables evenly. A deep traversal is more *adventurous* since it opts for less preferred, i.e., riskier, instantiations.

Fair strategies observe the Pareto ordering, meaning that no tuple dominates any of the previous tuples. For instance, the sequence $(0, 0), (0, 1), (1, 0), (1, 1)$ respects Pareto ordering but $(0, 0), (0, 1), (1, 1), (1, 0)$ does not because $(1, 0)$ Pareto-dominates $(1, 1)$. Note that both of these examples respect the Pareto graph in the sense that a node is visited only if at least one of its predecessors has been visited.

In the remainder of the section we introduce techniques considered in the experimental evaluation in Section V. On a technical note, in practice the number of possible candidates per variable may vary, but for the sake of clarity, we assume that each variable has the same number of possible candidate terms. This means that every element of the tuple (digit) is in the range $0..M$ for some fixed $M \in \mathbb{N}$. Effectively, this means that we are looking for systematic enumerations of tuples from the space $[0..M]^n$, with a fixed set of n variables.

A. Stages by maximal digit [8]

This ordering interprets tuples as numbers in increasing base $b \in 2..(M + 1)$. As an example, consider two variables and $M = 2$. The enumeration starts with base 2, yielding: $(0, 0), (1, 0), (0, 1), (1, 1)$. Subsequently, it switches to base 3, while skipping already enumerated tuples, giving the rest of the tuples: $(2, 0), (2, 1), (0, 2), (1, 2), (2, 2)$.

This is a natural alternative to interpreting the tuples as numbers in base $M + 1$, which would lead to a highly unfair strategy because large values of M would lead to changing significant digits very late.

This ordering observes Pareto domination and the enumeration algorithm runs in constant space.

B. Stages by sum of digits

The maximum digit approach mitigates unfairness in large value of M (large number of candidate terms). However, it still leads to an imbalance with a large number of quantified variables, i.e., with large tuples. Indeed, even with $M = 1$ already 10 variables require 2^{10} iterations before the most significant digit is changed. The alternative is to iterate over combinations stratified by the *sum of all the digits*. Tuples with the same sum of digits are ordered lexicographically.

This leads to a breadth first traversal of the Pareto graph and its effect is more pronounced with large number of variables. The initial sequence has the following form:

$$(0, 0, \dots, 0), (1, 0, \dots, 0), (0, 1, \dots, 0), \dots, (0, 0, \dots, 1), \\ (2, 0, \dots, 0), (1, 1, \dots, 0), (0, 2, \dots, 0), \dots$$

This ordering also observes the Pareto domination and can be calculated in constant space.

C. Leximax

Arguably the most fair strategy is enumeration according to the *leximax order* [1] since all the variables are in equivalent roles: let t_1, t_2 be n -tuples of integers. We say that t_1 is *leximax preferred* to t_2 if t_1^\downarrow is lexicographically smaller than t_2^\downarrow , where t^\downarrow denotes t sorted in descending order. Enumeration can be done in constant space. We observe that all permutations of a tuple are incomparable. This enables us to stage the enumeration by gradually worsening a sorted tuple and enumerate all its permutations through standard means. The incomparable permutations are enumerated lexicographically. For two variables the sequence starts as follows, $(0, 0), (0, 1), (1, 0), (1, 1), (0, 2), (2, 0)$. Contrast that with the sum of digits $(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0)$.

D. Iterative Deepening and Random-walk Search

Strategies discussed so far never violate Pareto domination, which would be violated by depth-first but that would have a large degree of unfairness. Instead, we propose to use *iterative deepening* where the maximum depth is incremented by some fixed parameter $k \in \mathbb{N}^+$. Maximum depth 2 yields $(0, 0), (0, 1), (0, 2), (1, 1), (1, 0), (2, 0)$, where $(1, 0)$ Pareto-dominates $(1, 1)$, even though it comes later in the sequence.

As another very adventurous strategy, we propose *random-walk traversal*, which is similar to DFS but instead of a stack we use a set where the next element is chosen randomly.

IV. DISCARDING REDUNDANT INSTANTIATIONS

When solving quantified formulas, SMT solvers are often hindered by an overabundance of generated instantiations. Thus, it is paramount to avoid instantiations that are *redundant*. At a high level, an instantiation is considered redundant if it does not help rule out models in the current context. Methods for discovering redundant instantiations are particularly important in the context of enumerative instantiation, where typically we are iterating over similar domains of terms on multiple instantiation rounds, and are looking for the first instantiation that is not redundant.

In our implementation, we consider three criteria for determining that an instantiation $\varphi \cdot \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ is redundant, in increasing order of cost:

- 1) (Duplicate Term Vector) For each φ , maintain a trie containing all term vectors of its previous instantiations. If (t_1, \dots, t_n) is already in this trie, then the instantiation is redundant.
- 2) (Entailed) As described in [8, Section 4.1], a fast incomplete method for entailment is used for discovering

when an instantiation lemma is already implied by the current set of constraints known by the SMT solver. All instantiations that are entailed are considered redundant.

- 3) (Duplicate Formula Modulo Rewriting) Maintain a set of previous formulas returned by quantifier instantiation. Construct the formula $\varphi \cdot \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ and normalize it using rewriting techniques. If the resulting formula is already in our set, it is redundant.

If none of these criteria hold, the instantiation is not considered redundant.

It is important to note that the latter two methods allow one to learn that a *class* of instantiations is redundant. For this purpose, we introduce the concept of a *fail mask* for an instantiation. A fail mask \mathcal{M} for a substitution $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ is a sequence of n bits such that all substitutions that extend $\{x_i \mapsto t_i \mid \text{the } i^{\text{th}} \text{ bit of } \mathcal{M} \text{ is set}\}$ when applied to φ result in a redundant instantiation.

For example, let φ be the formula $P(x_1, x_2) \vee Q(x_2, x_3)$, and consider the substitution $\sigma = \{x_1 \mapsto a, x_2 \mapsto b, x_3 \mapsto c\}$. Let $E = \{P(a, b), \neg Q(b, c)\}$ be the current set of assertions from the ground solver. The instantiation $\varphi \cdot \sigma$ is redundant; a fail mask for σ is 110, since $P(a, b) \vee Q(b, x_3)$ is entailed by E for any value of x_3 .

We incorporate fail masks into our implementation in the following way. When an instantiation $\varphi \cdot \sigma$ is discovered to be redundant, we construct the fail mask \mathcal{M} containing all 1s. Starting with $i = 1$, we drop the entry $\{x_i \mapsto t_i\}$ from σ . If the instantiation is still redundant based on the latter two criteria above, then we set the i^{th} bit to 0. If not, then we re-add the entry $\{x_i \mapsto t_i\}$ to σ , and proceed with $i + 1$. Notice this means that our computation of the fail mask is greedy.

The fail mask is incorporated into the enumerative strategies as follows. After each failed instantiation, combine the tuple of term indices and the fail mask into a tuple with wildcards, denoted “?”. So for instance, if the tuple $(5, 4, 3)$ fails with the mask 101, construct the tuple $(5, ?, 3)$ meaning that if the first variable is instantiated with the 5th term and the third variable with the 3rd term, the instantiation is bound to be redundant. Such combinations we wish to avoid. This is checked independently of the enumeration algorithm by storing the disabled patterns into a trie and discarding any combinations matching one of the previously disabled patterns. The trie handles the wildcard character ? specially by always matching on it.

V. EXPERIMENTS

This section reports on our experimental evaluation of different tuple enumeration strategies implemented in the cvc5 SMT solver (the successor of CVC4 [3]). We performed all experiments on a cluster with Intel Xeon CPU E5-2620 CPUs with 2.1GHz and 128GB memory, providing one core, 300 seconds, and 8GB RAM for each job.

Enumerative instantiation is extensively compared with other techniques in [8], where it was concluded that interleaving E-matching with enumeration gives the best results. However, as the focus of the paper is the different enumeration

TABLE I
SUMMARY OF PROBLEMS SOLVED. BEST NON-PORTFOLIO RESULTS ARE IN BOLD.

Library	#	e	u	id2	id4	lmax	sum	rwlk	allu-port	eu-port	eallu-port	z3
TPTP	18627	7765	6989	6801	6834	6832	6922	6839	7330	9056	9292	-
UF	7668	3243	3016	2975	2963	2959	3009	2992	3120	3433	3452	2905
UFLIA	10137	7424	6024	6018	5897	6001	5980	5994	6188	7595	7615	6912
UFNIA	13509	5715	7458	7396	7384	7426	7437	7430	7620	7740	7843	6491

strategies, we run enumeration on its own. For succinctness, we omit certain details, such as relevant domain heuristic, run as proposed in [8].

Benchmarks are selected from first-order benchmarks from the TPTP library [10], version 7.4.0, and from SMT-LIB [4], 2020 release. Of 19287 first-order TPTP problems, we excluded 660 which contained polymorphic types, leaving 18627 for consideration. For SMT-LIB, we considered all problems from logics containing quantifiers and integer arithmetic, i.e., UF, UFLIA, and UFNIA, totaling 31314 problems. This selection of benchmarks was inspired by the evaluation from [8], where enumerative instantiation was shown more effective in the above sets.

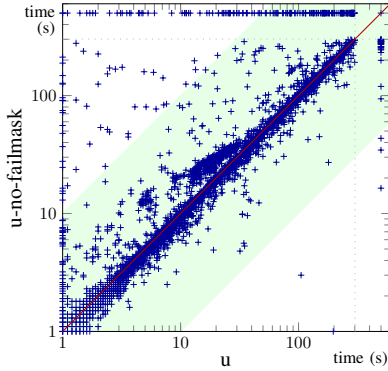


Fig. 3. Impact of elimination of redundant instantiation via fail masks.

The evaluation covers a number of cvc5 configurations. The default enumeration, maximal digit, is denoted as **u**. Its variations according to different enumeration strategies described above are **id-n** for iterative deepening with increment n ; **lmax** for leximax; **sum** of digits; and **rwlk** for random walk. We also run, for control, cvc5’s E-matching (denoted **e**) and z3 4.8.10 (denoted **z3**). By default z3 uses a combination of E-matching and model-based quantifier instantiation. All the cvc5 configurations run with the fail-masks technique enabled; further, they use conflict-based instantiation [2], [9] as a “fail-fast” technique, given its strong focusing effect. The implementation of E-matching in cvc5 already uses a redundancy checking mechanism [2], which is always enabled in our experiments. The z3 evaluation is restricted to SMT-LIB, given its limited support for TPTP.

The results are summarized in Table I. The column **allu-port** is a virtual best solver (**vbs**) of all the enumerative configuration, **eu-port** of a vbs of only e and u, and eallu-

TABLE II
SUMMARY PROBLEMS SOLVED UNIQUELY PER STRATEGY.

Library	#	e	u	id2	id4	lmax	sum	rwlk	z3
TPTP	18627	1862	27	5	22	12	14	17	-
UF	7668	160	0	0	5	3	1	2	126
UFLIA	10137	370	0	3	1	3	1	1	49
UFNIA	13509	76	3	8	9	12	2	9	547

port a vbs of all cvc5 configurations. We first emphasize the tremendous advantage in UFNIA of u over e, which can be explained by many benchmarks needing instantiations with key arithmetic constants, such as 0, to enable the necessary ground reasoning to solve the problem. However, a large number of these benchmarks may be impossible to solve via E-matching alone: if matching needs to be done on terms containing arithmetic operators, e.g. to match $x + 1$ with 1, E-matching will fail, whereas enumerative instantiation would instantiate the formula regardless. Moreover, the different enumeration strategies do lead to significant orthogonality among the different configurations. The number of uniquely solved problems per strategy is shown in Figure II. Note also that the vbs of the enumerative configurations versus u reduces the number of *unsolved* problems in UFNIA in almost 3%, while eallu-port vs eu-port reduces the number of unsolved in almost 2%. These improvements are also present in TPTP, with similar reductions in the number of unsolved problems when considering all the enumeration strategies in a virtual best solver. This clearly shows the benefit of integrating into actual portfolios different enumeration strategies rather than having just the default one.

We also evaluated an even more adventurous enumeration strategy than those in Table I, which randomly changes the strategy at each instantiation round, thus effectively simultaneously trying all the strategies. This random strategy performs similarly to the others but can be deeply influenced by the random seed chosen for selecting a strategy each round, to the extent that changing the seed from 0 to 7 makes it go, in UFLIA, from 6007 successes to 6047. This further reinforces the usefulness of diversifying the set of strategies used for quantifier instantiation in practice.

Discarding classes of redundant instantiations using fail masks gives a clear advantage as illustrated in Figure 3 (default enumerative instantiation strategy, on all benchmarks). Using the fail masks leads to 217 uniquely solved problems, whereas without it only 31 problems are solved uniquely.

Moreover, a large number of commonly solved problems have very significant speed-ups, as the plot makes clear. These improvements can be explained by the technique being the most effective in problems containing quantifiers with many variables, which are common occurrences among the benchmark sets we considered. On problems where the fail masks do not help, the overhead of computing and checking them is noticeable (see the often prevalent crosses just below the diagonal line). However, it is far from a deterrent, given the significant gains.

VI. CONCLUSIONS

Enumerative instantiation is powerful, versatile, and offers a lot of freedom for strategies. We presented several ordering heuristics for instantiation that contribute to the orthogonality of the strategies, and ultimately improve the SMT solver's performance and robustness. This is especially useful when a user is willing to employ a barrage of solver configurations to tackle a high-priority problem instance.

In future work, we plan to investigate the applications of enumerative instantiation strategies for portfolio approaches to SMT solving. We also would like to pursue more advanced techniques where tuple and term orderings are not fixed and may be influenced by previous successes or failures.

ACKNOWLEDGMENTS

We thank Mathias Preiner for helping with scripts for computing the experimental results. The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

REFERENCES

- [1] Salvador Barberà and Matthew Jackson. Maximin, leximin, and the protective criterion: Characterizations and comparisons. *Journal of Economic Theory*, 46(1):34–44, 1988.
- [2] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 214–230, 2017.
- [3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification (CAV)*, pages 171–177. Springer, 2011.
- [4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.
- [5] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking.*, pages 305–343. Springer, 2018.
- [6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
- [7] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Computer Aided Verification, 21st International Conference, CAV*, pages 306–320, 2009.

- [8] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10806, pages 112–131, 2018.
- [9] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods In Computer-Aided Design (FMCAD)*, pages 195–202. IEEE, 2014.
- [10] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

Mathematical Programming Modulo Strings

Ankit Kumar  and Panagiotis Manolios 

Northeastern University

Email: {ankitk, pete}@ccs.neu.edu

Abstract—We introduce TranSeq, a non-deterministic, branching transition system for deciding the satisfiability of conjunctions of string equations. TranSeq is an extension of the Mathematical Programming Modulo Theories (MPMT) constraint solving framework and is designed to enable useful and computationally efficient inferences that reduce the search space, that encode certain string constraints and theory lemmas as integer linear constraints and that otherwise split problems into simpler cases, via branching. We have implemented a prototype, SeqSolve, in ACL2s, which uses Z3 as a back-end solver. String solvers have numerous applications, including in security, software engineering, programming languages and verification. We evaluated SeqSolve by comparing it with existing tools on a set of benchmark problems and our experimental results show that SeqSolve is both practical and efficient.

I. INTRODUCTION

The problem of solving string equations has interested mathematicians and computer scientists for decades. Security, software engineering and verification applications, in particular, have generated a renewed interest in string solvers. Security applications include finding cross-site scripting vulnerabilities in Web applications, SQL injection attacks and fuzzing [1], [2], [3], [4], [5]. Software engineering applications include testcase generation, symbolic evaluation and flow analysis [6], [7], [8]. Programming language applications include type inference for array processing languages [9][10].

The basic problem is easy to define. Let Γ be a non-empty set of constants. The elements of Γ^* form a free monoid, *i.e.*, a structure with a single associative operation, corresponding to concatenation, and an identity element ϵ . Elements of Γ^* are called strings or words. Let \mathcal{X} be a set of variables over Γ^* and let \mathcal{Y} be a set of variables over Γ such that Γ , \mathcal{X} and \mathcal{Y} are disjoint. Elements in \mathcal{Y} are also called *unit variables*. Let $\mathcal{Z} = \mathcal{X} \cup \mathcal{Y}$. Elements of the free monoid $(\Gamma \cup \mathcal{Z})^*$ are called sequences, again with ϵ as the identity. A *normal substitution* is a partial function $\rho : \mathcal{Z} \rightarrow (\Gamma \cup \mathcal{Z})^*$. Every substitution can be extended to the domain $(\Gamma \cup \mathcal{Z})^*$, by defining $\rho(a) = a$ for all a not in the domain of ρ . We can also extend the domain to $(\Gamma \cup \mathcal{Z})^*$ in the standard way. $w\rho$ stands for the application of substitution ρ to the sequence w and it extends naturally to sequence equations. A solution of a set of equations $\{u_1 = v_1, u_2 = v_2, \dots, u_n = v_n\}$ is a substitution ρ that when applied to each equation yields identical sequences, *i.e.*, $\{u_1\rho = v_1\rho, u_2\rho = v_2\rho, \dots, u_n\rho = v_n\rho\}$ is a set of syntactic equivalences over $(\Gamma \cup \mathcal{Z})^*$. The problem statement is: given a set of sequence equations $\{u_1 = v_1, u_2 = v_2, \dots, u_n = v_n\}$ find a solution if there exists one, otherwise return *unsat*.

Related Work. Makanin, in 1977, proved that the satisfiability of string equations is decidable [11]. A series of results on complexity followed, after which Plandowski showed that the problem is in polynomial space [12]. String solvers supporting a variety of theories are available, *e.g.*, Z3Str3 [13], CVC4 [14], [15], S3P [16], Norn [17], TRAU [18], StrSolve [19], Sloth [2], Kepler₂₂ [20] and HAMPI [1]. Z3Str3 and CVC4 are multi-theory SMT solvers which consider unbounded string equations with concatenation, substring, replace and length functionality. Together with S3P and Norn, these tools handle a variety of string constraints including string equations, length constraints and regular language membership. However, these tools are incomplete. HAMPI works only for problems with one string variable of fixed size. Kepler₂₂ is a decision procedure for the straight line and quadratic fragments of string equations. Norn and TRAU can decide only the acyclic fragment whereas Sloth decides straight line and acyclic fragments. To the best of our knowledge, there is no solver that for decidable fragments is both theoretically and practically complete, *e.g.*, none of the above solvers are able to solve the string equation $xcyczvycya = yacwazvbx$. Therefore it is important to explore new techniques for solving string equations. One of the most promising existing techniques uses context-dependent techniques to improve the reasoning of string constraints in the context of DPLL(T)-based SMT solvers [15]. Similarly, our work introduces new techniques for reasoning in the context of BC(T)-based (Branch and Cut Modulo T) MPMT solvers [21], [22].

Contributions. Our contributions include (1) TranSeq, a new non-deterministic, branching transition system that can be used as part of the MPMT framework for combining decision procedures, (2) the SeqSolve solver, an implementation of TranSeq which resolves non-deterministic choices in a way designed to infer as much as possible with as few computational resources as possible, (3) proof sketches of soundness, completeness and termination for TranSeq and (4) an evaluation of SeqSolve using a set of benchmarks from related work, as well as Remora examples [9], [10]. We use publicly available benchmarks, being careful to evaluate only the string solving capabilities of our tool, not irrelevant aspects of the underlying SMT/MPMT tools. The integration of our solver into SMT/MPMT tools is briefly discussed. There are over 1,100 problems in our benchmark and no existing string solver can solve all of them. Experimental results show that

SeqSolve is more efficient and complete than existing solvers.

Paper Outline. Section II illustrates some techniques we use to reason about string equations through motivating examples. Section III defines basic terms used to define our transition system and algorithm. Section IV describes TranSeq and SeqSolve. Section V gives proofs sketches of correctness and termination; due to space limitations full definitions and proofs will appear in a full version of the paper. Section VI describes implementation considerations of our prototype and Section VII contains our evaluation. Conclusions and future work appear in Section VIII.

II. ILLUSTRATIVE EXAMPLES

In this section, we highlight some of the techniques used in our string equation solver, via a collection of examples, where $a, b, c \dots$ are constants (elements of Γ) and u, v, w, x, y and z are string variables (elements of \mathcal{X}).

Example 1 [ConstUnsat] Consider the string equation $b = a$. The constant b differs from the constant a so this equation is unsatisfiable. Our algorithm determines by performing partial evaluation that includes evaluating constant prefixes and suffixes of equations.

Example 2 [Trim] Consider $xab = xbb$. Our algorithm trims common prefixes and suffixes from both sides of the input equation to get $a = b$ which is unsatisfiable by ConstUnsat.

Example 3 [Decompose] Consider $xyazy = xybyz$. Prefixes xy and yz have provably equal lengths. So do the suffixes zy and yz . Therefore our algorithm decomposes the input equation into three equations: $xy = yx$, $a = ub$ and $zy = yz$. Equation $a = ub$ can be further decomposed into $a = b$ and $u = \epsilon$, which is unsatisfiable by ConstUnsat.

Example 4 [EqLength] Consider $uvxayvu = vuyxuv$. Decomposition generates the two *distinct* equations $uv = vu$ and $xay = yx$. Notice that if an equation is satisfiable, then both sides have to have the same length and our algorithm generates the constraint $l_x + 1 + l_y = l_y + l_x$ where l_x and l_y denote the lengths of x and y , respectively, which is unsatisfiable.

Example 5 [EqConsts] Consider $ax = xb$. If the equation is satisfiable, then both sides of the equation must have the same number of occurrences of each constant. To enforce this, our algorithm generates the constraint $1 + c_a^x = c_a^x$, where c_a^x is the number of a 's in x , which is unsatisfiable.

Example 6 [VarElim] Consider the set of (implicitly conjoined) string equations $\{uv = vu, xa = ax, cy = x\}$. The last equation has the form of a definition and this allows our algorithm to eliminate x by applying the appropriate substitution to the set of equations, giving us $\{uv = vu, cya = acy\}$. Since $cya = acy$ is unsatisfiable, so is the set.

Example 7 [VarSplit] Consider $xxa = cyx$. One side starts with the constant c so the other side must also start with c , which means x cannot be empty and must start with a c . Our

algorithm detects this and adds the equation $x = c\hat{x}$, where \hat{x} is a new string variable. After eliminating x and trimming, we wind up with the equation $\hat{x}c\hat{x}a = yc\hat{x}$, which decomposes into $\hat{x}c = y$ and $\hat{x}a = c\hat{x}$. The EqConsts analysis (Example 5) infers that the second equation is unsatisfiable. Our algorithm also does this for suffixes.

Example 8 [VarSubst] Consider $wuzwuz = cywuz$. The equation is equi-satisfiable with $xxa = cyx$: we substitute a new string variable, x , for the sequence of string variables, wuz , thereby eliminating all occurrences of w, u and z from all string equations. The resulting equation is unsatisfiable by VarSplit (see Example 7).

Example 9 [Rewrite] Consider the set of (implicitly conjoined) string equations $\{zv = ba, xxazv = cyxba\}$. The first equality can be used to rewrite the second equality to $xxazv = cyxzv$ which can be trimmed to $xxa = cyx$, which is unsatisfiable, as per Example 7.

Example 10 [LenSplit] Consider $xbyu = caxzb$. The length of the prefix xb is strictly less than the length of the prefix cax , which allows us to infer that $yu = \hat{y}zb$ for some new string variable $\hat{y} \neq \epsilon$. We can rewrite yu to $\hat{y}zb$ (see Example 9) and after trimming, we wind up with the equation $x\hat{y} = cax$, which is unsatisfiable (see Example 5).

Example 11 [EqWords] Consider $xbca = ycbax$. Let W_{ca}^x and W_{ca}^y be the number of occurrences of a word ca in x and y respectively. If the equation is satisfiable, then both sides must have the same number of ca occurrences. To enforce this, our algorithm generates the constraint $W_{ca}^x + 1 + W_{ca}^y = W_{ca}^y + W_{ca}^x$, which is unsatisfiable. Consider $bwbxw = vbabw$, which shows that counting words requires more care than what the above example suggests, e.g., to count the occurrences of bc , we have to take into account whether c is a prefix of w , whether b is a suffix of x , whether x is empty, and so on. We use 0-1 indicator variables P_c^w, S_b^x and ϵ_x , denoting the above conditions, respectively. Now, with just the ab occurrence analysis, we can use variable splitting on w (w ends in an a) and then on v (v ends in an a) to derive a contradiction.

Example 12 [SAT] None of the string solvers we tried are able to solve the string equation $xcyczvycya = yacwazvbux$. This equation is outside the scope of Kepler22, StrSolve, Hampi and Sloth. Sloth, TRAU and S3P return *unsat*, which is wrong. Norn, Z3Str3 and CVC4 timed out after 1,000 seconds, which shows that existing tools are incomplete, in a practical sense. Our solver finds the assignment $x = aba, y = ab, u = cab$ and $v, w, z = \epsilon$ in a fraction of a second.

III. BLOCKS, SUBSTITUTIONS AND THEORIES

Suppose that a sequence u has an l length subsequence of consecutive occurrences of the constant a . This subsequence can be compactly represented by the pair (a, l) , which we refer to as a *block*: pairs in $\Gamma \times PExp$ where

$$PExp := \mathbb{P} \mid x \mid PExp + PExp \mid PExp - PExp$$

and x is a variable over positive natural numbers, \mathbb{P} . We

require that a *PExp* is positive. A sequence that allows blocks is called an *extended sequence* (*es*); an *extended sequence equation* (*ese*) is similarly defined. The set of extended sequences *es* is $(\Gamma \cup (\Gamma \times PExp) \cup \mathcal{Z})^*$. We define a function *compress* : $es \rightarrow ((\Gamma, PExp) \cup \mathcal{Z})^*$ which given an (extended) sequence, replaces contiguous occurrences of each constant by its block such that no two blocks of the same constant are adjacent to each other, thus returning a unique *maximally compressed sequence*. We define the following useful functions, which given an extended sequence *U*: (1) *Elms* : $es \rightarrow 2^{\Gamma \cup \mathcal{Z} \cup (\Gamma, PExp)}$ returns the set of elements of *U*; (2) *Atoms* : $es \rightarrow 2^{\Gamma \cup \mathcal{Z}}$ returns the set of variables and constants occurring in *U*; (3) *Consts* : $es \rightarrow 2^\Gamma$ returns the set of constants in *U*. (4) *Vars* : $es \rightarrow 2^\mathcal{Z}$ returns the set of variables in *U*. These functions extend naturally to *eses* and to sets of *ess* and *eses*. An extended sequence *U* represents a sequence *u* if *u* is obtained from *U* by replacing every block (α, n) by α repeated *n* times. Note that *n* needs to be a positive integer. Extended sequences *U* and *V* are syntactically equivalent if they represent the same sequence. We use \equiv to denote syntactic equivalence. For example, $(\alpha, 2)\alpha X \equiv \alpha(\alpha, 2)X$, as both of them represent the sequence $\alpha\alpha\alpha X$. Notice that syntactic equivalence is an equivalence relation.

We define a substitution σ to be a partial function of the form $\sigma : es \rightarrow es$. Given substitution σ , let σ_v be σ restricted to \mathcal{Z} and let σ_s be $\sigma \setminus \sigma_v$. Let $\text{dom}(f)$ and $\text{cod}(f)$ be the domain and codomain of function *f*, respectively. Note that $\text{dom}(\sigma_v) \subseteq \mathcal{Z}$, so σ_v is a normal substitution. Substitutions σ_v and σ_s partition σ and have disjoint domains. We say that σ_s is an *extended substitution*, as its domain may contain sequences. We require substitutions to be *well-typed*, i.e., σ_v must map unit variables to sequences of unit length. $U\sigma$ stands for the application of substitution σ to $U \in es$. This notation extends naturally to equations and sets of equations. In order for application to be well-defined, we require that σ is *consistent*, as defined below. We say that σ is *uniquely defined* if for all $x, y \in \text{dom}(\sigma)$, if $x \neq y$ then $\text{Atoms}(x) \cap \text{Atoms}(y) = \emptyset$. To see why we require this, consider the case where $\sigma_v = \{x:ab, y:a\}$ and $\sigma_s = \{yax:aba\}$; note that $(yax)\sigma$ is ambiguous.

Given two uniquely defined substitutions, σ and τ , we say that they are *equivalent*, written $\sigma \equiv \tau$, if for all $U \in es$, we have $U\sigma \equiv U\tau$. We say that σ is *consistent* if it is uniquely defined and $\langle \exists \tau :: \text{dom}(\tau) \subseteq \mathcal{Z} \wedge \sigma \equiv \tau \rangle$, i.e., σ is equivalent to a normal substitution. Consider $\sigma = \{xay:bbb\}$. Even though σ is uniquely defined, it can not be expressed as a normal substitution. From now on, unless we say otherwise, all substitutions are implicitly assumed to be consistent. A substitution σ is said to solve an *ese* $U = V$ if $U\sigma \equiv V\sigma$; σ solves *Q*, a set of *eses*, if σ solves every *ese* in *Q*. A word *ab* is an *es* in which no prefix is a suffix.

Theorem 1. *If σ is a consistent substitution and $x_1, \dots, x_n \in \mathcal{Z}$ are distinct variables such that $n \geq 0$ and $\{x_1, \dots, x_n\} \cap \text{Vars}(\text{dom}(\sigma)) = \emptyset$, then $\sigma \cup \{x_1:V_1, \dots, x_n:V_n\}$ (where*

V_1, \dots, V_n are extended sequences of the right type) is a consistent substitution.

A theory is a pair $T = (\Sigma, \mathbf{I})$, where Σ is a signature and \mathbf{I} is a class of Σ -interpretations, the models of *T*. A set of formulas, Ψ , entails in *T* a Σ -formula ϕ , written $\Psi \models_T \phi$, if every interpretation in \mathbf{I} that satisfies all formulas in Ψ satisfies ϕ as well. The set Ψ is unsatisfiable in *T* if $\Psi \models_T \perp$.

Let *LIA* be a theory with signature $(0, 1, +, -, \leq)$ interpreted over the standard model of integers \mathbb{Z} . A linear constraint is a formula of the form $\sum_{i \in [1..n]} a_i x_i \leq b$, where x_i are variables and a_i and b are integer constants. For a collection of linear constraints *C*, $C \models_{LIA} \perp$ means that *C* is unsatisfiable in *LIA*, whereas $C \not\models_{LIA} \perp$ means that a model exists for *C*. Our algorithm accepts and generates linear constraints on the conjunction of input string equations. It assumes a sound, complete and terminating backend ILP solver for such constraints. Let *ES* be a theory of (extended) sequences over a signature Σ_{ES} with two sorts: extended sequences (*es*) and integers (\mathbb{Z}) along with an infinite set of variables over each sort. Σ_{ES} also includes constants in Γ , *PExp* expressions, blocks, (extended) sequences and functions *len* interpreted as the string length function, *countConst* interpreted as a function counting the number of a specified constant in a sequence and *countWords* interpreted as a function counting the number of specified words in a sequence.

IV. MPMT-BASED STRING SOLVER

Our algorithm, *SeqSolve*, accepts a conjunction of string equations *Q* as well as initial constraints C_{init} and returns either *unsat*, *unknown* or *sat* along with a solution. C_{init} is a set of *initemp*'s defined as

$$\begin{aligned} LExp &:= \mathbb{Z} \mid x \mid \text{len}(u) \mid LExp + LExp \mid LExp - LExp \\ initemp &:= LExp (< \mid \leq \mid > \mid \geq \mid = \mid \neq) LExp \end{aligned}$$

where x is an integer variable (\mathbb{Z}), u is an (extended) sequence and $\text{len} : es \rightarrow \mathbb{N}$ is a function that returns length of *u*. We refer to variables occurring in *PExp* and *LExp* expressions as *numeric variables*. Central to the algorithm is a non-deterministic transition system *TranSeq* with rules that operate on configurations consisting of (extended) sequence equations and sets of *LIA* constraints.

Our decision procedure can be integrated into MPMT solvers in a fine-grained way since MPMT is based on branching, using the branch-and-cut framework. However, in order to make the paper more self contained, we present *TranSeq* and *SeqSolve* with as few dependencies on the MPMT framework as possible.

Our decision procedure can be integrated into SMT solvers using the idea of *recursive solvers*: these are solvers whose decision procedures may depend on the solvers themselves. For example, we can integrate our decision procedure into Z3, even though our decision procedure uses Z3 as a backend solver, by using a separate Z3 process to handle the *LIA*

constraints and one can use this integration as a backend solver for yet another decision procedure, and so on. As far as we know, we are the first to propose the idea of recursive solvers. For SMT solvers like Z3 that provide contexts and a stack with a push-pop interface to manage constraints, integration can be achieved using these features by creating a new context or stack frame, thereby allowing decision procedures to query the SMT solver without polluting its state.

A. Configurations

The algorithm works on configurations that include tuples of the form $\langle \text{unsat} \rangle$, $\langle \text{unknown} \rangle$, $\langle \text{sat}, \sigma, \mathcal{C} \rangle$ and $\langle Q, \sigma, \text{vars}, \mathcal{C} \rangle$ where (1) Q is a set of *eses*, (2) $\sigma : \text{es} \rightarrow \text{es}$ is a (consistent) substitution, (3) vars is a superset of the variables in \mathcal{Z} which occur in Q , (4) \mathcal{C} is a union of constraints $C_{\text{len}}, C_{\text{con } t}, C_{\text{word}}$ and a set of linear constraints corresponding to C_{init} , where (i) C_{len} is a set of linear constraints regarding the lengths of variables in vars . For $x \in \text{vars}$, l_x is an integer variable denoting the length of x and ϵ_x is a 0-1 indicator variable indicating whether x is empty. Linear constraints in C_{len} and C_{init} are over these integer variables and over $PExp$ variables; (ii) $C_{\text{con } t}$ is a set of linear constraints regarding the number of occurrences of constants in variables from vars . For $x \in \text{vars}$, n_a^x is an integer variable denoting the number of occurrences of the constant a in x . Linear constraints in $C_{\text{con } t}$ are over these variables as well as over variables of C_{len} ; (iii) C_{word} is a set of linear constraints regarding the number of words occurring in variables from vars . Let $x \in \text{vars}$ and $s \in \text{consts}^*$. Then W_s^x denotes the number of s occurrences in x ; P_s^x and S_s^x are 0-1 indicator variables indicating whether x begins with s and ends with s , respectively. Linear constraints in C_{word} are over these variables as well as over variables of C_{len} .

The reason why we distinguish between $C_{\text{len}}, C_{\text{con } t}$ and C_{word} is that it makes it easier to consider simplified transition systems that include only a subset of these kinds of constraints. We define sets consts and C_{fuel} where (1) consts is a superset of the constants from Γ occurring in Q and (2) C_{fuel} is a set of linear constraints over the l_x variables, used to guarantee termination. Both consts and C_{fuel} are generated once and never modified by our transition system. The rules in TranSeq depend on auxiliary functions that are used to generate LIA constraints or to simplify equations. All of these functions are described in the full version of this paper.

B. Transition System TranSeq

We describe a non-deterministic transition system TranSeq. TranSeq consists of a set of rules called derivation rules. A derivation rule applies to a configuration K if all of the rule's premises are satisfied by K . Such a rule is *enabled* for K . A derivation tree is a tree where each node is a configuration and the children of any non-leaf node are exactly the configurations obtained by applying one of the derivation rules to the node. A configuration is *terminal* if no rules can be applied to it. We prove that terminal configurations are either of the form $\langle \text{unsat} \rangle$, in which case we call them *unsat* terminal nodes,

$\langle \text{unknown} \rangle$, in which case we call them *unknown* terminal nodes, or of the form $\langle \text{sat}, \sigma, \mathcal{C} \rangle$, in which case we call them *sat* terminal nodes and σ, \mathcal{C} can be used to generate a satisfying assignment to the equations appearing in the root of the tree.

A configuration $K = \langle Q, \sigma, \text{vars}, \mathcal{C} \rangle$ is *sat* (*unsat*) iff $Q \cup \mathcal{C} \cup C_{\text{fuel}}$ is *sat* (*unsat*). K is *C-sat* iff $Q \cup \mathcal{C}$ is *sat*. Notice that an *unknown* terminal node may be *sat* (or *unsat*). This discrepancy is due to the C_{fuel} constraints, which are provable upper bounds on the lengths of minimal solutions, but only if we have no length constraints in the input, so it is possible that K is *C-sat*, but the configuration is *unsat* and we generate an *unknown* terminal node. The derivation rules of TranSeq are given in guarded assignment form and can be categorized into three groups: (1) **Terminal rules:** Rules that yield terminal nodes. (2) **Inference rules:** Rules that generate new inferences. (3) **Branching rules:** Rules that generate multiple subproblems.

A derivation tree is *closed* if all its leaf nodes are terminal nodes. A derivation tree is *unsat-closed* if it is closed and all of its leaf nodes are *unsat*-terminal nodes. A derivation tree is *unknown-closed* if it is closed, has at least one *unknown* terminal node and has no *sat*-terminal nodes. We prove that if a derivation tree is *unsat-closed*, then the conjunction of the equations and constraints appearing in the root of the tree are unsatisfiable. A derivation tree for a set of sequence equations $Q = \{u_1=v_1, u_2=v_2, \dots, u_n=v_n\}$ and some initial length constraints C_{init} (if provided) is a tree whose root, $\text{genRoot}(Q, C_{\text{init}})$, is defined in Algorithm 1, where $\text{Choose}(X)$ is a function that given a non-empty set X , returns an element of X . $C_{\text{len}}, C_{\text{con } t}$ and C_{word} are initialized with linear constraints by functions initLen , initConsts and initWordCount respectively. These functions generate constraints which are satisfiable for any string variable. C_{fuel} comprises of constraints on the size of the minimum solution of each equation in Q which are calculated in function initFuel and are based on results from [23]. The sets consts and vars are supersets of the constants and variables occurring in Q , respectively.

We define the function toLIA , which given an *initexp* returns a linear constraint. Given $\text{len}(x)$, where x is a sequence variable, toLIA returns l_x ; we extend this to *initexp* expressions in the obvious way and use toLIA to also generate fuel constraints. We denote the set of words we are interested in counting as \mathcal{W} , which is global.

C. Rules in TranSeq

We now describe each rule in TranSeq. The conclusion of a rule describes how each component of a configuration is changed, if it does. Rules with two or more conclusions separated by \parallel , are branching rules, where each of the configurations are starting configurations for new branches in their derivation tree. In derivation rules, if Q is relevant, it appears on the top-left corner in the premise and as the last line of a concluding branch. A, t is an abbreviation for $A \cup \{t\}$ and $A \sim t$

Algorithm 1 $\text{genRoot}(Q, C_{\text{init}})$: Given input set of string equations Q , genRoot generates the root node of a derivation tree.

```

1:  $\sigma \leftarrow \{\}$ 
2:  $\text{vars} \leftarrow \{x \mid x \in \mathcal{Z} \wedge x \in v \wedge =v \in Q\}$ 
3:  $\text{consts} \leftarrow \{a \mid a \in v \wedge a \in \Gamma \wedge =v \in Q\}$ 
4: if  $\text{consts} = \emptyset \wedge \text{vars} \cap \mathcal{Y} \neq \emptyset$  then
5:    $\text{consts} \leftarrow \{\text{Choose}(\Gamma)\}$ 
6:  $C_{\text{len}} \leftarrow \bigcup_{v \in \text{vars}} \text{initLen}(v)$ 
7:  $C_{\text{con } t} \leftarrow \bigcup_{v \in \text{vars}} \text{initConsts}(v, \text{consts})$ 
8:  $C_{\text{word}} \leftarrow \bigcup_{v \in \text{vars}, w \in \mathcal{W}} \text{initWordCount}(v, w)$ 
9:  $\mathcal{C} \leftarrow \text{toLIA}(C_{\text{init}}) \cup C_{\text{len}} \cup C_{\text{con } t} \cup C_{\text{word}}$ 
10:  $C_{\text{fuel}} \leftarrow \text{initFuel}(Q)$ 
11: return  $\langle Q, \sigma, \text{vars}, \mathcal{C} \rangle$ 

```

abbreviates $A \setminus \{t\}$. We use $\equiv (\neq)$ for syntactic equivalence (in-equivalence) and $= (\neq)$ for semantic equality (inequality).

Terminal rules When Q is empty, if \mathcal{C} is unsatisfiable, LIAUnsat infers *unsat* otherwise Sat returns a *sat* configuration.

$$\frac{\mathcal{C} \models_{\text{LIA}} \perp}{\langle \text{unsat} \rangle} \text{LIAUnsat} \quad \frac{\{\} \quad \mathcal{C} \not\models_{\text{LIA}} \perp}{\langle \text{sat}, \sigma, \mathcal{C} \rangle} \text{Sat}$$

If the fuel constraints are needed to show unsatisfiability, then the rule FuelUnsat returns *unsat* if no initial linear constraints were provided, otherwise the rule Unknown returns *unknown*. Terminal rules are subject to fairness constraints, as described later.

$$\frac{C_{\text{init}} = \emptyset \quad \mathcal{C} \cup C_{\text{fuel}} \models_{\text{LIA}} \perp}{\langle \text{unsat} \rangle} \text{FuelUnsat}$$

$$\frac{C_{\text{init}} \neq \emptyset \quad \mathcal{C} \not\models_{\text{LIA}} \perp \quad \mathcal{C} \cup C_{\text{fuel}} \models_{\text{LIA}} \perp}{\langle \text{unknown} \rangle} \text{Unknown}$$

If there exists an equation with syntactically different extended sequences on both sides, ConstUnsat infers *unsat*.

$$\frac{\{U=V, \dots\} \quad U \not\equiv V \quad \text{Vars}(UV) = \emptyset}{\langle \text{unsat} \rangle} \text{ConstUnsat}$$

Note that we do not apply substitution σ to U and V when checking for syntactic equivalence, as shown below.

$$\frac{\{U=V, \dots\} \quad U\sigma \not\equiv V\sigma \quad \text{Vars}(UV) = \emptyset}{\langle \text{unsat} \rangle} \text{ConstUnsat}$$

This is because, for any equation $U=V \in Q$, we get the original rule due to $U\sigma = U$ as a result of the invariant $Q\sigma = Q$, which we prove later.

When one side of an extended equation contains a constant or a block, while the other side is empty, ConstEmpty deduces *unsat*. If both sides begin with blocks of unequal constants, DiffConsts deduces *unsat*.

$$\frac{\{U=\epsilon, \dots\} \quad \alpha \in \text{Atoms}(U) \quad \alpha \in \text{consts}}{\langle \text{unsat} \rangle} \text{ConstEmpty}$$

$$\frac{\{(\alpha, l)U=(\beta, m)V, \dots\} \quad \alpha \neq \beta}{\langle \text{unsat} \rangle} \text{DiffConsts}$$

If one side of an equation contains a unit variable while the other side is empty, then YVarEmpty infers $\langle \text{unsat} \rangle$.

$$\frac{\{U=\epsilon, \dots\} \quad e \in U \quad e \in \mathcal{Y}}{\langle \text{unsat} \rangle} \text{YVarEmpty}$$

The rules ConstEmpty and DiffConsts deduce *unsat* based on how terms in an equation start, but there is a symmetry here that allows us to define rules that make the same deduction based on how terms end. For example, the symmetric version of DiffConsts would start with $\{U(\alpha, l) = V(\beta, m), \dots\}$, but would otherwise be identical to DiffConsts . When rules have this kind of symmetry, we denote it by underlining the name of the rule in its definition. These symmetric rules help with efficiency, but are not needed for completeness, so to simplify the rest of the presentation, we proceed as if they do not exist.

Inference rules Trim removes syntactically equal prefixes and suffixes from both sides of an equation; note that one of a, b can be ϵ . EqElim removes *eses* whose both sides are syntactically equivalent. Observe that Trim can be used to reduce an equation $U=V$ which is syntactically equivalent on both sides, to get $\epsilon=\epsilon$, in which case we get syntactic equivalence of both sides trivially.

$$\frac{\{aUb=cVd, \dots\} \quad a \equiv c \quad \{U=U, \dots\}}{\{U=V, \dots\}} \text{Trim} \quad \frac{\{U=U, \dots\}}{\{\dots\}} \text{EqElim}$$

Decompose splits an *ese* $U=V$ into multiple equations using length constraints. A simple example is given in Example 3.

$$\frac{\{U=V, \dots\} \quad |\text{splitEq}(U, V, \mathcal{C})| > 1}{\text{splitEq}(U, V, \mathcal{C}) \cup \{\dots\}} \text{Decompose}$$

Compress converts an equation $u=v \in Q$ into a maximally compressed sequence. Observe that the premise requires that there is at least one constant element in $u=v$. Note that blocks such as $(a, 1)$ are not constants, as they are not elements of Γ .

$$\frac{\{u=v, \dots\} \quad \text{Elms}(uv) \cap \Gamma \neq \emptyset}{\{\text{compress}(u)=\text{compress}(v), \dots\}} \text{Compress}$$

VarSubst formalizes the idea from Example 8. Given W , a non-empty subsequence in Q satisfying the conditions below, the rule replaces W with a new variable z . We show later that for every node in a derivation tree generated by our algorithm, $Q\sigma = Q$ holds; hence, the first condition for consistency of substitutions is satisfied. The second consistency condition is satisfied due to the premise that requires atoms of W and $Q\{W:z\}$ to be disjoint. Hence, the substitution in the new configuration is consistent. The LIANewVar procedure generates numeric constraints for new variables. After this rule, it is called implicitly whenever a new variable is introduced.

$$\frac{\{U=V, \dots\} \quad \langle \exists S, T :: SWT=U \wedge |W| > 1 \rangle \quad \text{Atoms}(W) \subseteq \text{vars} \quad z \in \mathcal{X} \quad z \notin \text{vars} \quad \text{Atoms}(W) \cap \text{Atoms}(\{U=V, \dots\}\{W:z\}) = \emptyset}{\text{LIANewVar}(z) \quad \sigma \leftarrow \sigma, W:z \quad \{U=V, \dots\}\{W:z\}} \text{VarSubst}$$

Rewrite replaces a subsequence S of U by T , given that $S=T$ is an equation in Q . Rewrite can choose which occurrences to replace. Infinite derivation trees are ruled out with a fairness requirement that only allows us to use the Rewrite rule a finite number of times.

$$\frac{\{U=V, S=T, \dots\} \quad S \in U}{\{U\{S:T\}=V, S=T, \dots\}} \text{ Rewrite}$$

EqLength, EqConsts and EqWords generate length, constant count and word count constraints implied by an equation. Function `equateWordCount` returns a linear constraint equating the number of occurrences of a word w in U and V .

$$\frac{\{U=V, \dots\} \quad \text{equateLen}(U, V) \not\subseteq \mathcal{C}}{C_{len} \leftarrow C_{len} \cup \text{equateLen}(U, V)} \text{ EqLength}$$

$$\frac{\{U=V, \dots\} \quad \text{equateConsts}(U, V) \not\subseteq \mathcal{C}}{C_{const} \leftarrow C_{const} \cup \text{equateConsts}(U, V, const)} \text{ EqConsts}$$

$$\frac{\{U=V, \dots\} \quad w \in const \geq 2 \quad \text{equateWordCount}(U, V, w) \not\subseteq \mathcal{C}}{C_{word} \leftarrow C_{word} \cup \text{equateWordCount}(U, V, w)} \text{ EqWords}$$

VarElim allows us to eliminate variables.

$$\frac{\{x=V, \dots\} \quad x \notin V \quad x \in \mathcal{X}}{\sigma \leftarrow \sigma, x:V \quad \{\dots\}\{x:V\}} \text{ VarElim}$$

Given an equation where one side starts with c occurrences of variable x and the other starts with m occurrences of constant β , the rule `VarSplit` infers shape information about x involving fresh variable y . x can not be empty, and the prefix of x^c must be syntactically equivalent to (β, m) . Hence, `VarSplit` infers that x is $(\beta, k)y$, where $c * k \geq m$. Note that c is a constant, hence expressions such as $c * k$ do not take us out of the LIA fragment. Also note that if $k < m$, y will have to start with β as well, which we do not want. Hence we add an implication that if $k < m$ then y is empty. We extend the set of equations with $x=(\beta, k)y$. Anytime we extend a the set of equations with an equation of the form $x=\dots$, we call `VarElim` to eliminate the variable x .

$$\frac{\{x^c(\alpha, l)U=(\beta, m)V, \dots\} \quad \alpha \neq \beta, c > 0 \quad x, y \in \mathcal{X} \quad y \notin vars}{C_{len} \leftarrow C_{len}, k > 0, (c-1) * k < m \leq c * k, k < m \Rightarrow \epsilon_y = 1} \text{ VarSplit}$$

$$C_{word} \leftarrow C_{word}, k < m \Rightarrow S_{\beta}^x = 1$$

$$\{x=(\beta, k)y, x^c(\alpha, l)U=(\beta, m)V, \dots\}$$

Length constraints alone may not always be enough to split an equation. `LenSplit` introduces a new variable on one side of an equation such that the resulting equation is clearly split into smaller and possibly more tractable equations. Example 10 illustrates a simple example.

$$\frac{\{UW=SzV, \dots\} \quad C \models_{LIA} len(U) < len(Sz) \quad y, z \in \mathcal{X} \quad y \notin vars}{C_{len} \leftarrow C_{len}, \epsilon_y = 0} \text{ LenSplit}$$

$$\{Uy=Sz, W=yV, \dots\}$$

Inferences made by the backend LIA solver can be used to infer sequence variables. `LIAEmpty` concludes that a variable x is empty if $\epsilon_x = 1$ is derived by the solver. Similarly, x starts (ends) with α iff the solver derives $P_{\alpha}^x = 1$ ($S_{\alpha}^x = 1$).

$$\frac{C \models_{LIA} \epsilon_x = 1 \quad x \in vars}{\{x=\epsilon, \dots\}} \text{ LIAEmpty} \quad \frac{C \models_{LIA} P_{\alpha}^x = 1 \quad y \in \mathcal{X} \quad x \in vars \quad y \notin vars}{\{x=\alpha y, \dots\}} \text{ LIABegin}$$

$$\frac{C \models_{LIA} S_{\alpha}^x = 1 \quad y \in \mathcal{X} \quad x \in vars \quad y \notin vars}{\{x=y\alpha, \dots\}} \text{ LIAEnd}$$

Given an equation where one side is empty, `XVarEmpty` infers that a variable $x \in \mathcal{X}$ in the other side must also be empty. If the two sides of an *ese* start with unit variables x and y , then `DiffYVars` infers that both the variables must be equal.

$$\frac{\{U=\epsilon, \dots\} \quad x \in U \quad x \in \mathcal{X}}{\{x=\epsilon, U=\epsilon, \dots\}} \text{ XVarEmpty} \quad \frac{\{xU=yV, \dots\} \quad x \neq y \quad x, y \in \mathcal{Y}}{\{x=y, U=V, \dots\}} \text{ DiffYVars}$$

Branching rules Given an equation where one side starts with a block of α , while the other side starts with a unit variable e , `UnitConst` infers that either the length of the α block is greater than one, or equal to one. Observe that some constraints in this rule are emphasized with a wavy underline. If such constraints are implied by \mathcal{C} , we can directly jump to their corresponding branch. Practically, it helps to not branch, if one of the underlined constraints can be derived in the premise.

$$\frac{\{eU=(\alpha, l)V, \dots\} \quad e \in \mathcal{Y}}{C_{len} \leftarrow C_{len}, \underline{l=1} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{l \geq 1}} \text{ UnitConst}$$

$$\{e=\alpha, U=V, \dots\} \quad \{e=\alpha, U=(\alpha, l-1)V, \dots\}$$

Given an equation where one side starts with a unit variable e while the other side starts with sequence variable y , `UnitVar` infers that either y is empty, or e is a prefix of y .

$$\frac{\{eU=yV, \dots\} \quad e \in \mathcal{Y} \quad y, z \in \mathcal{X} \quad z \notin vars}{C_{len} \leftarrow C_{len}, \underline{\epsilon_y = 1} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{\epsilon_y = 0}} \text{ UnitVar}$$

$$\{y=\epsilon, eU=V, \dots\} \quad \{y=ez, U=zV, \dots\}$$

If both sides of an equation start with blocks of the same constant α , `SimConsts` infers that either both blocks have the same length or one of them has length more than the other. So this rule should have three branches, one equating l and m , while the other two deducing a strict inequality between them. However, there are two branches, one equating l and m , while the other deducing $\hat{m} > \hat{l}$. This is because, for the sake of conciseness we introduce “hatted” variables $\hat{U}, \hat{V}, \hat{l}, \hat{m}$ and $\hat{\beta}$. A branch with hatted variables signifies the presence of another branch where the hatted variables are replaced by their substitutions defined as:

$$\{\hat{x}:y, \hat{y}:x, \hat{X}:Y, \hat{Y}:X, \hat{U}:V, \hat{V}:U, \hat{l}:m, \hat{m}:l, \hat{\alpha}:\beta, \hat{\beta}:\alpha\}$$

Notice that we also have underlined constraints in the conclusion. So, the rule `SimConsts` represents six rules, three after expanding hatted variables where none of the underlined constraints is implied by \mathcal{C} , and the rest considering presence of each of the underlined constraints in the premise of its corresponding rule.

$$\frac{\{(\alpha, l)U=(\alpha, m)V, \dots\}}{C_{len} \leftarrow C_{len}, \underline{m=l} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{\hat{m} > \hat{l}} \quad \text{SimConsts}} \\ \{U=V, \dots\} \quad \{\hat{U}=(\alpha, \hat{m} - \hat{l})\hat{V}, \dots\}$$

Similar to `SimConsts`, `DiffXVars` also uses both hatted variables and underlined constraints which gives rise to a total of ten rules. If both sides of an equation start with syntactically different variables $x, y \in \mathcal{X}$, and none of the underlined constraints is implied by \mathcal{C} , then `DiffXVars` infers that either one of them is empty or they are semantically equal or one of them is a prefix of the other.

$$\frac{\{xU=yV, \dots\} \quad x \neq y \quad z \notin vars \quad x, y \in \mathcal{X} \quad z \in \mathcal{X}}{C_{len} \leftarrow C_{len}, \underline{l_{\hat{x}} > l_{\hat{y}}}, \parallel C_{len} \leftarrow C_{len}, \underline{l_x = l_y}, \quad \text{DiffXVars}} \\ \underline{\epsilon_{\hat{x}} = \epsilon_{\hat{y}} = \epsilon_z = 0, \hat{x} = l_{\hat{y}} + l_z} \quad \underline{\epsilon_x = \epsilon_y = 0} \\ \{\hat{x} = \hat{y}z, z\hat{U} = \hat{V}, \dots\} \quad \{x = y, U = V, \dots\} \\ \parallel C_{len} \leftarrow C_{len}, \underline{\epsilon_{\hat{x}} = 1} \\ \{\hat{x} = \epsilon, \hat{U} = \hat{y}\hat{V}, \dots\}$$

Finally, `VarConst` fires when one side of an equation starts with a constant block (α, l) while the other side starts with a variable x . Again, `VarConst` represents eight rules due to the presence of underlined constraints in its branching conclusions. Assuming none of these constraints is implied by \mathcal{C} , the first branch sets x empty; second branch sets length of x less than l ; third branch equated x to (α, l) , while the last branch sets x as a block of α whose length is greater than l , possibly followed by another variable y that does not start with α .

$$\frac{\{xU=(\alpha, l)V, \dots\} \quad x, y \in \mathcal{X} \quad y \notin vars}{C_{len} \leftarrow C_{len}, \underline{\epsilon_x = 1} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{0 < l_x < l} \quad \text{VarConst}} \\ \{x = \epsilon, U = (\alpha, l)V, \dots\} \quad \{x = (\alpha, l_x), U = (\alpha, l - l_x)V, \dots\} \\ C_{len} \leftarrow C_{len}, \underline{0 < l_x = l} \parallel C_{len} \leftarrow C_{len}, \underline{0 < l < l_x} \\ \{x = (\alpha, l), U = V, \dots\} \quad \{x = (\alpha, l_x)y, xU = (\alpha, l)V, \dots\}$$

D. SeqSolve definition

We define `SeqSolve` in Algorithm 2. It takes a set of sequence equations W and an optional set of length constraints C_{init} as input and either returns a *sat* with a solution, *unknown* or *unsat*.

V. CORRECTNESS OF SEQSOLVE

Full proofs of correctness of `SeqSolve` appear in the full version of this paper. In the interest of brevity, we outline the structure of proofs in this section. First, we define correctness.

Algorithm 2 `SeqSolve` takes a set of (extended) sequence equations W and optionally a set of linear constraints C_{init} as input and either returns a *sat* with a solution, *unknown* or *unsat*.

```

1:  $T \leftarrow \text{genRoot}(W, C_{init})$ 
2: while  $\exists$  a non-terminal leaf node  $n \in T$  do
3:   apply an enabled TranSeq rule to  $n$ 
4:   if sat terminal node  $\langle \text{sat}, \sigma, \mathcal{C} \rangle$  generated then
5:     generate a satisfying assignment  $\psi$  from  $\sigma, \mathcal{C}$ 
6:     return sat,  $\psi$ 
7:   if  $\exists$  leaf node  $\langle \text{unknown} \rangle \in T$  then
8:     return unknown
9:   else
10:    return unsat

```

Definition 1. A string equation solver is an algorithm that takes as input a set of string equations and a set of linear constraints. Its output is either “*Unsat*,” “*Unknown*,” or “*Sat*” and an assignment.

Definition 2. A string equation solver is sound if it never lies, by which we mean: (1) when it returns “*Sat*,” the conjunction of the string equations and the linear constraints is satisfiable and the assignment returned is a satisfying assignment and (2) when it returns “*Unsat*,” the conjunction of the string equations and the linear constraints is unsatisfiable.

Definition 3. A string equation solver is partially correct if it is sound and terminating.

Definition 4. A string equation solver is fully correct if it is sound, terminating and never returns “*Unknown*.”

Note that a sound solver can be turned into a partially correct solver by adding a timeout, which results in the solver returning “*Unknown*.” We prove that our solver is fully correct for the theory of string equations by showing that when the input consists of only a conjunction of string equations Q , our transition system generates a derivation tree that is unsat-closed iff the input is unsatisfiable; otherwise it generates a derivation tree containing a *sat* terminal node, from which we can extract a satisfying assignment for the input. When the input also includes linear constraints, our solver is partially correct as it may also generate an *unknown*-closed derivation tree. We show that `SeqSolve` is sound using the following theorems.

Theorem 5. Given inputs Q, C_{init} such that `SeqSolve` generates a tree T with a *sat* terminal node $\langle \text{sat}, \sigma, \mathcal{C} \rangle$, then σ, \mathcal{C} can be used to generate a solution for Q, C_{init} .

A configuration is *var-compliant* iff it is of the form $\langle Q, \sigma, vars, \dots \rangle$ where $\text{Vars}(\sigma) \subseteq vars$ (by $\text{Vars}(\sigma)$ we mean $\text{Vars}(\text{dom}(\sigma)) \cup \text{Vars}(\text{cod}(\sigma))$). A configuration is *numvar-compliant* iff (1) it is of the form $\langle Q, \sigma, vars, \mathcal{C} \rangle$ and all numeric variables appearing in it are also in \mathcal{C} and (2) for a variable $x \in vars$, $\text{initLen}(x) \cup \text{initConsts}(x, \text{consts}) \cup \text{initWordCount}(x, \text{consts}) \subseteq \mathcal{C}$. A configuration is *good* iff it is either terminal or it is disjoint, *var-compliant* and *numvar-compliant*. A derivation tree is *good* if all of its nodes are

good configurations. It turns out that all SeqSolve-generated derivation trees are good.

Lemma 7. *Given input Q, C_{init} where Q is a set of (extended) sequence equations and C_{init} is a set of linear constraints, $genRoot$ returns a good, non-terminal configuration.*

Lemma 12. *TranSeq rules preserve goodness, i.e., when applied to a good configuration, they produce good configurations.*

SeqSolve is subject to the following fairness conditions: (1) LIAUnsat, FuelUnsat and Unknown are *weakly-fair* rules. First note that once any of these rules is enabled, it stays enabled. We require that no branch of a derivation tree contains a suffix in which a weakly-fair rule is infinitely enabled, yet never applied. (2) Rewrite can only be applied a finite number of times along any branch.

A *fair* derivation tree is one which respects the above fairness conditions. SeqSolve generates fair and good derivation trees. We use good derivation trees to show that TranSeq is sound.

Theorem 6. *Every TranSeq rule is sound when applied to a good configuration.*

The termination of SeqSolve (and TranSeq) depends on a bound on the minimum lengths of solutions of string equations as described in [23] and on fair derivation trees.

Theorem 9. *SeqSolve is terminating.*

Theorem 10. *SeqSolve is a partially correct string equation solver.*

Theorem 11. *SeqSolve is a fully correct string equation solver when the input does not include any linear constraints.*

VI. IMPLEMENTATION OF SEQ SOLVE

Our implementation of SeqSolve along with all the benchmarks used is publicly available [24]. SeqSolve is implemented in ACL2s [25] which allows us to (1) define datatypes like blocks, sequences and valid Z3 expressions (used to query Z3) (2) define TranSeq rules, which requires proving termination and input/output contracts (input/output types) (3) prove basic theorems relating datatypes (subtypes, etc) and properties needed for above proofs and (4) make essential use of the Z3 interface ACL2s provides to solve ILP constraints. SeqSolve provides various settings that can be used to control how aggressively it generates linear constraints; however, all of the results reported in this paper are with the default settings. We implemented SeqSolve as a standalone decision procedure as opposed to making it a part of an MPMT solver. This makes it easier to compare our tool with other string solvers in an apples-to-apples way, avoiding the complications that would arise from the use of different underlying solvers and frameworks.

We apply a few TranSeq rules until we reach a fixpoint before generating the derivation tree in order to simplify the input problem. These preprocessing steps include Decompose,

VarElim, VarSubst and Compress. After reaching a fixpoint, we use LIAUnsat to check if the set of initial constraints and the linear constraints we generated above are *unsat*.

In our implementation of the rule EqWords, we only use words with the property that no non-empty prefix of w is a suffix of w . Since our solver makes many low-level calls to Z3, it does this in an incremental way. In addition, care is taken to avoid unnecessary calls to Z3, e.g., LIAUnsat is not checked after running Trim, EqElim, Decompose, Compress, VarSubst, Rewrite and VarElim, because in all of these rules, we do not update \mathcal{C} . We do not apply any branching rules, unless we have no other options. Our implementation supports string operations like `charAt`, `contains`, `indexOf`, `substr`, `prefixOf` and `suffixOf`. Each of these operations can be converted to a problem in the theory of extended sequences e.g., given `charAt` constraint $e = (str.at\ s\ n)$, we convert it into the conjunction of the string equation $s = xey$ and $len(x) = n$, where $e \in \mathcal{V}$ and $x, y \in \mathcal{X}$. Given the constraint $(str.contains\ s\ t)$, we convert it into the string equation $s = xty$ where $x, y \in \mathcal{X}$.

VII. EVALUATION

We compared our solver against Z3Str2 and Z3Str3 (Z3 version 4.8.8), Norn 1.0, Z3-Trau, Sloth 1.0 and CVC4 1.7. These are the only string solvers we know of that solve string equations with length constraints and ran without crashing. In [26], the tools CVC4, Z3Str2 and S3 are evaluated in which S3 is found to be 5 times slower than Z3Str2 and crashed on about 4.5% of problems in the Kaluza [27] benchmarks. We ran all of the selected tools on Kaluza and Stringfuzz-generated [28] benchmarks, as well as on benchmarks consisting of problem instances pertinent to type inference in Remora [9], [10], a dependently typed array processing language. The type of an array term in Remora encodes the shape of the array as a list of dimensions (natural numbers). Our work was motivated by the problem of inferencing these shapes which reduces to solving string equations. For example, suppose that X has dimensions $[a\ 3]b$ and Y has dimensions $b[3]z$, where a is a single dimension, while b and z are lists of dimensions, and juxtaposition indicates concatenation. If X and Y are used in a context where they must have the same dimensions, then for the program to be well-typed, we require that the string equation $a3b = b3z$ is satisfiable. One solution is $b = []$, $z = [3]$ and $a = 3$, in which case X and Y are 2-dimensional matrices with shape $[3\ 3]$.

We used all of the problems in the above mentioned benchmarks that were in the extended sequence theory, thus, excluding problems in Kaluza that used other constructs. This allows us to evaluate only our contribution, the string solver, not the underlying solvers. In total, we have 1,178 problems, of which 903 are *sat* problems and 275 are *unsat* problems. We cross-verified the tools and for all benchmark problems, all tools that gave definitive answers agreed on the classification of the problem. All experiments were performed on the same machine, which was running macOS Catalina 10.14.6 with a 2.7GHz Intel Core i5 CPU and 8 GB of memory. The timeout

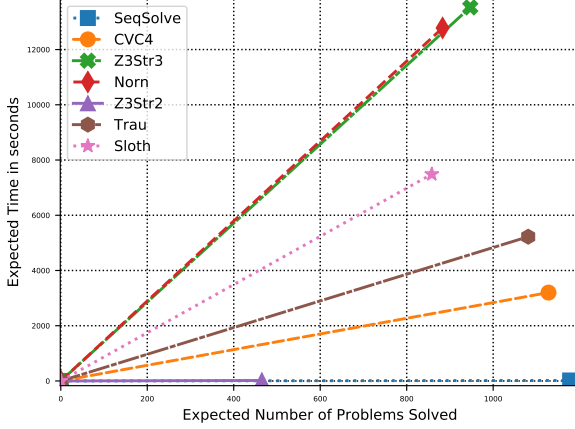


Fig. 1. Performance of SeqSolve, CVC4, Z3-Str3, Norn, Sloth, Trau and Z3-Str2 on solved benchmarks across all three benchmark sets.

for each problem was set to 60 seconds. Figure 1 shows the results of the performance evaluation, using what we call a *ray plot*. Ray plots are designed to visually depict the results of the evaluation in as simple a way as possible. On the x -axis we have the expected number of problems solved and on the y -axis we have the expected time in seconds. Suppose you want to determine how long it will take to solve n benchmark problems, say 800; just look at the line $x = 800$ and you will see that SeqSolve will take about 100 seconds, CVC4 will take over 2,000 seconds, Z3Str3 will take just under 12,000 seconds, Norn will take about 5,500 seconds and Z3Str2 can only solve about 500 problems, so it will never solve 800 problems. Symmetrically, if you want to determine how many problems you can expect to solve in t seconds, just look at the line $y = t$. This is a simpler plot than a cactus plot, which shows similar information, but with problems ordered, on a per-tool basis, from easiest to hardest. These orderings can vary significantly from tool to tool and there is no way for a user of the tool to determine how easy or difficult a problem will be, so it is not clear what benefit there is to this extra complexity. It is easy to generate ray plots; just run all the benchmark problems and draw a ray from the origin to the (p, t) coordinate, where p is the number of problems solved and t is the time taken. This is equivalent to shuffling the problems many times and taking the average of the running times for the shufflings.

In Table I, we show a table version of the experimental evaluation. Tuples under “Solved” give the number of problems solved for the Stringfuzz-generated, Kaluza and handcrafted benchmarks, respectively. In addition to the time in seconds, we also show the number of problems for which solvers returned *unknown*, timed out or returned incorrect result (X). We ran the tools without giving them a timeout and our scripts killed jobs that were taking too long, but some

TABLE I
PERFORMANCE OF SOLVERS ON ALL BENCHMARKS

Solver	Solved	Time (s)	Unknown	Timeout	X
SeqSolve	1,178: 780/344/54	176	0	0	0
CVC4	1,128: 736/344/48	3,200	0	50	0
Z3Str3	947: 552/344/51	13,527	6	225	0
Norn	883: 492/344/47	12,783	120	175	0
Z3Str2	465: 121/332/12	18	713	0	0
Trau	1,081: 692/344/45	5,223	18	78	1
Sloth	858: 462/344/52	7,486	0	319	64

tools returned *unknown* before timeouts occurred. Notice that SeqSolve beats all the other string solvers in terms of the standard ordering, which is based on first the number incorrect results, then on the number of problems solved and finally on the time taken.

Acknowledgements: We thank Andrew Walter for integrating Z3 with ACL2s, which was indispensable.

VIII. CONCLUSION AND FUTURE WORK


We introduced a new non-deterministic, branching transition system, TranSeq, for deciding the satisfiability of conjunctions of string equations and length constraints. TranSeq extends the MPMT framework for combining decision procedures and we prove that it is both sound and complete. We implemented a prototype, SeqSolve, which is based on TranSeq and resolves non-deterministic choices in a way designed to infer as much as possible with as few computational resources as possible. We evaluated SeqSolve by comparing it with existing tools on a suite of benchmark problems and found that SeqSolve solved more problems and was faster than existing solvers. In our ongoing work, we plan to extend the scope of TranSeq so that it supports richer classes of constraints. We also plan to reason about the implementation, as it is mostly written in ACL2s, which is built on top of the ACL2 theorem prover.


REFERENCES


- [1] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, “HAMPI: A solver for string constraints,” in *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.
- [2] L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar, “String constraints with concatenation and transducers solved efficiently,” in *Proceedings of the ACM on Programming Languages (PACMPL)*, 2018.
- [3] F. Yu, M. Alkhalaf, and T. Bultan, “Stranger: An automata-based string analysis tool for php,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2010.
- [4] P. Godefroid, N. Klarlund, and K. Sen, “Dart: Directed automated random testing,” in *Programming Language Design and Implementation*, 2005.
- [5] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing,” in *Programming Language Design and Implementation*, 2008.
- [6] A. S. Christensen, A. Möller, and M. I. Schwartzbach, “Precise analysis of string expressions,” in *Proceedings of the 10th International Conference on Static Analysis*, 2003.

- [7] R. Majumdar and R. Xu, “Directed test generation using symbolic grammars,” in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, 2007.
- [8] N. Bjørner, N. Tillmann, and A. Voronkov, “Path feasibility analysis for string-manipulating programs,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.
- [9] J. Slepak, O. Shivers, and P. Manolios, “An array-oriented language with static rank polymorphism,” in *European Symposium on Programming (ESOP)*, 2014.
- [10] J. Slepak, P. Manolios, and O. Shivers, “Rank polymorphism viewed as a constraint problem,” in *International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI*, 2018.
- [11] G. S. Makanin, “The problem of solvability of equations in a free semigroup,” *Mathematics of the USSR-Sbornik*, 1977.
- [12] W. Plandowski, “Satisfiability of word equations with constants is in PSPACE,” in *Foundations of Computer Science (FOCS)*, 1999, pp. 495–500.
- [13] M. Berzish, V. Ganesh, and Y. Zheng, “Z3str3: A string solver with theory-aware heuristics,” in *Formal Methods in Computer Aided Design, FMCAD*, 2017.
- [14] T. Liang, A. Reynolds, C. Tinelli, C. W. Barrett, and M. Deters, “A DPLL(T) theory solver for a theory of strings and regular expressions,” in *Computer Aided Verification (CAV)*, 2014.
- [15] A. Reynolds, M. Woo, C. Barrett, D. Brumley, T. Liang, and C. Tinelli, “Scaling up DPLL(T) string solvers using context-dependent simplification,” in *International Conference on Computer Aided Verification (CAV)*, 2017.
- [16] M. Trinh, D. Chu, and J. Jaffar, “Progressive reasoning over recursively-defined strings,” in *International Conference on Computer Aided Verification (CAV)*, 2016.
- [17] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, “String constraints for verification,” in *International Conference on Computer Aided Verification (CAV)*, 2014.
- [18] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, “TRAU: SMT solver for string constraints,” in *Formal Methods in Computer Aided Design (FMCAD)*, 2018.
- [19] P. Hooimeijer and W. Weimer, “StrSolve: Solving string constraints lazily,” in *Automated Software Engineering (ASE)*, 2012.
- [20] S. Eguchi, N. K. B., and T. Tsukada, “Automated synthesis of functional programs with auxiliary functions,” in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2018.
- [21] P. Manolios and V. Papavasileiou, “ILP modulo theories,” in *International Conference on Computer Aided Verification (CAV)*, 2013, pp. 662–677.
- [22] P. Manolios, J. Pais, and V. Papavasileiou, “The Inez mathematical programming modulo theories framework,” in *International Conference on Computer Aided Verification (CAV)*, 2015.
- [23] W. Plandowski, “Satisfiability of word equations with constants is in NEXPTIME,” in *Symposium on Theory of Computing (STOC)*, 1999.
- [24] A. Kumar, “SeqSolve string solver with benchmarks,” <https://github.com/ankitku/SeqSolve>.
- [25] H. Chamathi, P. C. Dillinger, P. Manolios, and D. Vroon, “The ACL2 sedan theorem proving system,” in *TACAS*, 2011.
- [26] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang, “Z3str2: an efficient solver for strings, regular expressions, and length constraints,” in *Formal Methods in System Design*, 2017.
- [27] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “Kaluza benchmark,” <http://webblaze.cs.berkeley.edu/2010/kaluza/>.
- [28] D. Blotsky, “StringFuzz-generated benchmark,” <http://stringfuzz.dmitryblotsky.com/suites/generated/>.
- [29] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *Symposium on Security and Privacy, S&P*, 2010.
- [30] A. Jez, “Recompression: A simple and powerful technique for word equations,” *Journal of the ACM (JACM)*, 2016.
- [31] W. Plandowski and W. Rytter, “Application of Lempel-Ziv Encodings to the Solution of Words Equations,” in *International Colloquium on Automata, Languages and Programming (ICALP)*, 1998.
- [32] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, “Stringfuzz: A fuzzer for string solvers,” in *International Conference on Computer Aided Verification (CAV)*, 2018.
- [33] L. M. de Moura, B. Dutertre, and N. Shankar, “A tutorial on satisfiability modulo theories,” in *International Conference on Computer Aided Verification (CAV)*, 2007.
- [34] H. Abdulrab, “Solving word equations,” in *Informatique Théorique et Applications (ITA)*, 1990.
- [35] S. Subramanian, M. Berzish, O. Tripp, and V. Ganesh, “A solver for a theory of strings and bit-vectors,” in *International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017.
- [36] N. Bjørner, “All strings attached: String and sequence constraints in Z3,” in *Rewriting Logic and Its Applications*, 2016.
- [37] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore, “ACL2s: The ACL2 sedan,” in *International Conference on Software Engineering (ICSE)*, 2007.
- [38] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *International Conference on Computer Aided Verification (CAV)*, 2011.
- [39] M. T. Trinh, D. H. Chu, and J. Jaffar, “S3: A symbolic string solver for vulnerability detection in web applications,” in *Computer and Communications Security (CCS)*, 2014.
- [40] A. Lin and P. Barceló, “String solving with word equations and transducers: Towards a logic for analysing mutation XSS,” in *Principles of Programming Languages (POPL)*, 2016.
- [41] C. Gutiérrez, “Solving equations in strings: On makanin’s algorithm,” in *Theoretical Informatics, Third Latin American Symposium (LATIN)*, 1998.
- [42] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang, “Effective search-space pruning for solvers of string equations, regular expressions and length constraints,” in *International Conference on Computer Aided Verification (CAV)*, 2015.
- [43] J. D. Day, F. Manea, and D. Nowotka, “The hardness of solving simple word equations,” in *Leibniz International Proceedings in Informatics (LIPIcs)*, 2017.
- [44] P. Aziz Abdulla, M. Faouzi Atig, Y.-F. Chen, B. Phi Diep, L. Holík, A. Rezine, and P. Rümmer, “Flatten and conquer: A framework for efficient analysis of string constraints,” in *Programming Language Design and Implementation (PLDI)*, 2017.
- [45] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.
- [46] Y. Minamide, “Static approximation of dynamically generated web pages,” in *Proceedings of the 14th International Conference on World Wide Web*, 2005.
- [47] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *IEEE Symposium on Security and Privacy*, 2010.

Lookahead in Partitioning SMT

Antti E. J. Hyvärinen 
 USI, Switzerland
antti.hyvaerinen@usi.ch

Matteo Marescotti 
 Facebook, UK
mmatteo@fb.com

Natasha Sharygina 
 USI, Switzerland
natasha.sharygina@usi.ch

Abstract—Lookahead in propositional satisfiability has proven efficient as a heuristic in pre- and in-processing, for partitioning instances for parallel solving, and as the main driver of a stand-alone solver. While applying similar techniques in satisfiability modulo theories is potentially equally useful, adapting lookahead to learning theory clauses and to estimating search space sizes in the presence of first-order structures is not straightforward. This paper addresses both of these observations. We give a hybrid algorithm that integrates lookahead into the state-based representation of an SMT solver and show that in the vast majority of cases it is possible to compute full lookahead up to depth four on inexpensive theories. We also show the role of first-order structures in SMT search space: while in most of our benchmarks the partitions are easier to solve than the original instance, we identify cases where lookahead results in sequences of increasingly difficult instances for a computationally expensive theory.

I. INTRODUCTION

Large scale parallel SMT solving that would result in linear speed-up reliably over any instance in a cloud environment is a lucrative prize that has been intensively studied over the recent years [26], [14], [13], [17]. A central sub-goal in this project is in understanding how to apply successfully the *cube-and-conquer* [24] approach in SMT solving. The lookahead heuristic in propositional logic [27], in addition to being efficient in solving certain types of structured problems [8], has recently proven to be a powerful tool in constructing partitions for divide-and-conquer-based parallel SAT solvers [10], [9]. The idea is to base the search-space traversal on the explicit principle of branching on literals that reduce maximally the remaining search space. In addition to SAT solvers, the heuristic has been implemented in SMT solvers such as Z3 [20], where it serves for in- and pre-processing, and by us in OpenSMT [11], [12] as an alternative implementation for the main SAT solver.

This paper studies how the literals chosen by lookahead algorithm for SMT affect the difficulty of the instance from the perspective of a standard CDCL-based SMT solver. This question is central to divide-and-conquer-style parallel SMT solving, where the lookahead heuristic is used to build a binary *lookahead tree* of depth d , with nodes labeled by the literals chosen with the lookahead heuristic, and root labeled with the true literal \top . Conjoining the literals in each rooted path to the leaves with the original instance produces 2^{d-1} partitioned instances that do not share models. The resulting instances can be solved in parallel, and the original instance is satisfiable if and only if one of the partitioned instances is satisfiable.

Our main contributions are rigorously defining what we mean by lookahead heuristic for an SMT solver, and an experimental study on how the use of this heuristic affects the difficulty of the partitions. In defining the heuristic, we show that lookahead can be integrated tightly into a CDCL(T)-style algorithm that fully leverages learned clauses, including determining unsatisfiability while constructing partitions. We summarize our experimental results as follows. First, in many cases the heuristic runs in seconds when producing a non-trivial number of partitions (say, 16). This is already a non-trivial observation given that the full lookahead heuristic in SAT is known to be in most cases prohibitively expensive. Second, usually the approach results in partitions that are easier to solve than the original. While this result seems rather implicit and obvious, it is made interesting by the next observation: There are instances where the above described lookahead-based parallel algorithm's run time *increases* compared to the original instance even when no overhead from partitioning or communication is considered, and the number of partitions is in the thousands. We show some details on the latter cases that help to understand the underlying phenomena, and identify a possible reason arising from the way the theory solving algorithm for linear real arithmetics is implemented in most SMT solvers. These cases serve to illustrate the complexity of the ultimate goal of an efficient and general parallel solver.

Combining a lookahead algorithm with a CDCL-based SMT solver in a meaningful way is not straightforward. First, the lookahead heuristics assumes that the clauses of an instance are known at computing time. In contrast, an SMT solver produces a new clause whenever a propositional model is inconsistent in the theory. A potentially very large number of clauses remain invisible for the heuristic. Second, the explanation clauses guide the search through non-chronological backtracking. This means that the heuristic scores of variables change with each backtrack, and the algorithm may determine unsatisfiable entire sub-trees of the lookahead tree. The subtrees need to be re-computed to ensure that the approach produces 2^d partitions. Finally, it is not clear how SMT solver's theory specific reasoning part interacts with the lookahead-heuristic that only measures the reduction in the propositional space.

To the best of our knowledge, this paper is the first to build lookahead partitioning into the SMT framework in a way that observes the search space reduction resulting from learned clauses, and guarantees the unit-propagation consistency of the resulting partitions in case instance satisfiability is not de-

terminated. We consider the theories of uninterpreted functions with equality [3] and linear real arithmetic [4]. These are the two central algorithms that constitute, together with a SAT solver, the core of most SMT solvers. Combinations of these two theories with pre-processing techniques are capable of handling the quantifier-free subset of the SMT-LIB benchmark library instances. The algorithm either produces exactly 2^{d-1} instances none of which can be shown unsatisfiable through (theory-aware) unit-propagation in the current state of the SMT solver; or shows the original instance either satisfiable or unsatisfiable. The partitioning algorithm compromises in certain cases the exactness of the lookahead scores for decreased run time. We believe that the efficiency of our proof-of-concept implementation forms a solid basis for future research in this direction. Since the approach also sheds light to the observed slowdowns, we believe that the work will prove useful for designing more general parallelization algorithms for SMT.

The paper is organized as follows. After discussing related work, in Sec. III we define our SMT-related logical notation. In Sec. IV we adapt the rule-based description of SMT from [25] to the specific case of lookahead and introduce a running example. In Sec. V we present our lookahead partitioning algorithm, then provide experimental results in Sec. VI, and conclude in Sec. VII.¹

II. RELATED WORK

The lookahead heuristic was first introduced in the context of DPLL-based SAT solving in [27]. The original idea uses the number of propagated literals as a measure of search space reduction [23], and is further extended to consider, e.g., equivalence reasoning [5], the clause-based Jeroslow-Wang heuristic [16], and approaches for choosing which variables to consider for lookahead [7].

Lookahead as a pre- and in-processor for clause-learning SAT solvers was formalized in [6]. However, it was not integrated into the CDCL algorithm in the sense that is done in this work. A similar pre- and in-processing approach was recently implemented for the SMT solver Z3 [20]. When used as a pre- and in-processor for an ordinary, CDCL-based solver, the lookahead implementation can be conceptually fairly straightforward. Lookahead is not directly involved in the CDCL search, and therefore the artifacts related to non-chronological backtracking need not be necessarily considered. In [12] we formalized an algorithm inspired by the lookahead heuristic for solving quantifier-free first-order formulas based on CDCL SMT solving. The approach is implemented in our SMT solver OpenSMT [11] and was shown experimentally to be efficient for solving linear integer arithmetic problems with Boolean structure. Compared to the publication, in the current work we give a more formal treatment of the implementation,

¹An extended version of the paper, available at <https://usi-verification-and-security.github.io/opensmt-doc/publications/lookahead-in-partitioning-smt-extended.pdf>, provides an appendix detailing some of the optimizations we implemented for the lookahead approach, further experiments, and a comparison to an alternate scoring for the lookahead algorithm.

define the lookahead algorithm for partitioning, and provide experimental data and analysis for parallel solving based on cube-and-conquer.

Our focus is in how SMT lookahead can implement partitioning in divide-and-conquer for parallel solving. The idea was introduced for parallel SAT solving in [10], and an implementation for parallel SMT solving was used in [13], [17]. However, the details of this partitioning approach have not been discussed before. The lookahead-based partitioning implementation in [10] applies essentially lookahead-based binary partitioning recursively. The downside of this design is that it does not use the full information in the CDCL solver, and producing the partitions might miss an unsatisfiability high up in the tree. As a result it constructs partitions that are known to be unsatisfiable in an intermediate state of the partitioning algorithm.

The substantial amount of research in SAT heuristics, overviewed in [1] from the perspective of parallel solving, provides a promising foundation for partitioning in SMT. Recent relevant approaches include [15], where the authors recognize high-level information that can be used for better clause learning.

III. PRELIMINARIES

The *Satisfiability Modulo Theories* (SMT) problem [22], [3] consists of determining whether a propositional formula is satisfiable, given that some of the atoms have an interpretation in first-order logic. A *conflict-driven clause learning* (CDCL) SMT solver searches first for propositional models, which are then checked for consistency with respect to the theory. If found inconsistent, the propositional structure is enriched with an *explanation*, that is, a clause containing in general theory atoms. If instead during the process the propositional part becomes unsatisfiable, the solver has shown the whole formula unsatisfiable. The formula is satisfiable if the solver finds a theory-consistent model.

1) *SMT solving*: This section fixes the notation for first-order logic and SMT. We define sets of function symbols, terms, constants, and predicate symbols as usual, the last containing the special symbols \top , \perp , and $=$ that represent, respectively **true**, **false**, and equality. We call applications of predicate symbols on terms *atoms*. Let U be a possibly infinite set of elements containing at least the truth values **true** and **false**. A *model* \mathcal{M} assigns to each constant a unique element from U , to each function symbol of arity $n \geq 1$ a total function $U^n \rightarrow U$, to each predicate symbol of arity zero a truth value **true** or **false**, and to each predicate symbol of arity $n \geq 1$ a total function $U^n \rightarrow \{\mathbf{true}, \mathbf{false}\}$. An *interpretation* \mathcal{A} is the extension of \mathcal{M} to general terms in the usual sense.

Given a finite set of atoms At , a *clause* is a set of *literals*, that is, positive and negative atoms $x, \neg x$, $x \in At$. We extend the negation to clauses, and write $\neg(l_1 \vee \dots \vee l_n)$ for $\neg l_1 \wedge \dots \wedge \neg l_n$. A *propositional formula in conjunctive normal form* (CNF) is a conjunction of clauses. Throughout the text we use both a set of literals and disjunction, and a set of clauses and a conjunction, interchangeably. We also treat conjunctions of

unit clauses (*cubes*) as sets of literals when this cannot be confused with a disjunction. A sequence of literals is written $l_1 \dots l_n$, and when the order plays no role, we equate the sequence with the corresponding set $\{l_1, \dots, l_n\}$.

A set of literals X is *consistent* if for no x both $x \in X$ and $\neg x \in X$. A consistent set σ is called an assignment. An assignment is *total* if for all atoms $x \in \text{At}$ either $x \in \sigma$ or $\neg x \in \sigma$. An atom x is *assigned* if either $x \in \sigma$ or $\neg x \in \sigma$. The assignment σ satisfies a clause c when $\sigma \cap c \neq \emptyset$, and a formula ϕ if it satisfies all clauses of ϕ . A *theory* T is a non-empty set of models. A CNF formula ϕ is *T-satisfiable* if (i) there exists a satisfying total assignment σ for ϕ and an interpretation \mathcal{A} that is an extension of a model $\mathcal{M} \in T$, and (ii) for each $l \in \sigma$, $l^{\mathcal{A}} \equiv \text{true}$ if l is of the form x ; and $l^{\mathcal{A}} \equiv \text{false}$ if l is of the form $\neg x$, where x is an atom of ϕ . In particular, given a formula ϕ and an assignment σ that is total (with respect to ϕ), we write $\sigma \models_T \phi$ if σ is such an assignment. In addition we write $\phi' \models_p \phi$ if all assignments that satisfy ϕ' also satisfy ϕ propositionally, and $\models_T c$ if c is entailed by the theory, that is, a *theory lemma* of a theory T . For a formula, clause, literal, or assignment ξ we denote by $\text{Ats}(\xi)$ the set of atoms appearing in ξ .

In this work we study two theories: the theory of linear real arithmetic (LRA) and the theory of uninterpreted functions with equality (EUF). The universe of LRA consists of real numbers, function symbols $*$ and $+$ of arity two restricted to expressing linear terms, and the predicate symbol \leq ; all three have their usual interpretations. The EUF theory places no restrictions on the interpretations of constants, functions, or predicates (apart from the inherent ones for equality, \top , and \perp).

2) *Parallel SMT solving*: Given an SMT instance ϕ , *partitioning* produces instances ϕ_1, \dots, ϕ_k such that the satisfiability of ϕ is equal to the satisfiability of the disjunction $\phi_1 \vee \dots \vee \phi_k$. In addition, we are interested in partitionings such that no two partitions ϕ_i, ϕ_j , $i \neq j$, share a total satisfying assignment. The *partitioning approach* $\text{Part}(k)$ consists of solving an SMT instance ϕ by first constructing the partitions ϕ_1, \dots, ϕ_k , and then solving each resulting partition ϕ_i in parallel until one of them is shown satisfiable, or all of them are shown unsatisfiable.

IV. CONFLICT-DRIVEN CLAUSE-LEARNING LOOKAHEAD IN SMT

The *CDCL lookahead algorithm* intuitively guides an SMT solver in a binary tree, using the solver's state to determine how to expand the tree. To more precisely describe the algorithm, we adapt here the rule-based presentation of CDCL(T) from [25], [21] to our needs. As usual, in the first phase an input SMT formula is converted into an equisatisfiable propositional formula ϕ in CNF while preserving the atoms in the theories T . The *state* $\langle \sigma \mid F \rangle$ of an SMT solver consists of σ , an initially empty assignment, and F , a set of clauses initially consisting of ϕ . The execution of the solver proceeds according to a set of rules described below. In general, the algorithm alternates between *propagation*, choosing a *decision literal*,

denoted by x^δ , and analysing conflicts found in propagation. The labels L and E refer to *learned* and *explanation* clauses. When they appear on the left side of \rightarrow , the corresponding rule matches only to clauses that have the label.

- The *propagation* rule $\langle \sigma \mid F \wedge (c \vee l) \rangle \xrightarrow{\text{Prop}} \langle \sigma l \mid F \wedge (c \vee l) \rangle$ where c is a clause, and $\neg c \subseteq \sigma$, $l \notin \sigma$ and $\neg l \notin \sigma$, expands the assignment with literals that are logical consequences in the current state.
- The *theory propagation* rule $\langle \sigma \mid F \rangle \xrightarrow{\text{TProp}} \langle \sigma l \mid F \wedge (c \vee l)^L \rangle$ uses theory lemmas to lift information to the propositional level allowing new literals to propagate. It can be applied if $\sigma \models_T l$, l or $\neg l$ appears in F , $l \notin \sigma$ and $\neg l \notin \sigma$, and c is a clause such that $\sigma \models_T \neg c$ and $\models_T c \vee l$.
- The *decision* rule $\langle \sigma \mid F \rangle \xrightarrow{\text{Dec}} \langle \sigma l^\delta \mid F \rangle$ decides a literal l , where l or $\neg l$ appears in F , and $l \notin \sigma$ and $\neg l \notin \sigma$.
- The *theory explanation* rule $\langle \sigma \mid F \rangle \xrightarrow{\text{TExp}} \langle \sigma \mid F \wedge c^E \rangle$ is used to lift theory to propositional level based on observed conflicts in the theory solver. It can be applied when each atom of c appears in $\langle \sigma \mid F \rangle$, $\sigma \models_T \neg c$, and $\models_T c$.
- the *propositional explanation* rule $\langle \sigma \mid F \rangle \xrightarrow{\text{PExp}} \langle \sigma \mid F \wedge (c_1 \vee c_2)^E \rangle$ is the standard resolution rule, which can be applied if $c_1 \vee x \in F$ and $c_2 \vee \neg x \in F$. However, due to the invariants of the underlying SAT solver, we require in addition that $\neg c_1 \subseteq \sigma$ and $\neg c_2 \subseteq \sigma$.
- the *backjump* rule $\langle \sigma l^\delta \sigma' \mid F \wedge c^E \rangle \xrightarrow{\text{BJ}} \langle \sigma l' \mid F \wedge (c' \vee l')^L \rangle$ learns clauses that steer the search. It is applicable if $\neg c \subseteq \sigma l^\delta \sigma'$, there is a clause $c' \vee l'$ such that (1) $F, c \models_p c' \vee l'$ and $\neg c' \subseteq \sigma$; (2) $l' \notin \text{Ats}(\sigma)$ and $\neg l' \notin \text{Ats}(\sigma)$; and (3) l' or $\neg l'$ occurs in $\sigma l^\delta \sigma'$ or $F \wedge c$.
- The *fail* rule $\langle \sigma \mid F \wedge c \rangle \xrightarrow{\text{Fail}} \perp$ corresponds to determining unsatisfiability. It is applicable if $\neg c \subseteq \sigma$, and σ contains no decision literals.
- The *reset* rule $\langle \sigma \mid F \rangle \xrightarrow{\text{Reset}} \langle \emptyset \mid F \rangle$ can be applied at any time.
- the *forget* rule $\langle \sigma \mid F \wedge c^L \rangle \xrightarrow{\text{Forget}} \langle \sigma \mid F \rangle$ is used for forgetting learned clauses, essentially to keep memory usage in control
- The *undo* rule $\langle \sigma l^\delta \sigma' \mid F \rangle \xrightarrow{\text{Undo}} \langle \sigma \mid F \rangle$ is finally required to implement the backtracking while computing lookahead.

A CDCL(T)-based SMT solver works by applying the above rules with two restrictions. (i) The solver always computes the *unit propagation closure* before deciding a new literal, i.e. the rule *Dec* is never applied if the rule *Prop* is applicable; and (ii) to notice any theory inconsistencies when a propositional assignment is found, if the rule *Dec* cannot be applied (i.e., all atoms are assigned) the solver applies the rule *TProp*. The solver always terminates if both the rules *Reset* and *Forget* are applied with an increasing interval [2].

Since the unit-propagation closure has a central role in computing lookahead, we give here two useful, related definitions in the above notation. Given a solver state $\langle \sigma \mid \phi \rangle$, the *unit*

propagation closure $UP(\sigma, \phi)$ is the set of literals $\sigma' \supseteq \sigma$, where $\langle \sigma' \mid \phi \rangle$ is the state obtained by applying the rules *Prop* and *TProp* until neither one applies. A solver state $\langle \sigma \mid \phi \rangle$ is called *unit propagation consistent* or *consistent* if the set $UP(\sigma, \phi)$ is consistent.

The following running example illustrates the use of the rules. The notation $Prop^*$ indicates a sequence of propagations.

Example 1: Consider the conjunction $F = (\neg x \vee (b \leq c))^{(1)} \wedge (\neg x \vee (a \leq b))^{(2)} \wedge (\neg(a \leq d) \vee \neg(a \leq b) \vee \neg(a \leq c))^{(3)} \wedge ((c \leq d) \vee \neg(b \leq c) \vee (a \leq d))^{(4)} \wedge ((c \leq d) \vee \neg(a \leq d) \vee (a \leq c))^{(5)}$ where the numbers in parentheses label the clauses. The following is a possible computation of the CDCL(T) system.

$$\begin{aligned} &\langle \emptyset \mid F \rangle \xrightarrow{Dec} \langle x^\delta \mid F \rangle \xrightarrow{Prop^*} \langle x^\delta(b \leq c)(a \leq b) \mid F \rangle \xrightarrow{Dec} \\ &\langle x^\delta(b \leq c)(a \leq b)\neg(c \leq d)^\delta \mid F \rangle \xrightarrow{Prop^*} \\ &\langle x^\delta(b \leq c)(a \leq b)\neg(c \leq d)^\delta(a \leq d)\neg(a \leq c) \mid F \rangle \xrightarrow{PExp} \\ &\langle x^\delta(b \leq c)(a \leq b)\neg(c \leq d)^\delta(a \leq d)\neg(a \leq c) \mid \\ &\quad F \wedge ((c \leq d) \vee \neg(b \leq c) \vee (a \leq c))^E \rangle \xrightarrow{BJ} \\ &\langle x^\delta(b \leq c)(a \leq b) \mid F \wedge C_1^L \rangle \end{aligned}$$

where the learned clause, obtained by resolution, is $C_1^L := (c \leq d \vee \neg b \leq c \vee \neg a \leq b)^L$. Continuing the example, we get

$$\xrightarrow{TProp} \langle x^\delta(b \leq c)(a \leq b)(c \leq d)(a \leq c) \mid F' \rangle$$

where $F' := F \wedge C_1^L \wedge (\neg(a \leq b) \vee \neg(b \leq c) \vee (a \leq c))^L$, the last being a valid clause in the theory, and

$$\begin{aligned} &\xrightarrow{Prop^*} \langle x^\delta(b \leq c)(a \leq b)(c \leq d)(a \leq c)\neg(a \leq d) \mid F' \rangle \\ &\xrightarrow{TExp} \langle x^\delta(b \leq c)(a \leq b)(c \leq d)(a \leq c)\neg(a \leq d) \mid \\ &\quad F' \wedge (\neg(a \leq c) \vee \neg(c \leq d) \vee (a \leq d))^E \rangle \\ &\xrightarrow{BJ} \langle \neg x \mid F' \wedge \neg x^L \rangle \end{aligned}$$

where $\neg x^L$ is obtained through a resolution derivation on clauses in F' and the explanation.

V. LOOKAHEAD-BASED PARTITIONING FOR SMT

This section describes the lookahead-based algorithm for partitioning an SMT instance into 2^d partitions or determining whether the instance is satisfiable.

A. The Lookahead Score

Lookahead in a backtracking search consists in general of repeated trial and backtracking on all available branches at a certain point of the search, and committing to the one that seems most promising. We define the relation between SMT solver states before and after the trial branch, and the lookahead score as the difference between the two. The approach is oblivious to the details on how the lookahead score between two states s and s' is defined. Our implementation supports two scoring functions, one based on the number of free atoms in the instance globally [23], and the other on unassigned atoms in the clauses of the instance [8]. Our examples and experiments in this paper use the former.

Lookahead aims to assign with the rule *Dec* the literal that minimizes the upper bound for the remaining search space. Given a state s where neither *Prop* nor *TProp* applies, we define the *lookahead step* on a literal l as the sequence of rules starting from s , having *Dec* on l as the first rule, followed by unit propagation closure computation resulting in the state s' , and finally an *Undo* on l ending in state s . This sequence is not always possible, and we describe in Sec. V how we handle the failed cases. For a consistent state $\langle \sigma \mid \phi \rangle$, the set $UP(\sigma, \phi)$ is unique. Therefore we can define the lookahead score of a literal l based on a difference between $\langle UP(\sigma, \phi) \mid \phi \rangle$ and $\langle UP(\sigma l, \phi) \mid \phi \rangle$. We denote the *lookahead score* of literal l by $score(l) = |UP(\sigma \cup \{l\}, \phi) \setminus UP(\sigma, \phi)|$, that is, the number of propagated literals after deciding l , and extend the definition to atoms x as

$$score(x) = \min(score(x), score(\neg x)), \quad (1)$$

which minimizes the sum of the upper bounds for the remaining search spaces [23].²

B. Lookahead-Based Partitioning

Algorithm 1: The lookahead partitioning algorithm.

Input : An SMT instance ϕ in CNF; Tree depth d
Output: Sat, Unsat, or a balanced binary tree of depth d
Data : Solver s , DFS stack $stack$

```

1 restart ← true
2 while restart do
3   restart ← false;
4   r ← empty node;
5   stack.push(r);
6   while stack.size ≠ 0 do
7     n ← stack.pop();
8     res ← setSolverToNode(s, n);
9     if res = Unsat then return Unsat;
10    if res = BackJump then
11      restart ← true;
12      break;
13    if Depth of n is d then continue;
14    c, c', res ← expandTree(s);
15    if res = Unsat then return Unsat;
16    if res = Sat then return Sat;
17    if res = BackJump then
18      restart ← true;
19      break;
20    stack.push(c);
21    stack.push(c');
22  end
23 end
24 return the tree rooted at r;
```

The approach is presented in Alg. 1. The algorithm constructs a tree with nodes labelled with literals. The tree is constructed depth-first using the *stack*, with the help of a CDCL(T) SMT solver s . The intuition is that the tree is being built by guiding the SMT solver along the rooted paths and lookahead heuristic is used to expand a leaf node. The

²There are other definitions for lookahead score, but they all favor atoms that minimize the remaining search space on both polarities [8].

algorithm limits the search depth to the input value d , and is also a sound but incomplete (if $|\text{Ats}\phi| > d$) SMT solver.

Let n^i denote a node n at depth i in the tree. Then each path in the tree from the root n^0 to a leaf n^i corresponds to a partition as follows. We label the nodes n with a literal $\text{Lab}(n)$, and n^0 is labelled $\text{Lab}(n^0) = \top$. A path $n^0 \dots n^i$ is interpreted as a cube, and $n^0 \dots n^d$ in the tree corresponds to the partition $\phi \wedge \text{Lab}(n^0) \wedge \dots \wedge \text{Lab}(n^d)$.

The main work, done in the loop between lines 6 – 22, consists of two phases: *setting the solver s to a given node* on Line 8, and *expanding the lookahead tree* on Line 14. We describe both phases, referring to the rules in Sec. III.

1) *Expanding the lookahead tree*: The lookahead tree is expanded with new nodes c, c' by the function *expandTree* on Line 14. Using the solver s the function computes the lookahead step for each literal $x, \neg x$ not assigned in σ as described in Sec. V-A. The process may be interrupted by three special conditions:

- The rule *Fail* becomes applicable. In this case the function returns *Unsat*.
- A total assignment is found: the function returns *Sat*.
- The rule *BJ* becomes applicable. In this case:
 - If *BJ* becomes applicable with $l^\delta = x$ or $l^\delta = \neg x$, the function does a *local restart*: it forgets the computed lookahead scores and restarts the lookahead computation.
 - If *BJ* is applicable with $l^\delta = y$ or $l^\delta = \neg y$ for some earlier decision literal $y \neq x$, the function does a *complete restart* by returning *BackJ mp*.

If *expandTree* determines satisfiability, the algorithm terminates and reports the result immediately. The distinction between local and complete restarts is motivated by efficiency and has deep implications to the algorithm. We discuss this point in Sec. V-B3.

2) *Setting the solver to a given node*: A lookahead path obtained from the stack is used to set the solver s to the correct state where the lookahead scores of literals can be computed. This is done in Line 8 by the call to the function *setSolverToNode* that takes as arguments the solver $s = \langle \sigma \mid F \rangle$, and the current node $n = n^k$. The function initially applies the rule *Reset* on the solver, and computes the unit propagation closure at the root by $\sigma = \text{UP}(\emptyset, F)$. Then, for each $n^0 \dots n^k$ the function applies *Dec* with $l = \text{Lab}(n^i)$, and sets $\sigma = \text{UP}(\sigma l, F)$. The process may be interrupted in two cases:

- *Fail* becomes applicable. This corresponds to the derivation of unsatisfiability, and the process returns *Unsat*.
- *BJ* becomes applicable. The node is locally unsatisfiable and our implementation restarts the construction of the lookahead tree to avoid unbalancedness.

Otherwise, setting solver to the node succeeds and the algorithm proceeds with expanding the tree.

To clarify the behavior of the algorithm, we show its execution on the running example (Example 1).

Example 2: Let $\phi = F$ from Ex. 1 and $d = 2$ for Alg. 1. The algorithm advances to line 14 to compute the lookahead scores of the variables using solver s . No conflicts are detected by s , literal x propagates $\{b \leq c, a \leq b\}$, and literals $\neg b \leq c$ and $\neg a \leq b$ propagate $\{\neg x\}$. No other branch results in propagations. Hence the score from Eq. (1) is zero for all atoms.

Say the algorithm expands the tree, that up to now consisted only of the empty root, with nodes labeled $\neg x, x$, and pushes both nodes to the DFS stack. Assume that the algorithm first branches on $\neg x$. None of the free literals propagate, and tree is expanded for example with $\neg a \leq d$ and $a \leq d$. Once these are popped from the stack, the tree would consist so far of branches $(\neg x(a \leq d))$, $(\neg x \neg(a \leq d))$, and (x) .

The algorithm will now pop x on line 7. On line 14, during the execution of the lookahead heuristic, the algorithm will do the lookahead step on $b \leq c$. This triggers the conflict-handling sequence shown in Ex. 1 resulting in the solver state $\langle \neg x \mid F \wedge ((c \leq d) \vee \neg(b \leq c) \vee \neg(a \leq b))^L \wedge (\neg(a \leq b) \vee \neg(b \leq c) \vee (a \leq c))^L \rangle$. Backjump is on the earlier decision literal $a \leq c$, not on the most recent decision literal $b \leq c$ (see the description above for *expandTree*), and therefore *expandTree* will return *BackJ mp*, restarting the tree construction.

The algorithm builds now the tree similar to the first time, but when computing lookahead in state $\langle x(b \leq c)(a \leq b)(c \leq d)(a \leq c) \neg(a \leq d) \mid F' \rangle$ there are no free variables, and the algorithm reports satisfiability.

3) *Observations on the backjumps*: The backjump during the above execution is critical for the partition quality. It is relatively easy to see that applying recursively a lookahead algorithm on the original problem, as in [10], produces partitions that in a later state of the solver would not be unit-propagation consistent.

First, one could imagine a version of the algorithm that backtracks to the level indicated by the backjump, similar to the underlying SMT solver. This choice would intuitively result in less repeated work as the previously built lookahead tree would be preserved, and therefore conceivably in a more efficient algorithm. However, there are two reasons why the restart is necessary. First, a clause c learned in a backjump at *expandTree* on node n^i alters the lookahead scores in an unpredictable way in the solver states closer to the root. The current lookahead tree becomes in general invalid from the heuristic perspective. Without the restart, the clause should be considered in all previous invocations of *expandTree* at least in the nodes $n^0 \dots n^{i-1}$, and tracking such propagations would be expensive. Second, allowing backjumps in the lookahead tree means that when setting the solver to a new node (Line 8), a learned clause can cause a conflict not present when the node was pushed (lines 20 and 21). In this case it is unclear how the algorithm should proceed to construct the balanced binary tree with consistent partitions.

The distinction between local and complete restarts stems from the above two observations. Complete restarts are too expensive to be performed on every conflict, a relatively

common event during the lookahead computation. Instead, they are done only on the long backjumps that are rare in lookahead-based branching. The consequence of having the local restarts is that *setSolverToNode* may result in a conflict. While this introduces a performance overhead, it turns out to be very rare and therefore insignificant in practice.³

We still recompute the lookahead scores in a local restart, since the error caused by omitting this may grow very large, as shown by this example where not recomputing the lookahead after a conflict would mis-calculate a literal's score with a maximum possible error.

Example 3: Consider the following derivation, where a lookahead at $\langle \sigma \mid G \rangle$ on x^d fails with the learned clause $(c \vee \neg x)^L$:

$$\begin{aligned} \langle \sigma x^d \mid G \rangle &\xrightarrow{PExp} \langle \sigma x^d \mid G \wedge c'E \rangle \xrightarrow{BJ} \langle \sigma \neg x \mid G \wedge (c \vee \neg x)^L \rangle \\ &\xrightarrow{Prop} \langle \sigma \neg x \sigma' \mid G \wedge (c \vee \neg x)^L \rangle. \end{aligned}$$

Assume now that G has as a subformula $(x \vee v \vee p_1) \wedge \dots \wedge (x \vee v \vee p_n) \wedge (x \vee \neg v \vee q_1) \wedge \dots \wedge (x \vee \neg v \vee q_n)$, where p_i, q_i and v do not appear in $\text{Ats}(\sigma')$. Then the lookahead score of v at $\langle \sigma \mid G \rangle$ is 0 but in the state $\langle \sigma \neg x \sigma' \mid G \wedge (c \vee \neg x)^L \rangle$ the score is n . Note that n is upper bounded by $|\text{Ats}\phi|$ which in our scoring is also the highest heuristic value.

4) *Correctness and termination:* We finish the discussion with proofs on correctness and termination for Alg. 1

Theorem 1: The algorithm either determines the satisfiability of the instance or constructs a balanced binary tree with each rooted path leading to the leaves corresponding to a unit-propagation consistent SMT instance.

Proof. The correctness of the Sat and Unsat results reported by the algorithm follow immediately from the observation that the result is obtained by modifying the solver state with the rules outlined in Sec. IV. Each rooted path of the tree corresponds to a unit propagation consistent instance. This follows from two observations. First, if *setSolverToNode* succeeds on a node n , the instance corresponding to the node is unit propagation consistent. Second, if *expandTree* succeeds, similarly by construction the instances corresponding to the nodes c and c' are consistent. The resulting tree is balanced, since unless the execution terminates in lines 9, 15, or 16, the algorithm performs a DFS with a cutoff at depth d . \square

Theorem 2: The algorithm terminates.

Proof. The procedure *setSolverToNode* terminates since it performs a sequence that is bounded by the depth of the node and consists of rules *Dec* and unit propagation closure computations that both terminate. The procedure *expandTree* terminates in quadratic number of applications of *Dec*, *Undo* and unit propagation closure computations: the computation consists of lookahead steps each bounded by the number of atoms $|\text{Ats}(\phi)|$. The local restart at a node n can be done at most $|\text{Ats}(\phi)|$ times, since each related backjump will assign at least one atom in the truth assignment of the solver state at node n .

³We observed three conflicts while partitioning over 9000 instances in different ways.

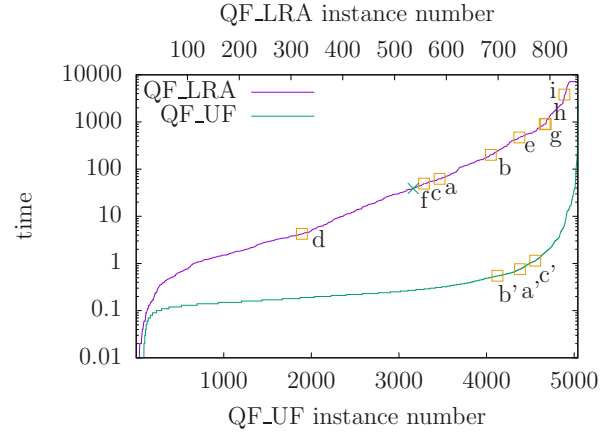


Fig. 1. Runtime for lookahead partitioning to 16 for QF_LRA and QF_UF. Labeled boxes and crosses refer to specific instances discussed below. Unsatisfiable instances are denoted with boxes (\square), and satisfiable with crosses (\times).

The restarts in tree construction on lines 11 and 18 will not cause non-termination since the solver state is persistent (modulo possible applications of *Reset*) over such restarts. Following [18], the assignments of the solver together with the literals can be seen as a finite ordered sequence that is increased by every backjump and has a maximum element where every atom is assigned with no decision literals. \square

VI. EXPERIMENTS

We report experiments on our implementation on the non-incremental benchmark divisions QF_UF and QF_LRA of SMT-LIB.⁴ The two divisions are chosen since they constitute the foundation of most other SMT logics and allow us to directly observe the behaviour of the congruence closure (egraph) and the Simplex algorithms under lookahead. All the experiments were run using the SMT solver OpenSMT [13]. The partitions are constructed with the implementation of Alg. 1, and, when applicable, solved with OpenSMT's default CDCL(T) engine running the VSIDS heuristic [19], a setup similar to most CDCL(T) solvers. The CPU time consumed by the experiments is slightly under 338 CPU days. We used a Linux cluster, equipped with two Intel Xeon E5-2650 v3 @ 2.30GHz CPUs, yielding (2×10) cores per node. Each node has 64GB of DDR4@2133MHz memory. We ran at most ten solvers on each node simultaneously, limiting the memory available for a solver to 4GB. The time out was 7200 s for both the partitioning and solving, except in Fig. 2 where the timeout was 1200 s. We first report on the efficiency of the partitioning implementation, and then show that the partitioning in general works well. Finally we study instances showing a slowdown anomaly. All times are given in seconds and refer to wall-clock times.

1) *Lookahead partitioning efficiency:* The plots in Fig. 1 illustrate the run times of Alg. 1 on the QF_LRA and QF_UF

⁴The benchmarks are available at <https://clg-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks> under commit hash 33961bc4.

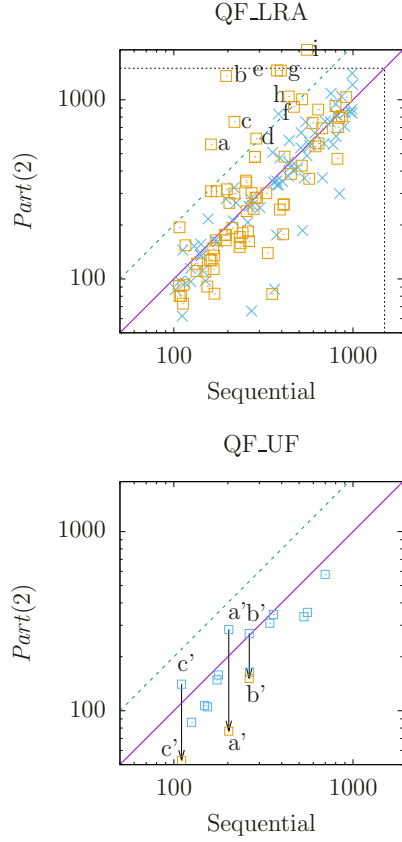


Fig. 2. Comparing sequential and *Part*(2) run times for QF_LRA (top) and QF_UF (bottom). On the top figure the boxes pointed to by the arrows are from *Part*(64) and show the approach efficient. The efficiency for QF_LRA is studied separately.

instances when partitioning into 16. The instances are ordered based on the run time. We only report the instances not solved during partitioning. The implementation is efficient in particular for QF_UF, where the maximum stays in the majority of cases within a few seconds. The lookahead on QF_LRA is much more involved, perhaps due to the more expensive theory solving. Our implementation partitions 98% of the benchmarks within two hours, showing that the approach is realistic.

2) *Effect of partitioning on instance difficulty*: To measure how partitioning affects the instance difficulty, we study instances that *OpenSMT* can solve between 100 and 1000 seconds sequentially, a range where parallelization is useful but the baseline can still be computed within a reasonable time. This resulted in 13 instances for QF_UF and 144 instances for QF_LRA. The reported times do not include partitioning.

Figure 2 compares *Part*(2) to sequential solving for QF_LRA (top) and QF_UF (bottom). We plot the line $y = x$ corresponding to no speed-up, and the dashed line $y = 2x$ corresponding to two-fold slowdown. The dashed horizontal and vertical lines in the top figure show the timeout of 1200 seconds. Crosses (\times) and boxes (\square) indicate satisfiable and unsatisfiable instances, respectively.

Except for three cases, *Part*(2) provides a consistent speed-up in QF_UF. We ran these instances in *Part*(64) and each became easier to solve than the original instance (as shown by the downwards arrows that point to the corresponding *Part*(64) measurement). As a conclusion, it seems that lookahead is efficient when combined with the congruence closure algorithm. This is somewhat expected since lookahead is efficient in purely propositional solving, and the congruence closure algorithm is scalable.

It is interesting to compare these results to QF_LRA, where lookahead is efficient in 60% of the instances, but we also observe significant slowdowns, corresponding to up to 6-fold increase in run time. Repeating the experiment of partitioning with *Part*(64) did not result in a positive result similar to QF_UF (see figures 3 – 4), suggesting that this phenomenon has a different origin.

The partitioning run times for the anomalies are shown with the labels in Fig. 1. Typically their run times are above the average.

3) *Slowdown analysis for partitioning*: Despite *Part* resulting in most cases in a consistent speed-up, the significant slowdowns in QF_LRA warrant a separate study, as it poses a threat for lookahead partitioning in SMT. We label with (a) – (i) in Fig. 2 (top) nine instances where the run time more than doubles. We removed the randomness common in heuristic search by solving each partition several times with the *OpenSMT* VSIDS engine while changing the branching heuristic’s random seed. We refer to this approach as the *simulated parallel solver*.

We ran as a pre-processing phase *Part*(k) for $k = 2, 4, 8, \dots, 2048$ for the instances (a) – (i) and stored the resulting partitions if the instance was not solved by *Part*. As a result of time outs and one of the instances being solved during partitioning, we could run the full experiment set only for the instances (a), (d), and (f). We concentrate on these three instances since they seem representative for the others as well.

Figure 3 (top) shows run times for the simulated parallel solver on the only satisfiable instance (f). While the slowdown is consistent for *Part*(2), we observe speedup for *Part*(k), $k \geq 4$. Figure 3 (bottom) shows the simulated parallel median run times on instance (d). The partitions are easy only once a big number, 1024, is reached. We show in addition run time ranges (green bars) and medians (blue stars) for the individual partitions. The instance (i) behaves similarly to this. Figure 4 shows the results for the instance (a), where the minimum, median, and maximum run times consistently increase. We show also the individual *Part* runs as yellow boxes. Instances (b), (c), (e), (g), and (h) behave similarly to (a). While the lookahead clearly identifies easier partitions, the hardest partitions seem to get more difficult. In particular Figs. 3 (bottom) and 4 show a significant amount of partitions having the median time higher than the sequential median. The slowdown can be argued to result in part directly from these partitions.

The slowdown, affecting not uniformly all instances, seems

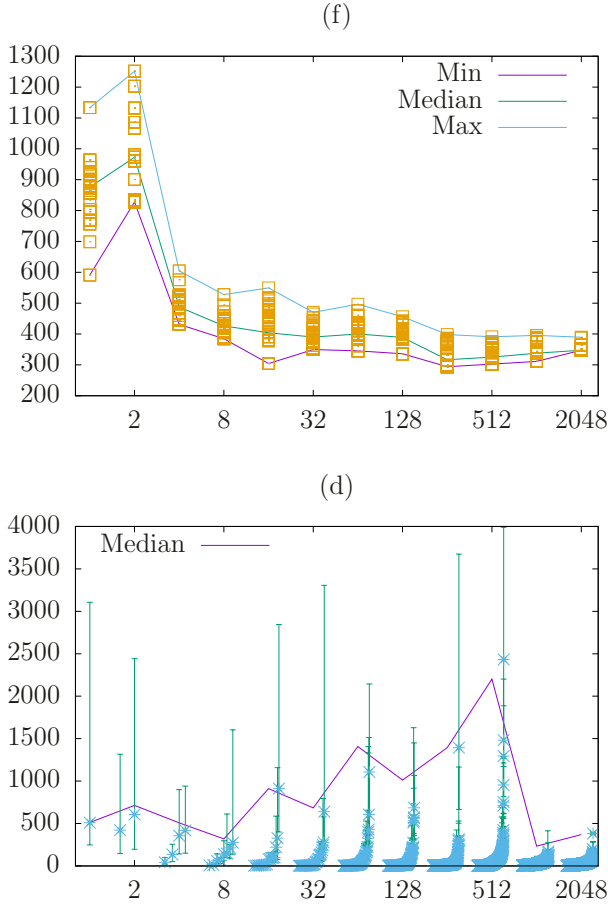


Fig. 3. Scalability for a satisfiable instance (*top*) and partition difficulty for an unsatisfiable instance (*bottom*). The horizontal axis refers to number of partitions produced, and the vertical axis to run time in seconds.

to be the result of an intricate interaction between lookahead and the incremental Simplex implementation typically used in SMT solvers [4]. The implementation maintains an internal model for its real valued variables that satisfies all currently asserted inequalities. If a new inequality is not satisfied in the model, this triggers the pivoting sequence of Simplex that is in the worst-case exponential. SMT solvers try to avoid this behavior by branching as much as possible on inequalities that are consistent with the model. Because of lookahead, Simplex is sometimes forced to follow such a sequence, causing the increasing run times for some of the partitions. It is a natural further question how to generalize lookahead to mitigate or avoid these cases.

To conclude, we note that the lookahead partitioning produces in the vast majority of cases very balanced partitions and good speed-up. Nevertheless, the instance run times increase in a significant portion of the benchmarks. In the studied SMT-LIB benchmark divisions, we observed slowdown only for QF_LRA. We believe that it is possible to obtain speed-up also for these instances by developing a version of the lookahead heuristic that considers also the configuration of

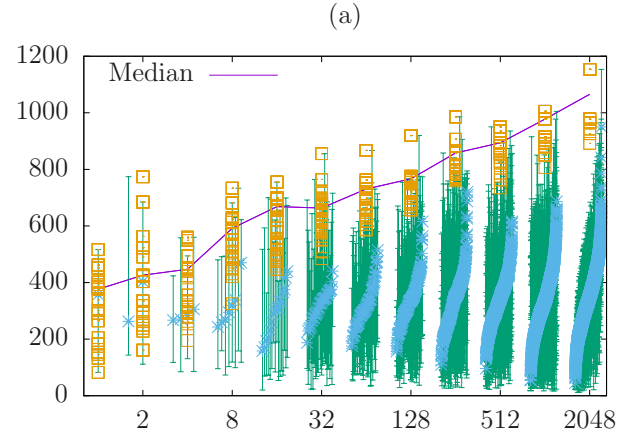


Fig. 4. Scalability and partition difficulty for an unsatisfiable instance. The horizontal axis refers to number of partitions produced, and the vertical axis to run time in seconds.

the theory solvers run inside the SMT solver.

VII. CONCLUSIONS

We present an algorithm for partitioning SMT with lookahead based on $CDCL(T)$ calculus and show experimentally that the approach is highly promising. We also demonstrate that the classical propositional lookahead is not in general sufficient in SMT, where the theory reasoning engines may unexpectedly interfere with lookahead heuristic's view of the search space. In particular we found that in combination with Simplex as implemented in many SMT solvers, lookahead partitioning sometimes creates instances that are increasingly difficult to solve.

In future we plan to extend the lookahead heuristic to better consider the theories. In parallel, we will also study lookahead partitioning in a more applied setting, including theory combinations and non-convex theories, when new atoms are introduced.

Acknowledgements. This research was supported by the Swiss National Science Foundation grant number 200021_185031.

REFERENCES

- [1] Balyo, T., Sinz, C.: Parallel satisfiability. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 3–29. Springer (2018)
- [2] Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
- [3] Dettlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. *Journal of the ACM* **52**(3), 365–473 (2005)
- [4] Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Proc. CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer (2006)
- [5] Heule, M., Dufour, M., van Zwieten, J., van Maaren, H.: March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In: Hoos, H.H., Mitchell, D.G. (eds.) Proc. SAT2004. LNCS, vol. 3542, pp. 345–359. Springer (2005)

- [6] Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Proc. HVC 2011. LNCS, vol. 7261, pp. 50–65. Springer (2012)
- [7] Heule, M., van Maaren, H.: March_dl: Adding adaptive heuristics and a new branching strategy. JSAT **2**(1–4), 47–59 (2006)
- [8] Heule, M., van Maaren, H.: Look-ahead based SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 155–184. IOS Press (2009)
- [9] Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the Boolean Pythagorean triples problem via Cube-and-Conquer. In: Proc. SAT 2016. LNCS, vol. 9710, pp. 228–245. Springer (2016)
- [10] Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning SAT instances for distributed solving. In: Proc. LPAR 2010. LNCS, vol. 6397, pp. 372–386. Springer (2010)
- [11] Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Proc. SAT 2016. pp. 547 – 553. No. 9710 in LNCS, Springer (2016)
- [12] Hyvärinen, A.E.J., Marescotti, M., Sadigova, P., Chockler, H., Sharygina, N.: Lookahead-based SMT solving. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) Proc. LPAR-22. EPIc Series in Computing, vol. 57, pp. 418–434. EasyChair (2018)
- [13] Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Proc. SAT 2015. LNCS, vol. 9340, pp. 369–386. Springer (2015)
- [14] Hyvärinen, A.E.J., Wintersteiger, C.M.: Parallel satisfiability modulo theories. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 141 – 178. Springer (2018)
- [15] Iser, M., Kutzner, F., Sinz, C.: Using gate recognition and random simulation for under-approximation and optimized branching in SAT solvers. In: Proc. ICTAI 2017. pp. 1029–1036. IEEE Press (2017)
- [16] Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Ann. Math. Artif. Intell. **1**, 167–187 (1990)
- [17] Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Proc. ATVA 2016. pp. 428–443 (2016)
- [18] Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (1999)
- [19] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC 2001. pp. 530–535. ACM (2001)
- [20] de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS 2008. LNCS, vol. 4963, pp. 337 – 340. Springer (2008)
- [21] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937 – 977 (2006)
- [22] Sebastiani, R.: Lazy satisfiability modulo theories. JSAT **3**(3–4), 141–224 (2007)
- [23] Simons, P.: Extending and Implementing the Stable Model Semantics. Ph.D. thesis, Helsinki University of Technology (2000)
- [24] van der Tak, P., Heule, M., Biere, A.: Concurrent cube-and-conquer - (poster presentation). In: Proc. SAT 2012. LNCS, vol. 7317, pp. 475–476. Springer (2012)
- [25] Tinelli, C.: A DPLL-based calculus for ground satisfiability modulo theories. In: Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23–26, Proceedings. pp. 308–319 (2002)
- [26] Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: A concurrent portfolio approach to SMT solving. In: Proc. CAV 2009. LNCS, vol. 5643, pp. 715–720. Springer (2009)
- [27] Zabih, R., McAllester, D.: A rearrangement search strategy for determining propositional satisfiability. In: Proc. AAAI-88. pp. 155–160. ACM (1988)

A Multithreaded Vampire with Shared Persistent Grounding

Michael Rawson and Giles Reger
University of Manchester

Abstract—Automated theorem provers (ATPs) typically run in a single thread. Hardware parallelism is then exploited through *portfolios*, in which distinct and disjoint strategies are launched as fully-independent processes and do not cooperate. Whilst there has been some historic exploration of cooperation, the technical challenge has prevented this from being fully explored in modern ATPs. The following describes the non-trivial engineering effort required to make the Vampire theorem prover multithreaded, such that multiple proof attempts coexist in the same memory space. This lays the foundations for a new generation of proof search techniques able to cooperate with other proof attempts running in parallel. As an initial demonstration, we implement a shared *persistent grounding* daemon that receives all clauses generated by all proof attempts and checks whether a heuristically-grounded version is unsatisfiable. The resulting multi-threaded system achieves limited contention compared to the previous process-based implementation, and persistent grounding improves performance in certain cases.

I. INTRODUCTION

Whilst parallel computational resources have become abundant and used with effect in many areas of computer science, they are yet to make a significant impact on automated theorem proving. We have seen substantial developments in SAT solving [1], [2], [3] and progress within SMT [4], [5], [6] but, to date, parallel automated theorem proving is typically historic with no modern implementation [7], [8], [9], or parallel at the level of portfolios without shared memory. The popularity of parallel portfolios is likely due to their ease of implementation and practical impact: it is common folklore that a good way to combat explosive proof search is a set of complementary search strategies. This success goes some way to explaining why research in other directions has been slow.

In this paper we discuss our initial work on a new shared-memory architecture for the VAMPIRE automated first-order theorem prover [10]. VAMPIRE is a saturation-based theorem prover that implements the superposition calculus [11] as its main mode, but also contains routines for instance-based reasoning [12] and finite model building [13]. It has won first place in the main track of the CASC competition for over 20 years [14] and implements advanced reasoning techniques for theory reasoning [15], [16], [17], inductive reasoning [18] and higher-order reasoning [19]. It consists of over 200k lines of C++ with contributions from over 15 developers and a permissive licence [20]. As such, it is a mature and highly-complex piece of software.

Since 2010, VAMPIRE has supported some form of multi-process parallelism where a portfolio of predetermined (and automatically generated) *strategies* (sets of proof search

heuristics) could be implemented by forked processes. This achieves good results, but limits options for cooperation between proof attempts due to reliance on inter-process communication. In 2015, we proposed a concurrent architecture [21] that interleaved proof attempts within a single process whilst sharing (some) memory to explore a novel method for cooperation. Our conclusion at the time was that we needed true shared-memory parallelism to make progress.

We experienced two main difficulties with such an approach in VAMPIRE. The first is that it is difficult to implement correctly: this is a well-known feature of parallel programming, and we discuss our approach and experience below. The second is *contention*, which for our purposes is negative performance impact caused by multiple threads using the same resource simultaneously, typically by having to wait for a lock held by another thread. Avoiding contention requires careful design of shared-memory schemes within an ATP.

A reasonable line of questioning raised in review asks whether it would be easier to start from scratch. It would probably be technically easier to do so: however, ATP systems at VAMPIRE's level of maturity take significant time to develop, even with the benefit of hindsight, so instead we offer pragmatic suggestions to convert existing systems.

The two main contributions of this paper are (1) A detailed discussion of the technical challenges and experience involved in transitioning a complex, mature theorem prover from a process-based model to a thread-based, shared-memory architecture (Section II), and (2) A new *persistent grounding* technique designed to take advantage of the shared memory concurrency provided by the architecture (Section III).

II. CHALLENGES AND EXPERIENCE

This section reflects on the engineering challenges we faced when converting Vampire into a multi-threaded solver, and the approach we took to overcome them. We include this discussion to provide guidance for others attempting to complete a similarly-challenging task. Currently, the implementation is available in a branch of the VAMPIRE repository¹.

A. Design

The architecture is based on the previous process-based architecture, which has not previously been described elsewhere. As illustrated in Fig. 1, the input problem is first parsed into a set of initial formulas over a signature (that is, the symbols

¹<https://github.com/vprover/Vampire/tree/caps>

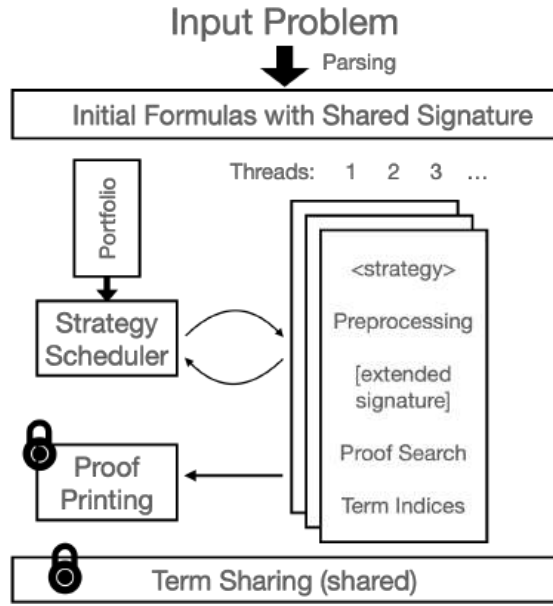


Fig. 1. Schematic of Architecture.

appearing in the problem) shared between all proof attempts. A strategy scheduler uses a portfolio of strategies to generate a set of k threads. The parent scheduler supervises the child threads, reporting success if any child succeeds and spawning new threads to keep available CPU cores busy. Each thread preprocesses the problem, potentially extending the signature by e.g. introducing names for subformulas, and then performs proof search. This typically involves the use of complex data structures (*term indices*) for storing and searching for relevant clauses. VAMPIRE’s complex custom memory allocator is disabled for this work, incurring a small performance hit.

Two complex parts of the architecture are currently protected by a coarse-grained lock. Only one proof attempt should print a proof, so this process is gated such that subsequent successful attempts block forever. A more difficult issue is *term sharing*. Part of the standard VAMPIRE is a hash-consing structure used to implement perfect term sharing, i.e. avoid duplication of terms. This is very convenient as it allows rapid identification of terms by pointer comparison, a property which is assumed throughout VAMPIRE. In our multithreaded architecture we share this structure and protect it by a lock. Term sharing must be able to distinguish between terms built solely from the shared signature and terms involving thread-specific symbols: that is, terms that could appear in any attempt versus terms that only have meaning in a single attempt.

B. Approach

Converting a large, complex and performance-sensitive system such as VAMPIRE to work in thread-parallel is not especially easy. The approach outlined previously [21] in which proof attempts *interleave* in a single thread of execution, rather than exist concurrently, at first seemed like a good intermediate step before starting work on a fully thread-parallel, shared-

memory system. However, we found that bugs introduced by interleaved proof attempts were very difficult to track down, not least because very often they had no observable effect.

Instead we take a more chaotic approach, leaning heavily on tooling for developing multi-threaded applications, particularly tools for detecting *data races*. Data races, for our purposes, are execution scenarios in which two threads access shared memory without synchronisation, and at least one access is a write. Detection of races is extremely useful in our case as it provides a good proxy for identifying when one proof attempt influences the execution of another. Nearly all thread-related bugs — of which there were many — could then be squashed by examining the context in which races occur and introducing synchronisation or data reorganisation where appropriate.

Tools for detecting dubious constructs and execution states in low-level programming have improved significantly. We were particularly impressed by the LLVM-based [22] linter *clang-tidy* [23], which helped to identify and remove existing discouraged constructs in VAMPIRE’s codebase, and the *ThreadSanitiser* [24] compiler instrumentation for the detection of data races. Armed with these tools, we simply introduced threads into VAMPIRE and waited for the tool reports. Races happened frequently in VAMPIRE at first, where code written under the implicit assumption of single-threaded execution breaks down, triggering a ThreadSanitiser report.

In general, data races tend to lead to crashes rather than unsound behaviour but to avoid the latter we rely on (i) existing mechanisms for automated testing utilising large sets of labelled benchmarks [25], and (ii) VAMPIRE’s support for proof checking which allows us to independently verify the correctness of proof search [26].

C. Thread-Local Storage, Atomics and Locking

The most common source of the races was the re-use of heap-allocated temporaries such as stacks or maps, often used in iterative translations of recursive algorithms present throughout the system. Reusing these values once allocated can improve performance in the single-threaded case by avoiding repeated (de)allocations. The majority of such cases can be resolved by the use of thread-local storage as a compromise, incurring one allocation per thread. The 2011 C++ standard [27] provides a `thread_local` keyword and associated machinery.

Another problem area is integer counters, often used for computing statistics and satisfying freshness constraints such as “select a fresh symbol for the Skolem function”. Usually the only operation required is “read-and-increment”, but this must sometimes be reflected across threads to maintain soundness of e.g. Section III. This operation can be safely achieved atomically: C++’s `<atomic>` proved useful here.

Only surprisingly rarely was a full lock required to synchronise compound operations. This relatively-coarse technique was only required for widely-used modules with non-trivial internal invariants such as the implementation of term sharing. Due to the small number of locks, deadlock was mostly avoided.

D. Data Organisation and Partitioning

Significant headaches can be avoided by carefully choosing which data are shared between proof attempts. A clever implementation could aggressively share all common data using very fine-grained synchronisation. For example, VAMPIRE maintains various term indices to quickly retrieve various syntactic data that satisfy some condition, like “retrieve all the literals that unify with L ”. In principle it would be possible to share at least some of these and save some memory, but in practice this is enormously difficult to implement correctly and efficiently. However, we remain interested in parallel term indices and may investigate these independently in future.

Currently, each proof attempt maintains its own clause space, computed properties and statistics, indices, introduced definitions, and ground reasoning systems such as those used in global subsumption [28] or AVATAR [29]. They do however share synchronised access to creating fresh symbols (although not all symbols are used in all proof attempts), term sharing, and persistent grounding (Section III). We feel this is a good initial trade-off.

E. Timing and Internal Control

One crucial difference between the multi-processing and multi-threading approaches to portfolio modes is that processes can be signalled to stop execution in a timely manner, whereas most threading abstractions do not have this ability. Threaded proof attempts must therefore frequently check for exit conditions, e.g. another proof attempt succeeded/time is up. Making these checks can be tricky: too frequently and there will be some performance impact; too infrequently and user experience or portfolio performance will begin to degrade. VAMPIRE executes a series of loops in its internal search routines: each iteration of these loops can take drastically different lengths of time depending upon the input problem.

F. Synchronisation and Performance

All the synchronisation measures introduced do incur some performance impact. Atomic operations are not quite *free*, but are very close in practice. Thread-local storage requires some checks for lazy initialisation, which can occur frequently if the compiler is unable to elide them, and is therefore not as cheap as we would like. VAMPIRE uses a global “environment” structure which was made thread-local: C++ semantics mean that this is considerably more efficient if an extra level of indirection is added such that the environment is accessed via thread-local *pointer*. Locks are currently a major bottleneck: while contention was expected to be high, another problem is that the locked sections are typically relatively short and inexpensive compared to the locking overhead. We will investigate finer-grained locking and alternative locking strategies in future.

G. Experimental Evaluation

To validate the resulting system we carry out two experiments using the 500 first-order problems from the 2020 first-order theorem division of CASC. All experiments in this paper

TABLE I
EVALUATING SCALABILITY OF THREADED ARCHITECTURE.

Threads	# solved	Avg time (s)	Total/Avg (s) on \cap	Speedup
1	399	7.05	2187 / 6.21	-
2	413	4.80	987 / 2.80	2.22
4	412	3.49	520 / 1.48	4.21
6	413	2.79	539 / 1.53	4.06
8	402	3.27	533 / 1.51	4.10
10	404	3.26	534 / 1.52	4.10

are run for 60 seconds per problem on a Ubuntu desktop machine with an 8-core CPU² and 16GB RAM.

Firstly, we compare the new thread-based architecture with the previous process-based implementation. The thread-based architecture solves 413 problems (10 uniquely) and the process-based architecture solves 424 problems (21 uniquely). The slight degradation in performance is unsurprising given the additional contention in the thread-based approach. The symmetric difference reflects the sensitivity of VAMPIRE to variations in timing and memory usage. On average, the new thread-based architecture took 1.25x longer to solve problems. However, this is heavily influenced by short-running problems. Excluding problems solved in under 1s, the slowdown is 1.02x.

Secondly, we examine the scalability of the thread-based solution using the same set of problems whilst varying the number of threads. The results are in Table I. The number of problems solved peaks between 2 and 6 threads. We achieve approximately-linear speedup with 2 and then 4 threads, but then plateau (based on the total time taken to solve the 352 problems solved by all attempts). The average solution time overall was the lowest for 6 threads — the lower average solution times for the intersection of solved problems suggests that these were the easier problems.

In summary, performance degrades slightly when replacing processes by threads (most likely due to contention) but the overhead is acceptable ($\sim 2\%$ on longer running problems).

III. PERSISTENT GROUNDING

As a first step to explore the benefits of the new architecture, we introduce a lightweight form of clause sharing. All clauses produced by all proof attempts are grounded, shared, and passed to a SAT solver to detect a form of *global* inconsistency, i.e. an inconsistency in the ground abstraction of the full search space explored by all proof attempts, past and present.

The idea of grounding the search space of a first-order prover in an attempt to detect inconsistency is not novel [30], [31] and some methods, such as instance generation [12] perform grounding as part of proof search already. What is new in our approach is the *persistence* of the grounding: grounded clauses escape from and outlive their thread, allowing clauses from different proof attempts to interact.

A. Extension to Architecture

We introduce a queue (synchronised by single lock) that proof attempts add produced (and grounded) clauses to and a

²Intel® Core™ i7-6700 CPU @ 3.40GHz

thread that loops, adding the grounded clauses to the MiniSAT solver [32] — yielding if the queue is empty — and checking for unsatisfiability. If the grounding is inconsistent the thread will report this immediately, interrupting other threads. Currently, full proof printing is not implemented and only the unsatisfiable core of grounded first-order clauses is identified. It is work-in-progress to rebuild the derivations that produced these clauses as a separate post-processing step.

We maintain a mapping from (grounded) first-order literals to SAT literals such that a fresh first-order literal leads to a fresh SAT literal, with the mapping stored for later. This mapping relies on the shared term indexing structure to efficiently identify atoms that are shared between proof attempts, ensuring they are represented using the same SAT variables.

B. Grounding Choices

There are numerous ways in which we could choose to ground first-order clauses. We implement three alternatives:

- **fresh**: all variables are replaced by a single fresh constant.
- **common**: all variables are replaced by the most common constant from the input problem.
- **input**: the clause is grounded repeatedly for every constant in the input problem.

Where the input problem is multi-sorted the above constants are selected per-sort. We compute constant frequency on the problem before preprocessing i.e. before subformulas are copied or reduced.

C. Experimental Analysis

We use the same 500 problems and experimental setup as above to analyse the impact of this new addition. Our first experiment is to isolate the impact of persistent grounding from threading by running with a single thread. In this setting, we solve 399 problems without persistent grounding and 398 with (using the **fresh** grounding) but with a symmetric difference of 11 problems — persistent grounding allows us to solve 5 problems we did not solve without it. Some problems were also solved significantly faster: for 8 problems the speedup was $> 2\times$, with one problem (SWB105+1) solved $15\times$ faster (from 25s to 1.6s).

Next, we compare the different grounding mechanisms (using 6 threads). The results are given in Table II (top 4 rows). The first observation is that we solve 8 problems that we did not solve without persistent grounding, and each grounding mechanism solves some problems uniquely.

However, the average time to solve each problem increases. The **fresh** grounding mechanism fares the worst with the **common** grounding mechanism producing proofs more than a second before other mechanisms 5 times. Within this there are some notable interesting cases. For example, GRP667+1 was solved using **input** in 15s whilst others failed to solve it using persistent grounding and it was eventually solved in the normal way after 50s. Similarly, ITP006+4 was solved using **common** in 9s rather than the 25s elsewhere.

TABLE II
PERSISTENT GROUNDING EVALUATION.

	# solved (uniq)	Best by $>1s$	Avg. time (s)
none	413 (6)	-	2.79
fresh	410 (1)	0	3.09
common	411 (2)	5	2.95
input	411 (2)	3	3.11
fresh	410 (2)	4	2.94
active-only	412 (3)	0	3.01
no-splitting	393 (5)	16	3.19
combination of PG	421 (12)	-	2.84 (best)

We explore two further variants (rows 5–7 of Table II): in *active-only* we restrict persistent grounding only to so-called *active* clauses [10] and in *no-splitting* we turned clause splitting off for all strategies. Clause splitting introduces additional (per proof attempt) propositional literals into split clauses, potentially reducing the amount of sharing between proof attempts. Active-only solves more problems and (not shown in the table) enjoys a slight reduction in solving times in cases where persistent grounding is not used to solve the problem. Turning clause splitting off solves fewer problems but is nicely complementary (solving 5 problems uniquely).

In summary, the persistent grounding method can drastically speed up proof search when it finds a proof but it generally adds a noticeable overhead. Overall, we solve 12 problems with variants of persistent grounding that we were unable to solve without it. The main observation is that it is possible to prove more by sharing information between proof attempts than simply running the union of proof attempts separately but more work is required to make this approach efficient.

IV. REFLECTION AND FUTURE WORK

We describe our initial efforts transforming VAMPIRE to a multi-threaded architecture and show how this new shared memory architecture can easily support methods for clause sharing. Whilst the concepts involved are straightforward, the engineering effort required to transform a mature codebase from a process-based single memory architecture to a thread-based shared-memory one is large. We have described our experience for others. Our general findings are:

- 1) It is more important to find a clean way to separate data and isolate points of sharing than it is to introduce “clever” fine-grained synchronisation. This ensures that debugging is manageable. We achieved a lot with `thread_local` and `atomic`.
- 2) In a large codebase like VAMPIRE there are tens or hundreds of little bottlenecks rather than few big ones and they interact in complex ways. Simply optimising one bottleneck rarely gives overall gains, improvements must be more architecturally-focussed.
- 3) Portfolio strategies are typically very short (often $<1s$) so “small” performance hits can have a large impact. Work is required to make portfolios robust to this setting.

The new shared persistent grounding method gave lacklustre results but only represents a first step in a number of oppor-

tunities presented by the new architecture. Directions we plan to pursue in the future include:

- Extending the shared signature. Currently, if two proof attempts introduce a definition for the same subformula this will be added to each local extended signature and the overlap will not be shared. A shared definition manager could increase the size of the shared signature, increasing the opportunity for cooperation.
- As originally proposed in [21], sharing the SAT solver used for clause splitting in AVATAR. Within a single proof attempt, this SAT solver is used to enumerate sub-problems. When shared, it can share information about previously proved sub-problems between proof attempts (similar to sharing learned clauses in parallel SAT [2]).
- Sharing simplification mechanisms (and associated data structures e.g. term indices). VAMPIRE contains a number of mechanisms for removing redundant parts of the search space. By sharing these mechanisms we can import information from other proof attempts that makes the current problem easier.
- Other clause sharing mechanisms. Whilst sharing many clauses risks proof attempts converging (undoing the complementary power), we can explore methods that aim to identify useful clauses to share. A fashionable approach would be to employ machine learning techniques to learn which clauses are good to share. Alternatively, we could take inspiration from SAT's *lazy clause exchange* [33] where clauses are only shared if useful locally. Finally, it is likely that not all clauses will be equally useful to all other proof attempts, which suggests a setting where clauses are *pulled* rather than *pushed* based on a local assessment of usefulness.

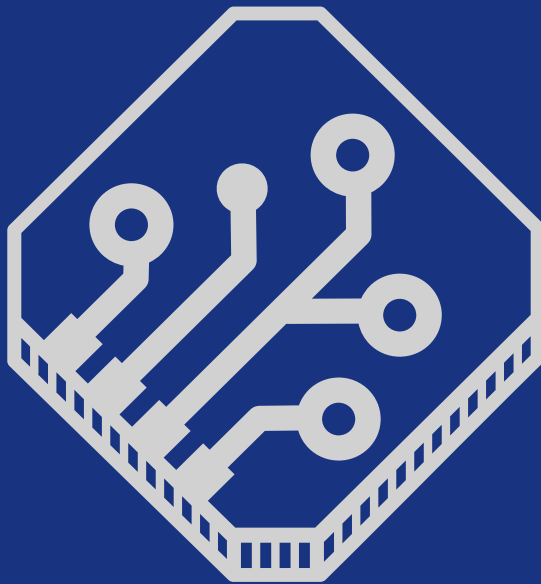
ACKNOWLEDGEMENT

This work was funded by EPSRC project EP/V000209/1: CAPS: Collaborative Architectures for Proof Search.

REFERENCES

- [1] Y. Hamadi, S. Jabbour, and L. Sais, “ManySAT: a parallel SAT solver,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, no. 4, pp. 245–262, 2010.
- [2] T. Balyo and C. Sinz, “Parallel satisfiability,” in *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 3–29.
- [3] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere, “Cube and conquer: Guiding CDCL SAT solvers by lookaheads,” in *Haifa Verification Conference*. Springer, 2011, pp. 50–65.
- [4] C. M. Wintersteiger, Y. Hamadi, and L. Moura, “A concurrent portfolio approach to SMT solving,” in *CAV ’09*, 2009, pp. 715–720. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_60
- [5] A. E. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, “OpenSMT2: An SMT solver for multi-core and cloud computing,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2016, pp. 547–553.
- [6] A. E. Hyvärinen and C. M. Wintersteiger, “Parallel satisfiability modulo theories,” in *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 141–178.
- [7] J. Denzinger and I. Dahn, “Cooperating theorem provers,” in *Automated Deduction—A Basis for Applications*. Springer, 1998, pp. 383–416.
- [8] M. P. Bonacina, “Parallel theorem proving,” *Handbook of Parallel Constraint Reasoning*, pp. 179–235, 2018.
- [9] J. Schumann and R. Letz, “PARTHEO: a high performance parallel theorem prover,” in *CADE*, ser. LNAI, vol. 449, Kaiserslautern, 1990, pp. 40–56.
- [10] L. Kovács and A. Voronkov, “First-order theorem proving and Vampire,” in *CAV 2013*, ser. LNCS, vol. 8044, 2013, pp. 1–35.
- [11] R. Nieuwenhuis and A. Rubio, “Paramodulation-based theorem proving,” in *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds., 2001, vol. I, ch. 7, pp. 371–443.
- [12] K. Korovin, “Inst-Gen – a modular approach to instantiation-based automated reasoning,” in *Programming Logics*, 2013, pp. 239–270. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37651-1_10
- [13] G. Reger and M. Suda, “The uses of SAT solvers in Vampire,” in *Proceedings of the 1st and 2nd Vampire Workshops*, ser. EPIC Series in Computing, L. Kovács and A. Voronkov, Eds., vol. 38. EasyChair, 2015, pp. 63–69.
- [14] [Online]. Available: <http://www.tptp.org/CASC/>
- [15] G. Reger, N. Bjørner, M. Suda, and A. Voronkov, “AVATAR modulo theories,” in *GCAI 2016*, ser. EPIC, vol. 41. EasyChair, 2016, pp. 39–52.
- [16] G. Reger, M. Suda, and A. Voronkov, “Unification with abstraction and theory instantiation in saturation-based reasoning,” in *TACAS 2018*, ser. LNCS, 2018.
- [17] G. Reger, J. Schoisswohl, and A. Voronkov, “Making theory reasoning simpler,” in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021*, ser. Lecture Notes in Computer Science, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 164–180. [Online]. Available: https://doi.org/10.1007/978-3-030-72013-1_9
- [18] G. Reger and A. Voronkov, “Induction in saturation-based proof search,” in *International Conference on Automated Deduction*. Springer, 2019, pp. 477–494.
- [19] A. Bhayat and G. Reger, “A combinator-based superposition calculus for higher-order logic,” in *Automated Reasoning - 10th International Joint Conference, IJCAR*, ser. Lecture Notes in Computer Science, N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12166. Springer, 2020, pp. 278–296. [Online]. Available: https://doi.org/10.1007/978-3-030-51074-9_16
- [20] [Online]. Available: <https://vprover.github.io/>
- [21] G. Reger, D. Tishkovsky, and A. Voronkov, “Cooperating proof attempts,” in *International Conference on Automated Deduction*. Springer, 2015, pp. 339–355.
- [22] 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*. IEEE, 2004, pp. 75–86.
- [23] B. Babati, G. Horváth, V. Májer, and N. Pataki, “Static analysis toolset with Clang,” in *Proceedings of the 10th International Conference on Applied Informatics (30 January–1 February, 2017, Eger, Hungary)*, 2017, pp. 23–29.
- [24] K. Serebryany, A. Potapenko, T. Ishkhodzhanov, and D. Vyukov, “Dynamic race detection with LLVM compiler,” in *International Conference on Runtime Verification*. Springer, 2011, pp. 110–114.
- [25] G. Reger, M. Suda, and A. Voronkov, “Testing a saturation-based theorem prover,” in *TAP 2017*. Springer, 2017, pp. 152–161.
- [26] G. Reger, “Better proof output for vampire,” in *Vampire 2016*, ser. EPIC, vol. 44. EasyChair, 2017, pp. 46–60.
- [27] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, 3rd ed. ISO, Sep. 2011.
- [28] G. Reger and M. Suda, “Global subsumption revisited (briefly),” in *Vampire 2016*, ser. EPIC, 2017.
- [29] A. Voronkov, “AVATAR: The architecture for first-order theorem provers,” in *CAV*. Springer, 2014.
- [30] S. Schulz, “A comparison of different techniques for grounding near-propositional CNF formulae,” in *FLAIRS Conference*, 2002, pp. 72–76.
- [31] —, “Light-weight integration of SAT solving into first-order reasoners — first experiments,” in *Vampire Workshop*, 2017, pp. 9–19.
- [32] N. Eén and N. Sörensson, “An extensible SAT solver,” in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.
- [33] G. Audemard and L. Simon, “Lazy clause exchange policy for parallel SAT solvers,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2014, pp. 197–205.

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 978-3-85448-046-4



www.tuwien.at/academicpress